UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

MSC DISSERTATION

FPGA Acceleration of GWAS Permutation Testing

Yaniv Swiel

Supervisors: Prof. Scott Hazelhurst (EIE) and Dr. Mitchell Cox (EIE)

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Electrical Engineering.

2022

Declaration

I, the undersigned, hereby declare that the work contained in this dissertation is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Yaniv Swiel

6 February 2022

Abstract

The large sample sizes of modern genetic datasets has necessitated the development of high-throughput accelerators in order to allow bioinformatics research to be performed in a reasonable amount of time. Although the inherently parallel nature of FPGAs makes them well suited to accelerating high-throughput workloads, they are not commonly employed as bioinformatics accelerators (in lieu of CPUs and/or GPUs) due to their high cost and the fact that developing FPGA-accelerated algorithms is a significantly more complex and time-consuming process than the development of software for CPUs or GPUs. The availability of cloud-based FPGA instances, however, has made powerful FPGAs accessible to bioinformatics labs and the continuous improvement of FPGA design tools has reduced much of the complexity of FPGA development.

This dissertation determines the efficacy of FPGAs (in terms of speed, accuracy, cost and power consumption) when applied to the acceleration of GWAS permutation testing — a computationally expensive bioinformatics algorithm that involves the repeated multiplication of a constant matrix with a changing vector. To demonstrate the effect of FPGA acceleration on GWAS permutation testing, this work presents the design and evaluation of an FPGA-based accelerator designed to run on an AWS EC2 FPGA instance.

This work shows that the FPGA accelerator is orders of magnitude faster than a popular CPU-based GWAS tool while generating comparable results. Furthermore, this work demonstrates that FPGA acceleration enables the handling of workloads which are almost unfeasible for current CPU-based methods. This dissertation, therefore, proves that FPGAs can effectively accelerate high-throughput bioinformatics workloads at relatively low cost.

Acknowledgements

I acknowledge the Pan-African Bioinformatics Network of H3Africa for funding this work.

I sincerely appreciate the contributions of my supervisors Professor Scott Hazelhurst and Dr Mitchell Cox. The guidance and insight they have provided throughout the course of this research has been invaluable.

I would like to acknowledge the postgraduate group at the SBIMB for fostering a welcoming and collaborative research environment. Thanks to Professor Christopher Mathew, Carl Chen and Mahtaab Hayat for allowing me to use their GWAS datasets for testing and to Dr Jean-Tristan Brandenburg for his insights on statistical hypothesis testing in a GWAS context.

Finally, I would like to give special thanks to my parents. I would not have reached this point without their support.

Contents

List of Figures

List of Tables

List of Abbreviations

GWAS	Genome-wide Association Study
GPU	Graphics Processing Unit
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
AWS	Amazon Web Services
EC2	Elastic Cloud Compute
DNA	Deoxyribonucleic Acid
AAF	Alternate Allele Frequency
LD	Linkage Disequilibrium
SNP	Single Nucleotide Polymorphism
PCA	Principal Component Analysis
LMM	Linear Mixed Model
FWER	Family-Wise Error Rate
FDR	False Discovery Rate
IC	Integrated Circuit
CLB	Configurable Logic Block
RAM	Random Access Memory
BRAM	Block RAM
FIFO	First In First Out
DSP	Digital Signal Processing
LUT	Lookup Table
FF	Flip-Flop
SRAM	Static RAM
DDR	Double Data Rate
HDL	Hardware Description Language
RTL	Register Transfer Level

VHDL Very High Speed Integrated Circuit Hardware Description Language

HLS High-Level Synthesis

XRT Xilinx Runtime

PCIe Peripheral Component Interface Express

AXI Advanced eXtensible Interface

API Application Programming Interface

DMA Direct Memory Access

MVM Matrix Vector Multiplication

GEMM General Matrix Multiplication

PE Processing Element

II Initiation Interval

SLR Super Logic Region

BLAS Basic Linear Algebra Subprograms

1 Introduction

The development of technology to accurately sample genome-wide genetic data at low cost has enabled the collection of large datasets that provide a representative sample of the genetic variation across a population. Genome-wide association studies (GWASs) analyse genetic variation over the genomes of many individuals in an attempt to identify *genetic variants* (specific locations in the genome that vary amongst members of a population) associated with complex traits and diseases (or *phenotypes*). Genetic variants found to be associated with a phenotype can be studied in more detail so as to gain a greater understanding of the genetic basis of the phenotype [1].

GWASs have been widely applied to the identification of genetic variants associated with complex human diseases and conditions including type 2 diabetes [9] and various cancers [10]. The identification of disease-associated genetic variants allows for the prediction of disease susceptibility as well as the development of improved treatment and detection methods [1]. In addition to the analysis of human genomes, GWASs have also been applied to plant [11] and animal [12] genomes in order to optimise agricultural processes.

To capture a large amount of genetic variation and increase the chance of detecting associated variants, modern GWASs include millions of genetic variants sampled from thousands of individuals. As demonstrated by Figure 2.5, a GWAS typically examines the independent effect of each variant on the phenotype by performing a series of independent statistical hypothesis tests (which require the computation of a *test statistic* for each variant) under the null hypothesis that no association exists. Rejection of the null hypothesis implies (but does not confirm) that some form of association exists between a genetic variant and the phenotype.

Including many variants in a GWAS gives rise to a phenomenon known as the *multiple testing problem*, whereby the cumulative probability of false rejections of the null hypothesis (or incorrect associations) increases with the number of independent hypothesis tests. A GWAS, therefore, needs to compensate for multiple hypothesis testing by controlling the number of false positives, but there is a critical trade-off between suppressing false associations while simultaneously maximising the number of valid associations.

There is a large body of work — driven primarily by genomics research — related to controlling the false positive rate in a multiple testing scenario but many of the methods assume the null distribution of the test statistics which can result in ineffective control of the false positive rate. Permutation testing is a non-parametric randomisation procedure which provides a robust and powerful method of limiting false positives when testing multiple statistical hypotheses [13]. It attempts to simulate the empirical null distribution of a test statistic using the observed data by repeatedly reassigning the phenotypes among the individuals, which destroys the relationship between genotype and phenotype (representing a sampling under the null hypothesis as illustrated by Figure 2.8), and recalculating the test statistic b times to generate an adjusted test statistic of resolution $1/(b+1)$ [1]. Generating adjusted test statistics which effectively control for a multiple testing scenario with millions of independent hypothesis tests is very computationally expensive (and slow) so, although a number of software libraries (such

as PLINK [14], GEMMA [15] and FaST-LMM [16]) have been developed to perform GWAS analyses on large data sets, there is a need for a GWAS permutation testing accelerator.

FPGAs (Field-Programmable Gate Arrays) are reconfigurable integrated circuits which provide a high level of parallelisation that can be harnessed to accelerate GWAS permutation testing and, although the widespread uptake of FPGAs has been inhibited by their high cost, the availability of cloud-based FPGA platforms has made powerful FPGAs more accessible to bioinformatics labs.

Work has been done on accelerating GWASs using multiple CPU cores [17, 18], GPUs [19, 20], FPGAs [20, 21], GPUs together with FPGAs [22, 23] and GPUs together with CPUs [24], but the majority of this work focuses on the acceleration of genome-wide interaction analysis (i.e. the association of interactions between genetic variants with phenotypes) for categorical phenotypes, not the acceleration of permutation testing.

Gundlach et al. [25] present an FPGA-based permutation testing accelerator for genome-wide interaction studies that runs on a cluster of 128 low-cost FPGAs, but the purchase and setup of an FPGA cluster is not a viable solution for a bioinformatics lab. PBOOST [26] is an example of a GPU-based permutation testing accelerator for genome-wide interaction association studies. PBOOST only supports categorical phenotypes, however, and cannot directly handle confounding variables. Terada et al. [27] present a GPU-accelerated implementation of GWAS permutation testing, but, like PBOOST, it only supports categorical phenotypes and does not support the inclusion of confounding variables.

1.1 Dissertation Aim

The aim of this dissertation is to determine the efficacy of FPGAs when applied to the acceleration of GWAS permutation testing. This is not a trivial problem as algorithm development for FPGAs is significantly more complex than software development for CPUs or GPUs due to the fact that FPGA development involves the design of a specialised hardware architecture [28]. Furthermore, FPGAs are not well suited to accelerating sequential algorithms with data dependencies and complex conditional branching logic [29] so, in addition to the hardware part of the accelerator, there is a requirement for well-optimised CPU software in order to manage the FPGA execution and to perform any complex conditional logic that is not a forte of FPGAs.

The research involves the development and evaluation of an FPGA-based permutation testing accelerator (designed to run on a cloud-based FPGA instance consisting of an FPGA together with a host CPU) that accelerates two known GWAS permutation testing algorithms. Finally, to prove that FPGAs can effectively accelerate GWAS permutation testing, the accelerator performance (i.e. the speed, accuracy, cost and power consumption) is compared to that of a widely-used CPU-based GWAS tool. The accelerator is evaluated based on the speedup it provides over the CPU implementation as well its accuracy (i.e. whether it can identify the same associated variants as the CPU implementation with similar precision).

This work aims to produce a tool which can be used by bioinformatics researchers to accelerate GWAS analyses. An additional aim is to develop a platform-agnostic FPGA design which can be implemented on a variety of FPGA platforms. Finally, this dissertation seeks to characterise the effect of the design optimisations used to maximise the throughput of the accelerator so as to provide a reference for FPGA-based

development.

1.2 Dissertation Scope

The scope of the research is limited to the development of a permutation testing accelerator for continuous phenotypes. While GWASs can analyse both continuous and categorical phenotypes, different numerical methods are used for the analyses and time constraints prevented the implementation of both methods.

As this research aims to investigate the use of cloud-based FPGA platforms to reduce the cost of FPGA-based permutation testing, and AWS EC2 FPGA instances (which provide access to Xilinx Virtex UltraScale+ VU9P FPGAs) were used for the development and testing of the accelerator, this dissertation will focus on Xilinx FPGAs and development tools, although the development process for other FPGAs is similar.

1.3 Dissertation Organisation

Chapter 2 includes the background information required for this dissertation and reviews the current state-of-the-art relating to the acceleration of GWAS permutation testing and the use of FPGAs as accelerators. It addresses the theory of genome-wide association studies — both the genetic theory and the numerical/statistical methods used to perform association tests — and the methods used to control for multiple testing before undertaking a review of modern FPGAs and Xilinx's FPGA development tools.

Chapter 3 addresses the design of the permutation testing accelerator. First, the FPGA-accelerated GWAS permutation testing algorithm is derived and the work is split between the FPGA and the host CPU. The remainder of Chapter 3 focuses on the design of the hardware and software components of the accelerator.

Chapter 4 analyses the final accelerator design and discusses the optimisations used to maximise the efficacy of the accelerator. Chapter 5 then compares the accelerator performance to that of a commonly-used CPU-based GWAS tool. Finally, Chapter 6 concludes the dissertation and provides recommendations for future work.

2 Background

This chapter summarises the relevant background information that informs this work. Section 2.1 covers the theory of GWASs — both the genetic theory and the numerical/statistical methods used to perform association tests. Section 2.2 addresses multiple statistical hypothesis testing and goes over some of the methods that are commonly used to control the false positive rate in a multiple testing scenario. Section 2.3 goes over the theory of permutation testing and highlights two popular algorithms that are employed to reduce the computational burden. Section 2.5 and Section 2.6 provide an overview of FPGA technology. Finally, Section 2.7 and Section 2.8 review state-of-the-art implementations of FPGAs as linear algebra and bioinformatics accelerators.

2.1 Genome-Wide Association Studies

A genome-wide association study (GWAS) attempts to associate genetic variants with a specific trait (or *phenotype*) by analysing genetic variation across the genomes of many individuals [1]. If a genetic variant is found more frequently in individuals with a specific phenotype, it implies that the variant is associated with the phenotype. Once phenotype-associated genetic variants have been identified by a GWAS, they can be studied in more detail in order to gain insight into the genetic mechanisms underpinning complex phenotypes.

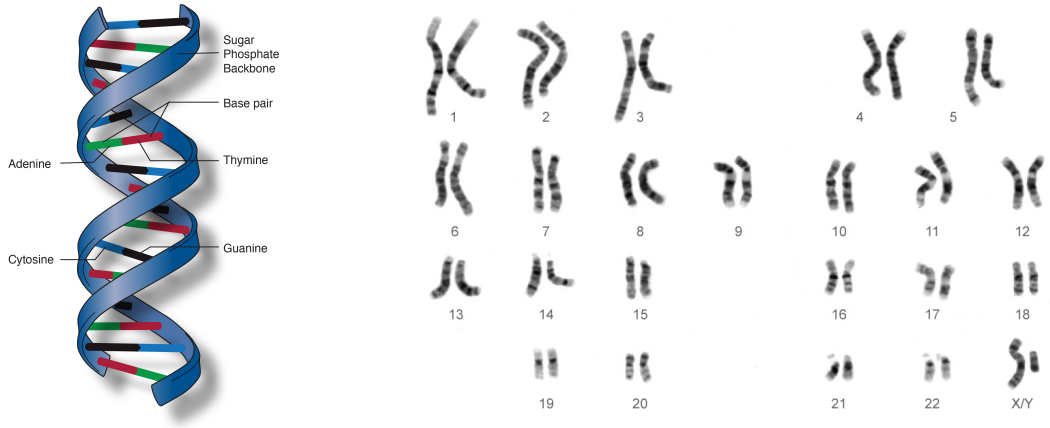
The first human GWAS, which identified genetic variants associated with age-related macular degeneration [30], was published in 2005. As of August 2021, over 5,000 human GWAS publications have discovered more than 275,000 genotype-phenotype associations [31]. GWASs have been widely applied to the association of genetic variants with complex human diseases including type 2 diabetes [9], late-onset Alzheimer's disease [32] and various cancers [10]. The identification and study of disease-associated variants can aid in the understanding of the genetic basis of a disease, allowing for improvements in treatment, detection and prediction of disease susceptibility. GWAS analyses have also identified genetic variants associated with complex human phenotypes such as height [33], body-mass index [34] and cognitive ability [35]. In addition to the analysis of human genomes, GWASs have also been applied to plant [11] and animal [12] genomes in order to optimise agricultural processes.

2.1.1 Genetic Theory

An organism's *genome* refers to the entirety of the genetic data stored in its DNA (deoxyribonucleic acid). The genetic information stored in the genome is required for an organism's growth and development, cellular function and reproduction. The genomes of all (known) species on earth are composed of long chains of *nucleotide* molecules which encode genetic information. Nucleotides consist of one of four bases — cytosine (C), guanine (G), adenine (A) or thymine (T) [36]. Two chains of nucleotides bond via hydrogen bonds between bases (forming nucleotide *base pairs*) in complementary configurations (A bonds to T and C bonds to G) to form double-helical DNA molecules (or *chromosomes*) as shown in Figure 2.1a.

Humans (along with almost all other mammals) are diploid organisms meaning that almost all human cells contain two copies of each chromosome — one inherited from

each parent. Figure 2.1b shows a representation of the human genome which consists of 23 pairs of chromosomes — one pair of sex chromosomes (i.e. chromosomes which determine an individual's biological sex) and 22 pairs of autosomes (i.e. chromosomes which are not sex chromosomes).



(a) DNA structure [37]. (b) Chromosomes of the human genome [37].

Figure 2.1: DNA and the human genome.

DNA is classified as either coding (the sequence of base pairs is used to transcribe a protein required for cellular function) or non-coding (the base pair sequence does not encode a protein). The majority of the human genome (more than 98%) is non-coding and a large proportion of non-coding DNA has no known biological function. A DNA sequence which has some effect on cellular function and gives rise to a phenotype is known as a gene. The human genome consists of more than 3 billion nucleotide base pairs with around 19,000 protein-coding genes.

A *genetic variant* is a specific location in the genome which can differ among members of a population [38]. The different variants found at a genomic location are known as *alleles*. A variant usually (although not always) has two alleles. The allele found in the majority of the population is known as the major allele, while less commonly occurring alleles are known as minor alleles. Different nomenclature is used in the context of genome-wide data as allele frequencies can vary significantly across different sub-populations so a 'major' allele in one sub-population may not be the most commonly occurring allele in a different sub-population. A reference allele is the allele found in the reference genome (i.e. a representative model of a species' genome used to standardise genetic research), while an alternate allele refers to any allele which differs from the reference allele. As a reference genome is composed of data sampled from a few randomly selected individuals, a reference allele may not always correspond to the major allele.

The alternate allele frequency (AAF) of a variant, which refers to the frequency of any allele which differs from the reference allele within a given population, is used to quantify the frequency of the variant within a population. Genetic variants which have an alternate allele frequency of less than 1% are classified as rare variants or mutations, while variants which have an AAF of greater than 5% are classified as common variants [39]. The pair of alleles (one inherited from each parent) at a genome location are known as an individual's *genotype*. A genotype is classified as either homozygous (identical alleles appear at the location) or heterozygous (two different alleles are found at the location). Alleles are classified as either dominant or recessive based on the associated phenotype. If a single copy of an allele results in a given phenotype, it is known as a

dominant allele, whereas if two copies of an allele are associated with a phenotype, the allele is classified as a recessive allele.

The majority of genetic variants have no known effect — they appear in non-coding regions or they do not affect the biological functioning of an organism. Some variants give rise to benign phenotypes such as eye colour or hair colour or can even convey positive effects (such as immunity to a disease). Other variants, however, can predispose individuals to a disease, affect their response to a drug or directly cause a disease by affecting cellular function. Many human diseases are caused by the interaction of multiple genetic variants, each with a small effect [39]. Figure 2.2, from Bush and Moore [1], shows the spectrum of genetic variants in terms of allele frequency and effect size. A GWAS is typically used to identify variants between the two diagonal lines. Rare variants with small effect sizes are difficult to identify with any statistical significance [40], while the complex phenotypes that are analysed with GWASs are not usually caused by common variants with a large effect size [1].

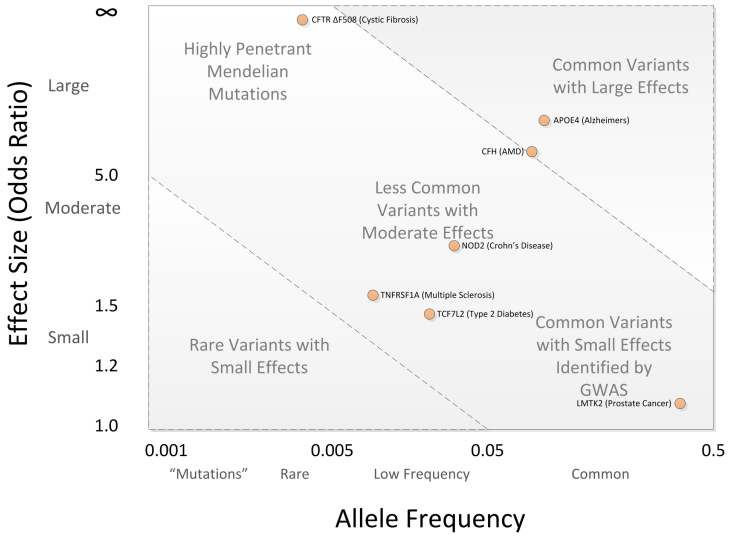


Figure 2.2: GWASs typically identify common variants with small effect sizes [1].

Genetic variants found to be associated with a phenotype can be either directly associated (the allele affects a biological pathway which causes the phenotype i.e. the allele is a *causal* variant) or indirectly associated (the non-causal, associated allele is in *linkage disequilibrium* (LD) with the causal allele). LD is a property of two (or more) variants on a chromosome which describes the frequency with which an allele of one variant is inherited with an allele of the other variant within a population (i.e. whether any correlation exists between the alleles within a population) [41]. A group of alleles on a single chromosome which are commonly inherited together within a population are collectively known as a *haplotype block*.

GWASs exploit the property of LD to analyse large regions of the genome using the knowledge of common variants and haplotype blocks. The discovery of haplotype blocks allows a number of variants to be uniquely identified using a few common variants. The Human Genome Project, which sequenced the entire human genome, together with the International HapMap Project, which analysed LD patterns across different populations, and the 1000 Genomes Project, which attempted to identify common variants (AAF > 1%), were instrumental in identifying common variants and generating reference haplotypes to be used for genome-wide analyses of different populations.

The most common type of genetic variant is the *single nucleotide polymorphism* (SNP). A

SNP is a single nucleotide deviation from the reference genome at a specific location [42] as illustrated by Figure 2.3. Over 600 million human SNPs have been identified [43]. They can be unique to individuals or occur commonly within populations. A typical human genome has 4.1 to 5 million sites which differ from the human reference genome and a majority (around 85%) of the variations are SNPs [44]. Due to their appearance throughout the genome, common SNPs in regions with high linkage disequilibrium (known as tag SNPs) are used as genetic markers to identify genes or alleles (in LD with the SNP) responsible for a specific phenotype.

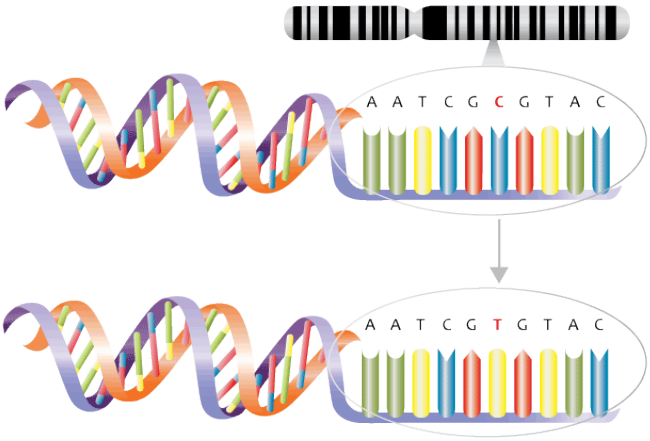


Figure 2.3: A single nucleotide polymorphism where a cytosine (C) nucleotide in the reference genome has been replaced by a thymine (T) nucleotide [2].

As SNPs are responsible for the majority of human genetic variation [44], they are the most well-studied and catalogued type of genetic variant. Therefore, the genome-wide association study was designed primarily to analyse SNPs, although the scope of genome-wide association analysis has been successfully extended to the association of genetic variants other than SNPs [39]. Copy number variants (a repeating sequence of nucleotides with a varying number of repeats) have been associated with phenotypes including schizophrenia and bipolar disorder [45] and obesity [46]. Inversions (a sequence of nucleotides which has an inverted order compared to the reference genome) have been associated with cognitive function [47] and mental disorders [48]. Indels (an insertion or deletion of a short nucleotide sequence) have been implicated in various cancers [49, 50]. Genome-wide interaction association studies are a class of GWAS algorithms used to associate the interaction between variants (a phenomenon known as *epistasis*) with phenotypes [51].

GWASs have become a viable method of identifying genotype-phenotype associations due to the development of low-cost SNP genotyping technology (SNP microarrays) which allow common SNPs to be accurately and quickly genotyped [40]. As an example, the H3Africa microarray [52], which is manufactured by Illumnia, can sample over 2 million SNPs. A limitation of SNP microarray technology is the associated difficulty of sampling rare SNPs, so genotyping with SNP microarrays does not capture the full extent of the genetic variation across the genome. A method called genotype *imputation* is commonly employed to accurately infer SNPs that are not directly genotyped using reference haplotypes [53]. Imputation cannot, however, identify extremely rare variants and variants which are not present in the reference haplotypes (as haplotypes vary across different populations). Whole genome sequencing can be used to capture almost all the genetic variation across a genome, but this technology is currently prohibitively

expensive for sequencing the genomes of many individuals.

2.1.2 Numerical Modelling

Encoding the phenotype of interest is a critical factor in obtaining meaningful results from a GWAS. Phenotypes are typically defined as categorical (usually case/control or phenotype/no phenotype) or continuous. Continuous phenotypes (e.g. height, blood cholesterol level) are preferred (but not required) for a GWAS as they improve the statistical power to detect associated variants and the results of the association tests are more interpretable than those testing categorical phenotypes [1]. Phenotypes which are not quantifiable, however, are defined as case/control.

The genetic data must also be represented numerically in order to perform association tests. *Allelic association* attempts to associate the presence of one allele of a genetic variant with a phenotype, so the genetic data for allelic association tests is encoded to denote the presence or absence of the allele being tested [54]. *Genotypic association*, which is more commonly used, attempts to identify the relationship between a genotype (i.e. an allele pair) and a phenotype [54]. Different genotypic encodings are used to represent different assumptions about the effect of a variant genotype on the phenotype. For a variant with two alleles, where A is the reference allele and a is the alternate allele, the following models can be used to denote the effect of a variant's genotype on a phenotype [54]:

- Dominant
  - The presence of at least one alternate allele in the genotype increases the chance of displaying the phenotype.
  - $AA = 0$, $Aa = 1$, $aa = 1$
- Recessive
  - Two copies of the alternate allele are required to affect the chance of observing the phenotype.
  - $AA = 0$, $Aa = 0$, $aa = 1$
- Additive
  - Each alternate allele present in the genotype results in a linear increase in the probability of observing the phenotype.
  - $AA = 0$, $Aa = 1$, $aa = 2$

The additive genotypic model is most commonly used as it can detect both additive and dominant effects. This work, therefore, focuses on the acceleration of the additive model GWAS.

2.1.3 Association Testing

In order to capture a large amount of genetic variation and increase the statistical power (i.e. the chance of correctly identifying associated variants), modern GWASs include millions of genetic variants sampled from thousands of individuals. Larger sample sizes usually lead to the identification of more associated variants [39], while the inclusion of more variants in a GWAS increases the genome coverage, allowing causal regions of the genome to be more easily identified.

GWAS analysis typically involves a series of independent statistical hypothesis tests to determine the relationship between each variant and the phenotype under the null hypothesis that no association exists. Regression analysis, a family of statistical methods which attempt to determine the relationship between a dependent variable (the phenotype) and one or more independent variables (the variant genotype), is commonly used to perform GWAS association tests. Regression calculates the strength of the

relationship between phenotype and genotype by estimating the function which best fits the data. The regression model used for a GWAS is based on the representation of the phenotype data. Linear regression is used for continuous phenotypes, while logistic regression is used for case/control phenotypes [55].

