Technical University of Crete

School of Electrical and Computer Engineering

Heterogeneous computing for large-scale Linkage-Disequilibrium analyses on the Aris supercomputer

---

A work submitted in fulfillment of the requirements for the degree of

Diploma in Electrical and Computer Engineering

by

Charalampos Theodoris

Committee

Prof. Apostolos Dollas, (Chair)	Technical University of Crete
Assoc. Prof. Vasilios Samoladas	Technical University of Crete
Asst. Prof. Nikolaos Alachiotis	University of Twente

December 2020

Abstract

Linkage disequilibrium (LD) is the non-random association between alleles at different loci. In the field of Genomics, due to several breakthroughs in DNA extraction and sequencing technologies, huge databanks of genomic data have been created, and continue to grow every day. Along with said data, grows the need for a highly-performing solution in analyzing them. The prevailing analysis method of calculation for the LD in genomes uses single nucleotide polymorphisms (SNPs) to detect the absence and/or presence of minor alleles. Most software implementations to-date are not yet capable to efficiently manage the expected time and memory requirements of future large-scale genomic analyses. To answer the need for fast, scalable genomic analysis, we engineered and created a standalone software, qLD (quickLD) (*https://github.com/StrayLamb2/qLD*). qLD relies on prior observations that a high-performance approach on LD can utilize general matrix multiplications. Therefore, existing optimized computational kernels that calculate LD are employed. Alongside the optimized kernels, qLD applies memory-aware techniques to lower memory requirements and parallel execution using both CPU and GPU to reduce execution times even more. qLD in single-thread execution in the Aris supercomputer, delivers up to 8x faster processing than the current state-of-the-art software implementation when run on the same CPU and up to 44x when the computation is offloaded to a GPU. When used in multi-threaded executions in an off-the shelf laptop, we observed speedups of up to 50x against the same state-of-the-art software, employing the same number of threads. qLD also addresses a missing feature of state-of-the-art tools, the ability to quantify allele associations between arbitrarily distant loci, thereby facilitating the evaluation of long-range LD and the detection of co-evolved genes. We showcase qLD on the analysis of 22,554 complete SARS-CoV-2 genomes.

Περίληψη

Η Ανισορροπία Γενετικής Σύνδεσης (LD) είναι η μη τυχαία συσχέτιση μεταξύ αλληλόμορφων σε διαφορετικούς τόπους στο γονιδίωμα. Στον τομέα της Γονιδιοματικής, λόγω των τελευταίων ανακαλύψεων στην τεχνολογία εξαγωγής και προσδιορισμού DNA, έχουν δημιουργηθεί τεράστιες τράπεζες γονιδιωματικών δεδομένων, οι οποίες αυξάνουν τον αριθμό των καταχωρήσεών τους καθημερινά. Παράλληλα, δημιουργείται η ανάγκη για την αποδοτική ανάλυσή τους με βάση τα νέα μεγέθη. Η επικρατούσα μέθοδος ανάλυσης για τον υπολογισμό του LD στα γονιδιώματα χρησιμοποιεί πολυμορφισμούς μονού νουκλεοτιδίου (SNPs) για την ανίχνευση της απουσίας ή/και παρουσίας δευτερευόντων αλληλίων. Οι κύριες υλοποιήσεις λογισμικού μέχρι σήμερα δε διαχειρίζονται αποτελεσματικά τις επερχόμενες απαιτήσεις χρόνου/μνήμης των μελλοντικών αναλύσεων μεγάλης κλίμακας. Για αυτό το λόγο, δημιουργήθηκε η αυτόνομη εφαρμογή qLD (quickLD) (*https://github.com/StrayLamb2/qLD*). Το qLD βασίζεται στην παρατήρηση ότι το LD μπορεί να υπολογιστεί με μεγάλη απόδοση κάνοντας χρήση μεθόδων πολλαπλασιασμού πινάκων, και χρησιμοποιεί υπάρχοντες βελτιστοποιημένους υπολογιστικούς πυρήνες. Μαζί τους πυρήνες, το qLD χρησιμοποιεί τεχνικές διαχείρισης της μνήμης και παράλληλης εκτέλεσης με χρήση επεξεργαστή και κάρτας γραφικών, για περαιτέρω μείωση των χρόνων ανάλυσης. Σε εκτελέσεις ενός νήματος στον υπερυπολογιστή Aris, το qLD επιτυγχάνει έως και 8 φορές ταχύτερη επεξεργασία από το τρέχον πρόγραμμα τελευταίας τεχνολογίας σε εκτέλεση στον επεξεργαστή, ενώ με τη χρήση της κάρτας γραφικών η εκτέλεση είναι έως και 44 φορές ταχύτερη. Σε εκτελέσεις με πολλαπλά νήματα σε ένα συμβατικό λάπτοπ, επιτύχαμε 50 φορές ταχύτερη επεξεργασία έναντι του ίδιου λογισμικού, αξιοποιώντας τον ίδιο αριθμο νημάτων. Επιπροσθέτως, το qLD συμπληρώνει ένα κενό των εργαλείων τελευταίας τεχνολογίας, παρέχοντας τη δυνατότητα συσχέτισης μεταξύ αυθαίρετων, απομακρυσμένων περιοχών στο γονιδίωμα, διευκολύνοντας έτσι την αξιολόγηση του LD σε δεδομένα μεγάλης εμβέλειας, και την ανίχνευση των συνεξελιγμένων γονιδίων. Για την παρουσίαση της ανάλυσης του qLD σε πραγματικά δεδομένα, χρησιμοποιήσαμε σετ δεδομένων με 22,554 πλήρη γονιδιώματα του SARS-CoV-2.

Contents

List of Figures

Acknowledgements

My deepest gratitude for his assistance to Dr. Nikolaos Alachiotis, who introduced me to the world of high-performance computing and bioinformatics. He entrusted me with this whole project and was beside me in every step towards the completion of my thesis. We had excellent communication and I hope that we continue working together, since I have a lot more things to learn from him.

Secondly, I would like to express my deep gratitude towards my supervisor, Dr. Apostolos Dollas, who trusted in me and my collaboration with Dr. Alachiotis in successfully completing our work, even when this project seemed demanding due to certain time restrictions. He enabled all of this, and without it, I would certainly not be the engineer I am today.

I would also like to formally thank Dr. Paulos Paulidis, who provided us with a biological background and real-life datasets for this project, as well as aiding our research in many more aspects.

Lastly, I take this opportunity to thank my friends and family for always being there for me, supporting all my life decisions. Their love and belief in me is my drive to keep pushing further, without them I would not be the man I am today.

1. Introduction

Understanding the way nature and evolution work in species -especially in humans- is one of humanity's oldest goals. The science behind the research on this topic, biology, dates back to Assyrians and Babylonians [4] from discovered illustrations and descriptions of then-current medical and biological knowledge. Fast forward in the last couple of centuries [5], the new, rising way of research using experimentation, leads to the discovery of DNA in 1871. Over a hundred years later, in 1977, the first complete genome was sequenced, leading to the creation of a new field of biology, genomics. Genomics is a field that focuses on the structure, function, evolution, mapping, and editing of genomes. While being relatively new, it is an ever-growing field of research. In the last decades, using modern and cost-effective ways [6] to extract genomic information, data-banks are increasing in size exponentially [7]. This increase yields the deployment of memory- and performance-optimized computational approaches prerequisite for the analysis of future large-scale datasets.

1.1 Motivation

This work focuses on a widely used statistic in population genomics, linkage disequilibrium (LD). LD is used to identify interactions among co-evolved genes by identifying complementary mutations [8] or to search for traces of positive selection by revealing particular patterns in subgenomic regions [9]. In genome-wide association studies (GWAS), LD facilitates the detection of polymorphisms of interest, e.g., associated with human diseases [10], thereby contributing to the design of more effective drug treatments [11].

The preliminary steps of an LD study include a) DNA sequencing for a set of individuals of interest and b) mapping to a reference genome to create a multiple sequence alignment (MSA). These are followed by a so-called SNP calling step that identifies the polymorphic sites in the MSA, which are commonly referred to as single-nucleotide polymorphisms (SNPs). Non-polymorphic sites are not informative for LD analyses, therefore

they are pruned. Computing LD requires the calculation of allele and haplotype frequencies per SNP and pair of SNPs, respectively. Thus, compute and memory requirements increase linearly with the number of genomes (sample size) and quadratically with the number of SNPs. While the number of SNPs is limited by the chromosomal length, sample sizes continue to increase rapidly, fueled by advances in DNA sequencing technologies, as previously stated. To further put the sample-size growth into perspective, the 1000Genomes [12] project that launched in January 2008 sequenced 2,504 human genomes in an 8-year span [12], while well over 200,000 SARS-CoV-2, at the time of this writing, complete genomes are already available on GISAID [13] since the beginning of the ongoing coronavirus disease 2019 (COVID-19) pandemic (December 2019). To limit the obstruction by computational inefficiencies and/or excessive memory requirements in future scientific discoveries in the fields of population genetics and computational biology, high-performance software implementations that employ the underlying hardware efficiently and scale well with an increasing number of samples are required.

1.2 Contribution

Previous research on the topic of LD calculation revealed the capability to use general matrix multiplications to speed up the execution. This resulted in the development of two highly optimized kernels for modern microprocessor [14] and GPU architectures [3]. An additional feature that these kernels provide, is the ability to not only calculate LD in a single region, but also between different pairs of regions. This feature enables the analysis between distant regions or even chromosomes, without the need to create special input files for this exact reason, which also comes with a calculation overhead due to the nature of this problem.

As part of this work, an underlying interface that manipulates the input, memory, and communication with the kernels was created. With this interface, we first developed a sequential solution using each kernel. We then introduced parallel programming using POSIX threads, populating instances of the CPU kernel, which proves to be especially

useful in clusters, where there is an abundance of available cores. Finally, we conjugated both execution modes in a heterogeneous multi-threaded approach. Along the way, we further optimized the platform for efficiency, with High-Performance Computing (HPC) practices. The result of all the previous steps was the deployment of a highly performing open-source software solution, qLD (quickLD). qLD allows the execution of large-scale LD analyses, on off-the-shelf workstations in a fraction of the time required by state-of-the-art software. This work provides insight into the inner workings of the project, as well as benchmarks in several use cases, in two different machines. Along with the validation of our theoretical performance, we compare qLD with the state-of-the-art software, PLINK [15].

qLD outperforms PLINK 1.9 [16] as the sample size increases, achieving up to 8x faster processing with the two CPU implementations compared. When qLD offloads the compute-intensive task of calculating haplotype frequencies to a GPU, analyses complete up to 44x faster than PLINK 1.9, which cannot employ a GPU. Finally, in multi-threaded applications, with the collective power of the CPU and GPU kernels, qLD achieved 50x speedup over PLINK 1.9, using the same number of threads.

Part of this work is already published [17] in the peer reviewed BioInformatics and BioEngineering conference (BIBE2020), under the title "*qLD - High-performance Computation of Linkage Disequilibrium on CPU and GPU*". The video presentation from the conference is also uploaded in Youtube, under the same title: *https://www.youtube.com/watch?v=rkN3zvlWSNc*.

1.3 Structure

The remainder of this work is organized as follows. In Chapter 2, we provide a background on Linkage Disequilibrium and the means to transfer this biological problem into a computational one. In Chapter 3 we highlight previous and related work, along with the basis of our software, the optimized CPU, and GPU kernels. The design and workflow of qLD are presented in Chapter 4, while we evaluate its performance in Chapter 5. Finally, we discuss conclusions and further improvements in Chapter 6.

2. Linkage Disequilibrium

Linkage disequilibrium is defined as the non-random association between alleles at different loci (genomic locations). In simple terms, when offspring are produced, evolutionary forces such as mutation, gene flow, genetic drift, and natural selection are be applied. At a chromosomal level, this means changes in the offspring's alleles. LD is a metric that identifies co-occurring combinations of alleles between generations in a genome, more frequently than at random.

2.1 Importance of Linkage Disequilibrium in Biology

It is proven since the 1980s that LD is an important metric in genetics [18]. It is useful in a wide range of different genomic analyses, affecting and being affected by many factors in evolutionary biology. LD is as well deemed invaluable to genetics [18], due to being used in gene-mapping, large-scale surveys upon genomic data, and most recently in genome-wide association studies. In a wider scope of a genome, it reflects the population history, providing knowledge and information about past events, the breeding system, and geographic patterns in the history of evolution. Narrowing it down to each genomic region, it provides information about the natural selection, gene conversion, mutation, and other forces that cause allele frequency changes.

LD in population genetics is preferable among other selection models [19], providing a better insight in selective sweeps [20, 21], the processes in which beneficial mutations become dominant throughout the history of a species. On an opposite, to the beneficial traits, subject, LD can also be used to research genes that underlie risks of complex diseases [22]. By doing so, scientists were able to track areas and variations of genes as well as random genetic events linked to cancer [23–26] and type 1 diabetes [27]. Recently, LD was used to locate recombination hotspots in the SARS-CoV-2 genome by assessing the reduction of association between mutations with an increasing genomic distance [28].

2.2 Data Preparation

To conduct any LD computation, there are some preliminary steps [29], involving the extraction of genetic material, the partial assembly and annotation of the genome, the alignment of said parts, and finally a calling step in which variable sites are identified. A top level pipeline is depicted in Figure 2.1.

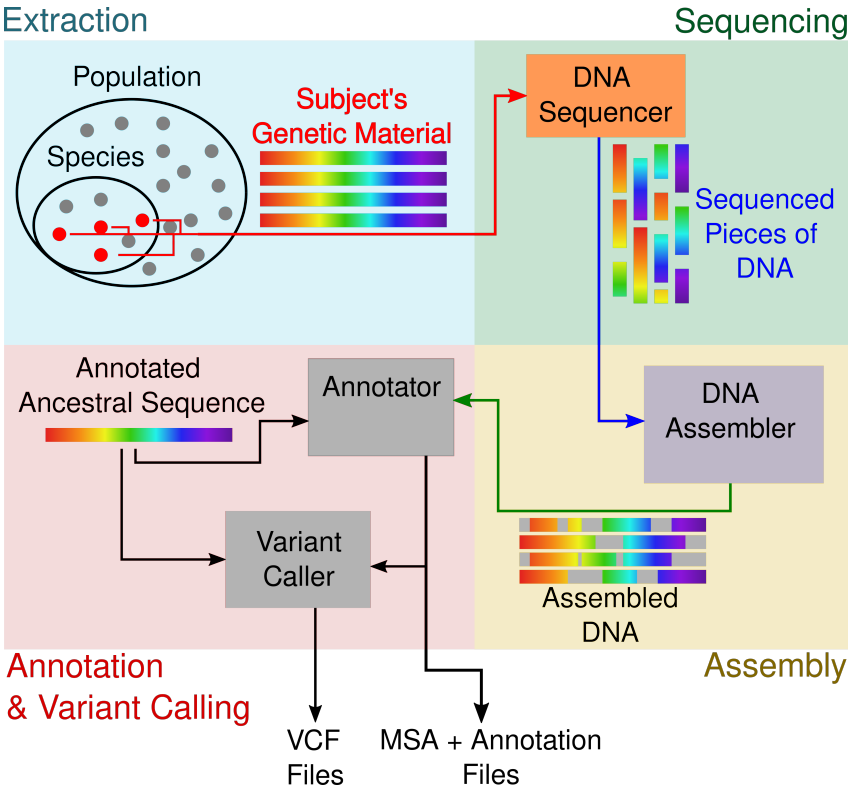

Figure 2.1: Top level example of the pipeline from DNA to Computer Data. The extracted DNA is first sequenced in multiple pieces that are then assembled, annotated and saved in files using the preferred data format.

Extraction To extract the DNA, individuals with representative characteristics of the species are selected, if capable to provide enough genetic material. DNA is then extracted, saving excess tissue from the same extraction in case of any later needs. Similarly, RNA can be extracted for RNA-sequencing, which helps in the research later on. The type, quality, purity, structural integrity, and preparation in general of the samples are integral to the quality of the whole study. Although not preferable, some pooling of individuals and amplification of the data can be used in cases of limited amounts of DNA. Contamination of the samples with other organisms is often inevitable to some degree, but in small amounts, it can be traced and filtered out in the next steps.

Sequencing To assemble a genome, the samples first pass a sequencing step, which includes multiple reads of the material, to gather as many repeat-free nucleotides as possible. Repeats are regions that occur in multiple copies, regardless their location in the genome. To produce quality results, a significantly greater number of nucleotides than the reference values of the organism under study, are needed. Approximations for the reference numbers can be found in several available databases, or else additional studies and investigations are needed. In terms of the technologies used in sequencing, the most prominent methods today involve next-generation sequencing (NGS), also known as second-generation sequencing (SGS), due to the extremely lower cost and much greater performance than the first-generation sequencing technologies. Many remarkable projects, as the 1000 genomes project, became a reality solely thanks to SGS. The state-of-the-art sequencing technology today though, is the third-generation sequencing (TGS). It focuses on producing the longest possible reads, tackling known issues of the previous technologies, such as repeats and high GC-content, that make the next step of the assembly difficult and sometimes ambiguous, although requiring more sophisticated correction steps due to error rates in the range of 10% to 15%.

Assembly Next is the actual assembly of the genome, along with pre- and post-processing steps of correction and polishing. This step requires heavy computational power, due to the sheer amount of data and can vary in performance amongst organisms and samples. From SGS to TGS the pipeline of this process slightly changes. SGS having trouble with aligning repeating areas correctly or having a bias on high GC areas due to the sequencing process [29] focuses more on a statistical analysis of all the possible alignments. Those alignments can also be way more in numbers than the TGS results, due to more, shorter pieces of sequenced DNA. On the other hand, TGS focuses on error correction, mainly by comparing the reads against themselves or by using shorter reads. In any case, ambiguous regions, or regions that were not sequenced, are marked as unknown and the extracted pieces are stitched together to complete the assembly.

Annotation The last step in the pipeline is the annotation. This is an entire chapter on its own, with many different processes, approaches, workflows, and choices to be made. It essentially is the part where each part of the assembled DNA is "mapped" to biological functions, genes, proteins, and all kinds of information about the genome. It is the step where mere raw genomic information is modeled and matched on the organism that provided it. The most important note in the annotation from the perspective of computer and data science though is the format in which the final annotated and assembled genome is provided. This is thankfully standardized in a few, widely accepted formats, such as GFF, GTF, Fasta, and others. These files are called multiple sequence alignments (MSAs), they contain several sites previously sequenced and annotated, and they, in turn, are processed to provide us with the final files needed for LD studies, SNP maps.

Variant Calling As mentioned previously, population genetics employ LD as a statistical measure to identify mutated alleles that are co-inherited more frequently than one would expect if these alleles were inherited independently. From a computational point of view, the input to an LD study is either a multiple sequence alignment (MSA), i.e., a $n \times m$ matrix that comprises n rows (one row per genome) of m columns each (also referred to as alignment sites), or a file that only comprises sites of interest, e.g., SNPs in a Variant Call Format (VCF [30]) file. An SNP is essentially an alignment site with two or more DNA states, i.e., at least one mutation has occurred at that site, whereas monomorphic sites are non-informative for computing LD and therefore are discarded. More information on the process of SNP mapping is provided in the next section.