Controlling for non-genetic effects (or *covariates*) is an important consideration when performing a GWAS. Traits which are known to have an effect on a phenotype (e.g. age, sex, weight) must be accounted for so they do not bias the result [1]. Population substructure (differences in variant allele frequencies among different ethnicities and subpopulations) can result in false associations and reduced statistical power [56], so a GWAS which includes samples from multiple populations (which is extremely likely due to the need for large sample sizes) must account for this effect. Principal component analysis (PCA) is used to estimate population substructure using variant genotype data, and principal components are included as covariates to account for genetic diversity among the individuals in a GWAS. Figure 2.4, from Choudhury et al. [3], plots the first two principal components of SNP data sampled from various sub-Saharan African populations, showing the extent of the genetic variation between individuals.

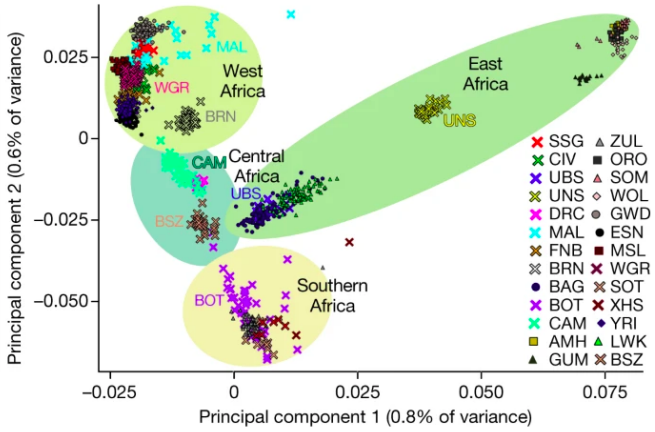


Figure 2.4: First two principal components from the PCA of various sub-Saharan African populations [3].

A consequence of performing multiple independent hypothesis tests is an increased probability of incorrect associations, so a GWAS has to compensate for the number of tests performed (see Section 2.2). The output of a GWAS is the effect size of each variant in the study together with a p-value (which has been adjusted based on the number of variants included in the study) that denotes the variant's genome-wide statistical significance (i.e. the probability that the result was not observed by chance).

2.1.4 Simple Linear Regression

A linear regression model to determine the effect of a single regressor on a dependent variable (i.e. simple linear regression) can be represented as a generic straight line with unknowns α (the y-intercept) and β (the slope of the line) together with an error term ϵ representing normally distributed noise with constant, unknown variance and zero mean.

$$Y = \alpha + X\beta + \epsilon \qquad \epsilon \sim N(0, \sigma^2) \tag{2.1}$$

The goal of linear regression analysis is to determine the *best linear unbiased estimators* for β ($\hat{\beta}$) and α ($\hat{\alpha}$) that optimally fit the data. For a GWAS which analyses genotype-phenotype samples from n individuals, Y is an $n \times 1$ vector of phenotypes, X is an $n \times 1$

vector of variant genotypes, β is a scalar value representing the effect of the variant on the phenotype and ϵ is an $n \times 1$ vector of errors representing environmental and sampling effects. The use of linear regression for GWAS analysis implies four general assumptions about the distribution of the genotype-phenotype data:

- There is a linear relationship between genotype and phenotype
- The samples are independent
- The phenotype is normally distributed for any genotype
- The variance of the error is constant over all samples (i.e. the variance is independent of the genotype)

A GWAS tests m variants simultaneously, so the encoded genotype data is stored in an $n \times m$ matrix X, where each row corresponds to an individual's genotype and each column represents the genotype of a single variant as shown in Figure 2.5. GWAS regression analysis attempts to determine β, an $m \times 1$ vector which indicates the effect size of each variant on the phenotype. GWAS sample sizes can range anywhere from $1,000$ to $1,000,000$ individuals, while the number of variants can be as high as 15 million.

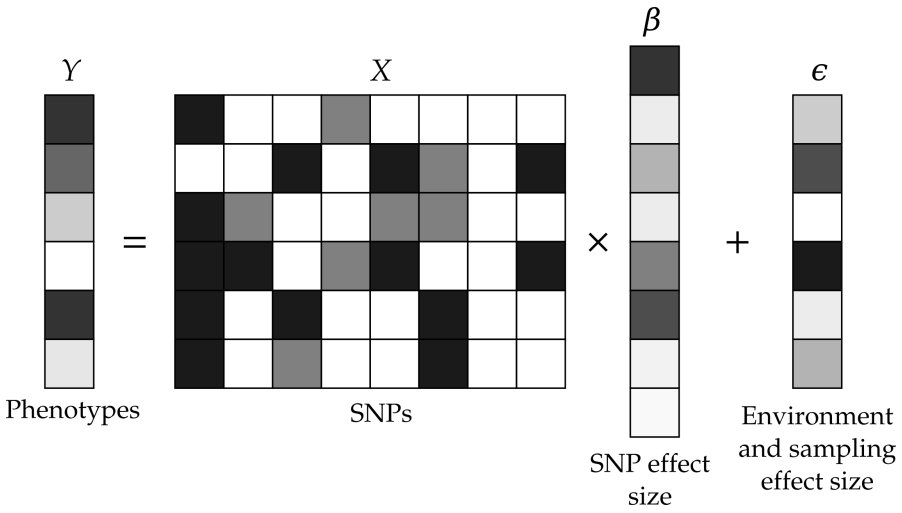


Figure 2.5: A visual representation of GWAS regression analysis.

A common method used to fit a straight line to a data set is the linear least squares method [57] which attempts to minimize the residual sum of squares $S(\alpha, \beta)$, where a residual (ϵ_i) represents the difference between an observed data point y_i and the estimated value $\hat{y}_i$, as shown in Figure 2.6.

$$\begin{aligned} S(\alpha, \beta) &= \sum_{i=1}^{n} \epsilon_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \\ &= \sum_{i=1}^{n} (y_i - \alpha - x_i\beta)^2 \\ &= \|Y - \alpha - X\beta\|_2^2 \end{aligned} \tag{2.2}$$

If the y-intercept is not included, minimising $S(\beta)$ results in $\hat{\beta}$, the best linear unbiased estimator of β.

$$\begin{aligned} \hat{\beta} &= \underset{\beta}{\operatorname{argmin}}\, S(\beta) \\ &= (X^T X)^{-1} X^T Y \end{aligned} \tag{2.3}$$

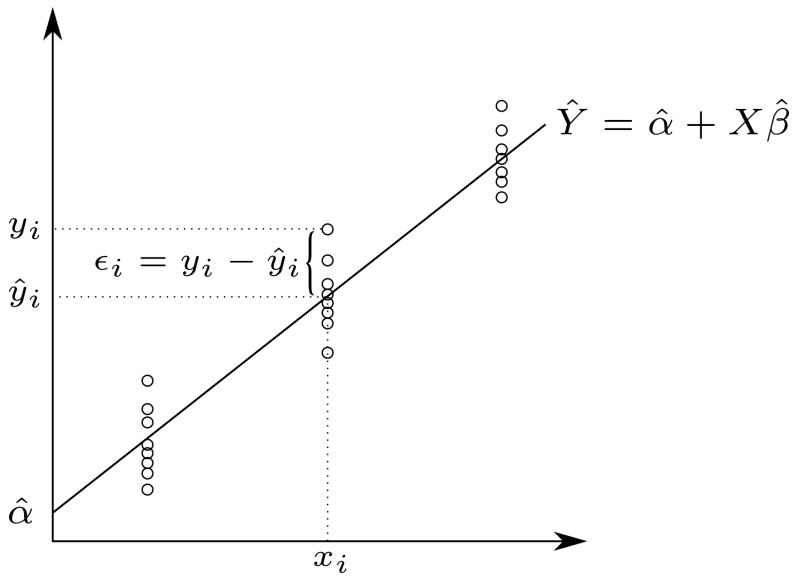


Figure 2.6: A straight line $\hat{Y}$ fitted to a genotype-phenotype data set showing the residual/prediction error ϵ_i.

Covariates (stored in an $n \times c$ matrix C, where c is the number of covariates) can be included in the regression analysis by replacing the phenotype vector Y, with the residuals obtained by regressing Y on C i.e.

$$Y^* = Y - \hat{Y}_C = Y - (C^T C)^{-1} C^T Y \tag{2.4}$$

An $n \times 1$ vector of ones is commonly appended to C so as to include the y-intercept in the regression analysis. If the y-intercept is the only covariate, Y^* is equivalent to the centred phenotype vector $Y - \bar{Y}$.

When a single regressor is used in a linear regression model, $\hat{\beta}$ can also be expressed as

$$\hat{\beta} = \frac{\sum_{i=1}^{n}(x_i - \bar{X})(y_i - \bar{Y})}{\sum_{i=1}^{n}(x_i - \bar{X})^2} \tag{2.5}$$

where $\bar{X}$ is the mean of X and $\bar{Y}$ is the mean of Y.

The F-statistic is used to determine the statistical significance of each genotype-phenotype association by quantifying the goodness of fit of the linear regression model. The F-statistic is defined as

$$F = \frac{\rho^2}{1 - \rho^2}(n - 2) \tag{2.6}$$

where ρ is Pearson's correlation coefficient

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \bar{X})(y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{Y})^2}} \tag{2.7}$$

2.1.5 Linear Mixed Models

High levels of genomic relatedness between individuals in a GWAS violates the assumption that samples are independent and can result in false associations due to the fact that phenotypes could be correlated with a shared ancestry rather than with a specific genotype.

Linear mixed models (LMM), an extension of simple linear regression to allow both fixed (genotypic) and random (relatedness) effects, are used to include the hidden relatedness effect in a GWAS regression model which reflects the fact that the phenotypes of individuals with a similar genotype should appear to be more correlated than the phenotypes of genetically different individuals.

The standard linear mixed model for a single variant is defined as

$$Y = C\alpha + X\beta + \mu + \epsilon \qquad \epsilon \sim N(0, \sigma_e^2) \qquad \mu \sim N(0, \sigma_g^2 K) \tag{2.8}$$

where C is an $n \times c$ matrix of non-genetic covariates with an appended column of ones, α is a $c \times 1$ vector of non-genetic effects, X is an $n \times 1$ vector of variant genotypes, β is a scalar value representing the effect of the variant on the phenotype, μ is an $n \times 1$ vector of random effects, ϵ is an $n \times 1$ vector of errors, σ_e is the variance of environmental effects, σ_g is the variance of genetic effects and K is an $n \times n$ kinship matrix representing the known genetic relationship between each individual which is inferred from the genotypic data of all included variants.

Algorithms used to solve LMMs attempt to estimate the variance parameters using restricted maximum likelihood or maximum likelihood methods which involve iteratively optimising the respective log-likelihood functions. Although more accurate than simple linear regression, solving LMMs is significantly more complex than simple linear regression and the computational time and memory usage of LMM-solving algorithms is orders of magnitude higher.

A number of algorithms have been developed to reduce the computational burden of solving LMMs. They are classified as either approximate or exact based on the computation of the variance parameters. Approximate methods assume that the ratio of the variance parameters remains constant for each variant, while exact methods compute the variance parameters individually for each variant.

GRAMMAR (Genome-wide Rapid Association using Mixed Model And Regression) [58] does not directly compute the variance parameters, but regresses the phenotype data on the kinship matrix and other non-genetic covariates under the null model (i.e. under the assumption that the genotype has no effect on the phenotype), and uses the computed residuals as quantitative phenotypes for genotype analysis using the simple linear model. As GRAMMAR does not directly compute the variance parameters, it is faster, but significantly less accurate than other LMM methods [15]. EMMAX (Efficient Mixed-Model Association Expedited) [59] estimates σ_g and σ_e using the kinship matrix once under the null model and then tests each variant for association. EMMAX assumes that the ratio between σ_g and σ_e remains constant for each variant, which does not hold true for strongly associated variants (which will have a smaller σ_g), so these variants may appear less significant.

Exact methods take advantage of the fact that the log-likelihood functions used to calculate the variance parameters can be rewritten such that they require optimisation of a single parameter [60]

$$\delta = \sigma_g^2 / \sigma_e^2 \tag{2.9}$$

EMMA (Efficient Mixed Model Association) [60] uses spectral decomposition to speed up the computation of δ via numerical optimisation, but requires a new computation of the spectral decomposition for each SNP [16]. FaST-LMM (Factored Spectrally Transformed Linear Mixed Models) [16] uses a single spectral decomposition of the relatedness matrix to determine the variance ratio of all SNPs simultaneously. The phenotypes, SNP data

and covariates are then transformed so that the data becomes uncorrelated, which allows simple linear regression to be used to detect associated variants. GEMMA (Genome-wide Efficient Mixed Model Analysis) [15] uses a similar method to FaST-LMM but is slightly more performant due to the use of a different optimisation algorithm.

2.1.6 Multiple Linear Regression

Although this work is focused on the acceleration of GWAS permutation testing using a linear regression model with a single independent variable, multiple regression models are increasingly being used in GWAS analyses in order to overcome some of the limitations of independently testing each variant. This section seeks to contextualise the current state of GWAS research by giving an overview of GWAS multiple linear regression.

In addition to detecting significant variants to understand a given phenotype, a goal of GWAS analysis is to develop a model which can be used to predict the occurence of a phenotype given an individual's genotype. Multiple linear regression can be used to develop a phenotype prediction model (as well as to improve the power to detect associated variants [61]) by simultaneously estimating the effect size of all variants i.e.

$$Y = \alpha + X_1\beta_1 + X_2\beta_2 + \ldots + X_m\beta_m + \epsilon \tag{2.10}$$

A solution for multiple linear regression cannot however be calculated using $\hat{\beta} = (X^TX)^{-1}X^TY$, where X is an $n \times m$ genotype matrix with significantly more SNPs than individuals ($m \gg n$), due to a phenomenon called *multicollinearity*. If different variant genotypes are correlated due to linkage disequilibrium and relatedness between individuals (i.e. if columns of X are linearly dependent on some of the other columns), X^TX cannot be inverted, and a unique solution to $\hat{\beta} = (X^TX)^{-1}X^TY$ does not exist.

Regularisation can be used to alleviate the effects of multicollinearity in linear regression by penalising the residual sum of squares function $S(\beta) = \|Y - X\beta\|_2^2$ with a penalty function $P(\beta, \lambda)$ with the regularisation parameter λ. The optimal degree of regularisation is unknown and is set experimentally.

Ridge regression [62] uses an l_2-norm penalisation function $P(\beta, \lambda) = \lambda \|\beta\|_2^2 = \lambda\beta^T\beta$ which has the effect of shrinking (or *regularising*) the effect size of correlated variants toward each other [63]. The ridge regression estimator is then

$$\hat{\beta}_R = (X^TX + \lambda I)^{-1}X^TY \tag{2.11}$$

which has a solution as $X^TX + \lambda I$ can be inverted. Heteroskedastic ridge regression is an extension of ridge regression where the penalisation parameter λ_m is set independently for each SNP [64].

Least absolute shrinkage and selection operator (lasso) regression [65] uses an l_1-norm penalisation function $P(\beta, \lambda) = \lambda \|\beta\|_1 = \lambda(|\beta_0| + |\beta_1| + \ldots + |\beta_m|)$ which performs both regularisation and selection. Lasso penalisation has the effect of shrinking the effect size of many SNPs to zero [64]. If $m > n$, lasso selects at most n variables, so it is not viable for a GWAS where $n \ll m$. In addition, if SNPs are correlated, lasso selects only one of the correlated SNPs [66]. The regularised sum of squares function $S(\beta) = \|Y - X\beta\|_2^2 + \lambda \|\beta\|_1$ does not have an explicit solution, so iterative optimisation methods are used to determine the lasso estimator $\hat{\beta}_L$.

Elastic net regression [66] uses the penalisation function

$$P(\beta, \lambda, \alpha) = \lambda(\alpha \|\beta\|_2^2 + (1 - \alpha) \|\beta\|_1) \tag{2.12}$$

which is a combination of both ridge regression and lasso and allows for regularisation as well as selection of all associated SNPs.

Bayesian methods can also be used to fit multiple regression models. Unlike least squares regression which generates a point estimate for a best fit, Bayesian regression estimates a probability distribution for each of the model parameters. Given model parameters θ and the observed data X, Bayes' theorem states that the posterior probability $p(\theta|X)$ (the probability of the model parameters given the observed data) is proportional to the likelihood $p(X|\theta)$ (the probability of the observed data given the model parameters i.e. the *likelihood* that the model best fits the data given the parameters) multiplied by the prior probability $p(\theta)$ (the assumed probability distribution of the model parameters without taking the observed data into account) i.e.

$$p(\theta|X) = \frac{p(X|\theta)}{p(X)} p(\theta) \tag{2.13}$$

Considering the multiple regression model $Y = C\alpha + X\beta + \epsilon$, where X is an $n \times m$ matrix of variant genotypes, β is an $n \times 1$ vector of variant effect sizes, C is an $n \times c$ matrix of non-genetic covariates, α is a $c \times 1$ vector of covariate effect sizes and ϵ is an $n \times 1$ vector of errors, Bayesian regression is used to estimate the posterior probability distribution for the variant effect size β. A number of Bayesian GWAS methods have been developed which differ primarily based on the prior probability distribution selected for β [67]. Determining posterior probability distributions analytically requires the computation of multi-dimensional integrals, which is typically not computationally feasible, so Markov chain Monte Carlo methods are used to simulate a posterior probability distribution based on the prior distributions selected for the model parameters [67].

2.2 Compensating for Multiple Hypothesis Testing

The aim of GWAS hypothesis testing, which simultaneously tests millions of variants for association with a phenotype, is to maximise the chance of detecting valid associations while minimising false associations [68]. A GWAS tests each variant under the null hypothesis that no association exists between genotype and phenotype, and rejection of the null hypothesis implies that some form of association exists. A false association (an incorrect rejection of the null hypothesis) is classified as a Type 1 error, while a missed association (a missed rejection of the null hypothesis) is classified as a Type 2 error.

The null hypothesis to test whether an arbitrary SNP i with effect size β_i is associated with a phenotype can be stated as:

$$H_{0,i} : \beta_i = 0 \tag{2.14}$$

while the alternative hypothesis is defined as:

$$H_{a,i} : \beta_i \neq 0 \tag{2.15}$$

To test the null hypothesis, a p-value is calculated using the F-statistic as defined in Equation (2.6). A p-value is the probability of obtaining a larger (i.e. more significant) test statistic than the one observed, under the assumption that the null hypothesis is correct [69].

$$p_i = Pr(T \geq t_{obs} \mid H_{0,i} : \beta_i = 0) \tag{2.16}$$

The standard approach to statistical hypothesis testing is to set an acceptable Type 1 error rate α (usually 0.05 or 0.01 by convention) and reject the null hypothesis when

$p_i < \alpha$ which implies that the probability of randomly observing the result is small (i.e. the result is *statistically significant*). Smaller p-values ($p_i \ll \alpha$) provide stronger evidence that the null hypothesis is incorrect. Using a Type 1 error rate of $\alpha = 0.05$ effectively means that 5% of tests will result in a Type 1 error.

Including millions of variants in a GWAS increases the chance of detecting associated variants, but gives rise to a phenomenon known as the *multiple testing problem*, whereby the use of the traditional p-value cut-off value $\alpha = 0.05$ makes false associations effectively inevitable. A GWAS which tests one million variants, for example, could result in 50,000 false associations ($1,000,000 \times 0.05 = 50,000$). Adjusting the Type 1 error rate to control for multiple hypothesis testing is, therefore, a critical factor in obtaining valid results from a GWAS and a number of statistical methods have been developed to control the number of false positives in a multiple testing scenario. Strong control of the Type 1 error rate, however, results in more Type 2 errors, so there is always a trade-off between false associations and missed associations. Publishing results with false associations is more damaging than missing associations, so researchers tend to prefer more conservative error controlling methods that aim to minimise false associations [70].

Procedures which control for multiple testing start with a collection of null hypotheses $H_1, \ldots, H_m$ with corresponding p-values $p_1, \ldots, p_m$ which are calculated using test statistics $T_1, \ldots, T_m$. Table 2.1, adapted from Benjamini & Hochberg [71], summarises the outcomes of a series of m hypothesis tests with n_{FP} Type 1 errors and n_{FN} Type 2 errors.

Table 2.1: Summary of the outcome of a multiple testing scenario.

	H_0 not rejected	H_0 rejected	Total
H_0 is true	n_{TN} (number of true negatives)	n_{FP} (number of false positives)	m_0
H_0 is false	n_{FN} (number of false negatives)	n_{TP} (number of true positives)	m_a
Total	n_N	n_P	m

As in the case of single hypothesis testing, a typical approach to multiple hypothesis testing is to set the Type 1 error rate at a predefined level α to provide an acceptable level of control over n_{FP} while simultaneously attempting to minimise n_{FN}. Weak control of the Type 1 error rate implies that control at level α is only guaranteed when $m_0 = m$ (i.e. when all null hypotheses are true), while strong control of the Type 1 error rate guarantees control for any distribution of true or false null hypotheses [68]. A GWAS requires strong control of the Type 1 error rate as at least one rejection of the null hypothesis is expected.

Different Type 1 error rates have been defined to enable different levels of control over n_{FP}. The family-wise error rate (FWER) [68] is the probability of at least one Type 1 error.

$$FWER = Pr(n_{FP} > 0) \tag{2.17}$$

The false discovery rate (FDR) [71] is the expected proportion of Type 1 errors to all rejections of the null hypothesis. To avoid division by zero, Benjamini & Hochberg [71] deal with the case where there are no rejections of the null hypothesis by setting n_{FP}/n_P to zero.

$$FDR = E\left(\frac{n_{FP}}{n_P} \mid n_P > 0\right) Pr(n_P > 0) \tag{2.18}$$

The positive FDR (pFDR) [72] only considers the case where there is at least one rejection of the null hypothesis (i.e. $n_P > 0$). The definition of the pFDR facilitated the development the q-value, an adjusted p-value which provides a measure of significance in terms of the FDR for each hypothesis [68]. A GWAS expects at least one rejection of the null hypothesis, so $FDR \approx pFDR$ as $Pr(n_P > 0) \approx 1$.

$$pFDR = E\left(\frac{n_{FP}}{n_P} \mid n_P > 0\right) \tag{2.19}$$

In most multiple testing situations, the FDR is smaller than the FWER as

$$E\left(\frac{n_{FP}}{n_P}\right) \leq Pr(n_{FP} > 0) = Pr\left(\frac{n_{FP}}{n_P} > 0\right)$$

which indicates that control of the FWER also provides control of the FDR [73].

In a GWAS context, statistical methods that control for multiple hypothesis testing are affected by the fact that the tests are not independent due to LD between SNPs. Methods which do not account for correlation between SNPs control for a 'worst case' scenario and are more conservative (i.e. there is a higher chance of Type 2 errors) than methods which account for dependence between tests. Research into LD patterns of common variants [74] has resulted in a genome-wide significance level of $\alpha = 5 \times 10^{-8}$ being accepted as a standard for GWASs, independent of the number of SNPs included in the analysis. This significance level is based primarily on LD patterns of European populations, so it may not be applicable for African populations, which have much lower levels of LD. An additional caveat is that, while $\alpha = 5 \times 10^{-8}$ provides acceptable control of the FWER for common variants, it cannot be relied upon to provide adequate control of the FWER when rare variants, which have unknown LD patterns, are included in the analysis.

2.2.1 Controlling the Family-Wise Error Rate

The Bonferroni correction is the simplest method of controlling for multiple testing. It replaces α with α/m and rejects the null hypothesis H_i when $p_i < \alpha/m$ [68]. The standard genome-wide significance cut-off corresponds to a Bonferroni adjustment for one million tests ($\alpha/m = 0.05/10^6 = 5 \times 10^{-8}$). If tests are independent, the Bonferroni method provides strong control of the FWER at $m_0\alpha/m$ [73], which is generally too conservative in a GWAS context where at least one rejection of the null hypothesis is expected. Moreover, the Bonferroni method is overly conservative when tests are correlated. As many of the variants in a GWAS are correlated due to LD, α does not have to be adjusted as low as α/m to provide effective control of the FWER [73].

Holm's method [75] is a less-conservative extension of the Bonferroni method. The algorithm orders the p-values from smallest (p_1) to largest (p_m) and finds the smallest i such that $p_i \leq \alpha/(m - i + 1)$. The null hypotheses $H_1, \ldots, H_i$ are then rejected. This method is more effective than the Bonferroni correction but, when m is large, the difference between m and $m - i$ is negligible when i is small, so Holm's method is not significantly better than the Bonferroni method when testing millions of variants [70].

Hochberg's algorithm [76] is similar to Holm's method, but is slightly more powerful. Like Holm's method, the p-values are ordered from smallest to largest, but Hochberg's method finds the *largest* i such that $p_i \leq \alpha/(m - i + 1)$ and rejects the null hypotheses $H_1, \ldots, H_i$. Hochberg's algorithm accounts for some correlation between tests, but it does not provide sufficient power for high levels of correlation [77]. Figure 2.7a compares

Holm's and Hochberg's algorithms for FWER control and demonstrates why Hochberg's method is more powerful.

2.2.2 Controlling the False Discovery Rate

An advantage of using methods which control the FWER is that they provide strong evidence that rejected null hypotheses are correct, but control of the FWER can be very strict in a GWAS context where at least one rejection of the null hypothesis is expected. Methods to control the FDR were developed with the aim of reducing the Type 2 errors which are common with the use of conservative FWER-controlling methods. FDR-controlling methods adapt to the data as they attempt to control the *proportion* of Type 1 errors, rather than strictly controlling the *number* of Type 1 errors [73].

Benjamini & Hochberg [71] defined the FDR and developed an algorithm for control of the FDR at level α. Benjamini & Hochberg's method orders p-values from smallest (p_1) to largest (p_m) and finds the largest i such that $p_i < i\alpha/m$. The null hypotheses $H_1, \ldots, H_i$ are then rejected. The use of $i\alpha/m$ as a criteria for significance allows more null hypotheses to be rejected compared to Hochberg's procedure for FWER control. Benjamini & Hochberg's method is valid when p-values are independent and uniformly distributed, so it does not account for correlation between tests. Figure 2.7b demonstrates why Benjamini & Hochberg's algorithm for FDR control is more powerful than the FWER-controlling methods displayed in Figure 2.7a.