2.3 Genomic Data Representation

Linkage Disequilibrium, as stated at the beginning of the chapter, is a metric of the non-random association between alleles at different genomic locations. This translates into differences between alleles in the genome of the subjects under investigation, and an ancestral sequence, which serves as a baseline. MSA files, which is the first useful data output of the genomic analysis, contains all the data to perform LD analyses, albeit

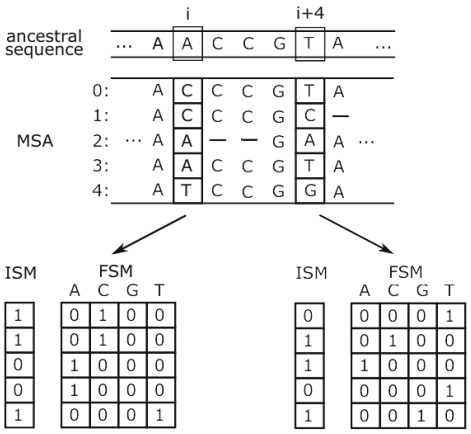

Figure 2.2: Example of a MSA that consists of an arbitrary number of sites, including two SNPs at locations i and i + 4, along with their respective representations under the ISM and the FSM evolutionary models. Adapted from [1].

containing much more information than needed. As depicted in Figure 2.2, an MSA file contains mapped alignments of DNA for each participant and an ancestral sequence, which are 'encoded' into new maps, using evolutionary models. A widely adopted evolutionary model in real-world analyses as well as *in silico* simulations is the infinite-site model (ISM) [31]. It assumes an infinite number of possible genomic locations where a mutation can occur, which leads to at most one mutation per site. In other words, every mutation appears on a site where no mutation has previously occurred. The ISM allows SNPs to be represented by binary vectors, where a '0' describes the allele state before a mutation (ancestral state) while '1' indicates the new state after a mutation (derived state). The other evolutionary model depicted is the finite-site model (FSM), which represents the same information in binary tables, along with the state after the mutation. We will focus on ISMs from this point on since this is the model of choice for most LD analyses. To identify mutations under ISM, each sequence in the MSA is compared, on a per-allele basis, with an ancestral sequence. Given an allele a_i at location i in the ancestral sequence, the binary vector s describing SNP i is constructed as follows, where [ ] is the Iverson bracket notation:

$$s_i^{a_i} = \{[s_{i,0} = a_i], [s_{i,1} = a_i], ..., [s_{i,N-1} = a_i]\}. \tag{2.1}$$

In this example, the alignment site i is the first SNP in the MSA, having mutations on

three out of five subjects. The ISM representation using the binary vector s should be:

$$s_i^A = \{1, 1, 0, 0, 1\}. \tag{2.2}$$

As we can see, the final information on the i^{th} site, given the ISM model, is that the sequences 0, 1, and 4 have mutated from the ancestral nucleotide base A. Calculating the s vectors of each site in the MSA results in an SNP matrix file, ready to use in any LD analysis. An example of the matrix G, which we henceforth refer to as the genomic matrix, is shown later in Chapter 4, where the input format is discussed in greater depths. Note that while it does not show the site locations, which are stored in separate memory space, the adjacent SNPs in G can be thousands of sites apart in the genome.

2.4 Calculating LD

Linkage Disequilibrium provides a pairwise score between SNPs, by calculating both the mutation frequency in each corresponding SNP and the pair, combined. These frequencies are denoted as allele and haplotype frequencies respectively, where haplotype is defined as a group of alleles inherited together from a single parent. In this context, it means the frequency in which a mutation has occurred concurrently in both alleles. Using those frequencies, we can compute the LD score of the pair using one of a few different formulas, which are explained later in this section.

2.4.1 Mutation Frequencies

Having extracted only the SNPs, the calculation of both allele and haplotype frequencies is relatively easy. We first need to count the number of mutations in each SNP (C_i for the i^{th} SNP), and the number of "shared" mutations, or rather the number of concurent mutations between the SNPs, (C_{ij} for the pair of i^{th} and j^{th} SNP respectively):

$$C_i = \sum_{k=0}^{N_{seq}-1} [s_{i,k} = 1] = s_i^T s_i, \tag{2.3}$$

$$C_{ij} = \sum_{k=0}^{N_{seq}-1} [s_{i,k} = 1]\,[s_{j,k} = 1] = s_i^T s_j. \tag{2.4}$$

From a biological standpoint, these counts are translated to the number of different sequenced genomes (amongst N subjects) than contain a mutation of the i^{th} allele, first in themselves and then as a pair. This means that the allele and the haplotype frequency, denoted as P_i and P_{ij} respectively, is the normalization of the aforementioned counts, by the number of sequences N_{seq} in the allignment:

$$P_i = \frac{C_i}{N_{seq}} = \frac{s_i^T s_i}{N_{seq}}, \quad P_{ij} = \frac{C_{ij}}{N_{seq}} = \frac{s_i^T s_j}{N_{seq}}. \tag{2.5}$$

2.4.2 Computing LD using D and D'

LD deals with the probability of independent events. The event that two mutations appear at different loci in the same sequence is said to be not independent, or in other words, the corresponding pair of SNPs are in linkage disequilibrium when the probability of the two mutations occurring at different loci in the same sequence is not the same as the product of the probabilities of these mutations occurring independently. Therefore, we compute

$$D_{i,j} = P_{i,j} - P_i P_j, \tag{2.6}$$

for every pair of SNPs, s_i and s_j, where $P_{i,j}$ represents the probability that a sample has mutations in both SNPs, s_i and s_j, and P_i and P_j are the probabilities for the independent events that a mutation has occured in s_i and s_j, respectively. When $D = 0$, s_i and s_j are in linkage equilibrium, i.e., mutations in s_i and s_j occur independently of each other. The two SNPs are in linkage disequilibrium when $D \neq 0$.

Using Equations 2.5 and 2.6, we can compute $D_{i,j}$ for all possible pairs of SNPs s_i and s_j in the following manner:

$$D_{i,j} = \frac{1}{N_{seq}}(s_i^T s_j) - \frac{1}{N_{seq}^2}(s_i^T s_i)(s_j^T s_j) \tag{2.7}$$

The LD formulation in Equation 2.6 is not widely employed because the sign and range of $D_{i,j}$ vary with the frequency at which different mutations occur, which hinders $D_{i,j}$ comparisons across different SNP pairs. Therefore, several standardization methods for D have been proposed. Amongst them, a proposal by R. C. Lewontin [32] suggested that D can be normalized by dividing by the theoretical maximum difference between the expected and observed allele frequencies:

$$D'_{ij} = \frac{D_{ij}}{D_m}, \tag{2.8}$$

where D_m is defined in accordance to the sign of D, eliminating it from the equation:

$$D_m = \begin{cases} max\left[-P_iP_j, -\left(1-P_i\right)\left(1-P_j\right)\right], & D < 0, \\ min\left[P_i\left(1-P_j\right), P_j\left(1-P_j\right)\right], & D > 0, \end{cases} \tag{2.9}$$

While D' is certainly a more accurate metric for linkage disequilibrium, providing scores that are disjoint from the fluctuation of allelic frequencies and effectively reducing their numeric range, it suffers in some corner cases. When the sample size is small, or in the case of rare alleles under examination, the produced scores are proven to be inflated. Such a bias can lead to skewed results and requires examination and error correction. This is one of the reasons that the most widely used measure to calculate LD is the next method, which we use in this work, the squared Pearson's coefficient, also known as r squared.

2.4.3 Computing LD using squared Pearson Coefficient

The formula to calculate LD using the squared Pearson coefficient [33] r^2_{ij}:

$$\begin{aligned} r^2_{ij} &= \frac{(P_{i,j} - P_iP_j)^2}{P_iP_j(1-P_i)(1-P_j)} \\ &= \frac{D^2_{i,j}}{P_iP_j(1-P_i)(1-P_j)} \end{aligned} \tag{2.10}$$

One of its main advantages is that all r_{ij}^2 values are in the range of 0 to 1, with higher values suggesting stronger association. This achieves the most optimal normalization to our knowledge, enabling linkage disequilibrium studies without the need for any compensation when comparing different genomic regions. Regardless of the employed measure, notice that the cost of computing the $r_{i,j}^2$ values for all pairs of SNPs is dominated by the cost of D.

Having established some basic knowledge in the background and the pipeline of the whole process of an LD study, the next section will proceed to the related work, covering existing applications and implementations.

3. Related Work

In the following sections, we present some of the most notable implementations and contributions to the LD studies, as well as state-of-the-art software that is most relevant to our work. Lastly, we present the basis of our work, two optimized, high-performance kernels that work in the heart of our program.

3.1 Software

Many ways to calculate LD exist, with many mathematical approaches surfacing throughout the years. Naturally, there exist many software implementations, either motivated by breakthroughs, or as performance-motivated alternatives to existing implementations. It is obviously impossible to track and count each successful attempt in calculating LD. Instead, the following are the most popular and widely used implementations, as per the recommendations of researchers in the field.

LDA is a Java based LD analyzer, released by Ding et al. [34]. It uses autosomal SNPs' genotypic data with unknown phase. The first stage of computations consists of either a Monte Carlo permutation or a x^2-test, to check whether the alleles at each locus are in Hardy-Weinberg equilibrium (HWE). For the alleles that are indeed on HWE, an expectation-maximization (EM) algorithm is deployed to calculate the four haplotype frequencies of pairwise loci. LDA uses several pairwise measures to quantify the degree of LD, such as the D, D' and r^2 and the significant association between two loci is tested through a Monte Carlo simulation-based likelihood ratio test or a x^2-test. LDA is programmed as a Java graphical user interface, presents a significant variety of choices to the user, allowing for several different analyses, and is capable of producing both text-based and graphical output. The only disadvantage of this tool, is the lack of multi-thread support, unfortunately making it unsuitable for large-scale analyses.

Haploview, released by Barett et al. [35], which is an open-source software for haplotype analyses written in JAVA. It boasts of a wide selection of input data formats, as well as

several genomic metrics, such as HWE, Mendelian inheritance errors and others. Through the use of a GUI environment, the program presents the option of pairwise LD calculations based on a wide array of LD measures, using either preset or user-defined groups of genetic markers. Haploview is another tool, widely used by the research community, that also suffers from a lack of a parallelized version. It has been updated and built upon for several years, but multi-threading is yet unimplemented, and no recent developments have been made.

SNPStats is a web-based application, developed by Sole et al. [36] for association studies analyses. It is designed from a genetic epidemiology point of view and is focused on enabling research around association studies based on both SNPs and biallelic markers. While being an easy solution to the average user, with a large amount of features, SNPStats is written in PHP and the computational core is implemented as a series of R packages. Small workloads can be executed with the easy-to-use web application, but larger ones require the user to deploy the R packages and manually develop the tests in R. This, along with the lack of parallelization hinders its performance.

Pfeifer et al. [37] released PopGenome, an R package for population genetic analyses that can compute a wide range of statistics, including LD. The vision of the project was the creation of a "Swiss army knife" that supports most widely-used input file formats, used in a well-established platform for geneticists, like R. It is established as a platform-independent, open-source framework with a great variety of implemented functionality, wide support for input files and many output formats, even in graphical representations. PopGenome supports multi-core execution, and most of the core functions are written in C or C++ for speed. Yet, the deployed LD kernel does not exploit neither the cache hierarchy, nor vector intrinsics, while the overall execution also suffers from the inherent overhead of interpreted languages, such as R.

Alachiotis et al. [38] released OmegaPlus, a scalable, open-source software for rapid detection of selective sweeps in whole-genome data based on linkage disequilibrium. OmegaPlus revolves around a bigger scope in research, utilizing LD as part of a bigger pipeline, internally. OmegaPlus' main statistic is the ω statistic [39], which is used

to measure positive-selection in populations. The ω statistic, in turn, bases on LD to calculate whether the mutations in a region are beneficiary, with r^2 being the default LD measure. Since the topic of selective sweeps requires large numbers of pairwise LD, among other, calculations, OmegaPlus is equipped with an optimized LD calculation implementation. In addition, it provides the ability of parallelization [40] with either POSIX Threads, or using the more generic, OpenMP API. While OmegaPlus uses efficiently the available system memory, similarly to PopGenome, it does not fully exploit the cache hierarchy either.

Chang et al. [41] released a comprehensive update to the widely used PLINK software [15] for whole-genome association and population-based linkage analyses. The updated implementation (PLINK 1.9/2.0) exhibits significant performance and scalability improvements in comparison to the initial software. It heavily relies on bitwise operations, multithreading, and high-level algorithmic improvements for the most compute-demanding functions, such as distance-based clustering and LD-based pruning. PLINK 1.9 implements the squared Pearson coefficient as a measure of LD and deploys the SSE2-based Lauradoux/Walish popcount algorithm to achieve high performance. Since PLINK 1.9 is the most used and latest stable high-performance implementation of LD, it became the benchmark for our performance.

3.2 Hardware

While software implementations provided many options to the LD research, there is significantly less research on hardware-based solutions. Most of the work on the general topic of population genetics in heterogeneous or solely bare-metal applications unfortunatelly does not revolve around LD. Three implementations in our knowledge can be considered related to this work, as high-performance LD platforms, one of which is used in qLD and presented in the next section.

First, Alachiotis and Weisz in their work on LD using FPGAs [2], utilize a database-like search model in two regions of an MSA to achieve pairwise correlation. This implemen-

tation uses the on-chip memory to act as a "query" of a subset of SNPs in a region, with the rest of the SNPs being streamed through the allele and haplotype calculators to finally enter the LD cores. Following this process, given an optimal number of LD cores for delay reduction, the pairwise score between the query and the streamed data is calculated. This implementation, while proven to be extremely efficient, lacks the ability to scale up with the trend of the genomic data, since one of its fundamental parts, the on-chip memory, limits the size of the datasets that can be supported.

Another implementation using FPGAs was the work of Bozikas et al. [1], which presented a novel hardware architecture for the calculation of LD in arbitrarily large datasets. By utilizing multiple levels of parallelism and efficient transformation of the memory data layouts, they created a bare-metal computation kernel capable of high-performance processing. Additionally, they mapped those kernels in a heterogeneous computing platform that enabled 4 parallel FPGAs to work simultaneously, using a high-speed memory interface. The resulting accelerator showed promising speedups, while supporting large datasets. As the authors noted in their publication, this work is not yet complete and set, but since hybrid computing is rising, it serves as a great contribution to the field.

3.3 Computing LD as GEMM

Alachiotis et al. [14] observed that the computational kernel for calculating LD can be cast in terms of dense linear algebra (DLA) operations. This allowed leveraging the collective knowledge in the DLA community in developing high-performance implementations for various microprocessor architectures, leading to the design of a highly efficient CPU kernel for LD that achieves between 84% and 95% of the theoretical peak performance of the machine.

Building upon the aforementioned DLA-based approach, which targeted modern CPU architectures, Binder et al. [3] presented a generic SNP-comparison framework that calculates LD on GPU architectures. The authors ported the proposed framework onto a variety of GPU platforms from different vendors, reporting between 55% and 97% of the

theoretical peak throughput of each specific GPU architecture.

These two kernels provide qLD with all the computational tools needed for an LD study. Using customized versions provided by their respective authors, tailored to use the same input data structures for compatibility, we were able to implement them with minimal effort into the platform. More detailed information on how the kernels work is provided in the next chapter.

4. Design and Usage

The design and functionality of the proposed software framework for high-performance computation of LD is presented in this chapter. We begin with the introduction of the supported input file format and the implemented memory optimizations for faster execution. We continue with presenting the two kernels used in qLD, their architecture and their approach on LD calculation using GEMM. Finally, we present qLD, its architecture, the general workflow in different use cases and the most noteable optimizations during its creation.

4.1 Input Files

Multiple data formats are used in LD studies, especially in the course of genomic informatics' history. HGVS [42], BED [43], GFF/GTF [43], GVF [44] are just a few of the most widely-used formats. In later years, VCF became the preferred format in LD studies, because it is unambiguous, scalable and flexible, allowing extra information to be added to the info field. Many millions of variants can be stored in a single VCF file. This is the reason why qLD supports the VCF file format for its input.

```
1  ##fileformat=VCFv4.1
2  ##FILTER=<ID=PASS,Description="All filters passed">
3  ##fileDate=20150218
4  ##reference=ftp://ftp.1000genomes.ebi.ac.uk//.../hs37d5.fa.gz
5  ##source=1000GenomesPhase3Pipeline
6  ##contig=<ID=2,assembly=b37,length=243199373>
7  ##ALT=<ID=CNV,Description="Copy Number Polymorphism">
8  ##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
9  ##INFO=<ID=CS,Number=1,Type=String,Description="Source call set.">
10 #CHROM POS   ID          REF ALT QUAL FILTER INFO            FORMAT HG00096 HG00097 ...
11 6      63854 rs544586840 T   G   100  PASS   AC=1;AF=0.00019 GT     0|0     0|0     ...
12 6      63979 rs561313667 C   T   100  PASS   AC=334;AF=0.066 GT     0|0     0|0     ...
13 6      63980 rs530120680 A   G   100  PASS   AC=354;AF=0.070 GT     0|0     0|0     ...
```

Listing 4.1: Variant Call Format example, partially derived from the Human chromosome 6. The header portion uses the # in front of the line. The body is Tab-delimited. This is a diploid genome, so the samples contain 2-bit information (X | x).

As depicted in Listing 4.1, the VCF file consists of a header that provides metadata describing the body of the file and the body itself. In the current state of qLD, header files are not processed semantically, apart from the validation of the format itself. The body of a VCF file is tab-separated into 8 mandatory columns, an optional format column that describes the samples, and an additional arbitrary number of sample columns. The mandatory columns provide information about:

- the name of the chromosome or sequence,
- the position of each variation in the sequence,
- the identifier of the variation,
- the reference base,
- the alternative alleles in the position,
- the quality score associated with the inference of the given alleles,
- the filters that the variation passed and
- a list of key-value pairs describing the variation

While being an easy to read format, VCF provides a lot of redundant information for qLD, while the user-friendly representation of each sample with a character does not fit the requirements of an HPC platform. Some more efficient alternatives have arisen through the years to accomodate the large data expansion, especially in GWAS. Honorable mentions in this regard is the proposal of BCF, the binary-VCF file format, found in the VCFtools package, the MVF [45] a format designed for phylogenomics and population genomic analysis, and SeqArray [46], built on-top of Genomic Data Structure (GDS) data format, which provides better data access and compression. Lastly, one of the most prevelant alternatives to VCF are the BED files, developed during the Human Genome Project [47] and later used by-default with PLINK. To this extend, qLD transforms the information into another, efficient file format, with perfect compatibility with our data structures, presented later in this chapter.

4.2 Data Representation in Memory

The ISM allows SNPs to be represented by binary vectors, where a '0' describes the allele state before a mutation (ancestral state) while '1' indicates the new state after a mutation (derived state). This representation is sub-optimal though, having each table entry take up 4 to 8 bits of memory, depending on the data type, while effectively needing only one. On the other hand, accessing each element separately could lead to suffocation of the memory controller, since the project is aimed towards parallel processing with a single, shared memory space, which could deal with several concurrent requests per cycle. Thus, to reduce the amount of memory space and accesses, we store each SNP as a group of N_{int} w-bit-long unsigned integers with N_{int} defined as follows:

$$N_{int} = \left\lceil \frac{N_{seq}}{w} \right\rceil, \tag{4.1}$$

with zero padding if $N_{seq} \mod w \neq 0$, and $w = 64$. The entire set of SNPs that collectively describe a genomic region of interest for computing LD is represented by a $(k \times w) \times n$ matrix, G, where $k = N_{int}$. An example of the matrix G, which we henceforth refer to as the genomic matrix, is shown in Figure 4.1. The genomic matrix G exclusively comprises SNPs. All monomorphic sites are already discarded during a preceding format conversion step, discussed in detail later in this section. For clarity reasons, Figure 4.1 does not show the site locations, which are stored in separate memory space, but note that adjacent SNPs in G can be thousands of sites apart in the genome. We henceforth represent an SNP as a column vector s.

4.3 Microkernels

The microkernels that qLD is based on, use general matrix multiplication operations (GEMM) to compute the haplotype frequency matrix, which is then used to calculate LD. Each of these kernels uses a different framework to speed up the calculations, based on the platform they are running. The CPU kernel uses the GotoBLAS implementation found in the open-source BLIS framework, while the GPU kernel uses custom functions,

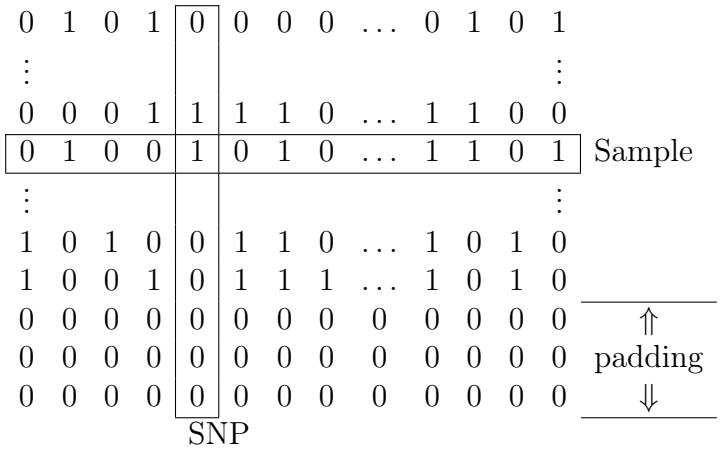

Figure 4.1: Pictorial representation of samples in the $(k \times w) \times n$ genomic matrix, G. The rows represent samples while the columns represent SNPs at different genomic locations. Adapted from [2].

running in the OpenCL framework for inter-platform communication.