Unlike the Benjamini & Hochberg procedure, Benjamini & Yekutieli's algorithm [78] controls the FDR under general dependence [73]. The procedure ranks the p-values from smallest to largest and finds the largest i such that

$$p_i < \frac{i\alpha}{m\sum_{j=1}^{m}(1/j)} \tag{2.20}$$

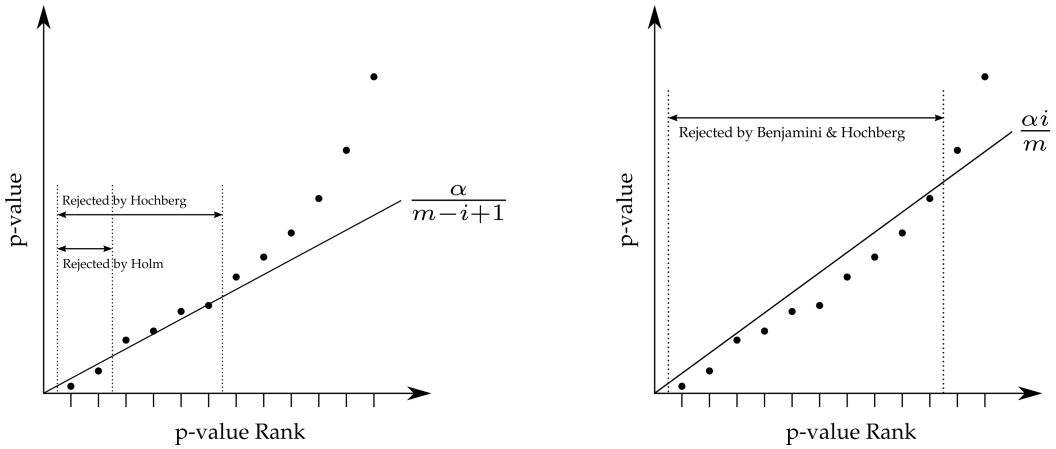
and rejects the null hypotheses $H_1, \ldots, H_i$. Benjamini & Yekutieli's procedure is much more conservative than the Benjamini & Hochberg algorithm (and, in some cases, is more conservative than FWER-controlling methods) as $m\sum_{j=1}^{m}(1/j) > m$.

Storey's method for FDR/pFDR control [79] differs from Benjamini & Hochberg's such that, rather than setting an acceptable FDR threshold α and estimating the region Γ where the null hypothesis should be rejected, it fixes the rejection region and estimates α. Storey's method has better power than Benjamini & Hochberg's algorithm for FDR control, but, like Benjamini & Hochberg's method, it is only valid for independent p-values.

New methods developed for FDR control include covariate data to improve power [80]. Although these methods are moderately more powerful than the classical FDR-controlling algorithms, their ability to reduce Type 2 errors is not significantly better [80].

2.3 Permutation Testing

If the true null distribution of the test statistics in a multiple testing scenario is known, more effective control of the Type 1 error rate can be achieved. In practice, however, the true null distribution is unknown, so the multiple testing procedures described above assume the null distribution in order to control the Type 1 error rate. A critical consideration when controlling for multiple testing is that the assumed null distribution



(a) Controlling the FWER (b) Controlling the FDR

Figure 2.7: Visualisation of methods used to control for multiple hypothesis testing.

provides acceptable control under the true null distribution [13] which may not be a valid assumption in a GWAS.

Permutation testing is a non-parametric randomisation procedure which provides a robust and powerful method of controlling the Type 1 error rate [13]. It avoids any assumptions about the null distribution of the data by simulating the true null distribution of the test statistics using the observed data [70]. Another advantage of permutation testing is that test statistics do not have to be computed with extreme precision as permutation-adjusted p-values are determined by comparing permuted test statistics rather than a direct calculation.

As demonstrated by Figure 2.8, randomly permuting the phenotypes among the individuals in a GWAS destroys the relationship between genotype and phenotype and represents a sampling under the null hypothesis [1]. While the genotype-phenotype relationship is broken by permuting the phenotypes, the structure of the correlation between variant genotypes is maintained together with the distribution of the test statistics corresponding to the true null hypotheses [73].

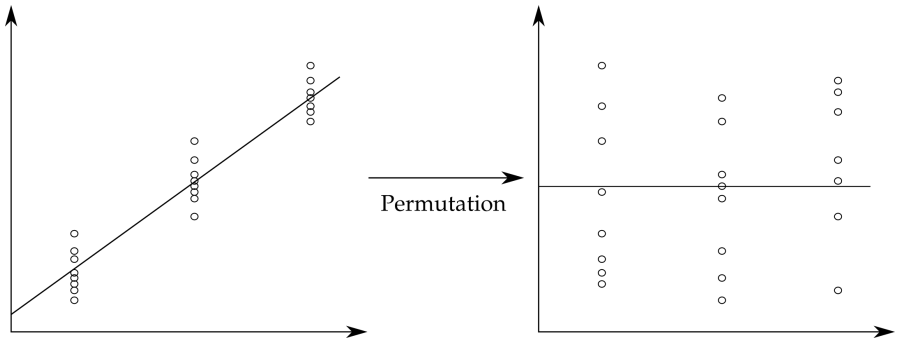


Figure 2.8: Permuting the phenotypes destroys the genotype-phenotype relationship and represents a sampling under the null hypothesis.

If test statistics are calculated for all possible permutations of the phenotypes (requiring $n!$ permutations, where n is the number of phenotypes), the empirical null distribution is generated, while an approximate null distribution is generated when test statistics are computed for a random subset of permutations. A permutation-adjusted p-value is obtained by ranking the observed test statistic among the permuted test statistics. In

order to avoid p-values of zero when no permuted test statistics are larger than the one observed, permuted p-values are defined as

$$p = \frac{x+1}{b+1} \tag{2.21}$$

where x is the rank of the observed test statistic and b is the number of permutations [81].

For a GWAS with a large sample size, recalculating test statistics for $n!$ permutations for even a single variant is computationally impossible, and generating permutation-adjusted p-values with enough resolution to compare against the standard GWAS significance threshold of 5×10^{-8} requires at least 2×10^7 permutations for each variant, so a number of methods have been developed to reduce the computational expense of permutation testing.

Westfall & Young's maxT permutation procedure [82] strongly controls the FWER (at the expense of a potential loss in power) by generating a null distribution using the *maximum* test statistic from each permutation iteration. A permutation-adjusted p-value can then be calculated by ranking the observed test statistic against the collection of maximum test statistics. Alternatively, the p-value cut-off can be adjusted by computing the α-percentile of the maximum test statistics [73]. As the maxT method simulates a permutation distribution using the maximum test statistic of each iteration, much fewer permutations are required to generate a representative null distribution. In fact, at $\alpha = 0.05$, one thousand permutations are typically sufficient, independent of the number of SNPs [73].

Research into permutation algorithms for FDR control is ongoing, although no algorithms have been developed which exhibit the simplicity and reliability of Westfall & Young's maxT permutation procedure for control of the FWER [73]. Most methods for FDR-based permutation testing implement the FDR/pFDR-controlling algorithms on permutation-adjusted p-values.

A class of algorithms called adaptive permutation algorithms have been developed to reduce the computational burden of GWAS permutation testing. Adaptive algorithms attempt to identify and stop permuting SNPs which do not appear to be significant early on in the permutation process so that variants which do appear to be significant can undergo more permutations, generating more accurate p-values for interesting SNPs. Adaptive permutation sets a limit R on the number of times the permuted test statistics can exceed the observed statistic. Permuted test statistics are calculated until either R test statistics are greater than the one observed (requiring B permutations) or the maximum number of permutations b have been performed. A p-value is then estimated as

$$\hat{p} = \begin{cases} \dfrac{x+1}{b+1} & \text{if } x < R \\ \\ \dfrac{x+1}{B+1} & \text{if } x = R \end{cases} \tag{2.22}$$

Although the development of the maxT and adaptive permutation algorithms has alleviated some of the computational burden of GWAS permutation testing, there is a need for a GWAS permutation testing accelerator as the computational cost of permutation testing for modern GWASs remains significantly high.

2.4 GWAS Permutation Testing Implementations

PLINK [14] is a widely used C/C++ command line tool which includes a variety of genome-wide analysis functions. PLINK supports adaptive and maxT permutation testing for both categorical and continuous phenotypes. Although PLINK includes a number of permutation testing optimisations from PERMORY [83] and PRESTO [84], these optimisations only apply to case/control phenotypes, so permutation testing for continuous phenotypes is significantly slower. PLINK is multithreaded and can distribute permutation testing over a multicore CPU and maxT permutation testing can be distributed over multiple CPUs by triggering multiple instances of maxT permutation testing on different CPUs and combining the results.

PLINK implements an adaptive permutation algorithm which, at specified intervals, calculates a confidence interval

$$CI = (1 - \frac{\beta}{2T}) \times 100 \tag{2.23}$$

for each permutation p-value (where β is a user-defined parameter and T is the number of SNPs) and drops SNPs from the permutation procedure when the confidence interval does not contain the user-defined value α. PLINK allows the user to set the minimum and maximum number of permutations for each SNP and sets the SNP removal rate as $ap + b$ where p is the number of completed permutations and a and b are set by the user.

Che et al. [8] recommend the parameters in Table 2.2 for adaptive permutation testing in order to generate p-values with enough precision (c) to compare against the standard GWAS significance threshold of $\alpha = 5 \times 10^{-8}$, where

$$c = SE(\hat{p})/\alpha. \tag{2.24}$$

Table 2.2: A parametrisation of GWAS adaptive permutation testing by Che at al. [8].

	b	R
$c = 0.2$	5×10^8	36
$SE(\hat{p}) = 1 \times 10^{-8}$		
$c = 0.1$	2×10^9	121
$SE(\hat{p}) = 5 \times 10^{-9}$		

GWAS-Flow [85] is a Python library which uses Google's machine learning framework TensorFlow [86] to accelerate LMM permutation testing using the EMMAX approximation method. TensorFlow allows the GWAS-Flow library to make use of CPUs or GPUs for parallelisation of the EMMAX algorithm. When few individuals ($n < 500$) are included in the analysis, the CPU version of GWAS-Flow (running on a 16-core Intel i9 processor) outperforms the GPU version running on an Nvidia Tesla P100 GPU. When more individuals ($n > 500$) are included, however, the GPU version of GWAS-Flow begins to outperform the CPU version. GWAS-Flow is 2-3 times faster than GEMMA, which is not a significant improvement when considering the fact that GWAS-Flow implements the approximate EMMAX algorithm while GEMMA is an exact method.

Terada et al. [27] present a GPU-accelerated implementation of the maxT algorithm for GWAS analysis called High-speed Westfall-Young (HWY). In addition to GPU acceleration, HWY stops permuting SNPs which do not affect the permutation null distribution and uses a lookup table to prevent the repeated calculation of identical permuted p-values. The authors state that HWY running on an Nvidia Tesla K20 GPU is 122 times faster than PLINK running on a 20-core Intel Xeon E5-2680v2 CPU. HWY calculates p-values using Fisher's exact test, so it only supports case/control phenotypes and does not support the inclusion of covariate data.

Zhang et al. [87] present an FDR-controlling permutation algorithm (Adaptive Monte Carlo Multiple Testing — AMT) which identifies significant SNPs via adaptive sampling. The algorithm takes inspiration from the multi-armed bandit problem to minimise the number of permutations required for accurate estimation of the significant p-values. The AMT algorithm is designed to control the FDR at level α using Benjamini & Hochberg's method. Unlike the other adaptive permutation algorithms which perform more permutations for small p-values, the AMT algorithm performs more permutations for p-values of uncertain significance (i.e. it quickly stops permuting very significant *and* insignificant SNPs), which allows the AMT algorithm to perform fewer overall permutations.

PBOOST [26] is a GPU-based accelerator for permutation testing of genome-wide interaction association studies. PBOOST uses a GPU to parallelise the computation of permuted test statistics which are used to determine the association between pairs of SNPs and a phenotype. PBOOST only supports case/control phenotypes and cannot directly handle continuous covariates. The authors state that PBOOST running on an Nvidia Tesla M2090 GPU is around 100 times faster than the same permutation algorithm running on an 8-core Intel Xeon E5-2650 CPU.

2.5 Field-Programmable Gate Arrays

FPGAs are reconfigurable integrated circuits (ICs) that can implement any arbitrary synchronous digital circuit [88]. In contrast to CPUs and GPUs, which have fixed architectures and instruction sets, FPGAs do not have a defined architecture and have to be configured to perform a specific function. Although the adaptability of FPGAs can increase the complexity of FPGA-based algorithm development, it can be exploited to design highly optimised architectures. Due to their inherently parallel nature and optimised use of hardware resources, FPGAs can be significantly faster and more power efficient than sequentially operating, general-purpose hardware such as CPUs, even though FPGAs are typically clocked an order of magnitude lower. For this reason, FPGAs are often used as *hardware accelerators* in heterogeneous computing platforms consisting of a *host processor* and one or more hardware accelerators. A heterogeneous architecture allows the host processor to perform complex conditional branching logic (which is difficult to effectively implement in FPGA hardware), while computationally intensive tasks are offloaded to an optimised hardware accelerator.

The need for large-scale parallel architectures to accelerate computational tasks has arisen due to physical constraints which limit the maximum frequency of CPUs. Parallelisation can be used to accelerate a computational task by splitting the task into identical, independent sub-tasks which can be processed concurrently by multiple cores operating in parallel. FPGAs, computer clusters and GPUs are examples of highly parallelised architectures.

A cluster consists of a networked set of computer nodes which work together as a single

system. Cluster software is typically developed using the Message Passing Interface (MPI), a standardised communication protocol for parallel architectures. Clusters generally offer multiple levels of parallelisation as the individual nodes which make up the cluster will typically contain multi-core CPUs and/or GPUs. Although clusters offer a very high level of parallelisation, they consume a large amount of power and take up a large physical space.

General purpose GPUs are often used as accelerators as they consist of many high-throughput parallel processing cores and they are easy to program using well-supported high-level libraries (such as CUDA, OpenGL and OpenCL). GPUs are more power-hungry than CPUs and FPGAs, however, and they have limited flexibility as they have a fixed instruction set and are optimised for high-throughput graphical processing.

In a bioinformatics context there is a need for high-throughput accelerators due to the increasing size of biological data sets. The adaptability and relative low cost of CPUs and GPUs has made it difficult to justify the time and expense required to develop and set up FPGA-based accelerators, but the availability of cloud-based FPGA instances which can be rented at an hourly rate has made powerful (but expensive) FPGAs more accessible. Cloud-based acceleration is a viable, potentially low-cost alternative to a local hardware setup, particularly in a scenario where the hardware is not constantly in use.

Table 2.3 compares the cost and power consumption of CPUs, GPUs and FPGAs in terms of comparable AWS (Amazon Web Services) EC2 (Elastic Cloud Compute) instances. It should be noted that Table 2.3 displays the total hardware cost and instance power consumption (i.e. it includes the cost and power consumption of a host CPU for the GPU and FPGA instances). Table 2.3 demonstrates that, although the cost of an FPGA-based hardware platform is higher than the cost of a CPU or GPU-based platform, the f1.2xlarge instance provides a potentially low-cost (depending on the instance rental time) option for FPGA-based acceleration. Unlike CPUs and GPUs which consume power at a constant rate (when all cores are active) independent of the work done, FPGA power consumption is dependent on the task, so Table 2.3 displays the power consumed by the FPGA circuit designed in this dissertation. It can be seen that the power consumption of the FPGA instance is lower than that of the other instances.

Table 2.3: Table comparing the cost and power consumption of CPUs, GPUs and FPGAs in terms of comparable AWS EC2 instances.

EC2 Instance/Hardware	Hardware Cost ($)	Instance Cost ($/hr)	Instance Power Consumption (W)
c4.8xlarge CPU — Intel Xeon E5-2660 v3 (36 cores)	1,350	1.591	375
g4dn.4xlarge GPU — NVIDIA T4 CPU — Intel Xeon Gold 6212U (16 cores)	2,500	1.204	180
f1.2xlarge FPGA — Xilinx Virtex Ultrascale+ VU9P CPU — Intel Xeon E5-2686 v4 (8 cores)	7,150	1.650	105

As this research aims to investigate the use of cloud-based FPGA platforms to reduce the cost of FPGA-based permutation testing, and AWS EC2 (which provides access to Xilinx

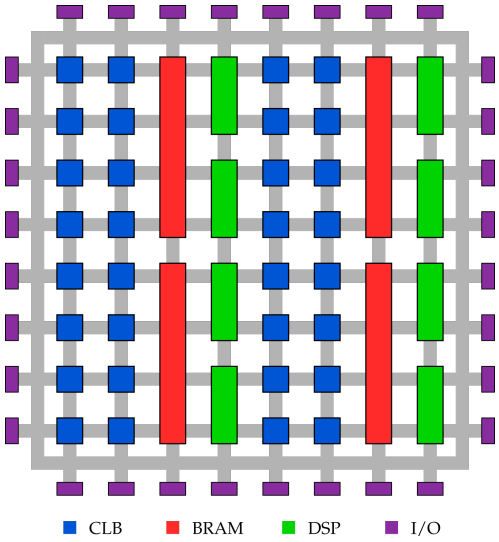
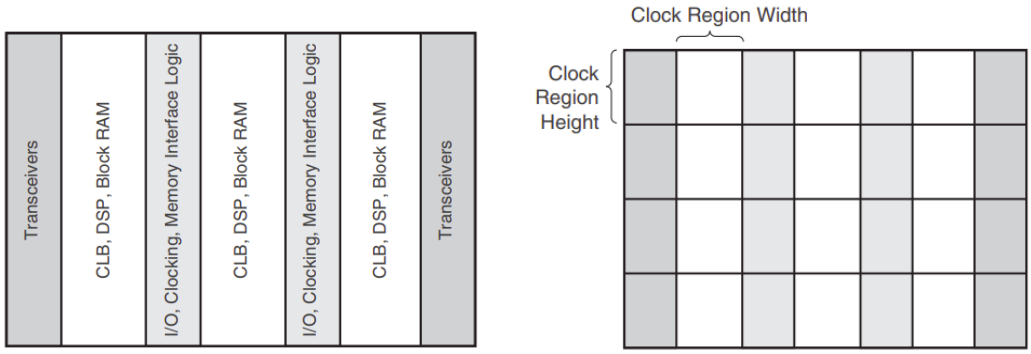


Figure 2.9: Simplified FPGA structure showing an interconnected array of logic elements arranged in a columnar layout.



(a) UltraScale columnar architecture.

(b) UltraScale clock regions.

Figure 2.10: Xilinx UltraScale FPGA architecture [4].

Virtex UltraScale+ VU9P FPGAs) was chosen as the cloud service for development and testing of the accelerator, this dissertation will focus on Xilinx FPGAs and development tools, although the development process for other FPGAs is similar.

2.5.1 FPGA Characterisation

Figure 2.9 shows a generic layout of an FPGA which consists of an array of configurable logic blocks (CLBs) surrounded by a matrix of configurable interconnect (or *routing*) hardware. Interspersed amongst the columns of CLBs are columns of specialised hardware elements including high-speed block RAM (BRAM), digital signal processing (DSP) blocks designed for fast floating-point and fixed-point multiply-accumulate operations, and I/O blocks which facilitate communication with external devices.

Figure 2.10 shows a high-level representation of Xilinx's UltraScale FPGA architecture. The architecture arranges the FPGA logic elements in a columnar configuration as illustrated by Figure 2.10a. Figure 2.10b shows how resources within each column are further divided into different clock regions. Each clock region contains 60 CLBs, 52 I/O blocks, 24 DSP slices or 12 BRAMs. The clock regions contain routing hardware which can be segmented at the clock boundary in order to optimise performance.

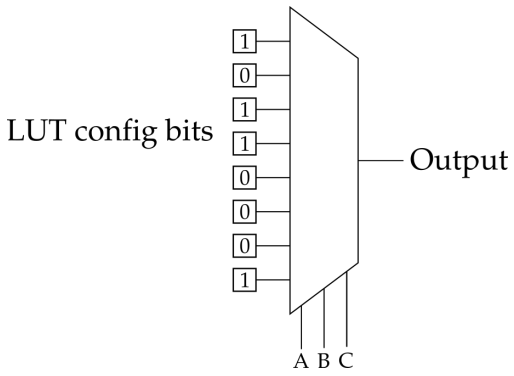
Configurable Logic Blocks

The fundamental building block of an FPGA is the CLB — a logic element used to implement general-purpose combinational and sequential circuits. A CLB comprises a collection of smaller logic elements including look-up tables (LUTs), flip-flops and multiplexers [89].

A LUT (as illustrated by Figure 2.11) consists of a small amount of SRAM which stores the possible outputs of a Boolean logic function and a multiplexer which selects the output based on the function's truth table. An n-input LUT can compute any arbitrary n variable Boolean logic function and the multiplexers within a CLB allow LUTs to be connected together to implement wider functions.

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(a) Truth table of an arbitrary 3-input Boolean function.



(b) 3-input LUT used to compute the Boolean logic function in Figure 2.11a.

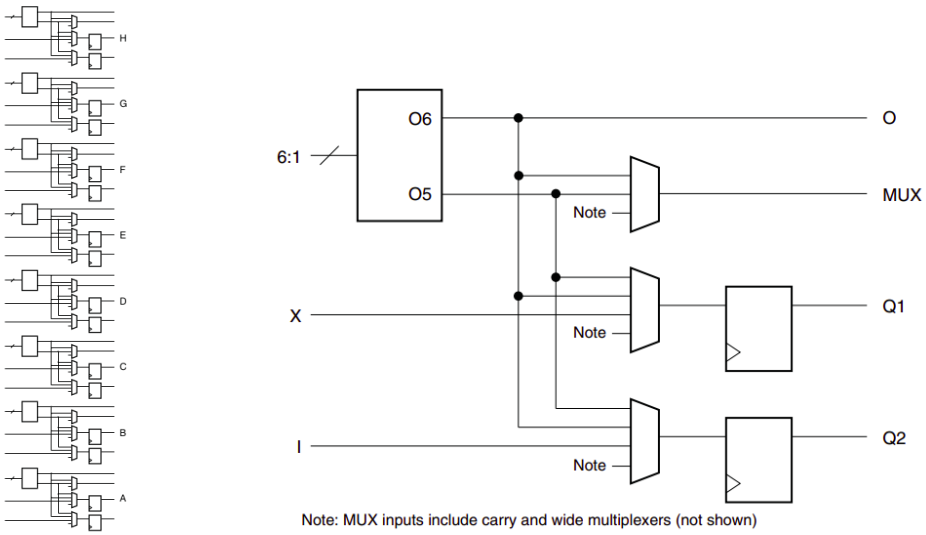
Figure 2.11: FPGA LUT logic

The structure of a Xilinx UltraScale CLB (or *slice*) is illustrated by Figure 2.12. An UltraScale CLB consists of eight 6-input LUTs, 16 flip-flops, a number of multiplexers and specialised carry logic which is used to perform fast addition/subtraction [5]. The UltraScale slice I/O connects the output of each 6-input LUT to the slice output, or optionally, two flip-flops. Each 6-input LUT can be configured to act as a 4:1 multiplexer, and all LUTs in a slice can be combined to implement a 32:1 multiplexer. Some UltraScale CLBs (SLICEM, as opposed to SLICEL which can only implement logic functions) can also be configured as 512-bit memory elements (where each LUT acts as a block of distributed 64-bit RAM) or as 256-bit shift registers.

Memory

One of the major considerations in FPGA design is the optimisation of memory usage as memory bandwidth is often a critical performance bottleneck. Xilinx UltraScale FPGAs have access to a variety of memory sources which differ in terms of size and speed. The RAM types available to UltraScale FPGAs are described below and are ordered in terms of increasing access latency and increasing capacity.

Distributed RAM is memory which is generated using SLICEM CLBs. The data access latency of distributed RAM is generally very low as the RAM is typically placed in close proximity to the logic elements which require the storage. As distributed RAM is composed of CLBs, however, the maximum capacity of distributed RAM is usually quite limited.



(a) One slice/CLB. (b) A simplified view of the LUT I/O.

Figure 2.12: Structure of the Xilinx UltraScale CLB [5].

Block RAM (BRAM) refers to 36 Kb blocks of dedicated memory with two independent ports which are interspersed among the CLB columns. Each BRAM can be used as a single block of 36 Kb RAM or two independent 18 Kb blocks. Ports used to access the 36 kB BRAM can be configured as $32K \times 1$, $16K \times 2$, $8K \times 4$, $4K \times 9$, $2K \times 18$, $1K \times 36$ or 512×72 and adjacent BRAMs can be cascaded to generate large, high-speed memory arrays and FIFOs. While the access latency of BRAM is usually quite low, BRAM is slower than distributed RAM as it is typically further away from the logic element which uses the storage (and may run on a different clock).

UltraRAM refers to blocks of 288 Kb dual-port dedicated memory with higher capacity, but lower speed, than BRAM. In contrast to BRAM, UltraRAM ports can only be configured as $4K \times 72$. If applicable, UltraRAM can be used as a high-speed replacement for external memory.

External DDR RAM is high-capacity RAM which is external to the FPGA logic, but placed on the same board. This RAM is generally used for large-scale data storage at the expense of high access latency.

DSP Blocks

DSP blocks are used to perform fast floating-point and fixed-point multiply-accumulate operations. Figure 2.13 shows a schematic of the UltraScale DSP48E2 slice, which consists of a 27×18-bit twos-complement multiplier, a 27-bit pre-adder and a 48-bit accumulator.

FPGA Hardware Resources

The reconfigurable nature of FPGAs makes hardware resource usage a critical consideration when designing an FPGA architecture. For this reason, FPGA designs are typically characterised in terms of both performance (i.e. speed and power consumption) and consumed hardware resources (or *area*). Table 2.4 summarises the hardware resources of three different Xilinx FPGAs which are featured on currently available development boards.

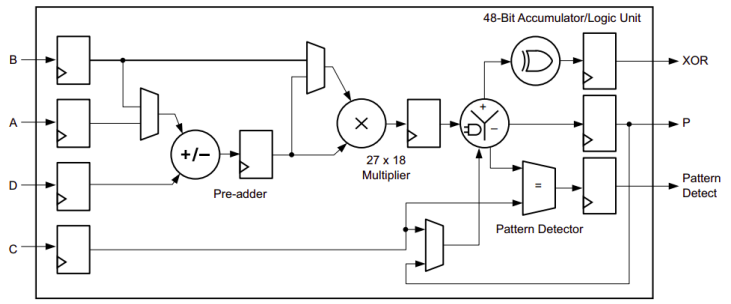


Figure 2.13: Simplified structure of the DSP block (the DSP48E2) used in UltraScale FPGAs [6].