4.3.1 CPU kernel

In a work published in 2016 [14], N.Alachiotis et al. proposed a reformulation of the LD computations to maximize the amount of operations that can be cast as matrix multiplication, based on their observation that LD formulas resemble matrix operations. As hinted in the equations 2.5, using a dense linear algebra (DLA) approach, we can find the desired frequencies with the following operations:

$$H = \frac{1}{N_{seq}} G^T G \tag{4.2}$$

$$D = H - PP^T \tag{4.3}$$

where the first operation computes all of the haplotype frequencies, storing them in a matrix H, while the second operation calculates and subtracts the product of the allele frequencies from H. P, similarly to the equations in Section 2, is the per-SNP allele frequency vector derived from an s SNP vector, in a genomic matrix G.

Using this formulation, the computation of H is an $O(n^3)$ operation, being a matrix

multiplication, while the subtraction is an $O(n^2)$ operation, given it is an outer product of two vectors. This became the predominant process in terms of computation cost as the number of SNPs gets larger and larger, which called for the following optimizations:

GotoBLAS approach To speed up the matrix operations, Alachiotis et al. turned to a popular approach in the DLA community, the Level 3 Basic Linear Algebra Subprograms (BLAS3), which are efficient algorithms for matrix multiplications on modern-day processors. In the BLAS approach, H could be translated into a general matrix multiplication (GEMM) operation, which is essentially the following high-performance, scalar formula:

$$C = \alpha AB + \beta C, \tag{4.4}$$

where A, B and C are of dimensions $m \times k$, $k \times n$ and $m \times n$, respectively. Without getting into too many details, the way GEMM works is by utilizing a rank-k update on the input matrices, breaking down the whole operation into much simpler blocked-dot products. A helpful visual representation is shown in Figure 4.2. Thankfully, Alachiotis et al. note that the format in which SNP data are allocated, is already optimal for use in a GEMM kernel, without further manipulation, making the integration of the kernel easy. The implementation of this approach was enabled by the employment of a well-known framework, the BLAS-Like Instantiation Software (BLIS) [48].

Popcount Another optimization in the overall process of LD calculation using this work as a basis, is the use of the intrinsic POPCNT operation for the calculation of the haplotype frequency P_{ij}. Remembering the Equation 2.5, given that s_i and s_j are binary values, P_{ij} equals 1 if and only if both alleles have mutated (having the value of 1, while 0 otherwise). Substituting the product with a count of ones in a pairwise AND logical operations greatly increases the efficiency of this step, further increasing the performance of the kernel. Therefore, haplotype frequencies are calculated using:

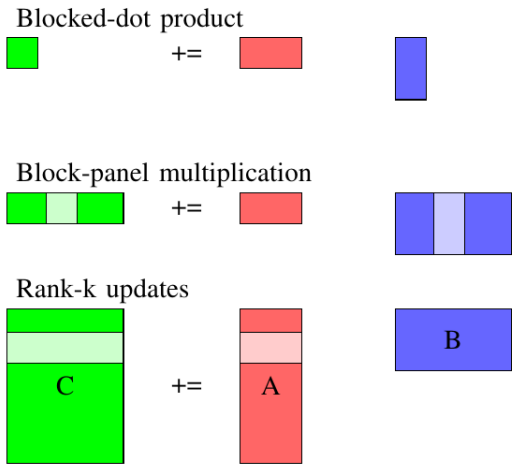

Figure 4.2: GotoBLAS layered approach to implementing a GEMM (rank-k) kernel on cache-based architectures. General matrix dimensions are first partitioned into rank-k kernels (bottom layer), which in turn are implemented as block-panel matrix multiplications (middle layer) These block-panel matrix multiplications are then implemented as blocked-dot products (top-layer). Adapted from [2].

$$P_{ij} = \frac{1}{N_{seq}} \text{POPCNT}\left(s_i \ \& \ s_j\right) \tag{4.5}$$

Data Format As stated in the beginning of the chapter, the data format in which the SNP maps are saved and used, employ a compression of the by-nature binary SNP elements into 64-bit unsigned long integers. Hence, accounting for this specific data format, which also introduces another optimization in the kernel, the new formula for the haplotype frequency is:

$$P_{ij} = \frac{1}{N_{seq}} \sum_{k=0}^{N_{int}} \text{POPCNT}\left(s_i^k \ \& \ s_j^k\right) \tag{4.6}$$

4.3.2 GPU kernel

The next kernel this work is based on, was developed by E. Binder et al. [3], albeit offloading the calculation of the haplotype frequencies to the GPU (second and third loop around in Figure 4.3), using the highly portable OpenCL framework. The model of the GPU architecture in Binder's et al. publication works on some assumptions, essential for the adaptation of the BLAS approach. Figure 4.4 provides a simplified visual representation. The main assumptions of their model, although stripped down to the minimum required features, briefly are:

- **Thread Group.** Each GPU comprises of N_{grp} groups of N_T number of threads that execute the same instruction at any given clock cycle. Known as warps and wavefronts.

- **Compute Cores.** A GPU is made up of N_C computational cores that perform independently. Known as streaming multi-processors or compute units.

- **Computer Clusters.** Each computer core comprises of N_{cl} clusters. Each cluster can execute a thread group independently.

- **Arithmetic Units.** Each cluster contains multiple arithmetic units, bound to execute a specific instruction. For any instruction fn there are N_{fn} arithmetic units in a cluster, varying based on the instruction, that have a latency of L_{fn} cycles. Each arithmetic unit can be pipelined through the use of L_{fn} different thread groups.

- **Shared Memory.** Each compute core contains a fast, shared memory of N_{shared} bytes, that are used from all the executed thread groups in the core. It is organized into parallel-access N_b banks.

- **Load/Store Architecture.** Data has to be loaded from memory before the computation. Each thread can load and store N_{vec} elements at the same time.

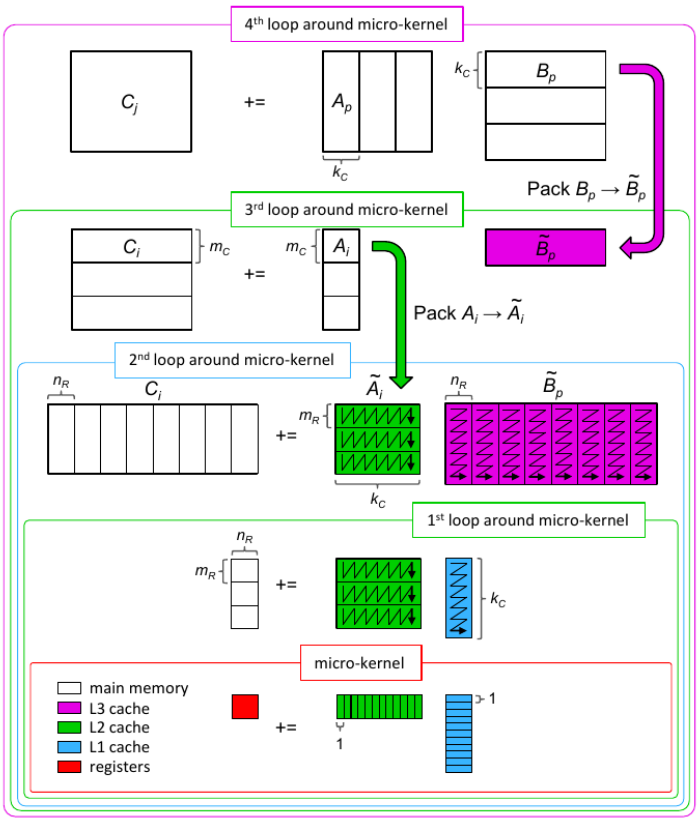

Figure 4.3: The GotoBLAS layered approach as implemented in the BLIS framework. Adapted from [3].

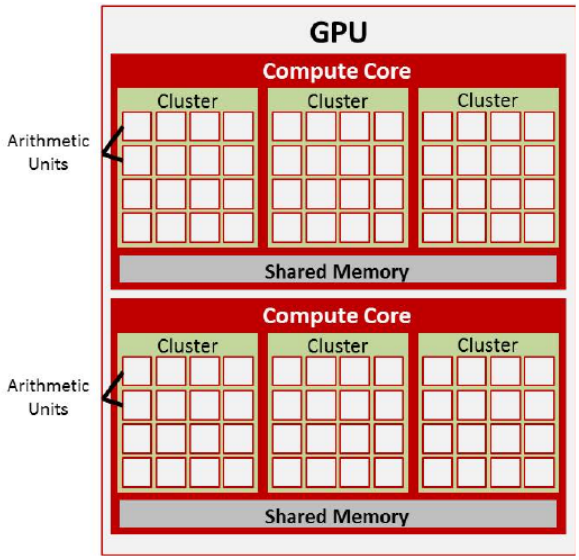

Figure 4.4: Organization of the hardware features on the model GPU architecture. The GPU above can be characterized by the following parameters: $N_C = 2$, $N_{cl} = 3$, and $N_{fn} = 16$. Adapted from [3].

Based on these assumptions, the sizes of the data structures need to be evaluated and parameterized, for the kernel to run at its full potential on each GPU model. For optimization purposes explained in their publication, the values are configured as follows. In Figure 4.3, which provides a more detailed insight of the BLIS architecture, the reader can easily see which size each value presents:

$$m_r = N_{vec}, \tag{4.7}$$

N_{vec} is the number of elements each thread in a thread group can load/store at the same time, while m_r is one of the dimensions in the second loop-around layer. Using this value maximizes the reuse of the output matrix.

$$m_c = \frac{N_b}{N_{cl}}, \tag{4.8}$$

m_c is one of the dimensions of the blocks packed into shared memory, found in the third loop-around layer. The number of banks N_b to the number of clusters N_{cl} minimizes the possibility of bank conflicts.

$$k_c = \frac{N_{shared}}{4N_b}, \tag{4.9}$$

k_c describes the number of elements in the shared memory space, given that each element is 4 bytes, and N_{shared} is the size of shared memory in bytes. Greater numbers than this would overflow the memory.

$$n_r \geq \frac{N_T m_r}{m_c} N_{vec} L_{fn} \tag{4.10}$$

Lastly, n_r, which is the last configurable dimensions, should be large enough to ensure the computation of a unique value each time, with a lower bound calculated as such, while also being as large as possible to speed up the process. There is no upper bound, because of the way each manufacturer evaluates its resources at a deeper level and how each compiler uses the available GPU registers. Given that the system is configured correctly,

this kernel promises immense speedups, while noting that, as is the case with the CPU kernel, popcount is still a bottleneck that stresses the need for better implementations that overcome this unavoidable limitation in the present.

In conclusion of this brief explanation of the kernels that we use, the optimizations they made to boost the performance of LD calculations are a key part of the success of this project. Using these kernels we managed to optimally calculate the most performance-critical part of the LD calculation pipeline, achieving sizeable speedup over the state-of-the-art software we benchmark with, as shown later in Chapter 5.

4.4 Execution Pipeline

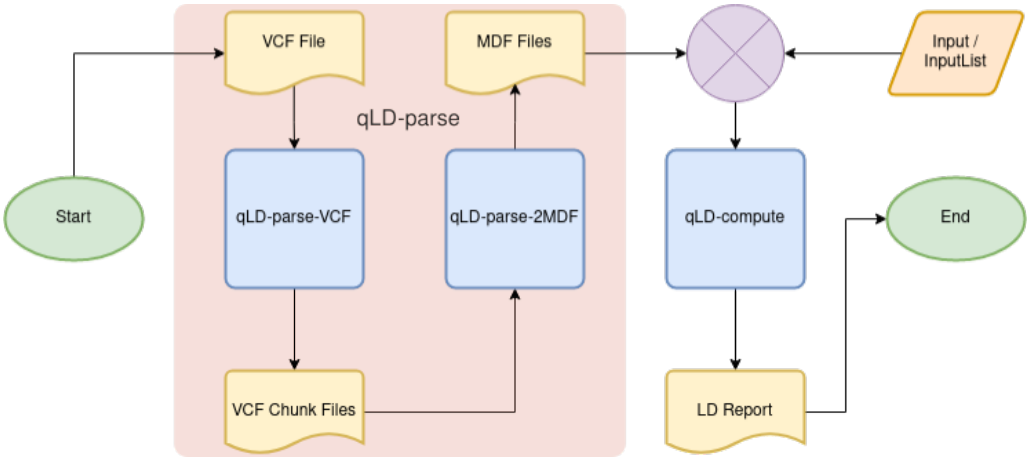

Figure 4.5: The qLD flowchart qLD-parse contains two subroutines, qLD-parse-VCF and qLD-parse-2MDF. If the MDF files already exist, only qLD-compute is executed.

Following the process from Chapter 2, we have VCF files consisting of SNPs. To start the calculation process, we must first preprocess and parse the data for validity, correctness, compatibility, and optimal use. In addition to performance, a major concern in designing an efficient software for large-scale LD studies is memory management, since requirements grow quadratically with the number of SNPs. For this reason, qLD implements a two-step process, as depicted in Figure 4.5, that separates parsing from processing, which allows pairwise LD calculations between arbitrarily distant SNPs to be conducted without increasing the memory space. We henceforth refer to the parsing and the processing modes of qLD as `qLD-parse` and `qLD-compute`, respectively.

4.4.1 qLD-parse

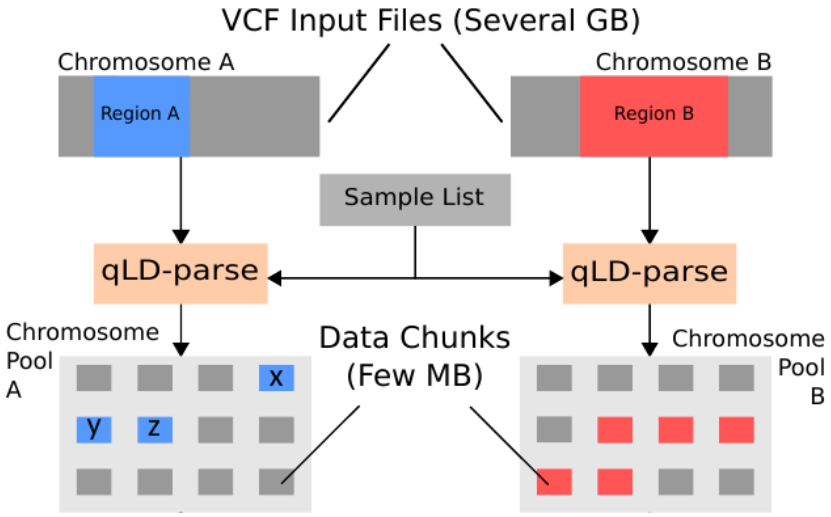

Figure 4.6: The qLD_parse workflow. Given a VCF input file, qLD-parse creates a chromosome pool containing corresponding MDF files. An optional sample list can select the samples to-be-included in the MDF files.

The first step of the process, `qLD-parse` (depicted in Figure 4.6), focuses on converting each–potentially large–VCF input file to an intermediate data representation that allows faster subsequent parsing and processing. The conversion process is performed once per VCF file and list of samples of interest. During this step, each VCF is split into a series of fixed-size files (`Chromosome Pool` in the figure) in a custom, LD-specific data format dubbed Matrix Data Format (MDF, see Listing 4.2). As presented, the VCF file is parsed and the valid samples (in the sample list) are converted into 4-bit unsigned words. A larger number of samples leads to a larger sequence of 4-bit words per MDF row. Note that MDF files to be processed using qLD contain 64-bit words. Each line also contains the total bit count per SNP (MDF row). The genomic coordinates (start and end position) of the subgenomic region stored in each MDF file are part of the filename. This convenient naming convention facilitates backward mapping of MDF files to the chromosomal region of the input VCF. Each MDF file is typically a few MB in size, regardless of the number of samples or SNPs in the VCF, with larger sample sizes leading to fewer SNPs per file.

```
1   ##fileformat=VCF
2   ... smp0 smp1 smp2 smp3 smp4 smp5
3   ...  1    1    0    0    1    1
4   ...  1    0    1    0    1    1
5   ...  0    0    0    1    0    1
6   ...  1    0    0    1    1    1

1   ##fileformat=MDF
2   ... Bitcount packed0
3   ...    4      15
4   ...    3      11
5   ...    1      1
6   ...    3      11
```

Listing 4.2: VCF-to-MDF conversion using the sample list “smp0 smp1 smp4 smp5" and assuming 4-bit MDF words for convenience. The bitcount of each resulting SNP is calculated in the process and included in the file.

4.4.2 qLD-compute

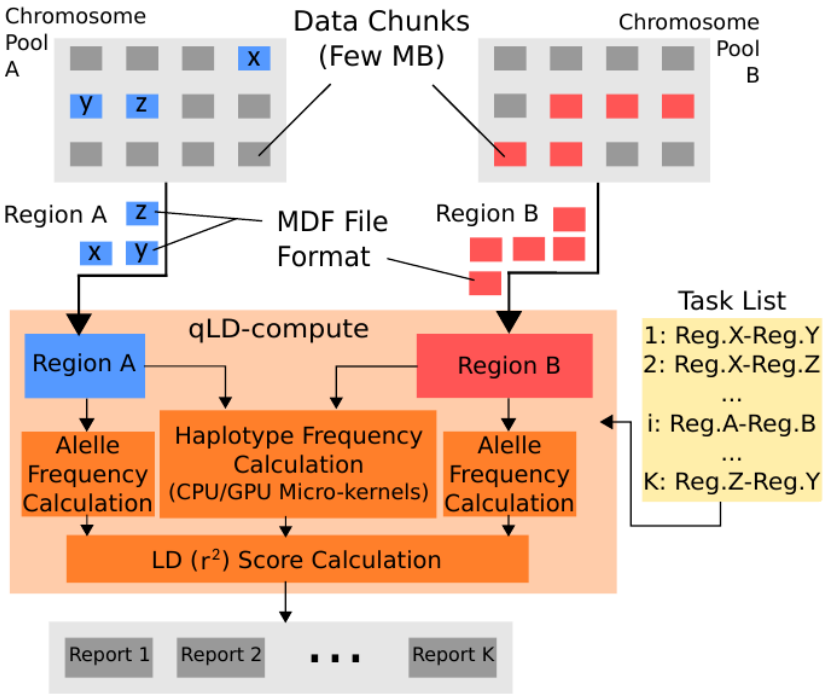

Figure 4.7: The qLD-compute workflow. Previously calculated MDF files get pulled into the program using a task list as input. The allele and haplotype frequencies are first calculated and then being used to produce the final LD score. A single report per task is saved in a user-designated directory.

The second step (`qLD-compute` in Figure 4.7) focuses on computing all LD scores between SNPs in pairs of genomic regions. A region pair, e.g., regions A and B in the figure, represents a compute task. qLD relies on the genomic coordinates that appear in the MDF filenames to parse the right subsets of MDF files and load the two chromosomal regions of a compute task to the main memory. For region A in Figure 4.7, for instance, only the MDF files x, y, and z are parsed. When the region pair is represented in the main memory, the kernels are employed, based on the user-selected computation mode, to compute the haplotype frequency. Thereafter, allele frequencies and the final r^2 scores are computed according to Equations 2.5 and 2.10, respectively. qLD produces a separate LD report per compute task.

4.4.3 Commands and Arguments

The following listings provide the required command lines for using qLD. A single asterisk indicates a required argument, while double asterisks indicate mutually exclusive arguments.

qLD-parse The first step of the qLD workflow is implemented through `qLD-parse`, which itself consists of two subfunctions: `qLD-parse-VCF` (splits a VCF to chunks) and `qLD-parse-2MDF` (converts each chunk to an MDF file). Listings 4.3 and 4.4 provide the basic input arguments.

```
./bin/qLD-parse-VCF
-inputList !$STRING!    **
-input     !$STRING!    **
-output    !$STRING!    *
-size      !$INTEGER!   *
```

Listing 4.3: qLD-parse-VCF command. Divides the initial VCF in zipped chunks. "input" and "inputList" arguments supersede each other. All of the arguments presented are required.

```
./bin/qLD-parse-2MDF
-input        !$STRING!   *
-output       !$STRING!   *
-sampleList   !$STRING!
```

Listing 4.4: qLD-parse-2MDF command. Transforms the previous zipped VCF chunks to MDF files. "sampleList" is an optional argument.

`qLD-parse-VCF` receives an input file that is either a single VCF (**-input**) or a list of several VCF files (**-inputList**) and creates a directory (**-output**) that contains VCF chunks of fixed size in MB (**-size**). The path to the produced output folder is input to `qLD-parse-2MDF` (**-input**), along with the list of samples of interest (**-sampleList**), which stores the generated MDF files in a user-given directory (**-output**).

qLD-compute In the second step, `qLD-compute` is launched to conduct the required LD calculations. Listing 4.5 provides basic input arguments.

```
./bin/qLD-compute
-input           $STRING    **
-input2          $STRING    **
-inputList       $STRING    **
-output          $STRING    *
-ploidy          $STRING    *
-r2limit         $FLOAT
-threads         $INTEGER
-mdf
-gpu
-compete
-sorted
```

Listing 4.5: qLD-compute command. Uses VCF chucks or MDF files to calculate the input tasks provided by the "input(2)" or "inputList" arguments. Arguments with an asterisk are required.

Using VCF chunks or, preferably, MDF files (**-mdf**) as input, `qLD-compute` generates

LD reports. Similarly to `qLD-parse-VCF`, either a pair of inputs (**-input**,**-input2**) are provided, in which case a single report is produced, or a list of several tasks (**-inputList**), in which case a report per task is stored to the output directory (**-output**). The ploidy, meaning the number of sets of chromosomes in an organism, therefore the number of alleles per loci, is also required (**-ploidy**). Optional parameters allow to deploy the GPU kernel (**-gpu**), set a number of threads (**-threads**) for parallel execution, apply optimizations (**-compete**, **-sorted**) and/or apply a threshold to the output (**-r2limit**).

4.5 Preprocessing

Before every LD analysis, the VCF files need to be preprocessed with `qLD-parse-VCF` and `qLD-parse-2MDF`, as mention previously. This is a serial execution step since the critical workload is bound by disk access, that cannot be parallelized. `qLD-parse-VCF` is a past contribution to the project, that manipulates the VCF files as we need, resulting in zipped chunks of data to minimize their footprint. Compatibility is maintained with these files in `qLD-compute`, as was intended before the start of the project. However, as the development proceeded, it was apparent that VCFs need further preprocessing to be used efficiently, which burdened the calculation step for no added benefits. This prompted the creation of `qLD-parse-2MDF`, which consists of all the VCF processing code. The data structure used in `qLD-compute` is, as mentioned earlier in this Chapter, arrays of 64-bit unsigned long integers, each representing 64 or fewer SNPs. Thus the MDF files, consisting of exactly this data structure, provide the information needed for the LD computation in a much cleaner and faster way, eliminating processing overheads, just by adding another small preprocessing step to the input data. The process of creating MDFs also handles the exclusion of non-polymorphic sites, that do not count as valid SNPs in an LD study. Although being a lossy procedure, the pruned information was of no interest in further steps of the process, thus providing further compression of the input, without loss of valuable data. Following this step, is the execution of `qLD-compute`, where the actual LD calculation takes place.

4.6 LD Computation

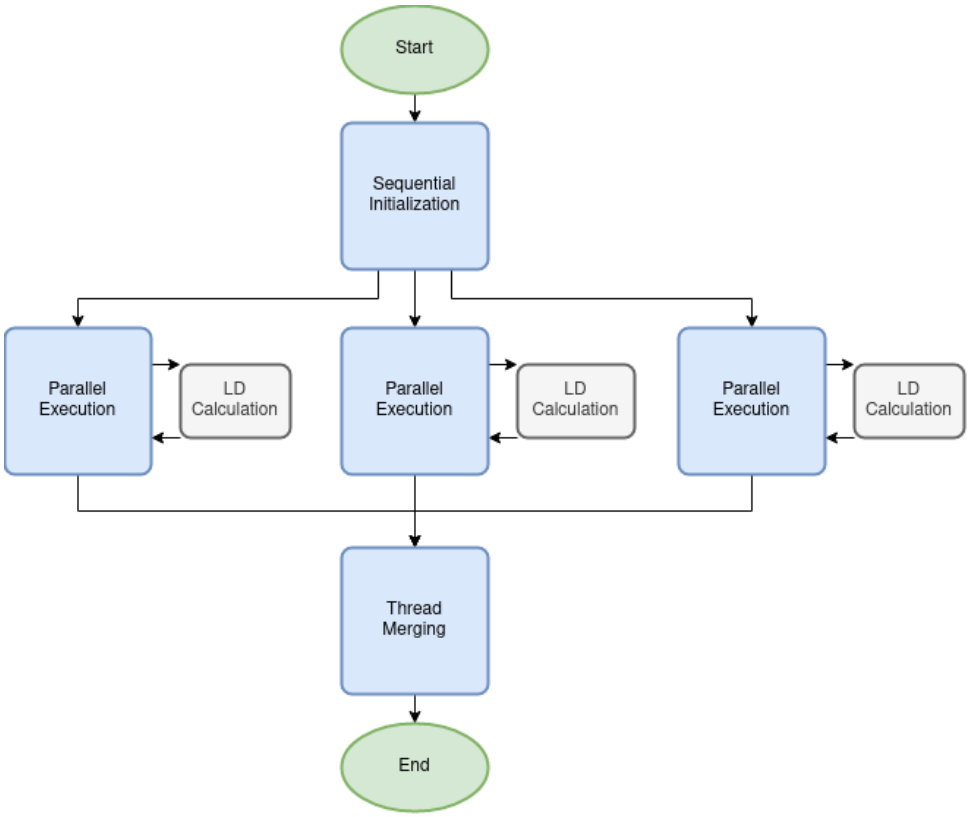

Figure 4.8: The qLD-compute flowchart. The initialization and argument parsing happens in a sequential process in the beginning. The LD calculations take place in the parallel threads, along with other complementary functions. Upon completion the threads merge and return a successful execution. Each process in this flowchart is further depicted in a deeper level in the following figures.

`qLD-compute` is capable of running in both single- and multi-threaded modes. Having said that, it should be explicitly specified that both modes execute within Pthreads, even if it is not strictly beneficial to the single-threaded execution. This design approach was decided after testing, where we found the overall benefits to heavily outweigh the potential downsides. Every contribution in the project revolved around the computational part of the platform, aside from some helper functions for proof-of-concept for the kernels. Therefore, it was fundamentally needed to develop a primal, serial version, before attempting to parallelize it. After completing this process, we immediately started to parallelize the core functions. Many techniques and different builds were tested, to ensure that our final model was the most scalable and memory-efficient approach given the nature of our problem. While building and testing the final versions of the parallel

approach, the truly serial and the single-threaded "serial" modes competed in several tests with different data sizes, to determine the realistic overhead of the deployment of Pthreads even for a single thread versus the serial code. We found out that given the workload in all use cases (minimum, average, heavy), the execution times were the same, within a margin of error in milliseconds while having almost no toll on the memory usage. Hence the serial approach was discontinued, for easier code maintenance, less clutter, and minimization of boilerplate code. The final version of `qLD-compute` works as depicted in Figure 4.8, and each step is further explored in the following subsections.

4.6.1 Preliminary Sequential Steps

In any kind of execution, the basic workflow consists of a serial execution part at the beginning and the end of the program, essential for proper initialization and completion of the program, and a parallel part that handles the calculations. The most notable functions (depicted in Figure 4.9) in the serial segment are:

Command Line Argument Parsing: Since the execution of `qLD-compute` happens exclusively from the command line, the first step is to parse the argument list, as shown in Listing 4.5. This updates any flags and variables needed, setting the execution path of the program. The *-threads* argument, in particular, is responsible for the selection between single- and multi-threaded operation, while the *-gpu* flag enables the computational offload to the GPU, establishing a connection to one of the graphics units if more than one exist in the system.

Task loading: A task is defined as a single, independent execution, be it in a single region or between a pair of regions. `qLD-compute` can work with single tasks or a list of multiple tasks as input, which are then loaded in memory, in a linked list data structure. Several checks ensure that the input is valid and remove invalid tasks while informing the user about the failed checks. When all of the tasks are loaded in memory, we proceed to the next step.

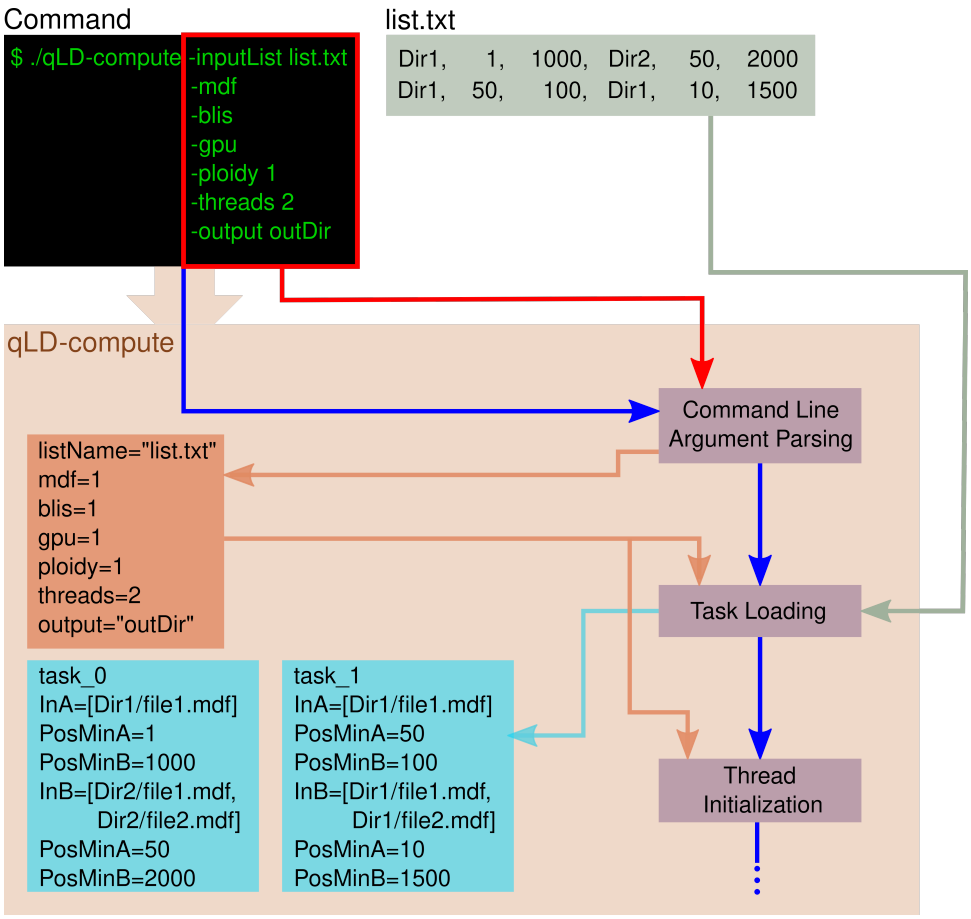

Figure 4.9: Sequential functions of qLD-compute. An example of the preliminary functions in a theoretical run with "list.txt". We assume that each input MDF file contains 1000 SNPs.

Thread Initialization and Work Assignment: `qLD-compute` uses POSIX threads for the parallelization of the computation step. One of the hardest choices in the whole project was deciding on which level we would perform the parallelization. The two ends of the spectrum in parallelization are fine and coarse grain. Grain refers to the routine which will run in parallel, fine being a small, specific function or calculation, while coarse meaning a broader part or even the entirety of the program. After small experimentation with fine-grain parallelization, we concluded that while it is plausible that in high scale calculations finer grains would be beneficial, they certainly didn't downscale well enough, having a low computation to communication ratio. Since the primal testing on coarse grain parallelization yielded much greater results, we opted to continue with a coarser grain, assigning a whole task execution as the parallel function. The communication between threads with this grain size is minimized due to the existence of the task queue

that concludes of independent tasks. Using this approach, we are also able to pre-assign the GPU handling in a single thread, if GPU offloading is enabled, even before the initiation of the parallel process. This prevents any bottlenecking between the two processing units and greatly simplifies the heterogeneity of the project. Hence, immediately after the thread initialization, we start the parallel execution, without any intermediate steps.

4.6.2 Parallel routine and sub-functions

The parallel routine utilizes POSIX threads and consists of all the steps depicted in Figure 4.7, under the `qLD-compute` module, including some helper functions. Each thread uses an internal loop to extract tasks from the task list, which is the only communication point between the threads, through a mutex. The mutex is responsible for an orderly dequeue of threads, without implications in case of simultaneous demand. The available memory for each thread is allocated beforehand when the thread first starts. The current model of the data structures uses simple statistical estimations based on the average frequency of SNPs in a genomic range, leaving open an expansion window, in case of miscalculations. In execution order, the main functions in a single thread are:

Preparation

Task Loading: In the preparation step, the thread dequeues a task from the list, if available, and loads the task-specific values such as data ranges, names, and directories for the input and output files, flags, and helper variables in local variable structs. It then proceeds to load the actual genomic data in memory, in the table creation step.

Table Creation step: The data tables each thread handles contain first and foremost the SNP vectors, then the position, the identifier, and the popcount of each site. Given the pruning and validation checks on the preprocessing step of creating the MDF files, no further manipulation of the data is needed, hence they are loaded immediately in the memory. Using the data provided from the loaded task, each thread opens the corresponding to the task's region files one by one, locates the lines containing valid data,

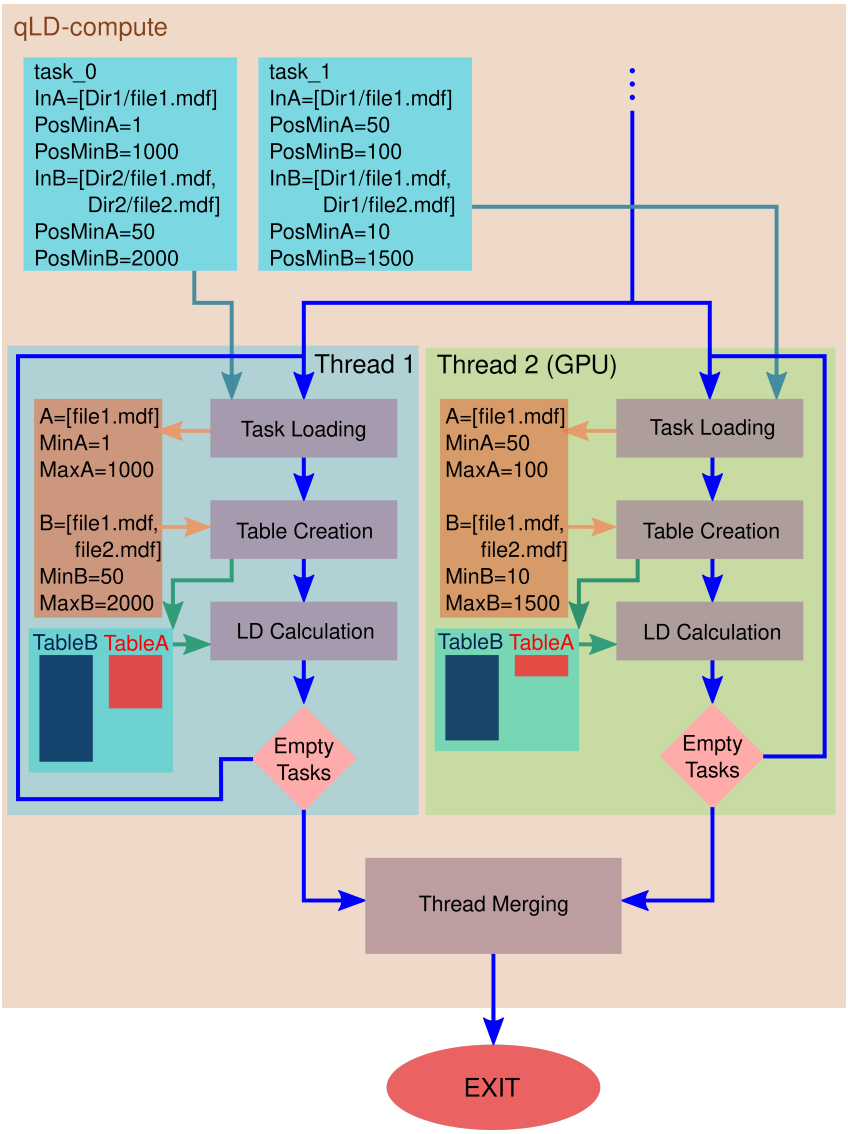

Figure 4.10: Parallel functions of qLD-compute. Continuing from the previous figure, 2 threads are used, utilizing the GPU. Each task is loaded in the corresponding thread and discarded when the calculations finish. When the tasks finish, the threads merge and the program ends the execution.

and loads the 64-bit unsigned integer values to the SNP table, along with all of the other information mentioned earlier. The memory space of the SNP tables, while technically being a matrix, is laid out in an array format, with an accommodating indexing function to this specific data structure. This layout is preferable and used by the kernels as well, reducing page faults and cache misses. After loading everything on memory, we proceed to the calculation of the allele and haplotype frequencies, the LD score calculation, and the output of the reports, using the CPU kernel in CPU threads and the GPU kernel on the GPU-offloading thread if enabled.

LD Calculation

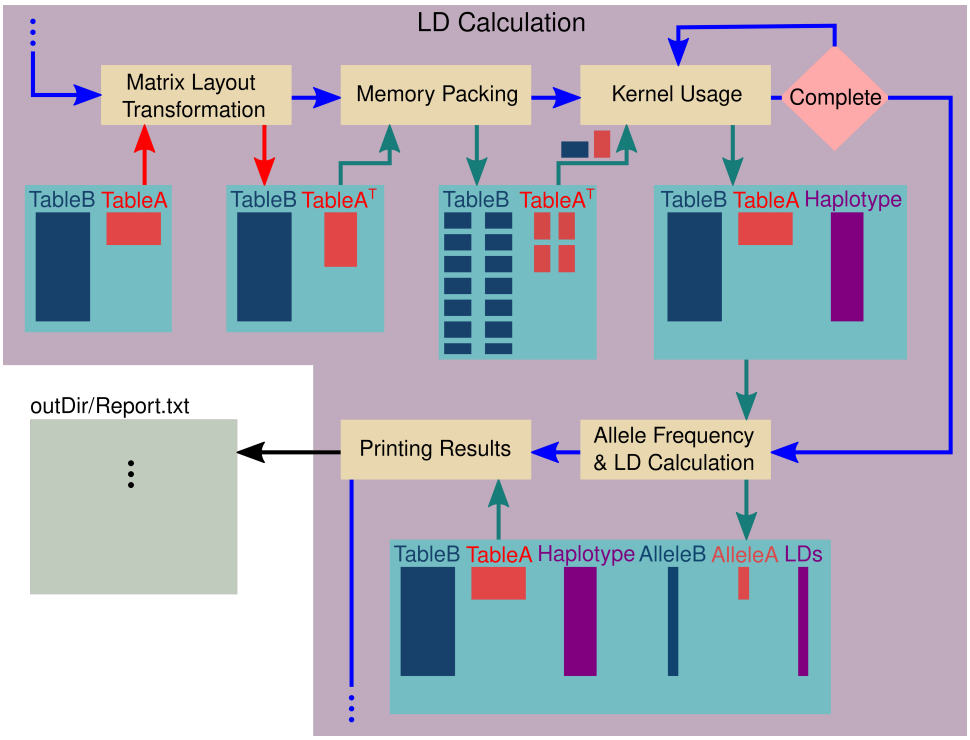

Figure 4.11: Inner functions of LD Calculation Step. The tables go through several memory transformations for better cache management. In the kernel usage step, the GPU is employed in the GPU-handling thread. To refrain from bloating the diagram it is not depicted differently from the CPU kernel employment.