Table 2.4: FPGA Hardware Resources.

Xilinx Development Board	Zedboard	ZCU102	VCU118
FPGA	XC7Z020	ZU9EG	VU9P
Cost	$475	$2,500	$7,000
LUTs (K)	53	274	1,182
Flip-flops (K)	106	548	2,364
BRAM (Mb)	5	33	76
UltraRAM (Mb)	0	0	270
Distributed RAM (Kb)	1,100	8,800	36,100
DSPs	220	2,520	6,840

2.5.2 FPGA Design Process

Algorithm development for FPGAs is a significantly different process than writing software for a CPU or GPU. In FPGA development, a hardware architecture is designed for a specific function, whereas software development for a CPU or GPU works within the framework of a pre-defined hardware architecture to implement the required functionality. This makes FPGA development a more complex and time-consuming process than software development for CPUs or GPUs. As FPGA algorithm development is primarily a hardware design exercise, a deep knowledge of the specific underlying hardware architecture is required for effective development [90].

FPGA circuit design was initially performed manually, but the exponential increase in circuit complexity due to shrinking transistor sizes necessitated the development of tools to automate the FPGA design process which is effectively a multi-objective optimisation problem under resource and timing constraints. Xilinx FPGA design tools are bundled into the Vivado development suite which includes tools for FPGA design automation, analysis of all steps of the FPGA design process and, if fine-grained control of the design is required, manual FPGA circuit design.

Design at the Register-Transfer Level

FPGAs are 'programmed' using a hardware description language (HDL) which implements a high-level register-transfer level (RTL) abstraction of a synchronous digital circuit in terms of the flow of signals between registers and logic elements. HDL is compiled to generate a physical digital circuit which can be implemented on FPGA hardware. The most widely used HDLs are VHDL (Very High Speed Integrated Circuit

Hardware Description Language) and Verilog. They are equally effective and the use of one over the other generally comes down to personal or industry preference [91]. While an HDL may look similar to a high-level software language like C or C++, HDL development is more complex than software development as it is operating at a much lower level of abstraction.

High-Level Synthesis

The knowledge required for effective circuit design using an RTL language has historically limited FPGA development to hardware design engineers, but an automated process called high-level synthesis (HLS), which compiles C/C++ code to an RTL representation [90], has made FPGA algorithm development more accessible. Although HLS can result in a potentially less-efficient RTL representation than HDL development in VHDL or Verilog (particularly in terms of hardware resource utilisation), HLS tools are constantly improving and the time saved in development can offset the loss of efficiency.

Although HLS abstracts some of the hardware-level details of FPGA algorithm development, an effective HLS design must inform the HLS compiler about the desired hardware architecture. In other words, an algorithm developed in C/C++ for a CPU cannot be directly converted to an RTL representation using HLS with the assumption that it will perform well (or at all) on FPGA hardware. Xilinx's HLS tool — Vitis HLS — requires the use of compiler directives to inform the HLS compiler about the desired hardware architecture.

The use of HLS can reduce design iteration time due to the higher level of abstraction compared to an RTL design. This enables the FPGA development process to focus on optimising the performance of the design rather than dealing with the complexities of HDL development. An HLS design is also more generic than one developed in an HDL — as HLS abstracts many of the hardware-level details, a single HLS design can be compiled for different FPGA platforms with minimal changes. Moreover, an inexperienced HDL developer can produce an inefficient hardware design, whereas HLS-aided development can result in an effective design without the need for prior FPGA development experience. Furthermore, Vitis HLS generates Verilog and VHDL code, so low-level optimisation can be performed if a sufficiently effective design cannot be generated solely through the use of HLS.

RTL Synthesis

Synthesis is the process of converting an RTL design into a logic gate-level representation of a digital circuit. A synthesis tool infers combinational and sequential logic elements from the RTL to generate a circuit netlist (as shown in Figure 2.14), which is then mapped to the target FPGA platform. RTL synthesis is an optimisation problem under two general constraints: timing constraints (i.e. the desired frequency of the circuit) and physical constraints (i.e. the constraints which define the placement of I/O pins and the absolute or relative placement of logic cells as well as the constraints based on the configuration settings of the target FPGA platform). The Vivado synthesis tool is timing driven (i.e. it attempts to generate a circuit which can operate at a defined frequency) and has a number of pre-defined design strategies which can be used to optimise for performance, area or synthesis runtime.

After synthesis is complete, Vivado performs logic optimisation on the circuit netlist in order to generate a more efficient circuit design for the specified FPGA platform. A number of different logic optimisations are available and Vivado supports different

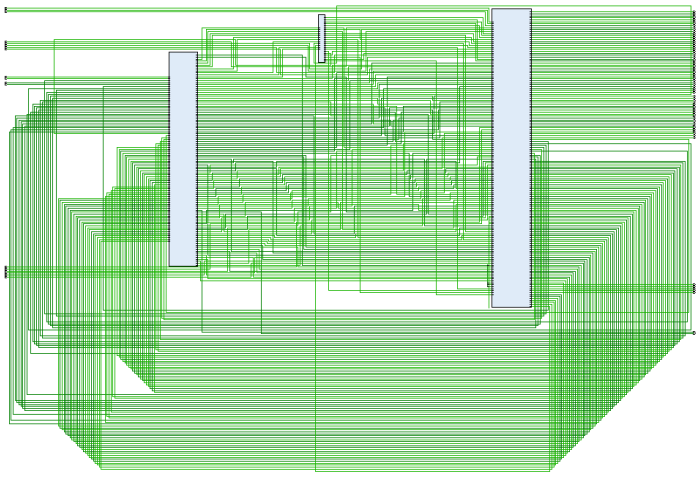


Figure 2.14: Circuit schematic generated by RTL synthesis of the FPGA architecture designed in this work.

strategies to control the behaviour of the logic optimisation process.

Logic Placement

Placement is the process of placing logic cells from the synthesised netlist onto specific sites on the target FPGA. The Vivado placement tool attempts to optimise for:

- Timing slack — cells are placed in order to minimise the effect of propagation delays which can affect the maximum frequency of the circuit
- Wire length — the overall wire length of circuit connections is minimised in an attempt to reach the target frequency
- Congestion — placement seeks to avoid situations where many logic cells are placed in very close proximity in order to reduce the volume of signals through the local circuit

Vivado performs post-placement optimisation in an attempt to further improve timing and congestion after all logic elements have been assigned to specific locations. Placement usually has the largest effect on the performance of the final circuit design, so Vivado offers a number of different placement strategies which optimise for performance, area and circuit congestion.

Routing

Routing of signals between the placed logic elements is the final step of the FPGA design process. When performing the routing procedure, Vivado attempts to route the design such that the target frequency is met. If placement has resulted in a congested design and the target frequency cannot be met, the router will adjust the frequency in order to meet the timing constraints.

Bitstream File

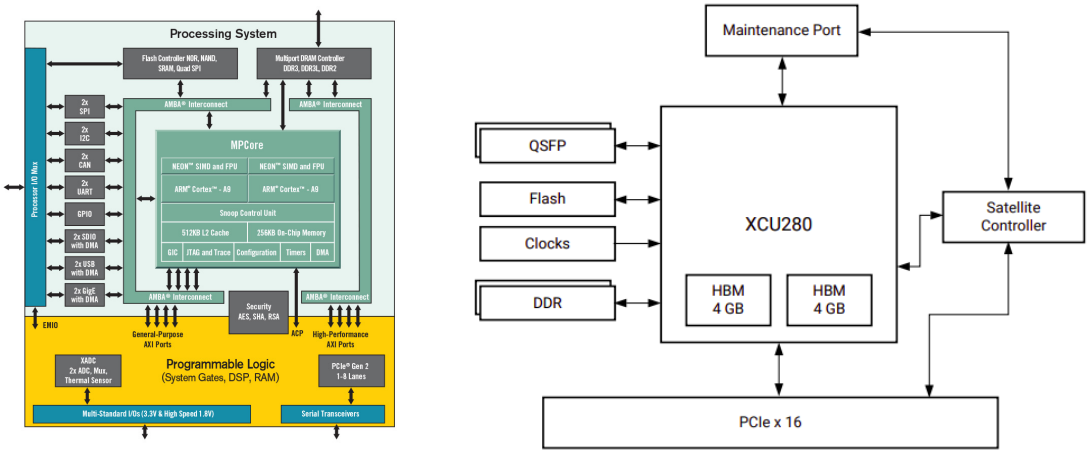
The output of the FPGA design process is a binary bitstream file which stores the data used to configure the FPGA hardware. As FPGAs store the circuit configuration in RAM, they have to be reconfigured whenever they are powered on. FPGAs can store the bitstream file in non-volatile flash memory and configure themselves on start-up or, alternatively, a host CPU can load a bitstream into the FPGA to reconfigure the hardware

when required.

2.6 Heterogeneous FPGA-based Architectures

As explained above, it can be beneficial to combine the raw computing power of an FPGA together with the adaptability of a general-purpose CPU. Some FPGA-based platforms (such as the Xilinx Zynq-7000 family) combine FPGA programmable logic together with a host microprocessor and peripherals in a single chip (called a system-on-chip or SoC). Alternatively, some FPGAs (such as the Xilinx Alveo family and the VU9P used in this work) communicate with a host CPU via a high-speed PCIe (peripheral component interconnect express) interface.

Figure 2.15a shows a schematic of the Xilinx Zynq-7000 SoC family which integrates a dual-core ARM processor and multiple peripherals with FPGA programmable logic, while Figure 2.15b shows a schematic of the Xilinx Alveo U280. Additionally, a soft processor (a microprocessor built out of the FPGA logic) can be instantiated at the expense of FPGA hardware resources [88]. ARM has developed Cortex-M1 and Cortex-M3 soft processors for Xilinx FPGAs [92].



(a) Xilinx Zynq-7000 SoC [93]

(b) Xilinx Alveo U280 [94]

Figure 2.15: Examples of heterogeneous FPGA-based architectures.

2.6.1 The Xilinx Vitis Development Platform

The Xilinx Vitis platform is a new (first released in late 2019) software development kit that provides a framework for Xilinx FPGA-based accelerator development. Vitis facilitates heterogeneous computing using the OpenCL framework and Xilinx runtime (XRT). OpenCL provides a set of APIs to allow an x86 or ARM host processor to offload tasks to accelerator *kernels* (which can be implemented on an FPGA, a GPU or another CPU), while XRT is a set of Xilinx-developed communication drivers which enable a host processor to communicate with and execute kernels on an FPGA using OpenCL APIs.

Vitis supports Xilinx FPGAs connected to an x86 processor via a PCIe interface or ARM processors though an AXI4 interface. Vitis FPGA kernels can be designed in C, C++, OpenCL C or an RTL language, while the host software is written in C or C++, and uses OpenCL API calls to communicate with the FPGA kernels. Vitis allows multiple kernels to be instantiated on an FPGA. The kernels can be connected together with streaming interfaces, and they can be configured to interact with the host application via

user-defined interfaces.

Vitis compiles an FPGA bitstream (using the Xilinx Vivado design tools) in a two-step process. First, individual kernels are compiled from RTL/C/C++ to create object files, and the kernel object files are then linked to generate the bitstream. The Vitis compilation tool supports different optimisation levels which can be used to optimise the FPGA design throughout the various design stages (at the expense of increased compiler runtime at higher optimisation levels).

Vitis kernels can be compiled for software emulation, hardware emulation or hardware, which allows the design to be validated and optimised before attempting to run on FPGA hardware, although Vitis does not currently support multiple emulated kernels. Kernels can also be compiled to output debug signals, but compiling a hardware kernel for debugging increases the amount of FPGA hardware resources consumed by the kernel which can limit the maximum frequency of the design. Vitis also includes an analyser application which can be used to visualise the profiling data generated by OpenCL, as well as the debug information produced by the kernel (if enabled).

FPGA development with Vitis is based on the concept of a *Vitis target platform*. The target platform comprises the FPGA logic for the specified FPGA hardware platform together with external DDR memory accessible by both the host and the FPGA (i.e. *global memory*). Xilinx provides a pre-configured target platform for AWS FPGA instances which consists of a static shell region (which contains I/O, status monitoring and lifecycle management logic) as well as a configurable region for user-developed kernels. Figure 2.16 shows the architecture of an AWS EC2 f1.2xlarge instance together with the Vitis target platform.

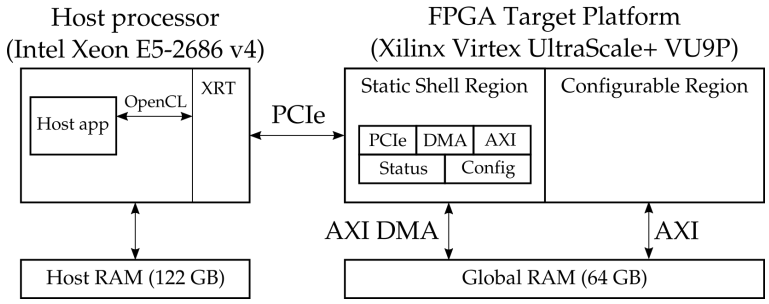


Figure 2.16: Architecture of an EC2 f1.2xlarge instance with the Vitis target platform.

The AWS F1 target platform provided by Xilinx has a maximum frequency of 250 MHz. Therefore, when instructed to compile a design for the F1 target platform, Vitis attempts to generate a circuit which can operate at 250 MHz, but router congestion/timing constraints can limit the maximum frequency of the design. If the compiled design cannot operate at 250 MHz, Vitis automatically sets the clock frequency based on the router timing constraints.

2.6.2 Vitis Kernel Execution and Interfaces

Vitis kernels make use of AXI4 (Advanced eXtensible Interface) interfaces to communicate with the host application via the shell region. AXI4-Lite interfaces are used for the transfer of scalar data types (i.e. any type which contains a 'single' value) which include the various control signals used to manage the execution of the kernel. Vitis kernels are limited to one AXI4-Lite interface. Memory-mapped AXI4 interfaces are used to transfer blocks of data to/from a Vitis kernel via global memory. A kernel can have more than one memory-mapped AXI4 interface, but different memory-mapped buffers can be transmitted through the same AXI4 interface if required. AXI4 streaming interfaces

can be used to stream data directly from the host memory to a kernel, bypassing global memory, but AWS FPGA instances do not currently support streaming interfaces.

To transfer data to a kernel via global memory, the host application must allocate a buffer in global memory and transmit the start address of the buffer to the kernel through the AXI4-Lite interface. The kernel can then read the data from global memory, operate on the data and write the result back to global memory. Once the kernel has notified the host that the kernel output data has been written to global memory, the host application can transfer the data from global memory to the host memory. The latency incurred by transferring data to and from global memory must, therefore, be offset by the acceleration provided by the kernel.

When the required buffers have been allocated in global memory, the host can initiate kernel execution using an OpenCL API call and wait to receive a notification from the kernel that processing is complete. This allows the host application to do other work while waiting for the kernel execution to complete. Kernel execution is typically pipelined so as to optimise performance (i.e. the host application can schedule a new task on a kernel before it has completed the execution of a previous task). The flow diagram in Figure 2.17 summarises the Vitis execution model. OpenCL API calls (which hide calls to XRT) allow the host application to allocate buffers in global memory, transfer data to and from global memory and interact with the FPGA kernels.

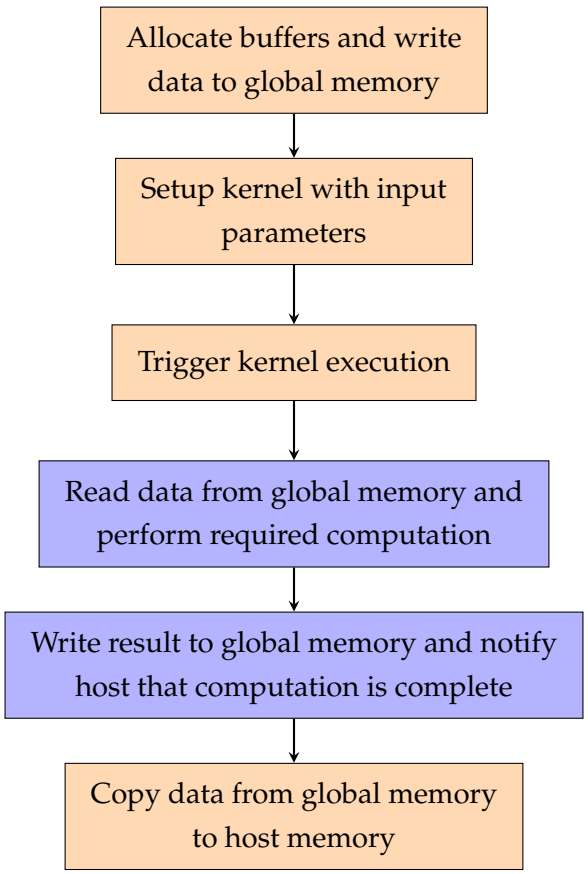


Figure 2.17: Flow diagram showing the execution of a Vitis kernel task by an OpenCL host application. Operations performed by the kernel are highlighted in blue, while those performed by the host processor are highlighted in orange.

2.7 FPGAs as Linear Algebra Accelerators

A large amount of work has been done on leveraging the parallel nature of FPGAs to accelerate a variety of linear algebra tasks. The fact that matrix-vector multiplication (MVM) and general matrix multiplication (GEMM) commonly appear in multiple disciplines has resulted in considerable research into FPGA-based acceleration of these operations. Furthermore, the popularity of machine learning (which makes extensive use of MVM and GEMM) has resulted in a reappraisal of FPGAs as high-throughput machine learning accelerators.

Kestur et al. [95] present a floating-point FPGA-based MVM accelerator for both dense and sparse matrices which uses a novel compression scheme (Compressed Variable-Length Bit Vector — CVBV) in an attempt to reduce storage and memory bandwidth requirements. The accelerator also allows for conversion from other matrix compression formats to CVBV at the expense of increased hardware utilisation. Dense MVM is performed by pipelined processing elements (PEs) which each compute the dot product between the vector and a single row of the matrix. Each PE contains a multiply-accumulator (MAC) to accelerate the dot product computation. The DMVM accelerator uses an FPGA-based controller to maximise performance. The controller is responsible for breaking up the input matrix into blocks which can fit into the FPGA's memory, as well as managing a DMA engine to stream the data to the PEs.

Dorrance et al. [96] present an FPGA-based sparse MVM accelerator which takes advantage of the sequential memory access patterns of the compressed sparse column compression scheme to calculate the output vector by performing column-wise vector additions of the matrix weighted by each element of the input vector. A co-processor is used to stream data to PEs containing a floating-point multiply-accumulator. The number of processing elements is scaled to match the memory bandwidth of the system. An on-board controller is used to handle scheduling errors and situations where multiple PEs attempt to write to the same memory location.

Cadambi et al. [97] present an FPGA-based accelerator for support vector machine (SVM) training and classification using the Sequential Minimal Optimization algorithm. SVM training requires repeated matrix-vector multiplications of a large, constant matrix with a changing vector. The matrix is stored in off-chip DRAM and streamed into the FPGA when required. Memory bandwidth is effectively increased by reducing the arithmetic precision, which allows more operations to occur in parallel by packing more data points into one memory access. The design utilises FPGA DSP blocks as low-precision MACs to accelerate the dot product kernel which constitutes the majority of the algorithm run-time.

Sankaradas et al. [98] also reduce data precision in order to improve the efficiency of an FPGA-based accelerator for convolutional neural networks. Data is converted from a double-precision floating-point representation to reduced precision fixed-point using an experimental approach to determine the best trade-off between speed and accuracy. Data is streamed from off-chip DRAM into dot product PEs which are composed of systolic chains of MACs. The design also takes advantage of the sequential memory access patterns of the MVM calculation in order to optimise the design of the accelerator.

Kyrkou et al. [99] use the Xilinx Spartan-6 FPGA as a hardware accelerator for a cascade SVM classifier with a view towards the development of a low-power, real-time embedded FPGA-based image classifier. Fixed-point decimal data is rounded to the nearest power of two (with a reduced classification accuracy trade-off) so

that multiplications can be performed using bit-shifts and data can be stored in a compressed format. The dot product of one vector is calculated in a fully unrolled parallel implementation with a bit-shifting (multiplication) stage followed by an adder tree pipeline which sums the bit-shift outputs to produce a scalar dot product. The authors state that if a fully parallelised dot product pipeline cannot be achieved due to memory bandwidth or insufficient hardware resources, the unrolling factor of the dot product pipeline should be reduced.

Rabieah et al. [100] propose an FPGA-CPU framework for SVM training. The design attempts to overcome memory bandwidth limitations by dividing a dataset with N data points into k blocks (of size N/k) which can each fit into on-chip memory. The data blocks are processed in parallel by k PEs that each have their own local memory. The CPU is responsible for grouping and sending data to the FPGA and for polling the FPGA to determine whether any PEs are idle.

Kara et al. [101] make use of the Intel Xeon+FPGA platform to accelerate the floating-point stochastic gradient descent algorithm (which performs multiple MVM computations) in order to accelerate the training of machine learning models. The design uses a novel compression scheme called stochastic quantization to reduce the effect of memory bandwidth so as to improve performance.

Moss et al. [102] present a configurable GEMM framework for the Intel Xeon+FPGA platform for use as a deep learning accelerator. The implementation supports arbitrary matrix sizes and different data types (as explained above, reduced data precision results in faster execution). Matrix multiplication is performed by a systolic array of PEs (fed by double buffered queues) which each contain a dot product operator. Data interleaving is used to take advantage of data reuse in the GEMM algorithm so as to minimise the effect of memory bandwidth. Pipelining is used to ensure that one dot product is produced every clock cycle. The design also implements a mixed precision 'dynamic dot product' (which switches between data types during runtime) and maximises the computing resources of the platform by partitioning large matrices into blocks which can be processed by either the FPGA or CPU.

2.8 FPGAs as Bioinformatics Accelerators

The large sample sizes of modern biological datasets has made parallelisation a critical consideration when developing bioinformatics algorithms. Although FPGAs have seen relatively little use in a bioinformatics context, their highly parallel nature combined with reduced power usage (compared to CPUs and GPUs) and optimised use of hardware resources makes them an attractive option to provide large-scale parallelisation.

Wienbrandt et al. [22] present a heterogeneous FPGA-GPU accelerator for genome-wide interaction analysis of case/control phenotypes. The accelerator uses a Xilinx Kintex UltraScale KU115 FPGA together with an Nvidia Tesla P100 GPU to reduce the runtime of logistic regression for interaction analysis. The accelerator uses a CPU host processor to facilitate data transfer between the GPU and the FPGA. The FPGA is used to compute contingency tables for each SNP pair in the study (which represent the degree of correlation between the alleles of a SNP pair), while the GPU is used to perform logistic regression using the contingency tables. The FPGA contingency table generator consists of a chain of 480 PEs which produce 480 contingency tables in parallel when SNP data is streamed into the FPGA. The authors state that the accelerator is 1,000 times faster than multithreaded PLINK running on 16 Intel Xeon E5-2667v4 CPU cores.

FPGASW [103], presented by Fei et al., is a heterogeneous FPGA-based accelerator for the Smith-Waterman (SW) algorithm for large-scale pairwise biological sequence alignment. The accelerator consists of a Xilinx Virtex-7 XC7VX485T FPGA together with a host processor. The accelerator uses a systolic array of PEs to parallelise the computation of the SW scoring matrix and the SW scoring matrix is processed in blocks which can fit into the FPGA memory. For the backtracking stage of the SW algorithm, the accelerator uses a 2-bit compression scheme and performs data prefetching in order to maximise memory bandwidth and reduce access latency.

Xia et al. [104] implement an FPGA accelerator for the GOR-IV algorithm for predicting a protein's 3D structure based on its amino acid sequence. The accelerator consists of a Xilinx Virtex-5 XC5VLX330 FPGA connected to a host processor and is shown to be 430 times faster than an unoptimised CPU-based implementation of the GOR algorithm and 110 times faster than an optimised, multithreaded CPU-based implementation. The accelerator is optimised for data reuse so as to avoid loading data from external memory, and each stage of the algorithm (data loading, processing and data writing) is pipelined to maximise efficiency. The accelerator also reduces data precision (with a minimal loss of accuracy) to optimise the memory usage requirements.

Gundlach et al. [25] present an FPGA-based accelerator for maxT permutation testing of genome-wide epistasis analysis of case/control phenotypes using the MB-MDR 3.0.3 algorithm. The accelerator uses a cluster of 128 low-cost Xilinx Spartan-6 FPGAs, controlled by a single host processor, to compute permuted test statistics. The host processor iteratively supplies each FPGA in the cluster with SNP genotype vectors, and systolic chains of PEs (which each buffer one genotype vector in local memory) compute contingency tables based on streamed genotype vectors. The accelerator achieves a 300-fold speed-up when compared to MB-MDR 3.0.3 running on a 160-core AMD Opteron CPU cluster, and exhibits similar performance when compared to an optimised version of MB-MDR (MB-MDR 4.2.2) running on a CPU cluster comprising 256 Intel Xeon cores.

Wienbrandt et al. [105] present an FPGA accelerator for haplotype phasing using the EAGLE2 algorithm. Haplotype phasing, which refers to the estimation of haplotypes from sampled SNP data using reference haplotypes, is an early stage of the genotype imputation process and is used to reduce imputation runtime and improve accuracy. The FPGA accelerator was developed to speed up the data preparation stage of the EAGLE2 algorithm as it comprises a large part of the algorithm runtime. The accelerator consists of a 16-core Intel Xeon E5-2667v4 CPU and a Xilinx Kintex UltraScale KU115 FPGA. The FPGA architecture consists of 48 independent pipelined kernels which can process different strings of SNP genotypes in parallel. The genotype data is encoded into a compressed 2-bit format, and the reference haplotype data is encoded as a 1-bit data type. During processing, the genotype data is stored in FPGA BRAM, while the reference haplotype data, which is too large to store in BRAM, is streamed to the FPGA and is buffered locally within the kernels. The FPGA-accelerated version of EAGLE2 is shown to be 2 times faster than a multithreaded CPU-based implementation of EAGLE2.

Illumina's DRAGEN (Dynamic Read Analysis for GENomics) platform [106] is a commercial FPGA-based bioinformatics accelerator that can be implemented as a cloud-based system or as a local hardware setup. DRAGEN accelerates a number of genome-wide analysis algorithms to facilitate high-speed whole genome sequencing. Illumina state that DRAGEN can yield a 36-fold speedup over CPU-based algorithms for whole genome sequencing.

3 Accelerator Design

The availability of cloud-based FPGA instances has made powerful FPGAs accessible to bioinformatics labs and has led to FPGAs becoming a viable option for accelerating large-scale bioinformatics workloads. As discussed in Section 2.3, there is a need for a GWAS permutation testing accelerator that can minimise the processing time of large GWAS datasets, and the parallelisation provided by FPGAs makes them well-suited to the task. This chapter will elaborate on the design of a heterogeneous FPGA-CPU accelerator using high-level synthesis (HLS) and the Xilinx Vitis development platform with the goal of determining the efficacy of FPGAs for accelerating GWAS permutation testing.

The following design goals/trade-offs, ordered in terms of priority, informed the design of the accelerator and will be expanded upon throughout this chapter:

- Processing speed — As minimising the runtime of GWAS permutation testing was the primary design goal, many of the design choices sought to maximise the throughput of the accelerator.
- Development time — Reducing the amount of time spent on the development of the accelerator was an important consideration due to time constraints. For this reason, the scope of the research was limited to the development of a GWAS permutation testing accelerator for the analysis of continuous phenotypes using a simple linear regression model (as addressed in Section 2.1.4) and high-level synthesis was used to simplify development.
- Accuracy — Although the accelerator is required to generate accurate permutation-adjusted p-values, high-precision test statistics are not a key requirement due to the fact that permutation p-values are calculated via the comparison between test statistics rather than a direct calculation.
- FPGA hardware resources — As AWS EC2 FPGA instances provide access to high resource capacity FPGAs (the Xilinx Virtex UltraScale+ VU9P — see Table 3.2 for details), optimisation of the hardware resource usage was not a critical consideration, albeit attention was given to ensuring that the design could operate at the maximum operating frequency of 250 MHz.
- Usability/accessibility — As the research aims to produce a tool that can be used by bioinformatics researchers, the usability of the accelerator informed some of the design choices.

This chapter is divided into three sections. Section 3.1 derives the permutation testing algorithm and divides the work between the FPGA and the host CPU. Section 3.2 then goes over the design of the FPGA part of the accelerator using Vitis HLS, while Section 3.3 focuses on the design of the host application.

3.1 GWAS Permutation Testing Algorithm

For a sample of n individuals, permutation testing for a single SNP requires the repeated computation of the test statistic F with each permutation of the phenotype vector Y.

$$F = \frac{\rho^2}{1-\rho^2}(n-2), \quad \rho = \frac{\sum_{i=1}^{n}(x_i - \bar{X})(y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{Y})^2}}, \tag{3.1}$$

Y is an $n \times 1$ vector of continuous phenotypes (or phenotype residuals), X is an $n \times 1$ vector of variant genotypes, $\bar{X}$ is the mean of X and $\bar{Y}$ is the mean of Y. When Y is permuted, the denominator of ρ remains constant while the numerator changes with each permutation. This suggests that the computation of the numerator of ρ is the critical part of the permutation testing algorithm.

The numerator of ρ — or the dot product of an $n \times 1$ vector of phenotypes ($\boldsymbol{y}$) and an $n \times 1$ vector of genotypes ($\boldsymbol{x}$) — can be defined as:

$$\boldsymbol{x}.\boldsymbol{y} = \sum_{i=1}^{n} x_i y_i \tag{3.2}$$

The genotype vector can be expressed as $\boldsymbol{x} = \boldsymbol{a} - \bar{\boldsymbol{a}}$, where, if the additive genotypic model is used, $\boldsymbol{a}$ is a vector of SNP alternate allele counts for each individual ($a_i \in \{0, 1, 2, \text{missing}\}$).

If the GWAS includes c covariates for each individual, the phenotype vector can be expressed as $\boldsymbol{y} = \boldsymbol{p} - C(C^TC)^{-1}C^T\mathbf{p}$, where $\boldsymbol{p}$ is the real-valued phenotype vector, and C is an $n \times (c+1)$ matrix of covariates with a prepended column of ones. If covariates are not included, $\boldsymbol{y} = \boldsymbol{p} - \bar{\boldsymbol{p}}$, which is equivalent to $\boldsymbol{y} = \boldsymbol{p} - C(C^TC)^{-1}C^T\mathbf{p}$ if C is an $n \times 1$ vector of ones.

Permutation testing for m SNPs sampled from n individuals requires the computation of the dot product between the permuted phenotype vector $\boldsymbol{y_p}$ and each genotype vector $\boldsymbol{x_m}$. The fundamental part of the permutation testing algorithm is, therefore, a matrix vector multiplication (MVM) between an $m \times n$ matrix of genotypes and the $n \times 1$ phenotype vector which results in an $m \times 1$ vector of dot products.

To get an idea of the computational complexity of GWAS permutation testing with n individuals, m SNPs and p permutations, the algorithm can be broken down as follows:

- The test statistic for a single SNP is computed with one dot product operation, requiring n multiplications and $n-1$ additions, which results in time complexity $O(n)$.
- The computation of m test statistics requires m dot product calculations and has time complexity $O(mn)$.
- Permutation testing requires the test statistic for each SNP to be computed p times resulting in an overall time complexity of $O(pmn)$

The runtime of the GWAS permutation testing algorithm therefore scales linearly with the number of SNPs, the number of individuals and the number of permutations.

As matrix vector multiplication is the most computationally expensive part of the permutation testing algorithm, the most effective way of accelerating permutation testing on a heterogeneous FPGA-CPU platform is to leverage the parallelism of the FPGA to accelerate the matrix vector multiplication, while utilising the CPU to perform the less computationally intensive (although still demanding) tasks. Table 3.1 summarises

the functions of the accelerator components, while Figure 3.1 shows a simplified flow diagram of the permutation testing algorithm.

Table 3.1: Work done by each accelerator component.

FPGA Xilinx Virtex UltraScale+ XCVU9P	CPU Intel Xeon E5-2686 v4
Matrix vector multiplication	Data preprocessing and buffer management FPGA setup and execution of kernels Permutation testing algorithm and result output

3.2 FPGA Design

This section describes the design of the FPGA part of the permutation testing accelerator using Xilinx Vitis 2020.2 and Vitis HLS. The decision to use a high-level synthesis tool was based primarily on practical time constraints. An RTL design developed in Verilog or VHDL can be very resource efficient and fast, but manually dealing with data dependencies and timing constraints while aiming to maximise performance can result in an extremely complex design. As resource usage was not a primary design consideration due to the large amount of resources provided by the VU9P (as summarised in Table 3.2), HLS was used to speed up the design of the accelerator.

Table 3.2: Xilinx Virtex UltraScale+ VU9P hardware resources.

CLB LUTs (K)	1,182
CLB Flip-flops (K)	2,364
Max distributed RAM (Mb)	36
Block RAM (Mb)	76
UltraRAM (Mb)	270
DSP Slices	6,840

The following metrics were used to quantify and optimise the performance of the FPGA design:

- Latency — The number of clock cycles required for computation of the output. Lowering the latency of a computational task is one of the primary reasons for employing large-scale parallelism. A reduction in latency directly corresponds to a reduction in runtime.
- Initiation Interval (II) — The number of clock cycles before the accelerator can can accept new input data. Minimising the amount of time that the FPGA hardware is idle is a critical consideration when designing an effective accelerator and a reduction of the initiation interval contributes to lower idle time. FPGA design attempts to achieve an initiation interval of one which implies that new data can be processed every clock cycle.
- Throughput — The amount of data that is processed per second. The FPGA throughput is closely related to the initiation interval as a reduction of the initiation interval results in an increase of the throughput.
- Area — The hardware resources (LUTs, BRAM, distributed RAM, DSPs, routing hardware) required to implement the design on the specified FPGA.

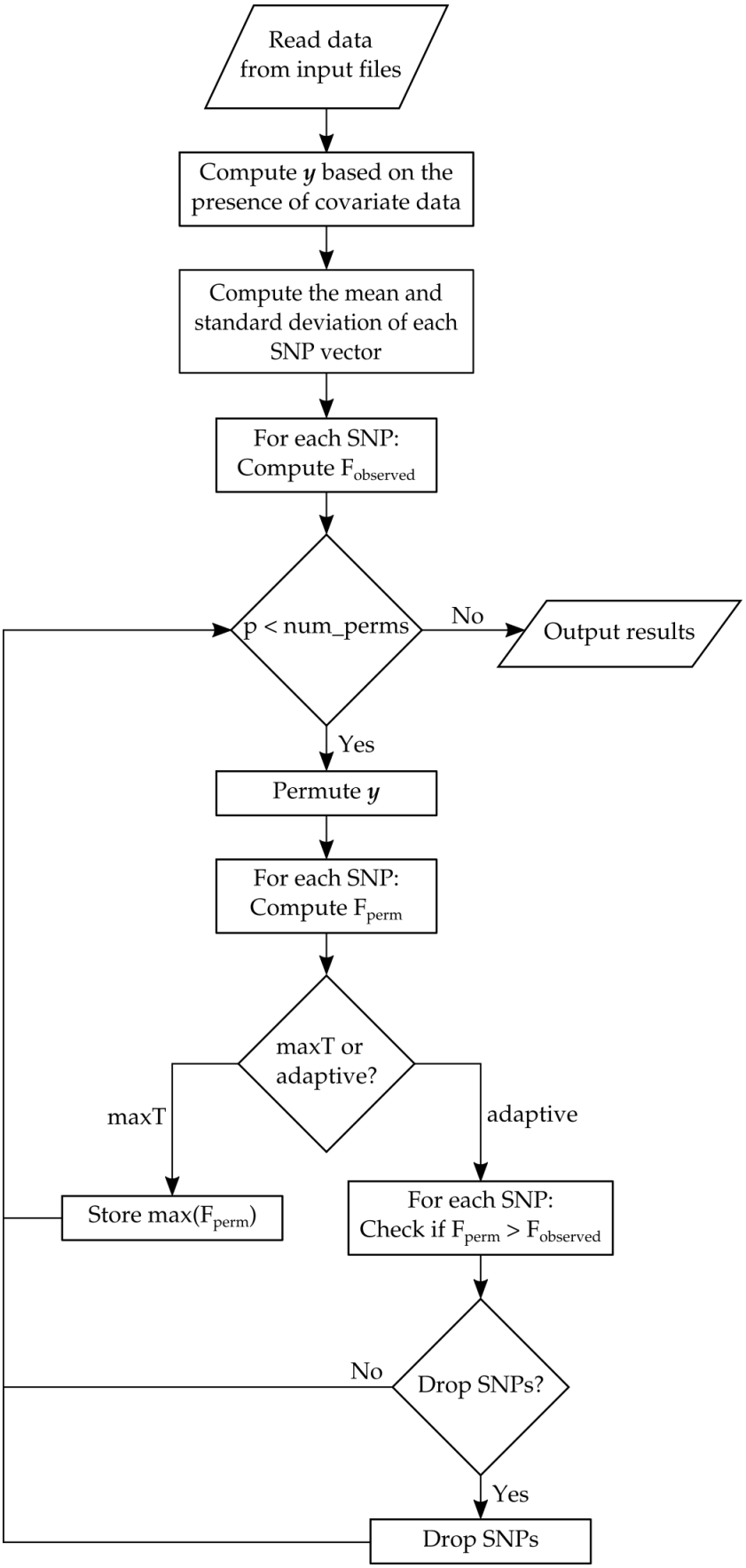


Figure 3.1: Simplified flow diagram of the GWAS permutation testing algorithm.

The aim of the FPGA design process was to develop a Vitis MVM kernel that provides an acceptable level of accuracy while minimising the overall latency and maximising the throughput in order to effectively accelerate the MVM calculation.

The fact that GWASs generally aim to identify potentially associated SNPs rather than confirming hypotheses about specific SNPs means than an approximately correct permutation p-value is typically sufficient for estimating statistical significance. Furthermore, as permutation p-values are not calculated directly from a test statistic but, rather, from the comparison between test statistics, the computation of extremely high-precision dot products was not a primary design goal.

The input to the MVM kernel is the genotype data and the phenotype vector, and the output is a vector of dot products which are subsequently used by the host application to compute the permuted test statistic for each SNP. The genotype data for any reasonably sized GWAS is too large to store in the onboard RAM of the VU9P FPGA (as shown in Table 3.2, the VU9P has around 48 MB of onboard RAM), so it has to be copied to global memory and streamed to the FPGA for each permutation iteration. The transfer of the genotype matrix is therefore a major performance bottleneck. As data is transferred between the host processor and the FPGA via global memory through a PCIe Gen3 x16 interface, the PCIe bandwidth (which is affected by a number of factors including motherboard type, operating system, transfer buffer sizes and communication drivers) is the limiting factor in the overall throughput of the FPGA accelerator.

Figure 3.2 shows a high-level view of the architecture of the AWS F1 target platform developed by Xilinx which consists of the FPGA logic together with four 16 GB banks of off-chip DDR4 DRAM (i.e. global memory). The FPGA logic comprises the Vitis shell region — which, as explained in Section 2.6.1, is a static region containing PCIe, DMA, AXI, status and control logic — as well as a reconfigurable region for user-developed kernels.

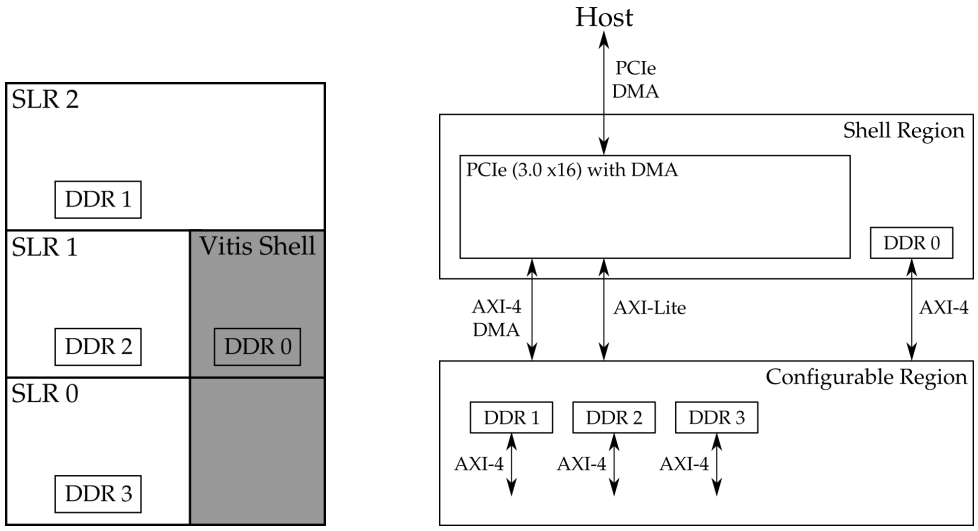
As illustrated by Figure 3.2a, the VU9P FPGA consists of three layers of stacked silicon dies or super logic regions (SLRs) and the Vitis shell is fixed in SLRs 1 and 0. Xilinx suggests distributing a design across the SLRs in order to reduce congestion. Furthermore, as connections between SLRs result in higher latency due to the fact that signals are forced to cross clock boundaries, Xilinx recommends minimising connections between the SLRs.

The AWS F1 target platform includes the following external interfaces:

- One PCIe 3.0 x16 interface that connects the FPGA (via the shell region) to the host processor
- Four DDR4 interfaces that connect the global memory banks to kernels instantiated in the configurable logic region

To optimise the hardware utilisation of the shell region, Xilinx placed one DDR4 interface in the shell region while the remaining three are implemented in the configurable region. Xilinx recommends connecting global memory banks to kernels placed in the same SLR in order to minimise latency. As shown by Figure 3.2b, kernels in the configurable region can access the PCIe interface through an AXI-4 interface and the host can manage kernel execution through an AXI-Lite interface.

The Vitis command line tool `xbutil` includes the `dmatest` command which measures the global memory bandwidth/PCIe throughput by transferring data between the host machine and the FPGA via global memory. A Vitis design that connects a kernel



(a) Architecture of the AWS F1 target platform.

(b) External interfaces of the AWS F1 Vitis target platform.

Figure 3.2: High-level view of the architecture of the Virtex UltraScale+ VU9P and the AWS F1 Vitis target platform showing the layered silicon architecture of the VU9P, the distribution of the global memory banks and the external interfaces which can be used to communicate with and transfer data to/from the FPGA.

to each of the four global memory banks was used to measure the global memory throughput of an AWS f1.2xlarge instance. As demonstrated by Listing 3.1, which shows the output of `xbutil dmatest`, the host-to-FPGA write bandwidth is around 8.5 GB/s and the FPGA-to-host read bandwidth is about 12 GB/s, which are both lower than the theoretical maximum PCIe 3.0 x16 bandwidth of 16 GB/s.

```
Buffer Size: 256 MB
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 8270.97 MB/s
Host <- PCIe <- FPGA read bandwidth = 12177.7 MB/s
Data Validity & DMA Test on bank1
Host -> PCIe -> FPGA write bandwidth = 8483.28 MB/s
Host <- PCIe <- FPGA read bandwidth = 12172.7 MB/s
Data Validity & DMA Test on bank2
Host -> PCIe -> FPGA write bandwidth = 9506.99 MB/s
Host <- PCIe <- FPGA read bandwidth = 12182.7 MB/s
Data Validity & DMA Test on bank3
Host -> PCIe -> FPGA write bandwidth = 8491.34 MB/s
Host <- PCIe <- FPGA read bandwidth = 12176.8 MB/s
```

Listing 3.1: Output of `xbutil dmatest` run on an AWS f1.2xlarge instance showing that the average host-to-FPGA write bandwidth is around 8.5 GB/s while the FPGA-to-host read bandwidth is about 12 GB/s.

In order to maximise the throughput of the FPGA part of the GWAS permutation testing accelerator, the FPGA design focuses on minimising the performance impact of transferring data via global memory by optimising the data types used within the MVM kernel (described in Section 3.2.1) and optimising global memory read/write performance (as discussed in Section 3.2.2), while taking advantage of the inherent parallelism provided by the FPGA hardware by implementing a number of loop optimisations (see Section 3.2.3), carrying out task-level pipelining (Section 3.2.5) and exploiting the AWS F1 FPGA architecture (addressed in Section 3.2.8).

3.2.1 Data Type Optimisation

As PCIe bandwidth is the limiting factor in the performance of the accelerator, it was critical to optimise the data types used by the MVM kernel in order to minimise the amount of data transferred via global memory. The selection of data types also has an impact on the latency of the MVM computation as different configurations of FPGA logic elements (which can vary in the number of clock cycles needed to produce a result) are used to process different data types.

Compression of the Genotype Data

The size of the genotype data is significantly larger than the size of both the phenotype vector and the result vector (for m SNPs sampled from n individuals, the genotype data consists of $m \times n$ elements), so the PCIe bandwidth is primarily used to transmit the genotype data from host memory to global memory.

One of the major advantages of FPGAs over CPUs and GPUs is their ability to natively operate on arbitrary precision data types. This allows FPGAs to take advantage of the compressed nature of SNP genotype data which can be one of four values (0, 1, 2 or missing). Genotype data can, therefore, be represented with a 2-bit data type, and performing the $\boldsymbol{x} = \boldsymbol{a} - \bar{\boldsymbol{a}}$ calculation within the MVM kernel (rather than the host application) can significantly reduce the impact of memory bandwidth on the efficacy of the accelerator. As the genotype data does not change throughout permutation testing process, $\bar{\boldsymbol{a}}$ only has to be calculated once for each SNP and, to ensure that the MVM kernel does not have to wait to receive an entire genotype vector before initiating a dot product computation, the mean of each SNP vector is calculated by the host application and transferred to the kernel along with the genotype and phenotype data.

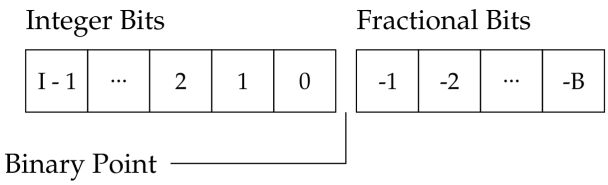
The computation of $\boldsymbol{x} = \boldsymbol{a} - \bar{\boldsymbol{a}}$ within the MVM kernel necessitates the identification of missing genotypes so that they are not included in the dot product calculation. If a missing genotype (encoded as 3) is identified, 0 is added to the dot product total in order to effectively parallelise the dot product computation, as skipping the addition causes data dependencies which limit the parallelism of the dot product computation.

Vitis HLS provides an arbitrary precision integer template class `ap_[u]int`, which allows four 2-bit SNP values to be packed into one byte, quadrupling the effective memory bandwidth of the MVM kernel. The `ap_[u]int` type takes the form `ap_[u]int<N>`, where N is the number of bits, so the genotype data type is defined as `ap_uint<2>`.

Fixed-Point Data Types

The MVM kernel uses fixed-point data types to represent fractional data (as opposed to floating-point types) as fixed-point operations use fewer LUTs, flip-flops and DSP48E2 blocks, can have a lower memory footprint and have lower latency than equivalent floating-point operations. Fixed-point designs also consume less power than equivalent floating-point designs due to the reduced use of hardware resources (although this did not drive the decision to use fixed-point data types).

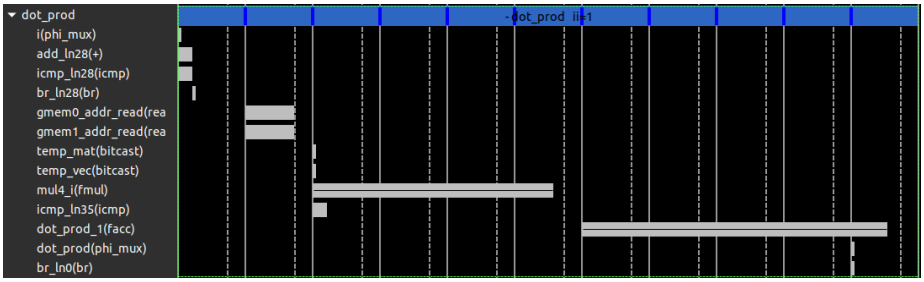
As demonstrated by Figure 3.3, fixed-point values are represented as a sequence of bits with a fixed binary (or decimal) point position (whereas the binary point of a floating-point value can be placed anywhere relative to the significant digit). Bits to the left of the binary point represent the integer part of the number, while bits to the right represent the fractional part.



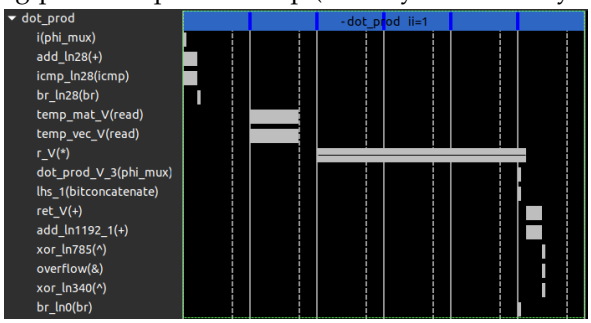
Word Length (W) = Integer Bits (I) + Fractional Bits (B)

Figure 3.3: Fixed-point representation.

Figure 3.4 compares the dot product loops synthesised by Vitis HLS for floating-point and fixed-point data types. It can be seen that the latency of a floating-point multiply-accumulate operation is five clock cycles more than the latency of a fixed-point multiply-accumulate. This is due to the fact that a floating-point accumulate operation requires the use of a DSP block which has a latency of five cycles, while a fixed point accumulate operation is performed using CLB logic which has an effective latency of zero cycles.



(a) Floating-point dot product loop (Latency = 11 clock cycles).



(b) Fixed-point dot product loop (Latency = 6 clock cycles).

Figure 3.4: Comparison between hardware-based fixed-point and floating-point dot product loops using synthesis graphs generated by Vitis HLS. Each row represents an operation performed by a logic element while the columns show the timing for each operation in terms of the number of clock cycles. The rows are ordered based on the sequence of logic operations needed to perform the respective multiply-accumulate operation.

As the permutation testing algorithm does not require extreme precision in the dot product computation, the high dynamic range yielded by floating-point data types was not seen to be a requirement for the design, and the precision provided by the use of fixed-point types was assumed to be adequate. A disadvantage of using a fixed-point type to represent the phenotype data, however, is that the resolution may not be sufficient when working with very small decimal values. Therefore, if the phenotype data requires high resolution fractional representation, it may be necessary to scale the phenotype data (which would have no effect on the computation of the test statistic) so

that the dot product calculation is not affected by the limited resolution of the fixed-point type. For this reason, the host application allows for the use of a constant phenotype scaling factor that can be set using a command line argument.

Vitis HLS provides the hardware-optimised arbitrary precision signed/unsigned fixed-point `ap_[u]fixed` template class which is also used in the host application to convert between floating-point and fixed-point data types. A major advantage of the `ap_[u]fixed` class is that it automatically aligns the binary point when performing calculations with variables of different bit widths and binary point positions. The `ap_[u]fixed` template takes the form `ap_[u]fixed<W, I, Q, O>`, where:

- W is the total word length in bits
- I is the number of integer bits (the implicit number of fractional bits is, therefore, $W - I$)
- Q is the quantisation mode
  - The behaviour of the `ap_[u]fixed` type when the result of an operation requires greater resolution than the defined type
  - `AP_TRN` (truncation) is used in the MVM kernel
- O is the overflow mode
  - The behaviour of the `ap_[u]fixed` type when the result of an operation exceeds the maximum value or is less than the minimum value
  - `AP_SAT` (saturation to the maximum or minimum value) is used in the MVM kernel in order to easily identify whether overflow is occurring

The range of the signed `ap_fixed` type is then

$$-2^{I-1} \leq x \leq 2^{I-1} - 1/2^{W-I}$$

with resolution

$$1/2^{W-I}$$

Xilinx UltraScale DSP48E2 blocks consist of a 27×18-bit multiplier together with a 48-bit accumulator, so the fixed-point types used in the MVM kernel (see Table 3.3) are defined to fit the dimensions of the DSP blocks in order to minimise the utilisation of DSP blocks.