Matrix Layout Transformation (MLT): Both the CPU and GPU kernels follow the same processes since they are based on the same approach to the LD calculation problem. Firstly, one of the tables corresponding to the input SNPs is transposed, to abide by the formats needed in Operations 4.2, 4.3. In the case of a single input region, we keep both the initial and transposed tables. Following this step, we have the memory packing.

Memory Packing: As mention in the kernel sections, the implementation of the GEMM operations involves looping through smaller blocks within the matrices. This allows efficient calculation in the limited cache space of a processor, with minimal cache misses. It also ensures that the memory size used in the calculation step is always bound and independent of the initial size of the calculated task. Thus, before the calculations, the memory is packed using a specific layout, creating blocks that are then fed into the kernels.

Using the CPU kernel: The CPU kernel uses the BLIS frameworks, which is linked at compilation time, and used in the form of a library. Thus, in this step, we simply call the GEMM function and provide the necessary arguments, alongside the SNP blocks.

Using the GPU kernel: The GPU kernel uses the OpenCL framework, and, as it offloads the computation to another processing unit, we first transfer the data needed through the connection established in the initialization phase. Without getting into details, since this kernel is not our work, the OpenCL framework initiates a program written in OpenCL language, running the GEMM micro-kernel. When the data are transferred successfully, this program is executed, calculating the resulting table. Upon finishing, it transfers the results back to CPU memory space. Appropriate testing was done to ensure the optimal fit of the kernel to the specific GPUs, as per the instructions of its developers.

Allele frequency & LD calculation: When the resulting haplotype frequency tables are complete, we proceed to the allele frequency calculation, along with the LD calculation, using the Squared Pearson Correlation Coefficient (Equation 2.10). This consists of four divisions, and the outcome is then saved in the list of the results.

Printing the Results: The results are printed into a file for every task. The data reported are the IDs and positions of the SNPs, along with the allele frequency of each SNP and the LD score of the pair. Depending on the limit set by the user (*r2limit*) and the task size, the output file size can vary from some MBs to several GBs. This puts a heavy load on the disk I/O, which can tremendously slow down the process, especially in the case of multiple threads. Apart from the application of r2limit from the user, which is fairly common practice in LD studies, which research **strong** correlations, we further optimized the writing to a sufficient level. The first optimization comes from utilizing C's string and file buffers efficiently, to reduce the amount of disk writes. The second optimization is presented in the next section.

4.6.3 Completion

Thread Merging and Program Completion After the successful printing of the report, the thread successfully finishes the task, resets to the beginning of the process, and pulls the next task. If it fails, it returns to the caller function, indicating the completion of the execution. Upon its return, each thread immediately goes into an inactivity state, waiting for the other threads to finish. When all threads are inactive, the master thread shuts down the program, completing a successful run.

4.7 Additional Optimizations

The qLD platform was developed using High Performance Computing (HPC) practices for input/output (I/O), memory management and overall performance. The kernels that calculate LD are also optimized. Therefore, to further boost the performance of qLD, we devised of qLD-specific alternatives to existing implementations, that perform faster than their generic counterparts. These optimizations include the creation of Matrix Data File format, a custom blocking routine for the input tasks, different task-to-thread allocation methods and a custom printing function using lookup tables:

4.7.1 MDF parsing

The Matrix Data Format (MDF) File

One of the most common file formats for gene sequencing variations is the variant call format (VCF). A typical VCF contains all the necessary information for an LD study, written in a human-friendly way, albeit being quite inefficient for a high-performance LD platform. Each element of the SNP matrix is represented with a single (or more, depending on the ploidy) character while providing binary information. This creates a significant overhead on the execution, where we must pack the information to fit it in the limited ram and cache spaces.

```
1  ##fileformat=MDFv0.1
2  #CHROM POS   ID       BC       compSNP_0            compSNP_1            compSNP_2
3  23     1     rs01     50104    728691394239593355   484860778953314344   5852295991283862418
4  23     16    rs85     50046    5889810797632891717  4067052671618687417  2187254492323864484
5  23     158   rs35     50031    4733340049119312308  3565397348654009321  1345173088689663059
6  23     160   rs520    50019    3080731532467135050  6110428055623626207  1674574188567292802
7  23     470   rs1050   50144    4357582525877649028  3817763505454594719  7760019307366416598
```

Listing 4.6: Matrix Data Format example, each sample block contains 64 samples encoded in unsigned integer values. CHROM, POS, ID are the same with the initial VCF values.

The kernels used in qLD, for those same limitations, handle the SNPs as arrays of 64-bit words, where each word represents 64 samples of the SNP. This was the deciding factor on the optimization of the input files, with the creation of MDF. The resulting MDF files, as shown in Listing 4.6, share the most basic SNP information with their VCF counterparts but differ in the packing of SNP arrays into 64-bit word arrays.

Another process that further optimizes the files both in size and execution time on the computation step, is the trimming of invalid sites. Since the creation of MDF files needs to first read each site, we capitalized on that read as much as possible. LD studies require SNP inputs, where VCFs provide sites as input. An SNP is always a site, while the opposite doesn't apply. Since the rules that validate an SNP require to first read the whole site, we prune non-SNP sites in the MDF creation, to both save space in the disk and calculation time.

Additionally, we calculate the bit count, meaning the number of 1s, in each SNP, and add it to the SNP's information in the file. This is used in the calculation of the allele frequencies in the calculation step, which is a part of the LD calculation.

Having already calculated the bit counts and pruned the invalid sites, the input from an MDF file is just directly fed into the correct tables in the calculation step, without any further processing.

Before the decoupling of file processing and the computation step, each run suffered from a significant overhead, that was deemed unnecessary. Since the VCF files are still processed into MDFs though, the trimmed time is not avoided per se. The performance gain is cumulative and related to the number of tasks a user needs from a specific dataset, therefore it is not accurately measurable. Even if we can not give a definitive peak

performance gain in the best-case scenario, it is certain that in the worst case, it does not hinder the process, while promoting a better workflow practice.

4.7.2 Custom Blocking Factor

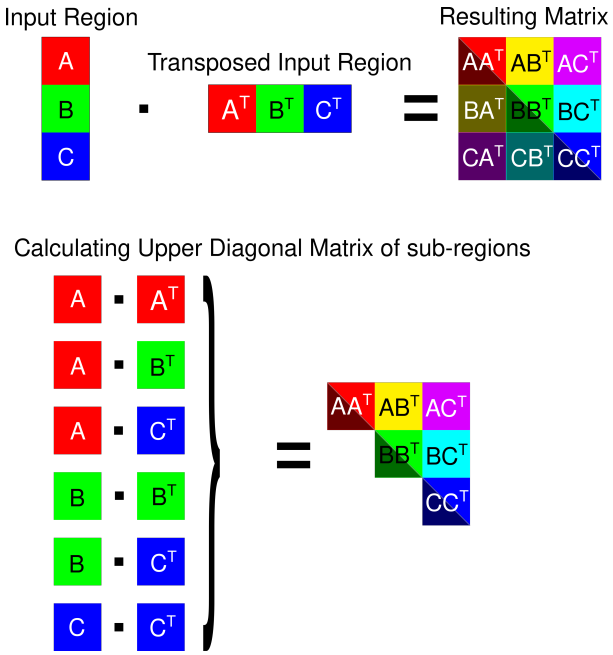

Figure 4.12: Example of a spliting input task. Each block contains a subdivision of the initial region. Splitting the input enables the trimming of redundant information.

An experimental feature that yields great results especially in single region executions, is the segmentation of the input task in smaller sub-tasks. The main idea behind this feature is to expand the concept of blocking, that our kernels use when calculating haplotype frequencies. Given that GEMM is designed to calculate whole matrices, single region calculations produce results that mirror on the diagonal. Using a blocking technique on the SNPs of the input task, as shown in Figure 4.12 we can approximately calculate only the results closer to the diagonal matrix. As depicted, the darker half of the resulting matrices is redundant information. When replacing the initial task with the new multiplications between the blocks, we can trim the calculations by removing the redundant regions. Having separated the input task in N blocks, the resulting "optimized" task list has a size of:

$$\#Tasks = \sum_{x=1}^{N} x, \tag{4.11}$$

and the number of redundant calculated LDs has shrunk dramatically. Specifically, given an input matrix with N SNPs, the number of elements of the upper triagonal matrix (excluding the diagonal, given it is always equal to 1) is:

$$\#Elements = \frac{N(N-1)}{2}, \tag{4.12}$$

as opposed to N^2 on the whole resulting matrix. Using this optimization step effectively, we aim to approach the triagonal matrix as close as possible. Additionally, by lowering the dimensions of every single sub-task, we ensure that the memory footprint of each thread is smaller, enabling better scalability when we add more threads in the execution since the available memory for all the threads is shared and also limited.

4.7.3 Competing Task Queue and Sorting

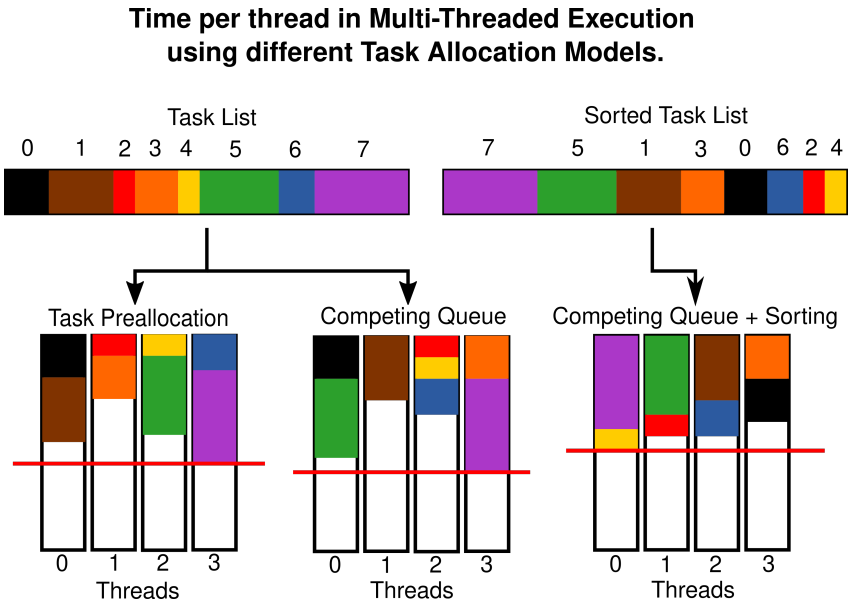

Figure 4.13: Example of the task allocation on threads. Task preallocation assigns regions of tasks to each thread, while the two competing implementations allow each thread to pull a task from the list whenever it finishes calculating the previous task.

The parallelization of several tasks of dynamic sizes can lead to imbalances in the load of each thread. To combat this issue, a competing queue was implemented, where each thread competes to pop the next task upon completion. A competing approach to the allocation of the tasks ensures balance over time. To further improve upon this idea, a sorting method was implemented, for corner cases such as when heavy tasks exist at the end of the queue. The sorting method is not taxing on the performance of the platform, since the queue is implemented as a linked list. Sorting such a structure with the merge sort algorithm has $O(N * logN)$ time complexity, which is low compared to the total execution time of an average LD workload. In Figure 4.13, a typical allocation with all the different methods is depicted.

Task Preallocation: Using the task preallocation method, where each thread is assigned to a region of tasks, it becomes apparent that the performance can vary depending on the order of the tasks. Each thread, in the example, gets a region of two tasks. This creates an imbalance, where the critical thread is thread number 3.

Competing Queue: Changing to a competing approach, we should expect to see better results, but this is not depicted in the example. On the contrary, thread 3 has been assigned an even greater workload. This is not the typical case, being an intentional exaggeration due to the small number of tasks and a badly weighted initial task list. Given a greater number of tasks, the average workflow would eventually be smoothly allocated, due to the competition. Cases like this though led to the implementation of the sorted competing queue.

Sorted Competing Queue: Sorting the task list, we ensure that the grain of our tasks gets finer as we get to the finish of the execution. This way, the last tasks act as fillers to the time gaps created by their larger counterparts. As we can see in the example, this leads to the most balanced execution, along with the best execution time.

4.7.4 Printing using Lookup tables

To improve write times, we experimented with a custom printing function specifically for the floating-point numbers, since fprintf is a general-purpose function and introduces a substantial overhead on type conversions. As shown in Listing 4.7, we finally opted to use Lookup tables, since it is the most common technique to combat mapping overheads, especially to text. Based on the fact that the output is dominated by floating-point values (FPVs), we map each FPV to a character array from '0' to '9', where each digit points to its representation in the array. This simple optimization achieves up to 1.7x faster printing, with 10^6 FPVs with 9-digit precision stored in a file in 0.123 seconds, while requiring 0.208 seconds when the standard C library function `fprintf` is used.

```
1  def reverse(num):
2      while num > 0:
3          digits++
4          rev = rev * 10 + num % 10
5          num = num / 10
6      num = rev
7      return digits, num
8
9  # Main function, separates the integer
10 # from the decimal part, then reverses
11 # (since printing works as a queue)
12 # and prints digit-by-digit the given
13 # number in the right order with the
14 # given precision
15 def custom_print(fp, value, precision):
16     char_array[10] = "0123456789"
17     pow10 = 10 ^ precision
18     int_part = (int)value
19     decimals = value - int_part * pow10
20     [int_dig,inv_int]=reverse(int_part)
21     [dec_dig,inv_dec]=reverse(decimals)
```

```
22      if inv_int == 0:
23          fputc('0', fp)
24      while inv_int != 0:
25          c2p = char_array[inv_int % 10]
26          fputc(c2p, fp)
27          inv_int = inv_int / 10
28          int_pos++
29      for i = [0:1:int_dig - int_pos]:
30          fputc('0', fp)
31      fputc('.', fp)
32      for i = [0:1:precision - dec_dig]:
33          fputc('0',fp)
34      for i = [0:1:dec_dig]:
35          c2p = char_array[inv_dec % 10]
36          fputc(c2p, fp)
37          inv_dec = inv_dec / 10
```

Listing 4.7: qLD custom printing routine for floating-point values based on a lookup table approach. (C code implementation available at : *https://github.com/StrayLamb2/qLD/blob/master/src/correlator/src/fast_print.c*)

5. Performance Evaluation

In this chapter, all of our test results are grouped by their scope, in their respective sections. The datasets used in the following evaluation derived from a single haploid dataset of 20k SNPs and 100k samples. Depending on the specific needs of each test, qLD-parse created a sub-set of MDF files, accordingly. We decided to evaluate our performance on haploid data, since using diploid data would only effectively double the sample size (2-bit information per sample). Real or realistically-made simulated datasets include a high percentage of monomorphic sites or sites with missing data, making it impossible to determine their SNP size prior to the experiment, so we refrained from using such datasets in the early development stages. When the project reached its development targets, it was already capable of running with real datasets. To showcase this, we used qLD for the computation of LD in the much recent and trending COVID-19 set.

The following sections introduce the systems in which we tested our code, and cover both the single- as well as the multi-threaded performance. The single-threaded section contains the baseline evaluation of qLD in terms of execution speed, throughput, and speedup. Additionally, a time breakdown of a typical qLD single-threaded execution in its main functions is presented, which provides a better insight into the allocation of the execution time. Finally, we present the results on the custom blocking of the input tasks, which is an optimization of the single region executions.

The multi-threaded section evaluates the heterogeneous CPU and GPU environment, involves the co-ordination of both computing units, and showcases the ability to simultaneously use them while balancing the load to compensate for the differences in execution speed. In addition, we tackle some of the challenges we faced and use performance graphs to establish the choices made. Lastly, we prove the ability of qLD to process real datasets, using a Covid19 dataset, derived from the available SARS-CoV-2 sequences in the GISAID database. (*https://www.gisaid.org/*).

5.1 Experimental Setup

In our experiments, we tried to evaluate the program in as many cases as possible, firstly because we had to ensure that it works in different machines with different hardware configurations and secondly to benchmark the differences between mainstream and server components. Given the aforementioned reasons, we performed experiments on an off-the-shelf personal laptop and the ARIS supercomputer (*https://hpc.grnet.gr/en/*). Table 5.1 provides the specifications of the employed test platforms. We compare our performance with the software PLINK 1.9 [41], since it is well designed, well known, and widely used. PLINK 1.9 is also able to directly compare in most of our use cases, both in single-thread, as well as in multi-threaded runs. The only limitations in terms of compatibility are the ability to run bi-regional correlations, which we found impossible in PLINK 1.9 without having at least a significant computational overhead of redundant data, and the utilization of the GPU as a processing unit. All runs were performed using the default r^2 limit of both qLD and PLINK 1.9 ($r^2_{limit} = 0.2$), which serves as a high-pass filter on the output. We opt-in filtering the output, because in most realistic cases the unfiltered outputs can grow in sizes of hundreds of GBs quite fast, while the useful information in them is only a few GBs or less.

	System 1	***System 2***
Description	Off-the-shelf laptop	Aris supercomputer
CPU Model	Core i5-8300H	Xeon E5-2660v3
Microarchitecture	Coffee Lake	Haswell
Nominal Frequency	2.3 GHz	2.6 GHz
Max. Turbo Frequency	4.0 GHz	3.3 GHz
Processors	1	2
Cores/Processor	4	10
Total Cores	4	20
Memory	8 GB	32 GB
GPU Model	GTX 1050-M	Tesla K40
Streaming Multiprocessors	5	15
Cuda Cores	640	2880
GPU Memory	4 GB	12 GB

Table 5.1: System Specifications

5.2 Single Thread Performance

The single-threaded performance tests of this project consist of 4 different factors; Execution speed, throughput, speedup, and breakdown. Execution speed is used as an evaluation of real-life user experience. Throughput, although tied to the execution speed, gives us a better understanding of the scaling capabilities of the interface, since the kernels are already tested and optimized for scaling. As speedup, we compare the gain in terms of execution speed against the state-of-the-art PLINK v1.9, as a benchmark of improvement over the current state of LD research platforms.

We then break down the execution time in the four major components of the program. These graphs are used to find out if qLD still has a critical path on the LD calculations, when the optimized kernels are used, or rather we should switch our focus on a newly found critical path in the future of the project.

Lastly, we provide testing results on a promising improvement in input manipulation, by fragmenting the input task into several sub-tasks.

5.2.1 Execution speed comparison

In the first test, we compare the execution speed of PLINK 1.9, the CPU kernel, and the GPU kernel in qLD. We aim to show, not only that we operate faster than the state-of-the-art solution, but also that we do not hinder the performance of the kernels compared to the previous work. Figure 5.1 illustrates execution times of qLD and PLINK 1.9 when the sample size increases up to 100k (Fig. 5.1A), and the number of SNPs increases up to 10k for fixed sample sizes of 2,5k (Fig. 5.1B), 10k (Fig. 5.1C), and 100k (Fig. 5.1D). Expectedly, execution times increase linearly with the number of samples and quadratically with the number of SNPs.

We can also observe that System 1 (off-the-shelf laptop) outperforms System 2 (ARIS supercomputer), exhibiting shorter execution times overall. This is expected because we test the sequential execution, and the operating frequency of System 1 (3.8 GHz turbo frequency) is about 40% higher than the nominal frequency of System 2 (2.6 GHz).