Data Type Summary

The data types used in the kernel are summarised in Table 3.3. The external types (i.e. data generated and used by the host application) are defined to simplify data alignment, while internal types (i.e. types used only within the MVM kernel) are defined to maximise the efficacy of the DSP48E2 blocks and to avoid overflow.

3.2.2 Read/Write Optimisation

The read/write logic of the MVM kernel was optimised so as to make more effective use of the PCIe bandwidth. Due to the latency incurred when accessing global memory, the MVM kernel was designed to minimise the amount of memory accesses. Furthermore, all read/write operations were split into separate functions to enable Vitis HLS to implement task-level pipelining (see Section 3.2.5 for an explanation of task-level pipelining).

Data Buffering

The phenotype vector is buffered within the kernel's internal memory to allow the phenotype data to be reused for each dot product computation. If the phenotype vector

Table 3.3: Summary of the data types used in the MVM kernel.

Data	Type	Range	Resolution
Genotype	ap_uint<2>	$[0, 3]$	1
Genotype Mean	ap_fixed<16,2>	$[-2, 1.999938965]$	0.000061035
Phenotype	ap_fixed<16,11>	$[-1024, 1023.96875]$	0.03125
Dot Product	ap_fixed<32,20>	$[-524288, 524287.999755859]$	0.000244141
Pre-adder Storage	ap_fixed<18,2>	$[-2, 1.999984741]$	0.000015259
Multiplier Storage	ap_fixed<48,30>	$[-536870912, 536870911.999996185]$	0.000003815
Accumulator Storage	ap_fixed<48,30>	$[-536870912, 536870911.999996185]$	0.000003815

is not buffered inside the kernel, it would have to be streamed into the kernel along with each SNP vector which would significantly impact the performance of the accelerator as the global memory bandwidth (which is primarily used to transmit the genotype data to the MVM kernel) would be split between transmitting both the phenotype and genotype data.

A disadvantage of buffering the phenotype data within the kernel is that Vitis requires that the size of the phenotype buffer is known at compile time. This constraint enforces a limit on the sample size of the datasets that can be analysed by the permutation testing accelerator. Kernels can be compiled with a large phenotype buffer to accommodate very large sample sizes, but the FPGA hardware resources will eventually be consumed. Furthermore, supporting a large phenotype buffer can limit the operating frequency of the design due to an increase in router congestion caused by the buffer partitioning scheme (described in Section 3.2.4), particularly when attempting to instantiate multiple kernels.

The phenotype buffer inside the MVM kernel is defined to support sample sizes of up to $262,144$ individuals. If the accelerator is required to handle datasets with larger sample sizes, the host application could be adapted to split the phenotype vector and calculate partial dot products using the MVM kernel, although this feature is not currently implemented.

When instructed to compile a design with four MVM kernels (see Section 4.2, which addresses why four MVM kernels is the optimal configuration), the Vivado compiler failed to generate a design which could simultaneously support a phenotype buffer of $262,144$ elements and operate at the maximum frequency of 250 MHz using the default settings, so a number of build directives (which are summarised in Table 3.4) were used to optimise resource usage and reduce routing congestion.

It was found that instructing the Vivado compiler to expect routing delays during placement (using the ExtraNetDelay_high directive) had the largest impact on reducing the routing congestion. The high runtime of the Vivado compiler prevented further experimentation, however, and it is not known whether a less-stringent placement directive can result in a design that can operate at 250 MHz.

Data Streaming

Data streaming is used to maximise the flow of data within the kernel and to allow for task-level pipelining. Data streams, which store data similarly to an array, have a

Table 3.4: Summary of the Vivado build directives used to generate a design which could operate at 250 MHz.

Design Stage	Build Directive	Description
Logic Optimisation	Explore	Performs multiple rounds of logic optimisations in an attempt to reduce or eliminate excess logic.
Placement	ExtraNetDelay_high	The placement tool anticipates delays caused by routing congestion when performing placement.
Physical Optimisation	Explore	Runs multiple passes of different optimisation algorithms to reduce delays if placement has resulted in a design which cannot operate at 250 MHz.
Routing	AggressiveExplore	Directs the router to aggressively explore different critical path placements if the initial routing path cannot operate at 250 MHz.

number of properties which allow Vitis HLS to generate an optimised design:

- Data is transferred sequentially from the first sample
- Streams have no defined size
- Streams do not require any address management
- Once data is read from a stream, it cannot be read again

This behaviour is ideal for the transfer of the genotype matrix, the genotype mean vector and the result vector within the kernel as the data is not reused.

Vitis HLS supports a template stream class (the `hls::stream<T>` class, where T is the data type stored in the stream) which provides a hardware-optimised implementation of a streaming data structure. An `hls::stream` object within a kernel is synthesised as a FIFO (first-in, first-out) queue with a default depth of two. The default FIFO depth was assumed to be sufficient for the `hls::stream` types defined in the MVM kernel as the kernel consumes/produces streams one sample at a time. Data stored in an `hls::stream` is accessed using class methods which stall execution if a read is attempted on an empty FIFO, or a write is attempted on a full FIFO.

Local Data Caching

Arrays are mapped to BRAM which has a limited number of ports, so repeated array accesses can affect the performance of the accelerator. Local data caching is used to improve performance by copying data from BRAM to local registers so that the BRAM ports are not occupied for the length of the multiply-accumulate operation. This is exemplified by Listing 3.2 which shows how local data caching is used in the MVM kernel to perform a multiply-accumulate operation.

Burst Data Transfer

As accessing global memory incurs significant latency, burst data transfer is used to minimise the volume of memory accesses when reading from or writing to global memory. The default burst size is 16, but Vitis HLS allows the burst size to be modified

```
ap_fixed<48,30> dot_product(const unsigned int num_elem,
                            ap_uint<2> *genotype_data,
                            ap_fixed<18,2> genotype_mean,
                            ap_fixed<16,11> *phenotype_data)
{
    ap_fixed<48,30> dot_product = 0;
    ap_fixed<18,2>  local_genotype_mean = genotype_mean;

    for(int i = 0; i < num_elem; i++)
    {
        ap_uint<2>      temp_genotype  = genotype_data[i];
        ap_fixed<16,11> temp_phenotype = phenotype_data[i];

        ap_fixed<18,2>  temp_pre_adder = temp_genotype - local_genotype_mean;
        ap_fixed<48,30> temp_product   = temp_pre_adder * temp_phenotype;

        dot_product += temp_product;
    }

    return dot_product;
}
```

Listing 3.2: A simplified dot product loop showing how data caching using local storage is used to minimise the volume of memory accesses in the MVM kernel.

(if required) to optimise performance. The Vitis HLS compiler infers burst data transfer with the use of a pipelined `for` loop.

Maximising the Global Memory Interface Width

The maximum width of the global memory interface of a Vitis kernel is 512 bits. Using the full width of the global memory interface provides another method of reducing the number of global memory accesses when reading from or writing to global memory, while simultaneously increasing the parallelism of the kernel. The kernel design had to be adapted to process data in blocks of 512 bits, and special care had to be taken when handling edge cases.

The `WideType` template class was adapted from Xilinx's open source HLS BLAS (Basic Linear Algebra Subprograms) library to simplify the kernel code. The class has the form `WideType<T, N, W>` where:

- T is the data type (e.g. ap_uint<2>, ap_fixed<16,2>) stored in the `WideType` variable
- N is the number of values to pack into 512 bits
- W is the size of the data type in bits

A `WideType` variable is effectively a 512 bit/64 byte array of variables of the defined type. The `WideType` class methods use the `HLS ARRAY_PARTITION` and `HLS UNROLL` compiler directives (see Section 3.2.3) to fully parallelise the processing of a `WideType` variable.

All memory-mapped kernel interfaces are defined as `ap_int<512>` to make use of the full width of the kernel's global memory interfaces when reading from and writing to global memory. When data is read from global memory, the `ap_int<512>` types are cast to the `WideType` of the respective data type, while the kernel output is cast from `WideType` to `ap_int<512>` when it is written to global memory.

The use of the `WideType` class to store and process genotype data has implications

for the hardware resources consumed by the kernel due to the fully unrolled loop implementation. Simultaneously processing 256 2-bit values can use a large amount of hardware and routing resources, but the hardware resources provided by the VU9P are sufficient for implementing the desired architecture, and the build directives used to compile the design (as described in Table 3.4) helped to mitigate the impact of the fully unrolled loop implementation on router congestion.

3.2.3 Loop Optimisation

Loop optimisation is an important aspect of maximising parallelism within the MVM kernel. Figure 3.5 illustrates the parameters which can be optimised to reduce loop latency and, therefore, maximise the kernel throughput.

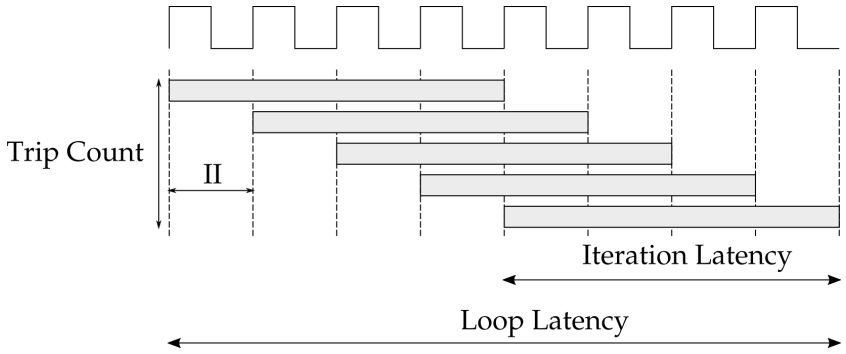


Figure 3.5: Loop parameters that can be optimised so as to maximise the kernel throughput. The primary aim of the loop optimisation process was to achieve an initiation interval of one for all loops.

- Loop initiation interval (t_{II}) — The number of clock cycles between the start of successive loop iterations
- Trip count (n) — The total number of loop iterations
- Loop iteration latency (t_{iter}) — Number of clock cycles needed for one loop iteration
- Loop latency (t_{loop}) — Number of clock cycles required to complete the entire loop

$$t_{loop} = ((n-1) \times t_{II} + t_{iter})$$

Data dependencies can have a significant effect on loop performance as some loops may not be parallelisable because subsequent loop iterations are dependent on the result of previous iterations. The MVM kernel was, therefore, designed to minimise data dependencies within loops so as to achieve $t_{II} = 1$ for all loops.

Loop Unrolling

The default behaviour of Vitis HLS is to keep loops rolled, which results in the loop hardware being reused for each loop iteration. Loop unrolling (enabled with the `HLS UNROLL` compiler directive) is used to perform multiple loop iterations in parallel (as shown in Figure 3.6) by replicating the loop hardware multiple times. If there are no data dependencies within a loop and there are sufficient hardware resources, a loop can be fully unrolled (which effectively sets $t_{II} = 0$) so that all loop iterations are performed concurrently, which reduces t_{loop} to that of a single loop iteration. Vitis HLS supports partial loop unrolling which provides a trade-off between throughput and area.

Unrolling loops with a large number of iterations or loops which contain complex logic within the loop body results in significant hardware utilisation and can increase the

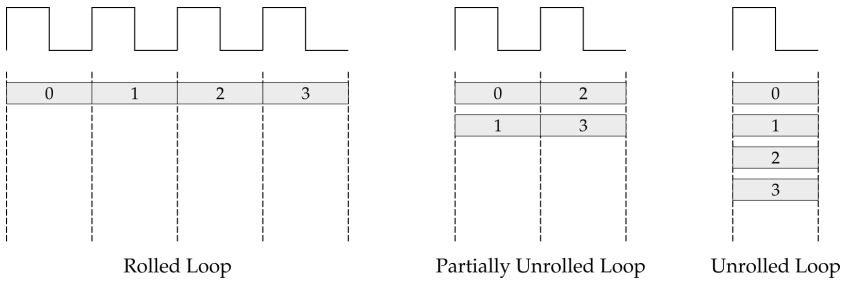


Figure 3.6: Illustration of how loop unrolling can be used to minimise latency by performing multiple loop iterations in parallel.

bandwidth requirements of the kernel, so small loops are unrolled while pipelining is used to accelerate large loops.

Loop Pipelining

By default, Vitis HLS executes loops sequentially which results in a loss of efficiency as the FPGA must wait for a previous loop iteration to complete before initiating a subsequent iteration. Pipelining is used to overlap loop iterations so that a new loop iteration can be started before the previous iteration is complete, allowing the FPGA hardware to be maximally utilised. Figure 3.7 shows how pipelining can be used to reduce t_{II} and, therefore, t_{loop} despite the fact that the loop iteration latency and the number of loop iterations remain unchanged.

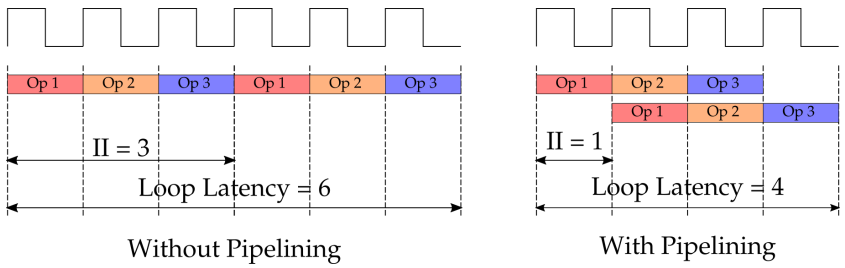


Figure 3.7: Demonstration of how pipelining can reduce loop latency by overlapping successive loop iterations.

The `HLS PIPELINE` directive is used to pipeline loops. When instructed to pipeline a loop, Vitis HLS attempts to achieve $t_{II} = 1$ (meaning that a new loop iteration is started every clock cycle), but this may not be possible due to data dependencies or timing constraints. When pipelining nested loops, the optimal trade-off between area and throughput is achieved by pipelining inner loops only. This is due to the fact that, when outer loops are pipelined, Vitis HLS attempts to fully unroll all inner loops which significantly increases the hardware utilisation and results in data dependencies which could prevent maximal parallelisation.

Matrix Vector Multiplication Loop Structure

Listing 3.3 shows how pipelining and loop unrolling are used to parallelise the MVM computation. The MVM kernel loops over the rows of the genotype matrix (which each represent one SNP) and processes each row in blocks of 512-bits/256 SNP values. The innermost loop, which performs the multiply-accumulate operation, is fully unrolled to parallelise the processing of each block of genotype data. The middle loop, which loops over a row of the genotype matrix in 512-bit blocks, is pipelined to overlap processing of

each 512-bit block. The outer loop, which loops over the rows of the genotype matrix, is not pipelined, as the Vitis HLS compiler would attempt to unroll the intermediate loop which would induce data dependencies in addition to significantly increasing the hardware utilisation of the kernel.

```
const unsigned int blocks_per_row = num_cols / 256;

for (int row = 0; row < num_rows; row++)
{
    for (int i = 0; i < blocks_per_row; i++)
    {
#pragma HLS PIPELINE
        for (int j = 0; j < 256; j++)
        {
#pragma HLS UNROLL
            // 256 fully parallelised multiply-accumulate operations
        }
    }
}
```

Listing 3.3: Structure of the MVM loop showing how pipelining and loop unrolling are used to maximise the parallelism within the MVM kernel.

3.2.4 Array Partitioning

Vitis HLS synthesises arrays as two port BRAM blocks which limits the performance of the accelerator when attempting to implement pipelining and loop unrolling as only two elements of the array can be accessed concurrently. Array partitioning (using the `HLS ARRAY_PARTITION` compiler directive) allows arrays to be accessed in parallel by splitting an array across multiple BRAM blocks or, in the most extreme case, into individual registers. Figure 3.8 demonstrates the different methods of array partitioning.

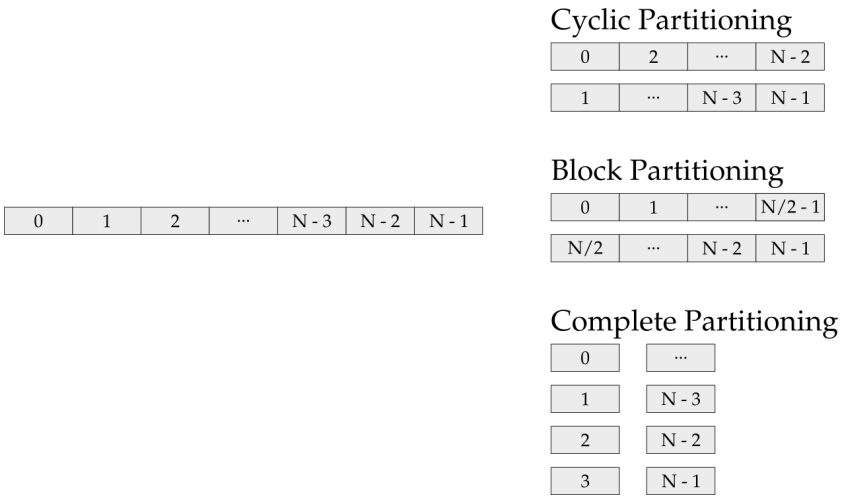


Figure 3.8: Illustration of the different ways an array can be partitioned using Vitis HLS.

For `block` and `cyclic` partitioning, the partition factor specifies the number of arrays which are created. Figure 3.8 shows arrays that have been partitioned with a factor of two, creating two separate arrays in memory which can be accessed concurrently. By default, the Vitis HLS compiler implements the `complete` partitioning scheme, which maps all elements of the array to individual registers. Although the `complete` partitioning scheme results in a fully parallelised array, a large amount of logic and

routing resources are consumed when sizeable arrays are partitioned. The MVM kernel uses `complete` partitioning to process all `WideType` variables in parallel, which is acceptable as a `WideType` array is never required to hold more than 256 elements.

The size of the phenotype buffer prevented it from being completely partitioned, so `cyclic` partitioning with a factor of 8 was used to allow the MVM kernel to process a new 512-bit block of genotype data every clock cycle as illustrated by Figure 3.9.

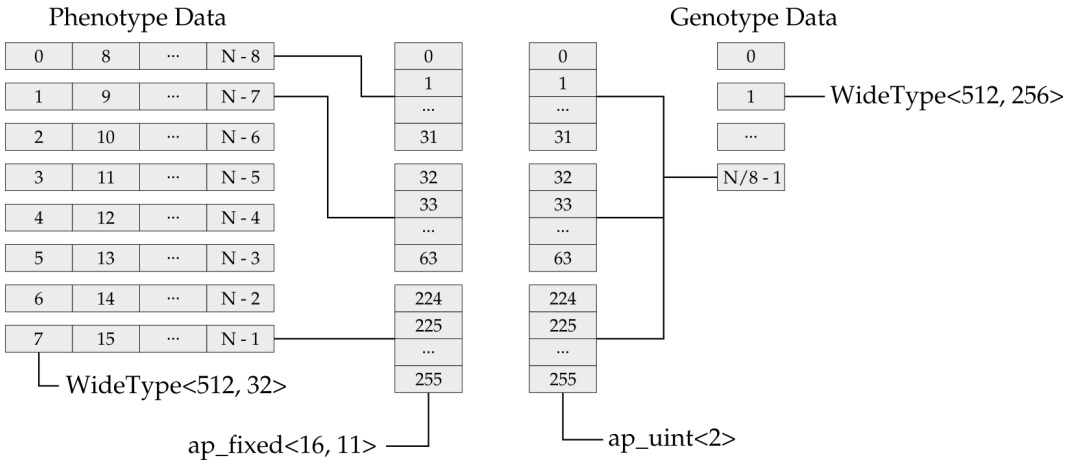


Figure 3.9: Illustration of how `cyclic` array partitioning is used on the phenotype buffer to allow a 512-bit block of genotype data to be processed every clock cycle.

3.2.5 Task-Level Parallelism

Vitis HLS supports task-level parallelism (where an architecture is generated which can execute a set of tasks or functions concurrently) using the `dataflow` optimisation. Figure 3.10 shows how task-level parallelism can improve kernel throughput and lower the overall latency.

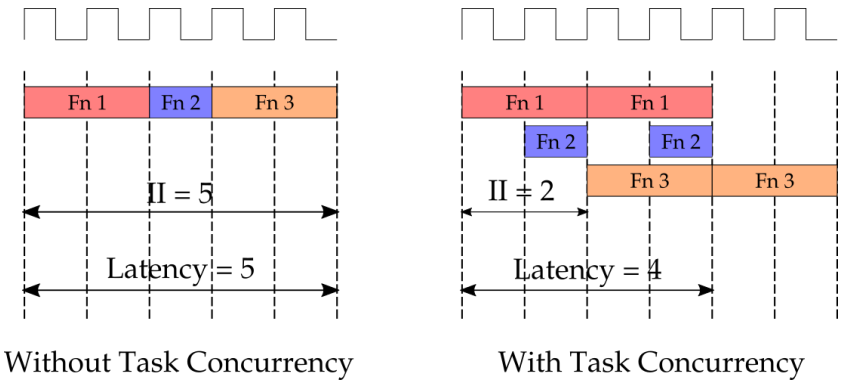


Figure 3.10: Demonstration of how task-level pipelining can be used to improve the efficacy of a Vitis HLS kernel by concurrently executing a set of functions.

When Vitis HLS is instructed to perform the `dataflow` optimisation on a set of functions, it creates channels (which model the flow of data through the functions) to buffer the output of each function, therefore allowing each function to execute independently. The throughput is then limited only by the availability of data at the input and output buffers, so task execution can be interleaved at the expense of higher resource usage (as more BRAMs are required to create the various buffers). Vitis HLS synthesises FIFOs to

handle scalar channels, while ping-pong buffers are synthesised to handle non-scalar channels.

Figure 3.11 shows the architecture generated by the `dataflow` optimisation, and Figure 3.12 demonstrates how Vitis HLS generates FIFOs (represented by the green arrows) and ping-pong buffers (represented by the brown arrows) to realise the `dataflow` architecture in the MVM kernel.

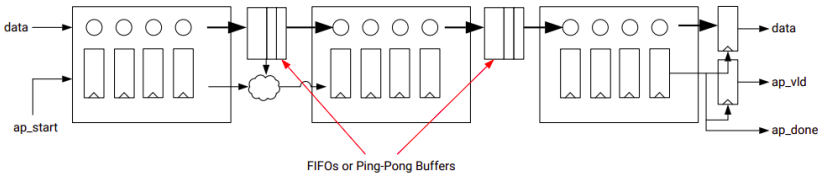


Figure 3.11: Architecture generated by the `dataflow` optimisation [7].

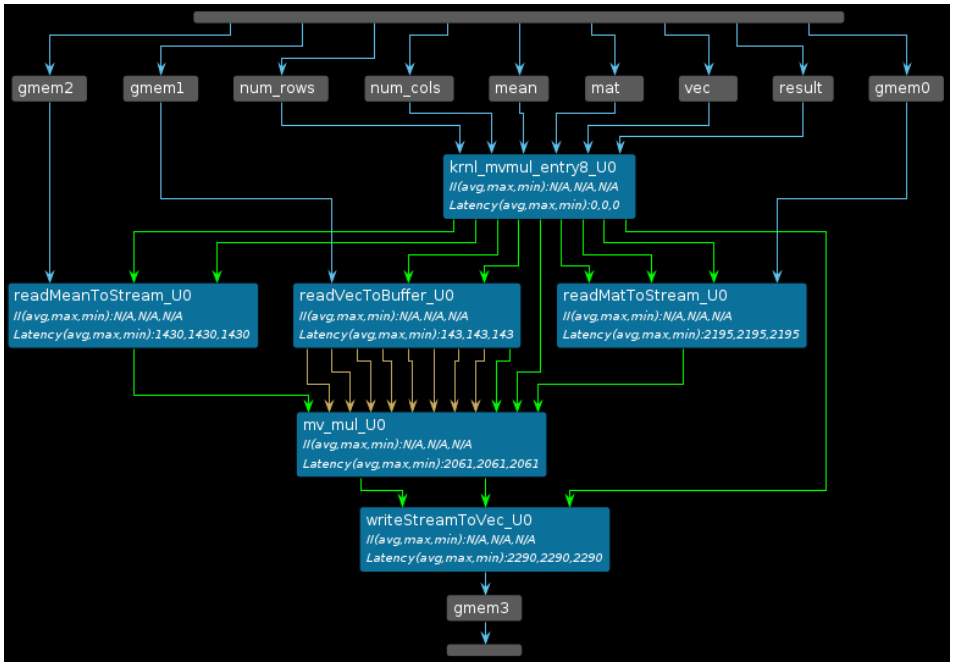


Figure 3.12: Vitis HLS visualisation of the MVM kernel's dataflow architecture showing how data moves through the kernel. Functions are represented as blue blocks and functions in the same row are executed concurrently. FIFOs (which buffer scalar data) are represented as green arrows and ping-pong buffers (which buffer arrays) are represented as brown arrows, while data transfer to/from global memory is represented by blue arrows.

3.2.6 Kernel Interfaces

The kernel interfaces/arguments are exhibited by Listing 3.4. All function arguments, together with a mandatory `return` port, are grouped into one `s_axilite` (AXI4-Lite) interface (called the `control` interface) to allow the host application to communicate with the kernel. The memory-mapped ports are bundled into the `control` interface so that the host application can pass the global memory address offset of each array to the kernel. The Vitis HLS compiler automatically assigns register addresses for each port in the `control` interface and generates an interrupt port to allow the kernel to notify the host application when processing is complete.

```
#define DATA_WIDTH 512

void krnl_mvmul(
    const unsigned int num_rows,
    const unsigned int num_cols,
    const ap_int<DATA_WIDTH> *mean,
    const ap_int<DATA_WIDTH> *mat,
    const ap_int<DATA_WIDTH> *vec,
    ap_int<DATA_WIDTH> *result)
{
#pragma HLS INTERFACE m_axi port = mat    offset = slave    bundle = gmem0
#pragma HLS INTERFACE m_axi port = vec    offset = slave    bundle = gmem1
#pragma HLS INTERFACE m_axi port = mean   offset = slave    bundle = gmem2
#pragma HLS INTERFACE m_axi port = result offset = slave    bundle = gmem3

#pragma HLS INTERFACE s_axilite port = num_rows  bundle = control
#pragma HLS INTERFACE s_axilite port = num_cols  bundle = control
#pragma HLS INTERFACE s_axilite port = mean      bundle = control
#pragma HLS INTERFACE s_axilite port = mat       bundle = control
#pragma HLS INTERFACE s_axilite port = vec       bundle = control
#pragma HLS INTERFACE s_axilite port = result    bundle = control
#pragma HLS INTERFACE s_axilite port = return    bundle = control
}
```

Listing 3.4: Top-level function of the MVM kernel showing the global memory interfaces.

All pointer arguments (specified as `m_axi` or AXI4 master interfaces) are defined as `ap_int<512>` pointers to use the full width of the memory bus. The memory-mapped arguments are summarised in Table 3.5.

Table 3.5: Summary of the memory-mapped interfaces used to transfer data between the MVM kernel and global memory.

Variable	Internal type	Number of elements	Size (B)
`mat`	`ap_uint<2>`	`num_rows` × `num_cols`	`num_rows` × `num_cols`/4
`mean`	`ap_fixed<16,2>`	`num_rows`	`num_rows` × 2
`vec`	`ap_fixed<16,11>`	`num_cols`	`num_cols` × 2
`result`	`ap_fixed<32,20>`	`num_rows`	`num_rows` × 4

3.2.7 Measuring the Effect of the Design Optimisations

This section demonstrates the effect of different design optimisations on area and performance by comparing five different simulated MVM kernel designs. In order to show the cumulative effect of the different optimisations, each new kernel design includes all previously implemented optimisations. Vitis HLS was used to simulate each kernel design running on the VU9P FPGA. Vitis HLS cannot simulate the effect of global memory bandwidth on kernel performance (i.e. it assumes that data is always available at the kernel interfaces), so a global memory bandwidth of 8.5 GB/s is used to estimate the effect of the different optimisations on data transfer latency.

A small data set consisting of 256 SNPs sampled from 2,018 individuals (which requires 516,608 multiplications and 516,352 additions) was used to measure the performance of the different kernel designs. A small data set was used to reduce simulation runtime, but performance can be expected to scale linearly with increasing SNPs and individuals.

In order to carry out a fair comparison, each kernel design was configured to support a maximum sample size of 262,144.

Design 1 — A minimally parallelised kernel which implements the MVM computation using single-precision floating point types. As the genotype data is not processed in a compressed format, the kernel is not required to perform the $\mathbf{x} = \mathbf{a} - \bar{\mathbf{a}}$ calculation. The kernel has a fully pipelined dot product loop ($t_{II} = 1$) which initiates one floating-point multiply-accumulate operation every clock cycle. Data is transferred directly via the kernel's global memory interfaces (i.e. streams are not used to transfer data within the kernel and the phenotype data is not buffered in local memory).

Design 2 — The floating-point types in Design 1 are replaced with signed fixed-point types. Genotype data is represented with `ap_fixed<16,2>`, phenotype data with `ap_fixed<16,11>`, and the dot product output with `ap_fixed<32,20>`.

Design 3 — The `dataflow` optimisation is used to overlap data reading/writing with data processing. The phenotype data is buffered in the kernel's internal memory and data streams (which are required by the dataflow optimisation) are used to transfer data within the kernel.

Design 4 — Parallelism is increased by using the full width of the global memory interface. The `WideType` class (defined in Section 3.2.2) is used to process data in blocks of 512 bits. The kernel has a fully pipelined dot product loop ($t_{II} = 1$) which initiates 32 multiply-accumulate operations every clock cycle.

Design 5 — The final kernel design further increases parallelism by processing compressed genotype data (as described in Section 3.2.1), which allows 256 multiply-accumulate operations to be completed every clock cycle.

Table 3.6 compares the kernel designs in terms of the respective hardware utilisation, while Table 3.7 compares the designs in terms of performance. To generate timing estimates for the performance comparison, the kernel operating frequency is set at 250 MHz, while the global memory bandwidth is set at 8.5 GB/s.

Table 3.6: Comparison of the hardware utilisation of the different kernel designs showing that, although the hardware utilisation increases as the kernel parallelism increases, the VU9P can easily support multiple instances of the Design 5 kernel.

	BRAM	DSPs	LUTs	FFs
VU9P Resources	2,160	6,840	1,182,000	2,364,000
Design 1	6	3	4,367	3,096
Design 2	6	1	3,815	2,577
Design 3	14	1	5,141	3,948
Design 4	218	1	13,751	12,369
Design 5	120	257	53,359	41,676

It can be seen that the use of fixed-point types in Design 2 leads to a reduction in area (~15-20%) compared to the use of floating-point types in Design 1, but does not lead to a significant reduction in latency. This is due to the fully pipelined dot product loop which hides the increased latency of floating-point multiply-accumulate operations. The use of a 16-bit type to represent the genotype data has, however, halved the amount of

Table 3.7: Comparison of the performance of the different kernel designs in terms of kernel latency and data transfer latency via global memory. Timing estimates use an operating frequency of 250 MHz and a global memory bandwidth of 8.5 GB/s.

	Kernel Latency (cycles)	Kernel Latency (µs)	Data Transfer Size (B)	Data Transfer Latency (µs)
Design 1	539,220	2,157	2,075,528	244
Design 2	537,933	2,152	1,038,276	122
Design 3	520,322	2,081	1,038,276	122
Design 4	16,621	66	1,053,696	124
Design 5	2,291	9	134,724	16

data transferred via global memory.

The performance of Design 3 indicates that task-level pipelining leads to a small reduction in latency (at the expense of increased area caused by the synthesis of additional FIFO and ping-pong buffers) due to the overlap of data reading/writing with data processing.

Widening the width of the global memory interfaces to 512 bits and using the `WideType` class to process 512-bit blocks of data concurrently provides the most significant performance improvement in terms of kernel latency. The latency of Design 4 is ~32 times lower than the latency of Design 3, which was expected as Design 4 performs 32 multiply-accumulate operations every clock cycle.

Compression of the genotype data allows 256 multiply-accumulate operations to be performed every clock cycle and significantly reduces the volume of data transferred via global memory. As expected, the latency of Design 5 is ~256 times lower than that of Design 3, and ~8 times lower than the latency of Design 4. Compression of the genotype data, therefore, provides the most significant performance improvement in terms of global memory transfer latency.

One advantage of Design 4 over Design 5 is the reduced usage of DSP blocks. As Design 4 does not require the MVM kernel to compute $\mathbf{x} = \mathbf{a} - \bar{\mathbf{a}}$, the HLS compiler performs the multiply-accumulate operations using CLBs rather than DSP blocks, but a consequence of handling uncompressed genotype data is an increase in BRAM usage — Design 4 requires almost double the amount of BRAMs compared to Design 5.

It is immediately apparent that increasing the parallelism of the MVM kernel comes with the trade-off of increased area. As shown by Table 3.6, the hardware resources of the Xilinx VU9P FPGA can easily support multiple instances of the Design 5 kernel, but smaller FPGAs could struggle to support the same level of parallelism. If a particular FPGA cannot support multiple instances of the Design 5 kernel, the number of kernel instances could be reduced, or Design 4, which provides a significant level of parallelism compared to Design 3, could be used.

3.2.8 Topological Optimisations

This section describes how the architecture of the Vitis target platform for the AWS F1 instance (shown in Figure 3.13) is exploited to further increase the parallelism of the accelerator.

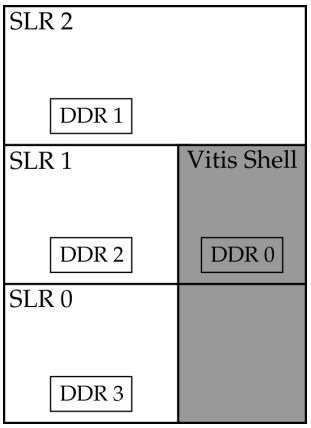


Figure 3.13: Architecture of the AWS F1 FPGA target platform.

Multiple Kernels

Multiple MVM kernels are instantiated on the FPGA to allow the accelerator to process different blocks of data in parallel. The parallelism is limited by the PCIe/global memory bandwidth and the available hardware resources. It was found that four MVM kernels (with each kernel connected to a different global memory bank) maximised the global memory bandwidth, and additional kernels did not result in an increase in performance. See Section 4.2 for an analysis of the effect of the number of kernels on the performance of the accelerator.

Utilising Multiple Memory Banks

Each of the four global memory banks is connected to one of the four MVM kernels in order to maximise the system memory bandwidth. As illustrated by Figure 3.14, all data transferred to a kernel via global memory is transferred using the same memory bank. This configuration minimises connections between SLRs which helps to reduce latency, but can slightly impact kernel performance as the kernel is required to wait for data due to the fact that different kernel ports cannot concurrently access global memory. Section 4.1 demonstrates the effect of transferring data between a kernel and global memory using a single memory bank.

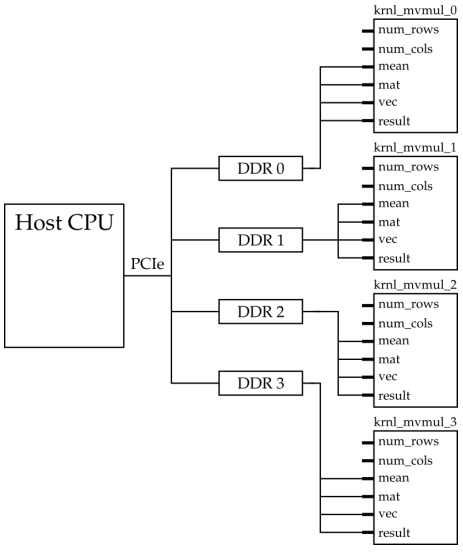


Figure 3.14: System architecture showing the configuration of the global memory banks.

Kernel SLR Assignment

Xilinx recommends connecting memory banks to kernels placed in the same SLR in order to minimise latency. Furthermore, distributing a design over the SLRs helps to minimise congestion. For this reason, `krnl_mvmul_3` is placed in SLR 0, `krnl_mvmul_0` and `krnl_mvmul_2` are placed in SLR 1 and `krnl_mvmul_1` is placed in SLR 2. Constraining the placement of the kernels resulted in an increase in router congestion in SLR 1 which contains two kernels and part of the Vitis shell, but the impact of router congestion was mitigated by the use of the Vivado build directives summarised in Table 3.4.

3.3 Host Application Design

This section describes the design of the C++ host application. The host application is required to efficiently perform various tasks including:

- Reading the GWAS data from input files and generating the data format expected by the MVM kernels
- Data blocking and buffer management
- Setting up the FPGA and executing kernels with OpenCL
- Implementing the permutation testing algorithms
- Recording the results

The host application is compiled with the GCC compiler (Red Hat 4.8.5-44) using optimisation level 3. The design of the host application focused primarily on keeping the FPGA kernels occupied in order to minimise FPGA idle time.

3.3.1 Data Preprocessing

GWAS data is read from PLINK-formatted data files and preprocessed before it is transmitted to the FPGA kernels. The accelerator supports PLINK files as they are an industry standard method of storing and handling SNP data which ensures that the accelerator is more accessible to users.

The data preprocessing stage of the permutation testing algorithm has multiple functions, namely:

- Performing the covariate adjustment on the phenotype data if required
- Converting the PLINK genotype storage format to the additive genotypic model encoding and generating the data format expected by the MVM kernels
- Calculating the mean and standard deviation of each SNP vector
- Converting floating-point data to the corresponding fixed-point `ap_[u]fixed` type

To simplify the development of the data preprocessing algorithm, it was assumed that all data supplied to the host application is to be used for the GWAS i.e. no filtering is performed on the input data.

Input Data File Types

The host application must be supplied with three (or four, if covariates are included) PLINK-generated input files, which are identified by file extension.

A **.bim** file is a text file which describes the genetic variants included in the GWAS. The

.bim file contains each variant's position in the genome, the variant descriptor (which is used when generating the result files) and the two alleles of the variant. The number of variants in this file must correspond to the genotype data stored in the .bed file.

A **.bed** file is a binary file which stores compressed SNP data. The first three bytes contain a magic number used to identify the file type, and genotype data is then stored as a sequence of V blocks of N/4 (rounded up) bytes, where V is the number of variants and N is the number of samples. The first block corresponds to the first SNP in the .bim file, etc. The following two-bit genotype encodings are used:

- 00 — Homozygous for the first allele (usually the alternate allele) in the .bim file
- 01 — Missing data
- 10 — Heterozygous
- 11 — Homozygous for the second allele (usually the reference allele) in the .bim file

The **.fam** file stores the phenotype data for each individual/sample, as well as the individual's sex and identifying information. The number of phenotypes in the .fam must correspond to the data in the .bed file.

A **covariate** file is an optional text file containing covariate information for each individual in the study.

Phenotype Preprocessing

A Python3 script was developed to perform the covariate/mean adjustment on the phenotype data using the NumPy, pandas and statsmodels packages. The script reads the phenotypes from the .fam file and writes the adjusted phenotypes (either $\mathbf{y} = C(C^TC)^{-1}C^T\mathbf{p}$ or $\mathbf{y} = \mathbf{p} - \bar{\mathbf{p}}$, depending on whether a covariate data file is provided) to a file which is read by the host application. The host application then converts the double-valued phenotypes to `ap_fixed<16,11>`.

Python was used for phenotype preprocessing in order to avoid the implementation of multiple linear regression in the C++ host application. Phenotype preprocessing has little influence on the overall runtime of the permutation testing algorithm as it only happens once and the Python packages used are well optimised and extensively tested.

Genotype Preprocessing

An OpenMP-accelerated (v3.1) algorithm was developed to handle genotype preprocessing. This stage of the permutation testing algorithm had to be accelerated as it was found to influence the overall runtime, particularly when the number of SNPs is large and the number of permutations is low ($p < 1000$). OpenMP is a simple API which parallelises loops by spawning multiple threads based on the number of available processor cores and running each loop iteration in a separate thread.

The genotype preprocessing algorithm performs the following operations for each variant in the .bed file:

- Converts each byte from the PLINK encoding to the genotypic model encoding
- Determines the number of non-missing genotypes for each variant
- Calculates the mean of each SNP vector and converts it to `ap_fixed<16,2>`
- Calculates the standard deviation of each SNP vector

The algorithm uses a nested loop pair to process a .bed file. The inner loop iterates through each byte of a variant data block and converts the PLINK-encoded genotype data to a representation which corresponds to the additive genotypic model (as summarised in Table 3.8). The inner loop also sums the converted SNP values and the number of non-missing genotypes to calculate the mean of each SNP vector. The outer loop (which is parallelised with OpenMP) iterates through each variant block in the .bed file, converts the double-valued mean of each SNP vector to `ap_fixed<16,2>` and calculates the standard deviation of each SNP vector.

Table 3.8: .bed file binary data conversion.

Genotype	.bed File Encoding	Additive Model Encoding
Homozygous recessive	00	10
Missing data	01	11
Heterozygous	10	01
Homozygous dominant	11	00

To show the effect of parallelisation on the performance of the genotype preprocessing algorithm, the runtime was measured with and without OpenMP-facilitated parallelisation using a dataset consisting of $1,898,052$ SNPs sampled from $2,018$ individuals. On an 8-core Intel i5-8265U CPU, the runtime of the unparallelised version of the algorithm was measured at 24.15 seconds, while the parallelised implementation yielded a runtime of 5.26 seconds. OpenMP-aided parallelisation of the genotype preprocessing algorithm, therefore, provides a significant 4.6-fold speedup.

3.3.2 Data Blocking

The host application takes advantage of the parallelism of the MVM accelerator by blocking the GWAS dataset so that different blocks of data can be processed concurrently by each of the four MVM kernels as illustrated by Figure 3.15. PCIe transfer efficacy is dependent on the size of the data buffer being transferred (see Section 4.3 for an investigation into the effect of the buffer size on PCIe throughput), so the host application blocks the genotype matrix based on the optimal buffer size, which determines the number of kernel invocations as well as the number of dot products computed by each MVM kernel i.e.

$$n_{blocks} = \frac{n_{snps}}{n_{snps/kernel}}, \quad n_{snps/kernel} = \frac{\text{Genotype buffer size}}{\text{Variant block size}} \tag{3.3}$$

The blocking algorithm was complicated by a requirement that all buffers passed to the FPGA are 4 KB aligned as the Xilinx runtime (XRT) driver allocates memory along 4 KB boundaries for internal memory management. If a buffer is not aligned on a 4 KB boundary, XRT performs a `memcpy` to achieve the correct alignment. As unnecessary `memcpy`s reduce the data transfer efficiency, the blocking algorithm was designed to ensure that all blocks of data transferred to/from the FPGA are correctly aligned. The algorithm was also designed to ensure that all data buffers can be transferred in blocks of 512 bits. Figure 3.15 illustrates how the blocking algorithm distributes the GWAS data among the MVM kernels in order to maximise concurrency.

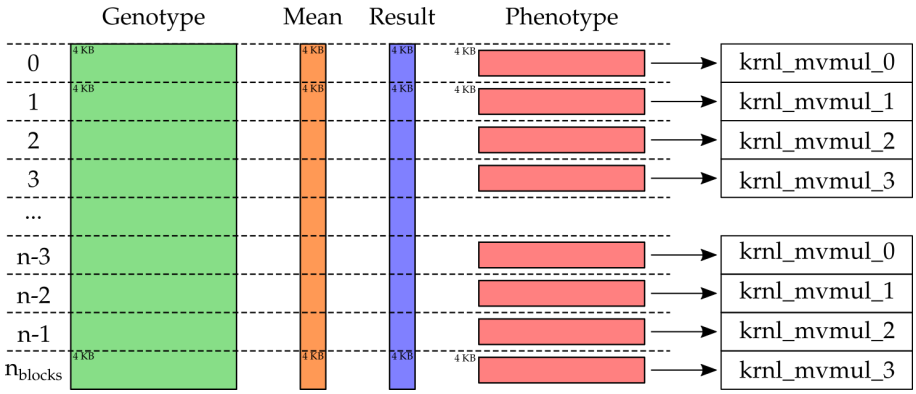


Figure 3.15: Illustration of the data blocking scheme implemented by the host application.

3.3.3 Kernel Execution with OpenCL

The host application uses OpenCL (v1.2) API calls to execute Vitis kernels instantiated on the FPGA and to manage data transfer via global memory. This section describes the OpenCL environment and kernel execution model.

Initialisation of the OpenCL Environment

To setup the OpenCL environment, the host application first searches for a Xilinx **platform** connected to the host processor. If a Xilinx platform is detected, the host application searches the platform for Xilinx FPGA **devices** and creates an OpenCL **context** (which is used to manage and execute kernels) for each detected device. An OpenCL **program** object is then created for each device context and the FPGA bitstream, which stores the circuit configuration data, is loaded into the program object to configure the device. Finally, an OpenCL **command queue** is created for the device to allow OpenCL/XRT to schedule and execute tasks on the FPGA kernels. The host application uses an out-of-order command queue which allows the XRT scheduler to dispatch kernel tasks in any order so as to maximise concurrency.

A class (called `OclApi`) was adapted from Xilinx's open-source Vitis Accel Examples Repository to initialise the OpenCL environment and configure the FPGA hardware (using a bitstream provided to the host application via a command line argument) in addition to encapsulating the OpenCL objects.

Kernel Execution

Once the OpenCL environment is initialised, the host application can communicate with and execute tasks on the FPGA kernels. OpenCL API calls are used to:

- Initialise the kernels
- Transfer data to/from the kernels
- Execute kernel tasks
- Synchronise kernel execution based on OpenCL events

OpenCL `kernel` objects are instantiated in the host application in order to access each of the four MVM kernels defined in the FPGA bitstream. Unlike the invocation of C/C++ functions, OpenCL requires that kernel arguments are set before the kernel is executed. Allowable kernel arguments are write-only scalar variables and read/write OpenCL memory buffers. An OpenCL memory buffer is a pointer to an OpenCL memory object

created with the context associated with a kernel object. Listing 3.5 demonstrates how the host application sets the kernel arguments for one invocation of an MVM kernel. Note how the order of the kernel arguments corresponds to the kernel interfaces defined in Section 3.2.6.

```
kernel.setArg(0, (const unsigned int)n_rows);
kernel.setArg(1, (const unsigned int)n_cols);
kernel.setArg(2, mean_buf);
kernel.setArg(3, mat_buf);
kernel.setArg(4, pheno_buf);
kernel.setArg(5, result_buf);
```

Listing 3.5: Setting the kernel arguments for one invocation of an MVM kernel.

All OpenCL API calls used to control kernel execution are asynchronous (i.e. they immediately return after the command has been queued on the command queue), so events are used to resolve dependencies between kernel commands. Event-based synchronisation allows the host application to perform other work while OpenCL/XRT manages the command queue. A class (named `OclTask`) was adapted from Xilinx's Vitis Accel Examples Repository to perform one invocation of an MVM kernel using event-triggered OpenCL API calls. An `OclTask` has three stages:

- Writing data to the kernel
- Triggering kernel execution
- Reading the result from the kernel

Each stage is placed on the command queue and event-based synchronisation is used to ensure that the host application does not try to read data that has not yet been computed. To ensure that kernels are idle for as little time as possible, an `OclTask` can triggered by an event fired by the completion of a previous `OclTask`. Listing 3.6 shows how the `OclTask` class facilitates event-based kernel execution with the `run` method.

One `OclTask` is created for each block of the genotype matrix. The host application iterates through the data blocks, sets the corresponding kernel arguments and queues the `OclTask` on the command queue. Figure 3.16 demonstrates how the `OclTask` class facilitates synchronisation using the events fired by OpenCL on the completion of each stage of an `OclTask`. Once all kernel tasks have been queued, the host application calls

```
api.queue.finish();
```

which blocks execution of the host application until the command queue is empty, indicating that all `OclTask`s are complete.

3.3.4 Permutation Testing Algorithms

The two permutation testing algorithms were designed with a view towards minimising the overall runtime by ensuring that the FPGA kernels are idle for as little time as possible.

MaxT Permutation Testing

For each permutation iteration of the maxT algorithm, the host application is required to:

- Receive the dot product vector from the FPGA

```
class OclTask
{
    std::vector<cl::Event> write_data_event = std::vector<cl::Event>(1);
    std::vector<cl::Event> enqueue_event = std::vector<cl::Event>(1);
    std::vector<cl::Event> read_data_event = std::vector<cl::Event>(1);

public:
    void run(
        OclApi &api,
        cl::Kernel kernel,
        ocl_task_buffers_t task_buffers,
        std::vector<cl::Event> *prev_event = nullptr)
    {
        // Copy data to the FPGA
        api.queue.enqueueMigrateMemObjects({task_buffers.mat_buf,
                                            task_buffers.pheno_buf,
                                            task_buffers.mean_buf},
                                            0 /* from host */,
                                            prev_event, &write_data_event[0]);

        // Enqueue kernel execution
        api.queue.enqueueTask(kernel, &write_data_event, &enqueue_event[0]));

        // Copy result from FPGA
        api.queue.enqueueMigrateMemObjects({task_buffers.out_buf},
                                            CL_MIGRATE_MEM_OBJECT_HOST,
                                            &enqueue_event, &read_data_event[0]);
    }
};
```

Listing 3.6: The `OclTask` class used by the host application to manage event-based execution of OpenCL `kernel` objects.

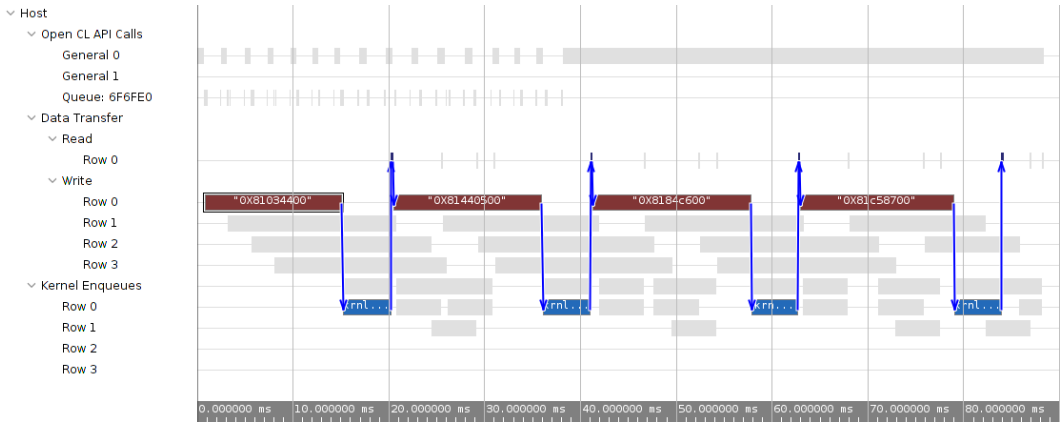


Figure 3.16: Vitis Analyser timeline showing the event-based kernel execution using OpenCL. The blue arrows represent dependencies between tasks. It can be seen that a new `OclTask` is triggered on a kernel after the host application has read the FPGA-computed result buffer from global memory.

- Calculate the F-statistic for each SNP
- Determine and record the maximum F-statistic

This work is done in a detached thread (using the `std::Thread` class) so that the host application can start the next permutation while computing the results of the previous iteration.

Once all permutations are complete, the host application ranks each observed test statistic against the recorded maximum F-statistics to determine the adjusted p-value for each SNP.

Adaptive Permutation Testing

The adaptive permutation testing algorithm is more complex than the maxT algorithm as SNPs are removed from adaptive permutation testing when they do not appear to be significant, which requires the FPGA data buffers to be regenerated whenever SNPs are dropped. For each permutation iteration of the adaptive algorithm, the host application is required to:

- Receive the dot product vector from the FPGA
- Calculate the permuted F-statistic for each SNP
- Determine and record whether the permuted test statistic is greater than the observed test statistic

As in the maxT implementation, this work is done in a detached thread to maximise the throughput of the accelerator.

Unlike the maxT algorithm, which has a runtime which is affected only by the number of permutations, the adaptive algorithm has a number of parameters, along with the number of permutations, which affect the runtime, namely:

- The SNP dropping interval
- The rate at which the dropping interval increases
- The minimum number of permutations for each SNP

The minimum permutations per SNP is a user-defined parameter. The SNP dropping interval is initially equal to the minimum permutations per SNP and increases by 20% every 5 drops. At each SNP dropping interval, the host application regenerates the data buffers to remove the SNPs which have been flagged as insignificant and re-runs the data blocking algorithm on the regenerated data.