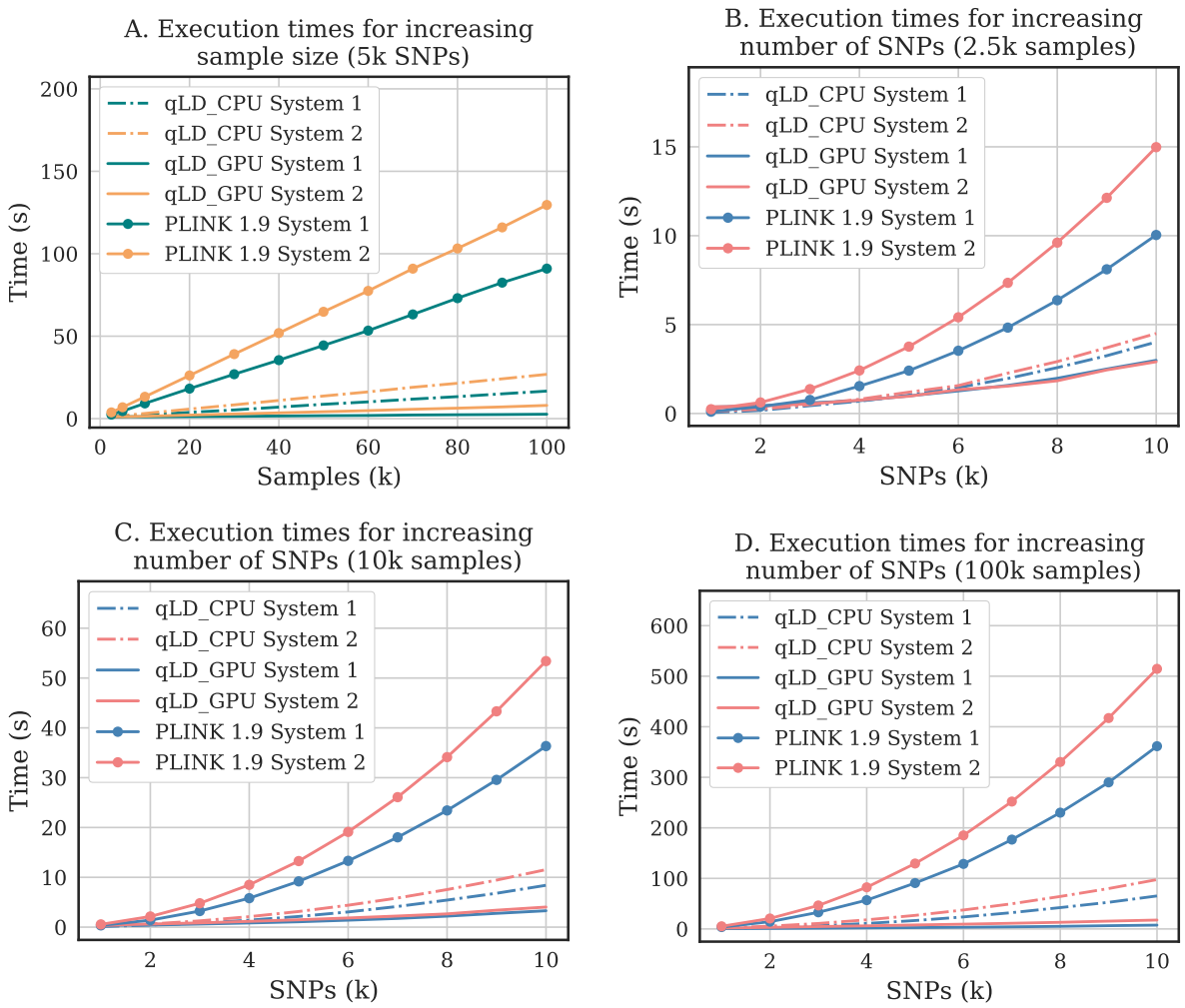

Figure 5.1: Execution times on System 2 for increasing number of samples Fig. 5.1A and increasing numbers of SNPs when the sample size is 2,500 sequences (Fig. 5.1B), 10,000 sequences (Fig. 5.1C), and 100,000 sequences (Fig. 5.1D).

This works under the assumption that our node on Aris ran without, or using only a small amount of boost, probably because of a conservative tuning for safety and stability. Tables 5.2 and 5.3 summarize these results in terms of speedup (discussed in detail in Section 5.2.2).

5.2.2 Throughput comparison

A major difference between qLD and PLINK 1.9 is the way pairwise LD scores are computed. As previously mentioned, qLD relies on the BLIS framework and exploits the observation that pairwise LD computations can be cast as a matrix-multiply operation. Computing all pairwise LD scores between all SNPs in a single region using BLIS results in the calculation of a symmetric output matrix, thus having evaluated the same scores twice. PLINK 1.9, on the other hand, only processes a single file/region and calculates a diagonal matrix with all the pairwise LD scores.

Samples ($\times 10^3$)	Single Region					Pair of regions	
	Throughput (LD $\times 10^6/sec$)			Speedup (x) over PLINK 1.9		Throughput (LD $\times 10^6/sec$)	
	qLD_CPU	qLD_GPU	PLINK 1.9	qLD_CPU	qLD_GPU	qLD_CPU	qLD_GPU
2.5	8.679	11.466	3.306	2.625	3.468	17.361	22.936
10.0	3.967	8.560	0.942	4.210	9.082	7.937	17.123
20.0	2.155	5.813	0.478	4.510	12.167	4.310	11.628
30.0	1.497	4.416	0.319	4.685	13.823	2.994	8.834
40.0	1.133	3.581	0.241	4.706	14.874	2.267	7.163
50.0	0.912	2.983	0.193	4.726	15.465	1.823	5.967
60.0	0.770	2.572	0.161	4.773	15.940	1.540	5.144
70.0	0.657	2.200	0.137	4.782	16.012	1.314	4.401
80.0	0.583	1.965	0.121	4.812	16.228	1.166	3.931
90.0	0.518	1.758	0.108	4.804	16.312	1.036	3.516
100.0	0.465	1.558	0.096	4.822	16.157	0.930	3.117

Table 5.2: Throughput performance on System 2 for increasing number of samples when a single SNP region and a pair of regions are processed by qLD_CPU, qLD_GPU, and PLINK 1.9. The table also provides the observed speedups of qLD_CPU and qLD_GPU versus PLINK 1.9. The region size is 5,000 SNPs, thus 12.5×10^6 and 25×10^6 LD scores are computed when a single SNP region and a pair of regions are processed, respectively.

Region size (SNPs)	Single Region						Pair of regions		
	number of LD scores	Throughput (LD $\times 10^6/sec$)			Speedup (x) over PLINK 1.9		number of LD scores	Throughput (LD $\times 10^6/sec$)	
($\times 10^3$)	($\times 10^6$)	qLD_CPU	qLD_GPU	PLINK 1.9	qLD_CPU	qLD_GPU	($\times 10^3$)	qLD_CPU	qLD_GPU
1.0	0.500	0.257	0.284	0.095	2.701	2.977	1.000	0.515	0.568
2.0	1.999	0.359	0.645	0.097	3.713	6.671	4.000	0.718	1.290
3.0	4.499	0.414	0.953	0.097	4.279	9.845	9.000	0.829	1.907
4.0	7.998	0.444	1.303	0.097	4.568	13.414	16.000	0.887	2.606
5.0	12.498	0.466	1.580	0.097	4.822	16.363	25.000	0.931	3.161
6.0	17.997	0.480	1.902	0.097	4.934	19.562	36.000	0.960	3.805
7.0	24.497	0.492	2.181	0.097	5.060	22.443	49.000	0.984	4.363
8.0	31.996	0.498	2.439	0.097	5.140	25.176	64.000	0.996	4.878
9.0	40.496	0.507	2.616	0.097	5.220	26.951	81.000	1.013	5.233
10.0	49.995	0.512	2.842	0.097	5.272	29.254	100.000	1.024	5.685

Table 5.3: Throughput performance on System 2 for increasing region size (number of SNPs) when a single SNP region and a pair of regions are processed by qLD_CPU, qLD_GPU, and PLINK 1.9. The table also provides the observed speedups of qLD_CPU and qLD_GPU versus PLINK 1.9. The sample size is 100,000 sequences.

Because of this, when a single region is processed, qLD computes twice the amount of scores that PLINK 1.9 computes. To perform a fair throughput comparison, since qLD can not compute only the diagonal matrix when a single region is processed, and PLINK 1.9 does not support two different regions as input, we report effective throughput performance for an increasing number of samples (Table 5.2) and SNPs (Table 5.3), distinguishing between processing a single region and a pair of regions of the same size in terms of SNPs. As can be observed in the tables, qLD is between 2.63x and 4.82x faster than PLINK 1.9 when the sample sizes increase from 2,500 sequences to 100,000 sequences, and between 2,70x and 5,27x faster when the region size increases from 1,000 SNPs to 10,000 SNPs. When qLD offloads computations to a GPU, qLD is between 3.47x and 16.31x faster than PLINK 1.9 (running on the CPU) when the sample size increases, and between 2.98x and 29.25x faster when the number of SNPs increases.

5.2.3 Execution time breakdown

Figure 5.3 presents execution time breakdowns for the processing step of qLD when the number of samples and the number of SNPs increase. `qLD-compute` consists of 4 main stages that collectively contribute to its total execution time:

- **Memory Layout Transformation (MLT):** transposition of one of the two input genomic regions, as required by BLIS for computing LD as a matrix-multiply operation. This stage is not performance-critical as it only relocates SNP data in memory.

- **General Matrix-Multiply (GEMM):** Invocation of the CPU/GPU LD microkernel through BLIS for computing haplotype frequencies as a matrix-multiply operation. This stage is performance-critical and dominates the total execution time in all CPU runs.

- **Linkage Disequilibrium (LD) score computation:** Calculation of the allele frequencies for all SNPs in the two regions and the final LD scores. This stage heavily relies on floating-point operations, but the amount of time spent on floating-point

operations for computing LD scores becomes negligible as the sample sizes grow because the overall execution time is dominated by GEMM. When qLD deploys a GPU, however, the GPU-based GEMM stage is between 2 and 20 times faster than the CPU one, which leads to the LD time dominating execution times for sample sizes as low as 10,000 sequences.

- **Write Output:** Storing LD scores in a text file. Note that we apply the default cutoff threshold for LD scores ($r^2_{limit} = 0.2$) to discard very low scores and prevent output reports from exploding in size. Our custom printing is also enabled by default.

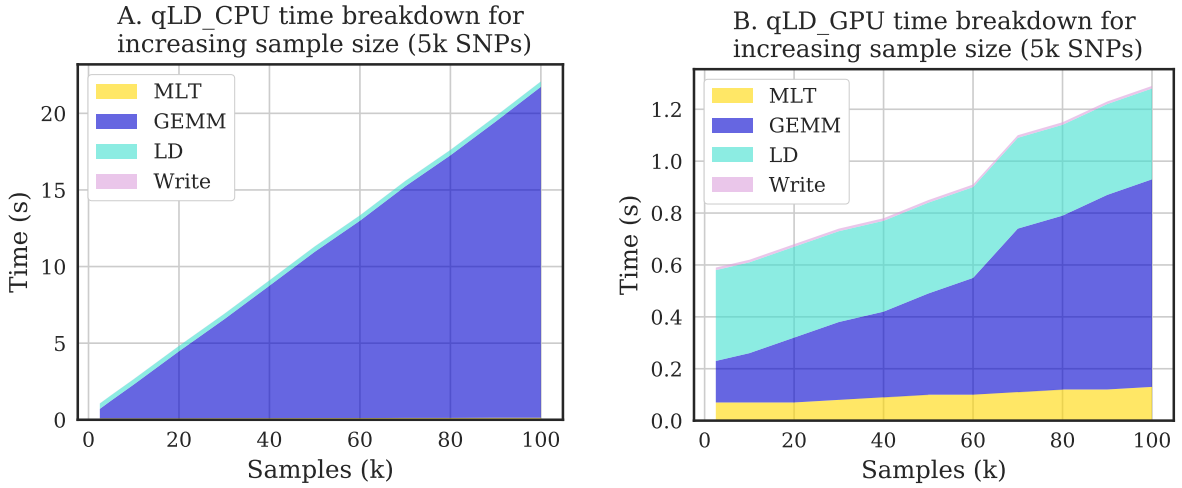

Figure 5.2: Execution time breakdown of qLD-compute when executing on a CPU and a GPU on System 2 when samples increase for set SNP size.

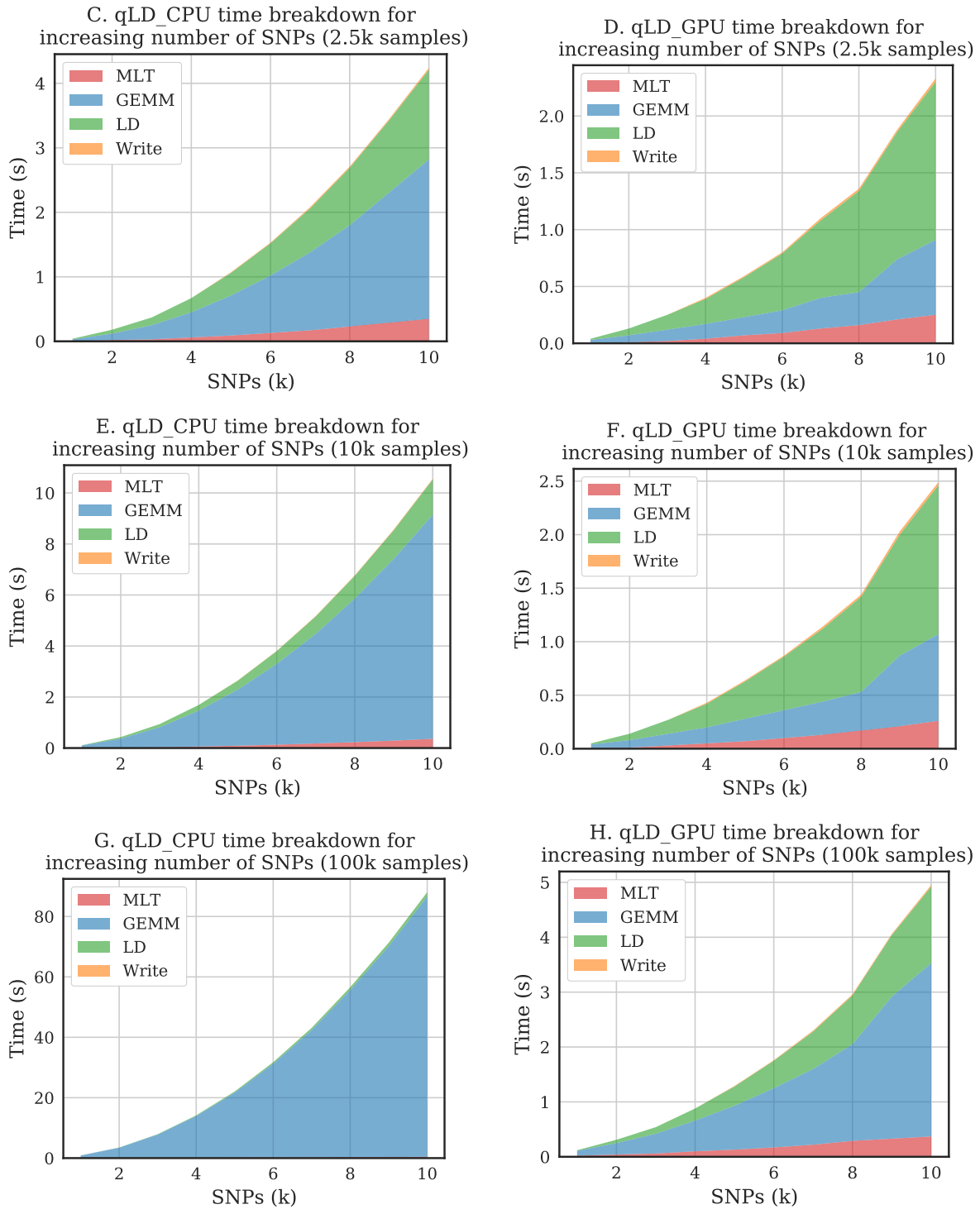

Figure 5.3: Execution time breakdown of qLD-compute when executing on a CPU (left column) and a GPU (right column) on System 2, when SNPs increase for set sample sizes.

5.2.4 Custom Blocking Factor

To evaluate the custom blocking optimization correctly, we need to find the trend between performance and blocking factor. The balance that we need to achieve is the reduction of redundant results, versus the loss of throughput using smaller tasks. As explained previously in this work, the workload of GEMM scales quadratically with the SNPs in the input. This means that as we break down the input in finer blocks, we should expect to see quadratic speedup over the initial time. In reality though, the finer we split the input, the more tasks are created, accumulating overhead from the task initialization. Additionally, the allele frequency calculation in the LD step is scaling linearly to the number of SNPs, providing less speedup than the GEMM function. Having all these restrictions in mind, we provide the following results from tests on several blocking factors, using both the CPU and GPU kernels. As was expected, the optimal blocking factor varies depending on the input task size, but using it, we achieved a significant speedup in all tests, with approximately 1.5x speedup in both kernels. This, on top of the 5x and 29x previously observed speedups of qLD over PLINK in similar workloads, leads to a total of 7.5x speedup over PLINK with the CPU and 43.5x with the GPU. It is also apparent, that when used incorrectly, blocking can lead to performance loss, especially when using the GPU kernel. Future work is needed to determine the correct implementation of this custom blocking, but given the provided speedup it is a promising optimization.

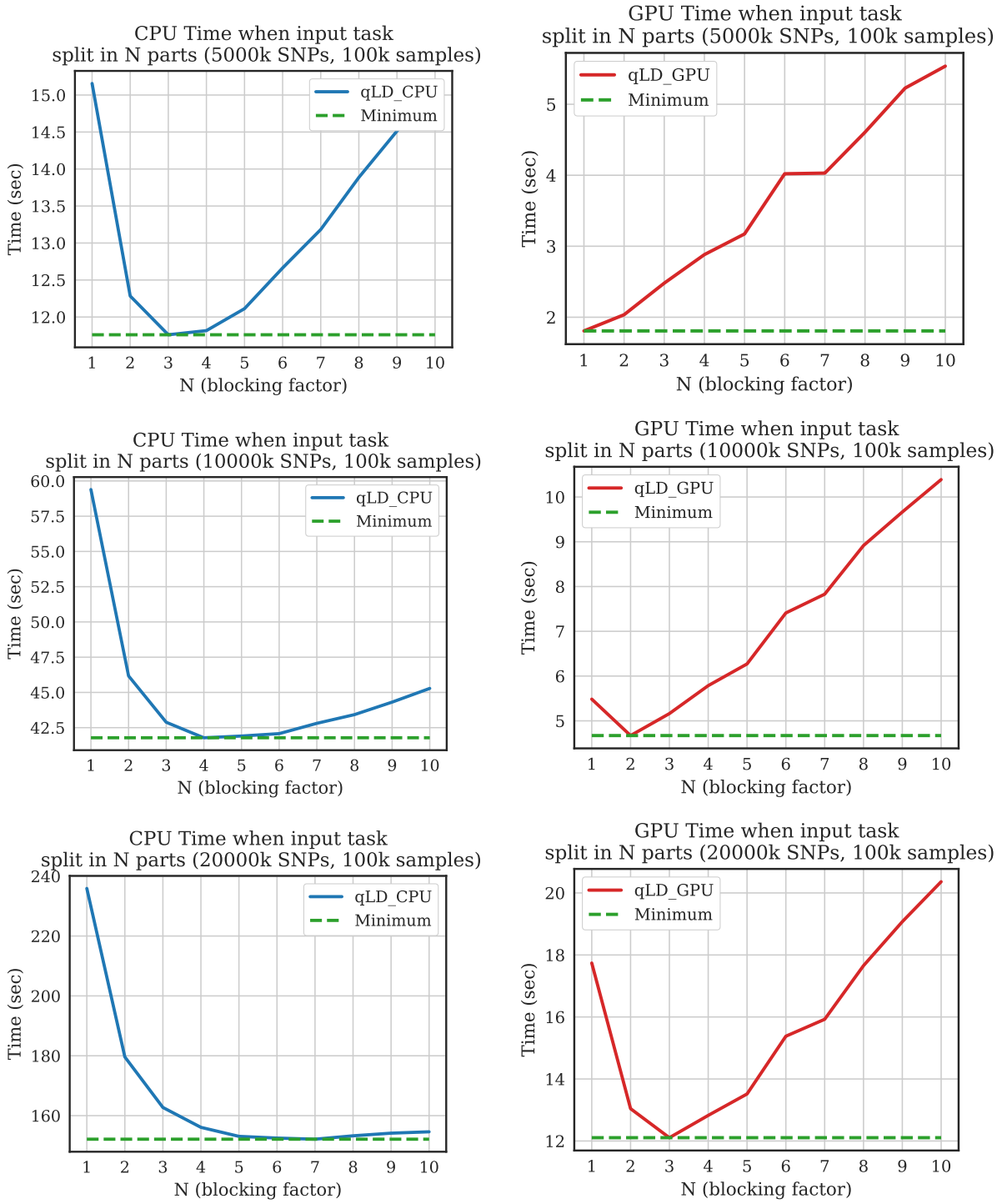

Figure 5.4: Execution Time in qLD using custom blocking in the input. The CPU kernel greatly benefits from blocking, overall. The GPU kernel only benefits from slight blocking, since it is prone to communication overhead in smaller datasets.

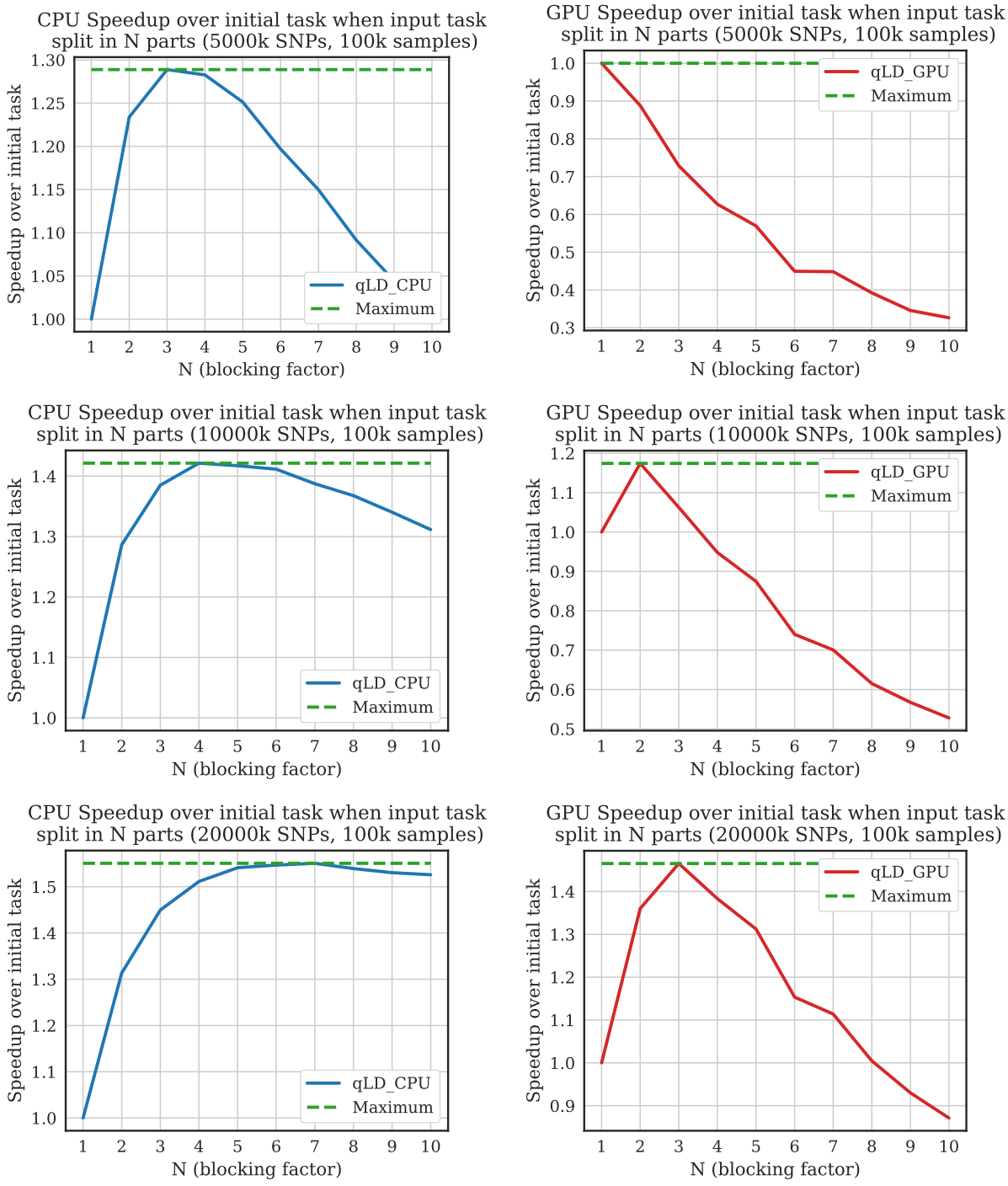

Figure 5.5: Speedup over the initial task in qLD using custom blocking in the input. The CPU kernel achieves up to 1.55x speedup using 21 sub-tasks (factor 6) against the un-blocked initial task in the biggest dataset. The GPU kernel achieves 1.45x speedup, with fewer and larger blocks (factor 3).

5.3 Parallel Performance

When we introduce parallelization with the help of Pthreads, we need to mainly evaluate the speedup and the scalability of the implementation. Since most of qLD's optimizations are applied in the parallelization process, we also need to evaluate the performance boost of each optimization applied. Therefore, we present all of the tests that provide insight into the multi-threaded performance of qLD, along with the final juxtaposition between qLD and PLINK.

5.3.1 Multi-core CPU Scalability with number of threads

An important aspect of the parallelization of qLD that needs to be evaluated is its scalability on the CPU cores. The CPU kernel, while being slower than the GPU kernel, is scaling well with the number of available cores, ranking third in the previous comparison, when using only 4 threads. To find out the cutoff of the CPU kernel speedup, we deployed a test on 1000 tasks of arbitrary SNP size in System 2, using up to 20 threads. The SNP size varied from 1k to 10k SNPs. Blocking was not applied. As we can observe in Figure 5.6, the speedup starts to drop when approaching the 20 thread mark, in all test cases. It also approaches linear speedup more when the sample size increases, while maintaining a high speedup overall.

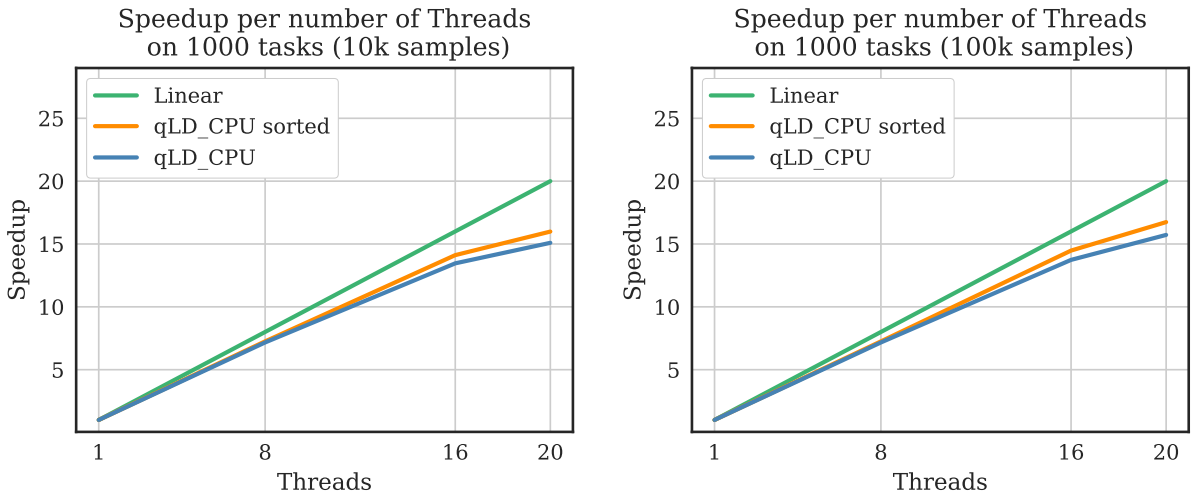

Figure 5.6: Speedup of execution with increasing number of CPU threads on the Aris supercomputer. 1000 tasks of arbitrary sizes are used.

5.3.2 GPU-handling Thread Pinning

Since most of the tests in this section inherently evaluate the GPU in multi-threaded executions, the first decision we ought to evaluate is the way qLD handles the "special" GPU-offloading thread, that uses the GPU kernel for the haplotype frequency calculations. Firstly, we should address that the kernels are developed for maximum efficiency in their assigned core usage, meaning that our thread limit is the physical core limit of the CPUs in each system. While this does not constitute a problem when the CPU is handling the performance-critical parts of the execution, a typical GPU-handling thread waits while the GPU calculates the workload. This results in CPU performance loss for that particular time window on that particular thread.

From the design perspective, we could resolve this issue by adding another CPU-handling thread and letting the GPU-handling thread unpinned. This would certainly introduce more latency to the thread that shares the same core for the CPU intensive part of the execution, but it could also be balanced out from the introduction of the new, pinned CPU thread in place of the former GPU-handling thread.

To test both the pinned and unpinned GPU-handling thread designs, we used System 1, due to its small physical core size, on a list of 1000 tasks with varying numbers of SNPs for a set number of samples. The pinned mode runs on 3 pinned CPU and 1 pinned GPU threads, while the unpinned uses 4 pinned CPU and 1 unpinned GPU thread. As depicted in Figure 5.7, while the unpinned design leads the performance by a small margin in low-to-mid sample sizes, when the samples get bigger, even the slight disruption of an unpinned thread is enough to heavily toll the execution. This test is verified by the breakdown of the execution time in the previous section, where we witnessed that the GPU kernel fires up and dominates the execution when the sample size is bigger. Apparently, in these cases, even the slightest delay in the execution of the ten-times-faster GPU kernel is enough to hinder the overall performance.

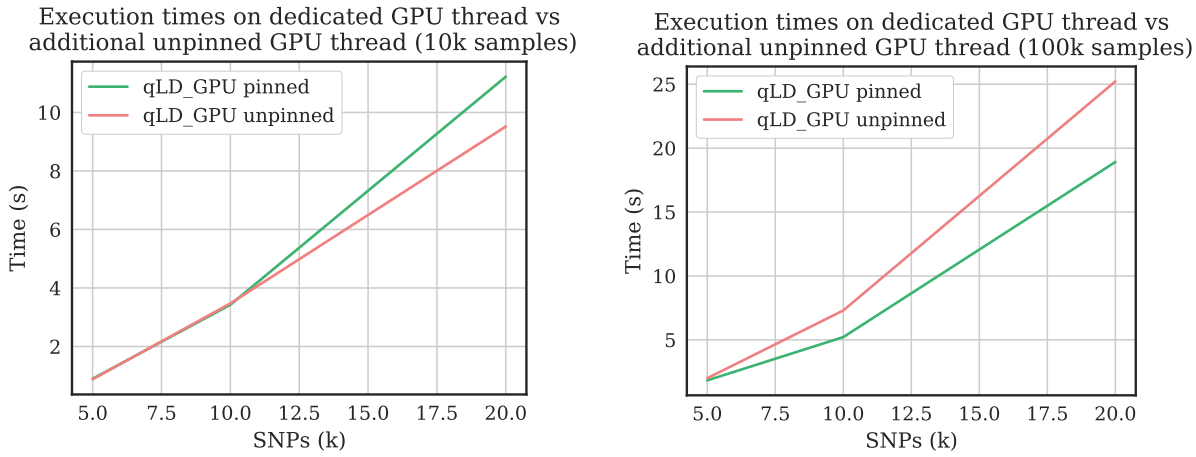

Figure 5.7: Execution time comparison of 4 pinned total CPU and GPU Threads vs 4 pinned CPU + 1 unppined GPU Thread. When calculating tasks with larger sample sizes, where the execution time is considerably slower, pinning the GPU-handling thread offers the best performance.

5.3.3 Competing Task Queue and Sorting

As mentioned previously, qLD uses the input task list as the means of parallelization of the workload, where each task is executed by a different thread. Since the workload is, most of the time, not balanced, there was a need to address the balancing of the tasks to the available threads.

In the design section, we addressed the 3 methods that qLD can use to assign the tasks to each thread. The regional assignment preallocates the tasks to specific threads based on the number of available threads. The competing queue pops a task from the task queue and assigns it to a thread whenever that thread finishes its previous execution. The sorted competing queue uses the same approach, after sorting the task queue first. In this test, we evaluate those methods in the worst-case scenario, since it is the cleanest way to visualize the benefits of the two optimized executions, eliminating randomness from our results. The input consisted of a total of 40 tasks, with an increasing workload every 10 tasks, from 1k to 7k in 2k intervals. The execution was performed on the first System, the off-the-self laptop, since we wanted to utilize all of the available computational power in this test, and doing so in a cluster with 20 cores would call for extreme input data sizes and execution time. The evaluation involved CPU only execution, heterogeneous execution with a pinned GPU-handling thread, and a single GPU-handling thread,

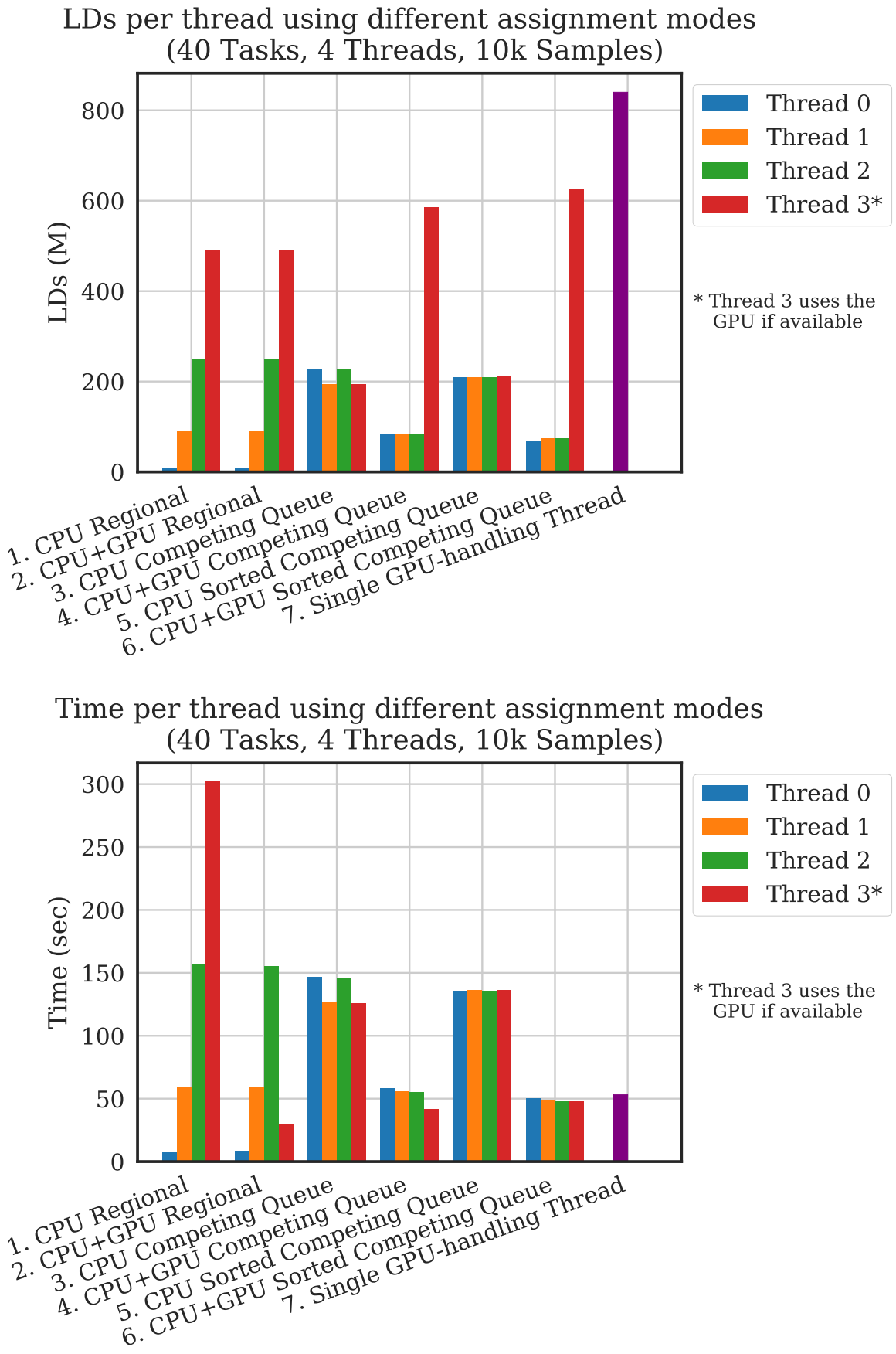

Figure 5.8: Comparison of Different Task Assignment Techniques. The LD graph (top) shows the workload per thread, while the Time graph (bottom) shows the execution time per thread. The workload and time using a single GPU thread is depicted in purple.

to test if the heterogeneous execution is slowed down by the slower CPU threads.

In Figure 5.8 we can observe the workload balancing with each assignment technique in the LD subfigure, while the execution times appear in the Time subfigure. The workload of the initial task is assigned linearly in the regional execution, so the first thread calculates the first 10 tasks, the second thread calculates the next 10, etc. The highest thread time in each execution is also equal to the total execution time of qLD, meaning that the fastest execution is the one with the lowest peak in the time graph.

As we can observe, while the competing queue approach seems to work sufficiently well, the slight modification of sorting the input first gives a completely balanced execution time. Another vital observation is that the execution time of the sorted competing queue parallel approach is better, although marginally, than the single GPU-handling thread.

This sets the premise for the following tests in multi-threaded heterogeneous executions. To achieve better (than the single GPU-handling thread) parallel execution, we need the sorted competing queue approach. It also raises the question of whether increasing the CPU threads helps the performance. To evaluate this, we proceed to the next test.

5.3.4 Aggregate System Performance

Evaluating the task assignment methods we observed a reduced performance when executing in parallel, compared to the single GPU kernel execution. The performance of qLD is bound by a lot of parameters, making it extremely difficult to evaluate properly as a whole. The main parameters we could address, given the previous evaluations, are the input data size, the blocking factor in the input, and the thread count. To utilize all previous knowledge, while using both kernels to their maximum potential, we chose an input of 20k SNPs and 100k samples. This set resulted to the best speedup for both kernels in the evaluation of the custom blocking optimization and should not favor one kernel over the other. Using the aforementioned data, we know that the optimal blocking factor for the CPU in this input is 7, while 3 for the GPU. As for the number of threads, we tested the maximum available threads (4), the combination of a single CPU and a single GPU thread (2), and a single GPU thread (1). The results are presented

in Figure 5.9, confirming that the combination of the CPU and GPU kernels using the maximum available threads is marginally slower than the single GPU kernel execution when the former is used with a blocking factor favoring the CPU kernel.

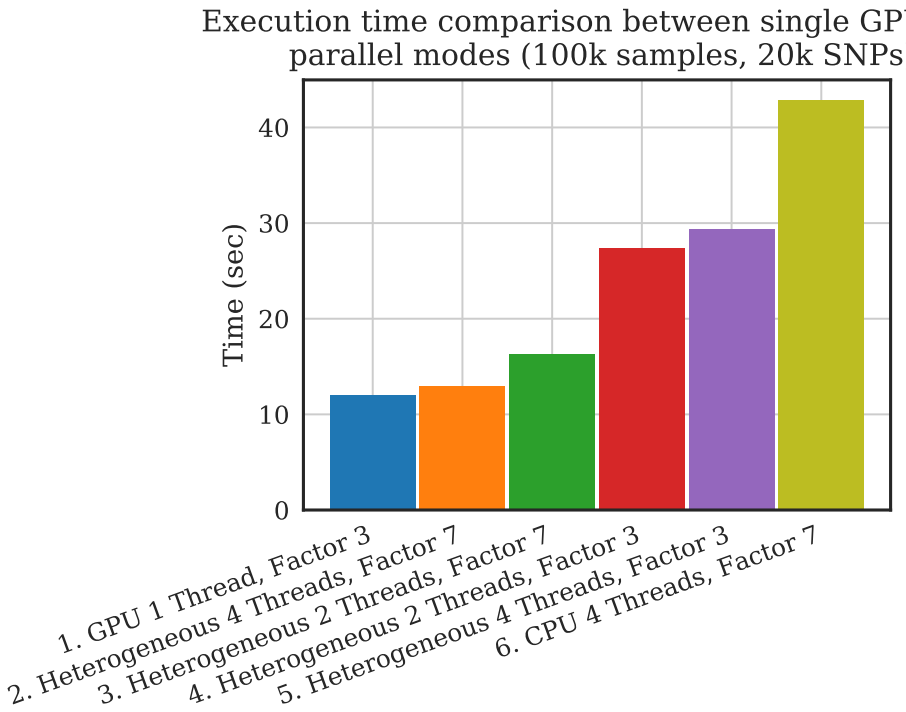

Figure 5.9: Time comparison between several parallel executions and single GPU kernel execution using optimal blocking factors on the input. The parallel heterogeneous execution provides good results when using an optimal blocking factor and more threads, but is marginally slower than the dedicated GPU execution.

5.4 qLD vs PLINK Performance Comparison

Conducting all the previous tests, we gained a good understanding of the most optimal multi-threaded executions. To measure qLD's speedup over PLINK in multi-threaded mode, we compare our CPU and heterogeneous parallel execution modes, alongside the seemingly best performing single-threaded GPU execution mode with the parallel PLINK. It should be noted again, that, PLINK does not utilize GPUs and handles the blocking of the input internally. Hence, PLINK gets the initial task as input, while qLD applies the custom blocking feature to enable the parallelization of each task. The results in Figure 5.10 show that overall, qLD is faster in any type of execution. When using smaller

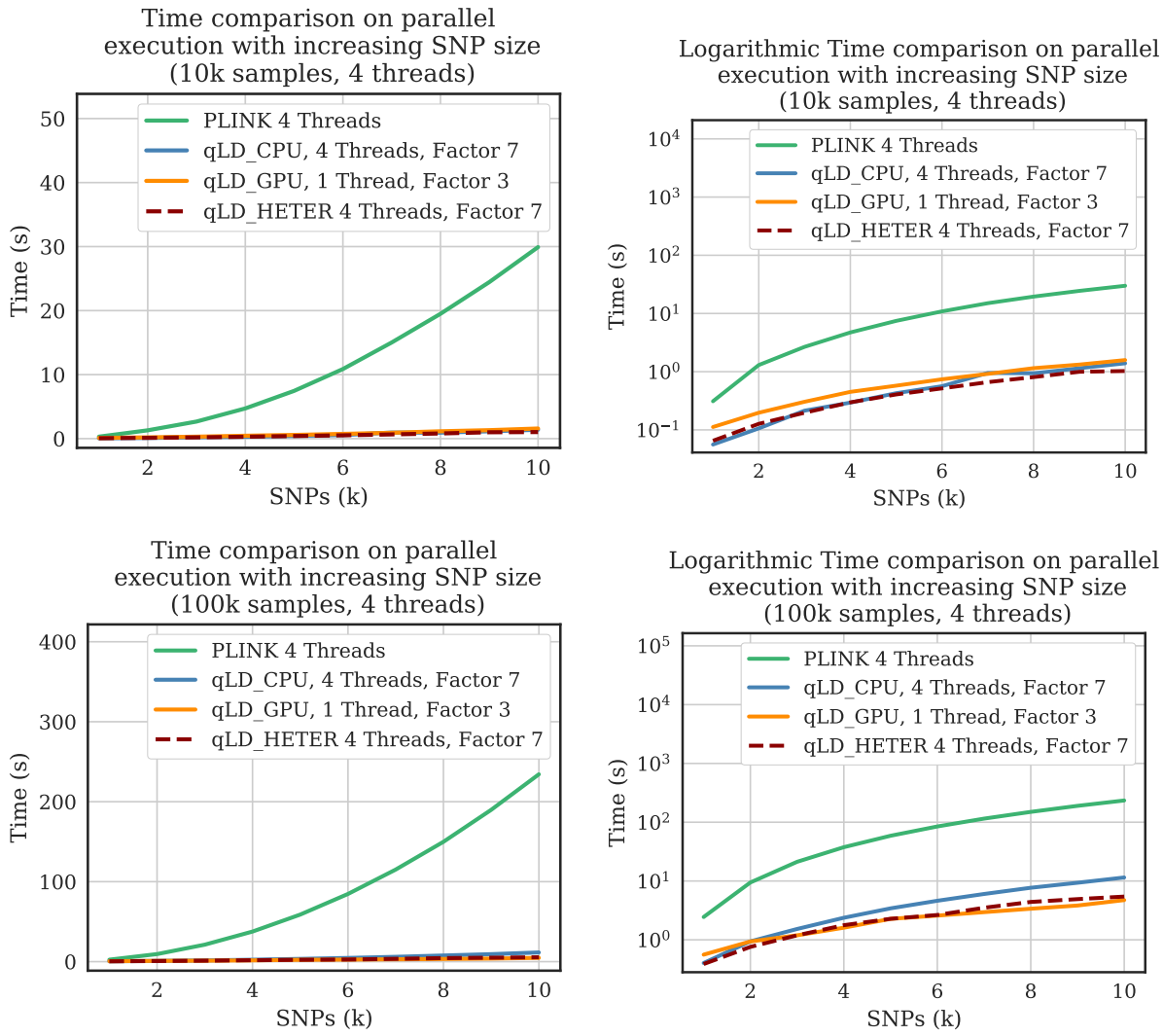

Figure 5.10: Execution time comparison between PLINK 1.9 and the top 3 performing modes of qLD. Time graphs are presented in the left column, with logarithmic graphs being on the right.

sample sizes, the CPU and heterogeneous executions outperform the single GPU-handling thread, while in larger sample sizes the GPU, alongside the heterogeneous execution show the best performance. In conclusion, qLD shows up to 29x speedup over PLINK in smaller sample sizes using the heterogeneous execution mode, closely followed by the CPU and GPU execution modes with 22x and 19x respective speedup over PLINK. In larger sample sizes qLD_GPU outperforms PLINK with 50x speedup, with the heterogeneous and CPU execution performing 43 and 21 times faster than PLINK.

5.5 Application on SARS-CoV-2 Genomes

To showcase qLD, we used 39,941 high-coverage SARS-CoV-2 genomes from the GISAID database (*https://www.gisaid.org/*), downloaded on July 9, 2020. We kept only complete sequences (genomes with base-pair lengths greater than 29,000), and trimmed the ambiguous states (N's) from both the beginning and the end of the genomes. Thereafter, we excluded all sequences that contained Ns, and employed the experimental version of MAFFT [49] for closely-related viral genomes to create a multiple sequence alignment in FASTA format. The final dataset, after discarding the sequences that did not pass the aforementioned filters, comprised 22,554 genomes. We used snp-sites [50] to convert the FASTA to VCF for processing with qLD, invoking the tool's built-in option for omitting columns that did not exclusively contain A, C, G, T.

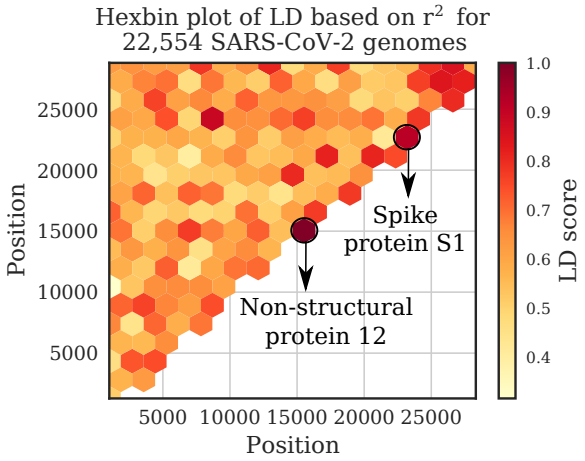

Figure 5.11: Hexbin plot (gridsize=16) of LD scores calculated using qLD on System 1 (laptop) in 18 seconds.

Figure 5.11 illustrates qLD scores (r^2) in a hexbin plot with grid=16. The two highest-score bins shown in the figure correspond to regions $15,042-15,517$ and $22,684-23,198$ with scores 1.0 and 0.92, respectively. Both regions are UniProt highlighted regions of interest as shown in the UCSC genome browser view of SARS-CoV-2 genomic datasets (*https://genome.ucsc.edu/cgi-bin/hgTracks?db=wuhCor1*), with the first region found in the non-structural protein 12 and is known to interact with RMP Rendemsivir, while the second region found in the Spike protein S1 and is a motif in the Receptor Binding Domain that binds to human ACE2.

6. Conclusions and Future Work

In this work, we presented a new scalable platform for calculating pairwise LD scores in heterogeneous environments. Using optimized kernels based on the BLIS framework (`qLD_CPU`) and the OpenCL framework (`qLD_GPU`), we achieved single-thread speedup in all test-cases, up to 5x and 29x using the CPU and GPU kernels respectively, over the widely used state-of-the-art PLINK 1.9. Based on HPC practices, the platform works on par with the theoretical performance of these computation kernels with minimal overhead from the rest of the processing steps. Furthermore, using certain experimental optimization techniques, we were able to achieve additional speedups of up to 1.5x with both kernels over the default execution. Cross-referencing our results, this leads to a maximum observed speedup of 7.5x and 43.5x over PLINK 1.9, in single-thread executions.

Using qLD in multi-threaded executions, we observed overall satisfactory results. In terms of scaling, the CPU execution seems able to handle many concurrent threads, maintaining close to linear speedup. Testing several optimizations on the allocation of tasks to threads, we managed to achieve almost perfectly balanced workloads, utilizing all of the available hardware to the maximum potential.

The default execution in a heterogeneous environment resulted in -at best- similar performance to the single-threaded GPU kernel execution in most cases. The sheer difference in power between the two kernels led to unbalanced executions hindering the performance of the GPU kernel, especially in workloads with many samples, where the GPU kernel performs the best. More testing on the optimization of the heterogeneous execution could lead to better results, though, as proven with the introduction of input blocking in the parallel heterogeneous execution. Using optimal parameters in the last experiment, we managed to achieve a significant 50% increase in performance with heterogeneous execution vs the single GPU execution in small sample sizes. Overall, using qLD optimally with all the performance optimization features enabled, we managed to achieve up to 50x speedup over the state-of-the-art PLINK 1.9 in scaling SNP sizes and large sample sizes (100k samples) with the parallel heterogeneous execution in qLD.

The future of this project revolves heavily around the fine-tuning of all the optimizations presented here. Given the nature of realistic data, where each site in the input region is not necessarily an SNP, uncertainty is introduced in the input parsing. Most of the optimizations are already usable with realistic data, utilizing estimations based on empirical statistics and memory reallocation mechanisms to prevent failure in executions. Looking back at the performance evaluation, especially when using the custom blocking feature, it becomes apparent that we need to set a designated block size and enforce an automated blocking mechanism in the input. This doubles up as a preventative mechanism to the uncertainty in memory usage with realistic data but enforces the mandatory use of MDF files since they contain SNP-only values.

Lastly, qLD is programmed to be flexible and modular, to be expandable, to get future kernel upgrades, and to support more genetic calculations in the future. Since the LD calculation tool-set is a completely separate function that runs inside the threads when selected, we could easily append another tool-set that can utilize the existing HPC platform for different calculations. On the other side, since the LD kernels simply calculate GEMM operations, we could refactor the existing qLD operation for different domains than utilize matrix multiplications. An example domain for this modification could be the assessment of chemical similarity using the Tanimoto coefficient [51] in Chemical Informatics, where the computational pipeline resembles LD both mathematically and in data structure.

Bibliography

[1] 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)*, pages 1–8. IEEE, 2017.

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

[3] 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*, pages 199–208. IEEE, 2019.

[4] K. Rogers, E. Green, et al. Biology, November 2019.

[5] Y. Smith. History of genomics, February 2019.

[6] K. A. Wetterstrand. Dna sequencing costs: data from the nhgri genome sequencing program (gsp), 2019.

[7] C. E. Cook, O. Stroe, G. Cochrane, E. Birney, and R. Apweiler. The European Bioinformatics Institute in 2020: building a global infrastructure of interconnected data resources for the life sciences. *Nucleic Acids Research*, 48(D1):D17–D23, 11 2019.

[8] R. V. Rohlfs, W. J. Swanson, and B. S. Weir. Detecting coevolution through allelic association between physically unlinked loci. *The American Journal of Human Genetics*, 86(5):674–685, 2010.

[9] J. M. Smith and J. Haigh. The hitch-hiking effect of a favourable gene. *Genetics Research*, 23(1):23–35, 1974.

[10] D. E. Reich, M. Cargill, S. Bolk, J. Ireland, P. C. Sabeti, D. J. Richter, T. Lavery, R. Kouyoumjian, S. F. Farhadian, R. Ward, et al. Linkage disequilibrium in the human genome. *Nature*, 411(6834):199–204, 2001.

[11] M. T. Alam, D. K. De Souza, S. Vinayak, S. M. Griffing, A. C. Poe, N. O. Duah, A. Ghansah, K. Asamoa, L. Slutsker, et al. Selective sweeps and genetic lineages of plasmodium falciparum drug-resistant alleles in ghana. *Journal of Infectious Diseases*, 203(2):220–227, 2011.

[12] 1000 Genomes Project Consortium and others. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[13] S. Elbe and G. Buckland-Merrett. Data, disease and diplomacy: Gisaid's innovative contribution to global health. *Global Challenges*, 1(1):33–46, 2017.

[14] 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)*, pages 418–427. IEEE, 2016.

[15] 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*, 81(3):559–575, 2007.

[16] 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*, 4(1):s13742–015, 2015.

[17] C. Theodoris, N. Alachiotis, T. M. Low, and P. Pavlidis. qld: High-performance computation of linkage disequilibrium on cpu and gpu. *20th IEEE International Conference on BioInformatics And BioEngineering (BIBE)*, 2020.

[18] M. Slatkin. Linkage disequilibrium—understanding the evolutionary past and mapping the medical future. *Nature Reviews Genetics*, 9(6):477–485, 2008.

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

[20] P. Pavlidis, J. D. Jensen, and W. Stephan. Searching for footprints of positive selection in whole-genome snp data from nonequilibrium populations. *Genetics*, 185(3):907–922, 2010.

[21] 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*, 15(11):1566–1575, 2005.

[22] A. G. Clark. Finding genes underlying risk of complex disease by linkage disequilibrium mapping. *Current opinion in genetics & development*, 13(3):296–302, 2003.

[23] P. E. Bonnen, P. J. Wang, M. Kimmel, R. Chakraborty, and D. L. Nelson. Haplotype and linkage disequilibrium architecture for human cancer-associated genes. *Genome research*, 12(12):1846–1853, 2002.

[24] R. A. Kittles, D. Young, S. Weinrich, J. Hudson, G. Argyropoulos, F. Ukoli, L. Adams-Campbell, and G. M. Dunston. Extent of linkage disequilibrium between the androgen receptor gene cag and ggc repeats in human populations: implications for prostate cancer risk. *Human genetics*, 109(3):253–261, 2001.

[25] B. A. Nexø, U. Vogel, A. Olsen, M. Nyegaard, Z. Bukowy, E. Rockenbauer, X. Zhang, C. Koca, M. Mains, B. Hansen, et al. Linkage disequilibrium mapping of a breast cancer susceptibility locus near rai/ppp1r13l/iaspp. *BMC medical genetics*, 9(1):56, 2008.

[26] J. Lu, Q. Wei, M. L. Bondy, D. Li, A. Brewster, S. Shete, T.-K. Yu, A. Sahin, F. Meric-Bernstam, K. K. Hunt, et al. Polymorphisms and haplotypes of the nbs1 gene are associated with risk of sporadic breast cancer in non-hispanic white women≤ 55 years. *Carcinogenesis*, 27(11):2209–2216, 2006.

[27] G. Morahan, D. Huang, S. I. Ymer, M. R. Cancilla, K. Stephen, P. Dabadghao, G. Werther, B. D. Tait, L. C. Harrison, and P. G. Colman. Linkage disequilibrium of a type 1 diabetes susceptibility locus with a regulatory il12b allele. *Nature genetics*, 27(2):218–221, 2001.

[28] M. Vasilarou, N. Alachiotis, J. Garefalaki, A. Beloukas, and P. Pavlidis. Population genomics insights into the recent evolution of sars-cov-2. *BioRxiv*, 2020.

[29] V. D. Del Angel, E. Hjerde, L. Sterck, S. Capella-Gutierrez, C. Notredame, O. V. Pettersson, J. Amselem, L. Bouri, S. Bocs, C. Klopp, et al. Ten steps to get started in genome assembly and annotation. *F1000Research*, 7, 2018.

[30] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, et al. The variant call format and vcftools. *Bioinformatics*, 27(15):2156–2158, 2011.

[31] M. Kimura. The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutations. *Genetics*, 61(4):893, 1969.

[32] R. C. Lewontin. The interaction of selection and linkage. i. general considerations; heterotic models. *Genetics*, 49(1):49, 1964.

[33] J. Benesty, J. Chen, Y. Huang, and I. Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[34] K. Ding, K. Zhou, F. He, and Y. Shen. Lda—a java-based linkage disequilibrium analyzer. *Bioinformatics*, 19(16):2147–2148, 2003.

[35] J. C. Barrett, B. Fry, J. Maller, and M. J. Daly. Haploview: analysis and visualization of ld and haplotype maps. *Bioinformatics*, 21(2):263–265, 2005.

[36] X. Solé, E. Guinó, J. Valls, R. Iniesta, and V. Moreno. Snpstats: a web tool for the analysis of association studies. *Bioinformatics*, 22(15):1928–1929, 2006.

[37] B. Pfeifer, U. Wittelsbürger, S. E. Ramos-Onsins, and M. J. Lercher. PopGenome: An Efficient Swiss Army Knife for Population Genomic Analyses in R. *Molecular biology and evolution*, 31(7):1929–36, 2014.

[38] N. Alachiotis et al. OmegaPlus: a scalable tool for rapid detection of selective sweeps in whole-genome datasets. *Bioinf.*, 28(17):2274–2275, 2012.

[39] Y. Kim and R. Nielsen. Linkage disequilibrium as a signature of selective sweeps. *Genetics*, 167(3):1513–1524, 2004.

[40] 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*, pages 56–68. Springer, 2012.

[41] 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*, (4), 2015.

[42] J. T. den Dunnen, R. Dalgleish, D. R. Maglott, R. K. Hart, M. S. Greenblatt, J. McGowan-Jordan, A.-F. Roux, T. Smith, S. E. Antonarakis, P. E. Taschner, et al. Hgvs recommendations for the description of sequence variants: 2016 update. *Human mutation*, 37(6):564–569, 2016.

[43] D. Karolchik, A. S. Hinrichs, T. S. Furey, K. M. Roskin, C. W. Sugnet, D. Haussler, and W. J. Kent. The ucsc table browser data retrieval tool. *Nucleic acids research*, 32(suppl_1):D493–D496, 2004.

[44] M. G. Reese, B. Moore, C. Batchelor, F. Salas, F. Cunningham, G. T. Marth, L. Stein, P. Flicek, M. Yandell, and K. Eilbeck. A standard variation file format for human genome sequences. *Genome biology*, 11(8):R88, 2010.

[45] J. B. Pease and B. K. Rosenzweig. Encoding data using biological principles: the multisample variant format for phylogenomics and population genomics. *IEEE/ACM transactions on computational biology and bioinformatics*, 15(4):1231–1238, 2015.

[46] X. Zheng, S. M. Gogarten, M. Lawrence, A. Stilp, M. P. Conomos, B. S. Weir, C. Laurie, and D. Levine. Seqarray—a storage-efficient high-performance data format for wgs variant calls. *Bioinformatics*, 33(15):2251–2257, 2017.

[47] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at ucsc. *Genome research*, 12(6):996–1006, 2002.

[48] F. G. V. Zee, T. M. Smith, B. Marker, T. M. Low, R. A. V. D. Geijn, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, et al. The blis framework: Experiments in portability. *ACM Transactions on Mathematical Software (TOMS)*, 42(2):1–19, 2016.

[49] K. Katoh and D. M. Standley. Mafft multiple sequence alignment software version 7: improvements in performance and usability. *Molecular biology and evolution*, 30(4):772–780, 2013.

[50] A. J. Page, B. Taylor, A. J. Delaney, J. Soares, T. Seemann, J. A. Keane, and S. R. Harris. Snp-sites: rapid efficient extraction of snps from multi-fasta alignments. *Microbial genomics*, 2(4), 2016.

[51] N. Alachiotis. Generating fpga accelerators for chemical similarity assessment. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2015.