Once enough SNPs have been dropped so that the SNP data can fit into one buffer, the host application transitions to performing one permutation per MVM kernel (as demonstrated by Figure 3.17) rather than splitting one permutation amongst the kernels. When handling small genotype buffers, the runtime of each kernel invocation is reduced to the point where OpenCL/XRT scheduler overhead begins to have an impact on performance. It was found that triggering one `OclTask` for each of the four kernels before calling `api.queue.finish()` reduced the overall performance of the permutation algorithm due to OpenCL/XRT scheduler overhead, so the host application places multiple `OclTasks` on the command queue for each kernel. This requires the FPGA data buffers to be replicated based on the number of `OclTasks`. It was found, through trial and error, that triggering 64 `OclTasks` per kernel before calling `api.queue.finish()` minimises the runtime of the adaptive permutation testing algorithm, although it is not known if this result is specific to the test dataset or if it can be generically applied to any dataset. Further analysis of the adaptive permutation testing algorithm could aid in the optimisation of the various parameters.

During this phase of the algorithm, the host application adaptively sets the size of the genotype data buffers to ensure that the accelerator throughput is maximised. If the SNP data does not fill the buffer, the buffer size is reduced. The host application continues to drop SNPs and regenerate the data buffers during this phase of the algorithm, but the SNP dropping interval increases more rapidly as only significant SNPs remain and fewer SNPs are removed at each interval.

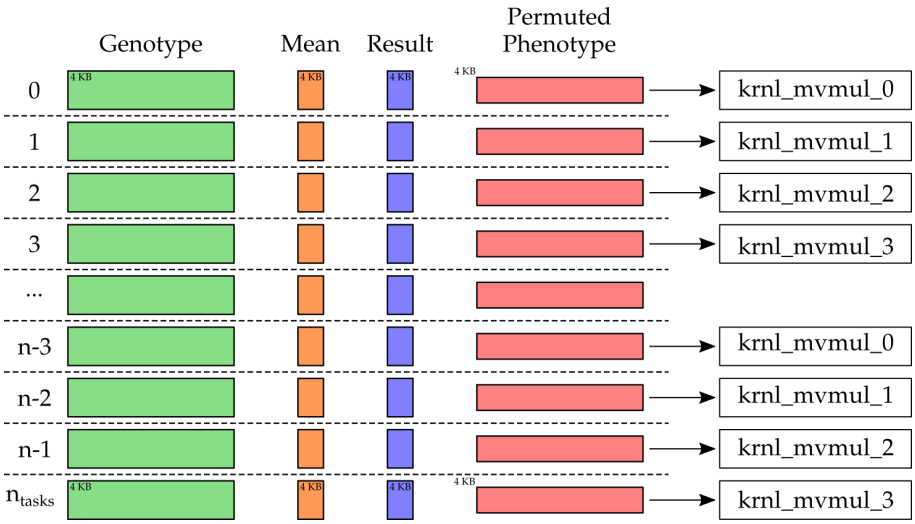


Figure 3.17: Buffer management for the later stages of the adaptive permutation testing algorithm when the SNP data can fit into one buffer. Note that each phenotype buffer contains a different permutation of the phenotype vector as each kernel is accelerating one permutation.

3.4 Summary

In summary, a GWAS permutation accelerator was designed for an AWS EC2 f1.2xlarge instance consisting of a Xilinx Virtex UltraScale+ VU9P FPGA together with a host CPU. As matrix vector multiplication is the most computationally expensive part of the permutation testing algorithm, the FPGA was tasked only with accelerating the MVM computation while the host CPU was used to: handle data preprocessing, manage the FPGA execution using OpenCL and perform both the maxT and adaptive permutation testing algorithms.

An MVM kernel was designed (using Vitis HLS) to take advantage of the compressed nature of genotype data which can be represented with a 2-bit data type. Processing compressed genotype data allows an MVM kernel to perform 256 concurrent multiply accumulate operations every clock cycle. To take advantage of the parallel architecture of the VU9P FPGA and the f1.2xlarge instance, an FPGA design consisting of four identical MVM kernels was built using Xilinx Vivado 2020.2.

The host application was designed in C++ and uses OpenCL to communicate with the FPGA and transfer data between the host CPU and the FPGA kernels. A data blocking algorithm was designed to break up a GWAS dataset and distribute the MVM computation between the four kernels so as to maximise the throughput of the accelerator. The host application uses multithreading in order to allow new permutations to be started while the results of previous permutations are being processed.

4 Design Evaluation and Optimisation

This chapter presents the optimisation exercises that informed the design process so as to optimise the performance of the accelerator. Section 4.1 evaluates the performance of a single MVM kernel running on an f1.2xlarge instance using OpenCL profiling data and the debug signals generated by the kernel. Section 4.2 addresses why four MVM kernels is the optimal configuration that maximises both the global memory bandwidth and the kernel concurrency. Section 4.3 demonstrates how the size of the buffers used to transfer data between the host CPU and the FPGA kernels affect the global memory bandwidth and the accelerator throughput. Section 4.4 shows how the size of the GWAS dataset affects the performance of the accelerator. Finally, Section 4.5 analyses the FPGA hardware utilisation of the four kernel design and demonstrates how the hardware utilisation is affected by different designs.

4.1 Kernel Evaluation

A simulated dataset of 2,048 SNPs sampled from 2,018 individuals was used to evaluate the performance of the MVM kernel running on an f1.2xlarge instance. A small dataset was used to allow for detailed kernel profiling but the use of a small dataset results in suboptimal global memory bandwidth due to the size of the buffer (1 MB) transferred via global memory (see Section 4.3 for the investigation into the optimal buffer size). Furthermore, the generation and sampling of profiling data has an adverse effect on the performance of both the host application and the FPGA kernel, so the final build (which does not generate or capture profiling data) can be expected to perform better.

Kernel profiling shows that the average kernel data transfer rate to/from global memory is 12.7 GB/s, which is higher than the host-to-FPGA global memory bandwidth (around 8.5 GB/s) measured by the `xbutil dmatest` command as exhibited by Listing 3.1. This indicates that further optimisation of the kernel read/write logic will not improve the overall accelerator performance — additional gains in performance must be achieved via improvement of the global memory bandwidth which will require optimisation of the AWS F1 Xilinx target platform and/or of the XRT drivers.

Figure 4.1 shows a Vitis Analyser timeline which displays both host and kernel events for one invocation of the MVM kernel. It can be seen that the FPGA execution time (which is represented by the `Compute Unit "krnl_mvmul_0_1"` trace) is a small proportion (about 6.5%) of the overall runtime. This underscores the need for multiple kernels as well as the need for optimisation of the OpenCL part of the accelerator in order to keep the FPGA supplied with data so that the FPGA idle time can be minimised.

Figure 4.2, which displays a Vitis Analyser timeline of only the FPGA part of the kernel execution, shows that the MVM kernel is never idle when the FPGA is supplied with data. No major stalls can be observed in the `m_axi_gmem3_DDR[0] (mat)` trace which indicates that new genotype data is constantly being processed by the MVM kernel.

The `m_axi_gmem3_DDR[0] (result)` trace in Figure 4.2 demonstrates the occurrence of burst data transfer as discussed in Section 3.2.2. The total size of the kernel output is $2,048 \times 4 = 8,192$ bytes, so each write is $8,192/8 = 1,024$ bytes which is

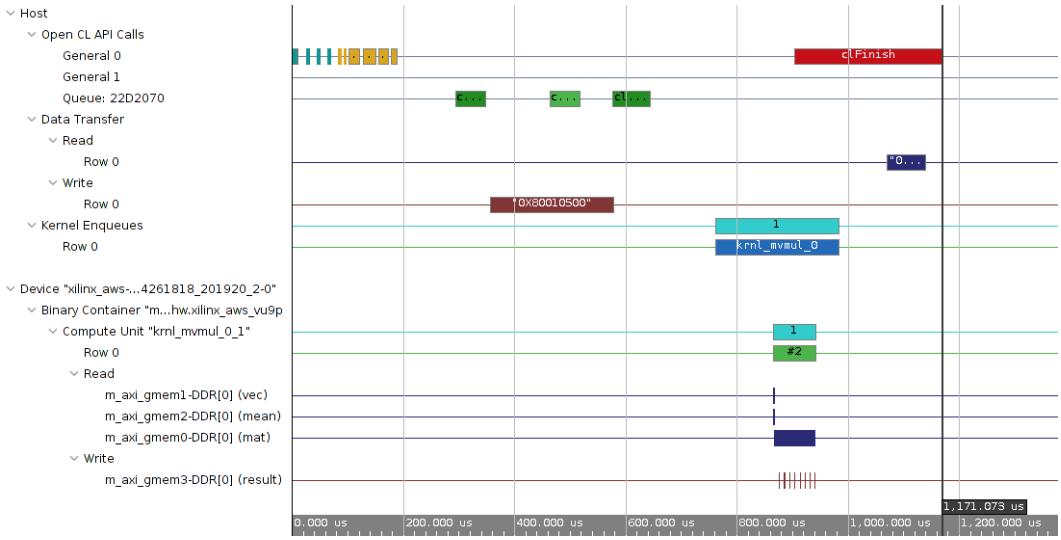


Figure 4.1: Vitis Analyser timeline showing that the FPGA execution time (which is represented by the `Compute Unit "krnl_mvmul_0_1"` trace) is a fraction of the total runtime.

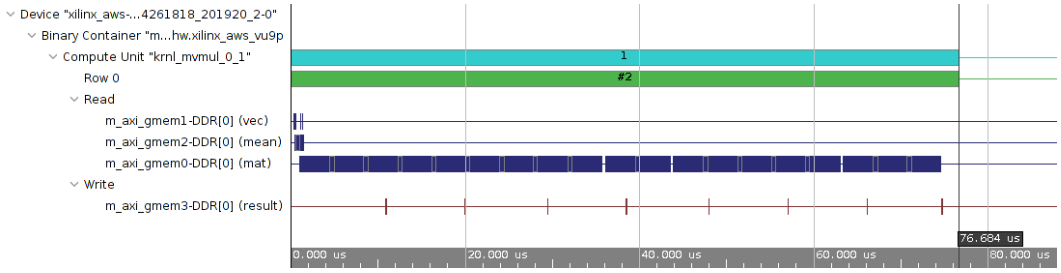


Figure 4.2: Vitis Analyser timeline demonstrating that the MVM kernel is never idle when data is available.

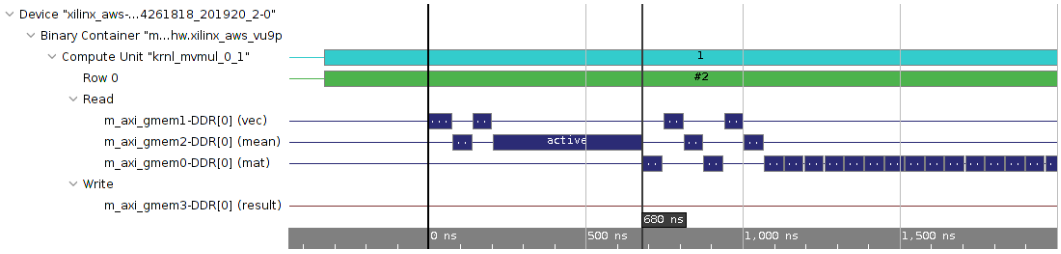


Figure 4.3: Vitis Analyser timeline showing the effects of data contention in the MVM kernel. As all data is transferred via the same global memory bank, a small delay occurs before the genotype data is read due to the fact that the first elements of the phenotype vector and the genotype mean vector must be buffered in the kernel's internal memory before the MVM calculation can begin.

equivalent to sixteen 512-bit words per memory access. Increasing the burst size will not have an effect on kernel performance due to the PCIe bandwidth bottleneck but, if the host-to-FPGA memory bandwidth is improved, it may be necessary to increase the burst size in order to maximise the bandwidth.

Figure 4.3, which magnifies the initial part of MVM kernel execution timeline, shows that a small delay occurs before the kernel begins the MVM calculation (i.e. before the kernel begins reading the genotype data). This is due to the fact that all data is transferred

via the same global memory bank and the initial elements of both the genotype mean vector and the phenotype vector must be buffered in the kernel's internal memory before the MVM calculation can begin. Connecting all of the kernel ports to the same global memory bank results in data contention when multiple ports attempt to access the memory bank simultaneously, but this is an acceptable trade-off to allow each of the four MVM kernels to be connected to a different memory bank. Moreover, the data contention events are minimal as the genotype matrix port is the only port in use for the majority of the kernel runtime as exhibited by Figure 4.2.

4.2 Optimising the Number of MVM Kernels

An important optimisation exercise was determining the optimal number of MVM kernels to maximise the global memory bandwidth and the accelerator throughput. As discussed in Section 3.2.8, the final FPGA design consists of four identical MVM kernels with each kernel connected to one of the four global memory banks. The primary aim of this optimisation exercise was to determine whether more than one kernel could be connected to a global memory bank to improve the accelerator throughput. An FPGA design consisting of eight MVM kernels (with two kernels connected to each global memory bank) was compiled using Xilinx Vitis 2020.1 so as to test the effect of different configurations of MVM kernels on the accelerator throughput. The eight kernel design was compiled without SLR constraints as high levels of router congestion prevented Vitis from generating a design which could run at 250 MHz when the kernels were constrained to SLRs. The size of the phenotype buffer was also reduced (from 262,144 elements to 131,072 elements) to limit the amount of congestion.

A full-sized GWAS dataset of 1.89 million SNPs sampled from 2,018 individuals was used for this analysis. The dataset was blocked based on the number of kernels so that each kernel is invoked once during each MVM calculation. Table 4.1, which displays the average global memory write bandwidth that was measured for the different kernel configurations when performing 200 maxT permutations, shows that the memory bandwidth increases when writing to multiple DDR banks simultaneously and that the bandwidth is maximised with either two or four kernels. The reduced bandwidth of the eight kernel design compared to the four kernel design suggests that connecting multiple kernels to each global memory bank leads to suboptimal global memory bandwidth.

Table 4.1: The global memory write (i.e. host-to-FPGA) bandwidth that was achieved with different configurations of MVM kernels.

Number of kernels	Data transfer per kernel (MB)	Host-to-FPGA bandwidth (GB/s)
1	976	9.4
2	489	11.6
4	244	11.7
8	122	9.9

Table 4.2 summarises the throughput that was measured for each of the kernel configurations when tasked with performing 200 maxT permutations of the test dataset (requiring 201 MVMs and around 198 GB of data transfer via global memory). It can be seen that the four kernel configuration provides the highest throughout.

Table 4.2: Table summarising the accelerator throughput for 200 maxT permutations using different configurations of MVM kernels.

Number of kernels	Total Runtime (s)	Estimated throughput (GB/s)
1	37.2	5.3
2	25.4	7.8
4	21.5	9.2
8	22.9	8.6

The Vitis Analyser timelines in Figure 4.4 help to visualise the effect of the number of kernels on the accelerator throughput. The `Kernel Enqueues` traces in Figure 4.4 give an indication of the amount of time that the FPGA is idle — large gaps in these traces are undesirable as large stretches of FPGA idle time typically result in a reduction in throughput. It can be seen that a trade-off exists between maximising the global memory bandwidth and maximising the kernel concurrency (and therefore the throughput of the accelerator).

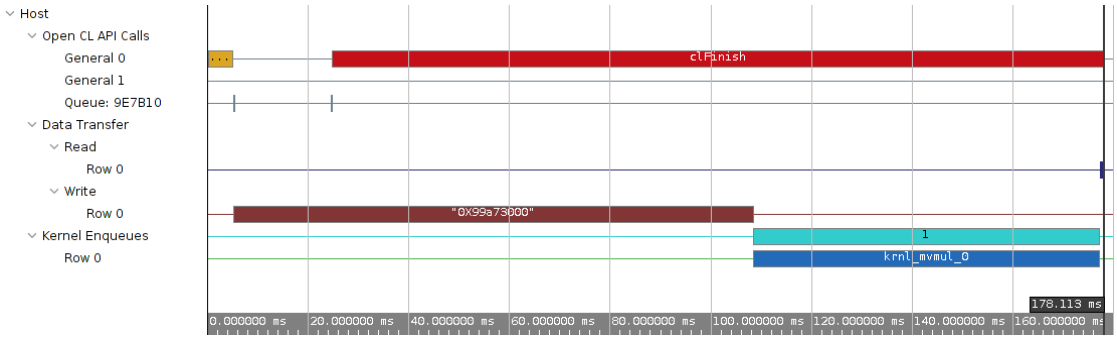
The lowest throughput is achieved with one MVM kernel — there is no concurrency and both the FPGA and the host application are idle for large stretches of time. Although both the two kernel design and the four kernel design have similar global memory bandwidth, the four kernel configuration has lower FPGA idle time due to the concurrent execution of kernels which results in higher overall throughput. The eight kernel configuration displays the lowest amount of FPGA idle time, but has slightly lower throughput than the four kernel configuration due to suboptimal memory bandwidth (as demonstrated by Table 4.1). As the four kernel configuration provides the best trade-off between global memory bandwidth and throughput, it was used for the final design.

The fact that the eight kernel configuration has very similar throughput to the four kernel design in spite of it having significantly worse global memory bandwidth suggests that effective data blocking can improve the accelerator throughput by minimising the FPGA idle time. Section 4.3 investigates the optimal block size to maximise the accelerator throughput.

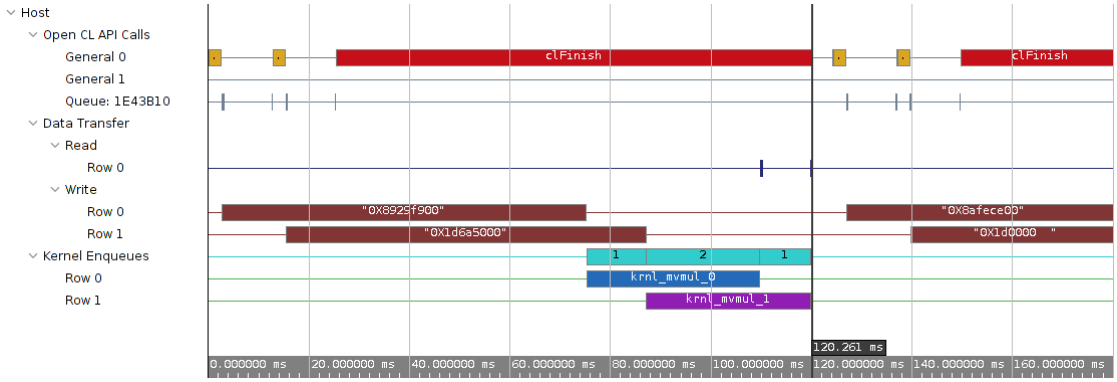
4.3 Buffer Size Optimisation

Minimising the FPGA idle time by optimising the size of the buffers that are transferred to the FPGA kernels via global memory is an important of maximising the accelerator throughput. The four kernel design was used to determine the throughput for different buffer sizes by measuring the time taken to perform 200 maxT permutations using the simulated dataset of 1.89 million SNPs sampled from 2,018 individuals.

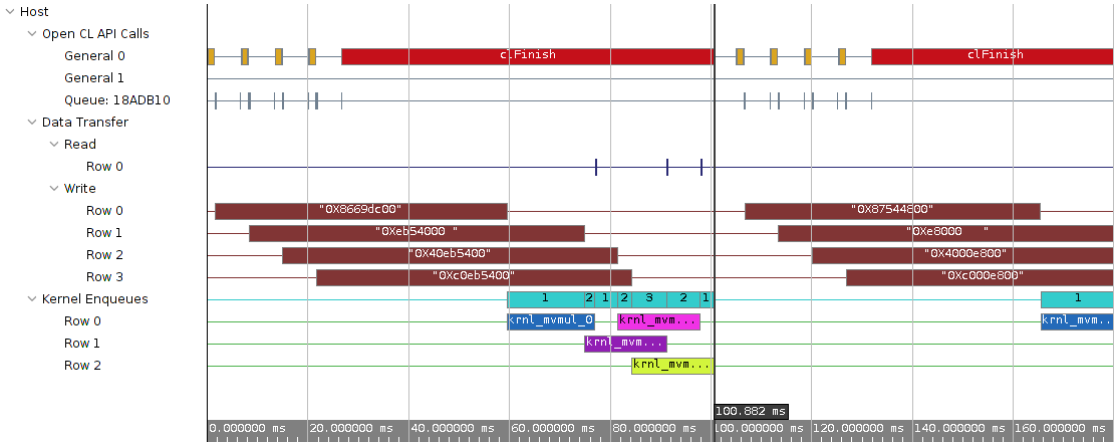
The buffer size in this section refers to the size of the genotype matrix buffer, but the data transferred to/from a kernel for each block of the dataset is composed of the genotype matrix buffer, the genotype mean buffer, the phenotype buffer and the result buffer. As the dataset blocking algorithm uses the specified genotype matrix buffer size to perform the blocking and each block of data transferred to the FPGA must be aligned to a 4 KB boundary, changing the size of the genotype buffer can result in a change in the total size of the data which is sent to the FPGA. Table 4.3 demonstrates how the size of the genotype matrix buffer affects the total data transfer for a single MVM computation.



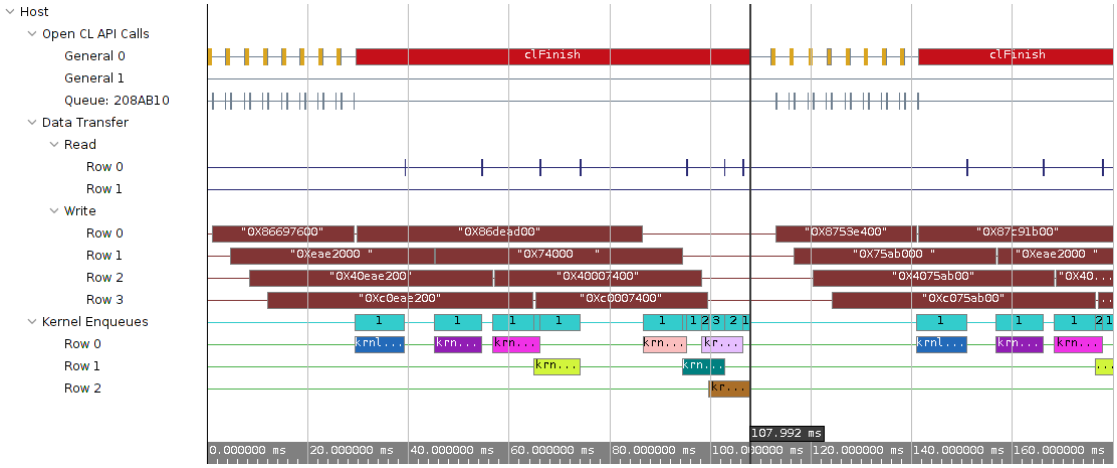
(a) One MVM Kernel



(b) Two MVM Kernels



(c) Four MVM Kernels



(d) Eight MVM Kernels

Figure 4.4: Vitis Analyser timelines which illustrate how the throughput of the accelerator is affected by the number of kernels.

Table 4.3: Table demonstrating how the size of the genotype buffer affects the total data transfer for one MVM computation.

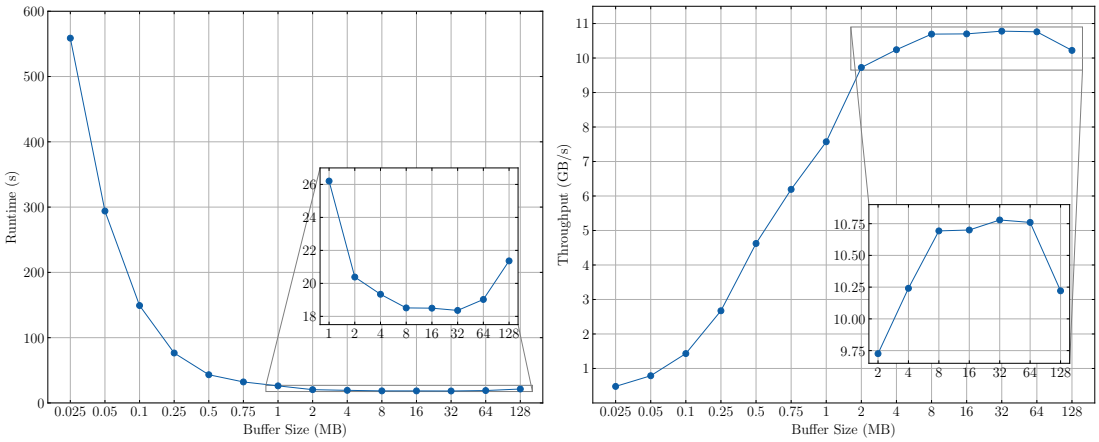
Buffer Size (MB)	Kernel Invocations	Total data transfer (MB)
0.025	29,658	1,336
0.05	14,829	1,154
0.1	7,415	1,063
0.25	3,708	1,018
0.5	1,854	995
0.75	1,236	992
1	927	987
2	464	986
4	232	985
8	116	985
16	58	985
32	29	985
64	15	1,018
128	8	1,086

Table 4.3 shows that the 4 MB, 8 MB, 16 MB and 32 MB buffers yield the lowest data transfer volume. It is also apparent that the total data transfer volume increases as the buffer size decreases. This is caused by the requirement for 4 KB aligned buffers, which enforces a 4 KB minimum buffer size on all buffers transferred to or from the FPGA, resulting in the total data transfer increasing when the buffer size decreases below 1 MB. Furthermore, the fact that the phenotype data must be transferred to the FPGA for each data block leads to an increase in data transfer as the block size decreases (and the number of blocks increases). This result suggests that an adaptive buffer sizing algorithm which automatically uses the buffer size that produces the smallest overall data transfer volume should be implemented in the host application.

Figure 4.5, which plots both the algorithm runtime and the FPGA throughput against the buffer size for 200 maxT permutations of the test dataset, shows that the performance of the accelerator decreases when the buffer size becomes small. It is clear that the buffer size begins to have a significant effect on the efficiency of the accelerator when it is reduced below 2 MB. It can also be seen that comparably high performance is achieved with any buffer size from 8 MB to 64 MB.

Figure 4.6 shows Vitis Analyzer timelines which illustrate (primarily through the `Kernel Enqueues` trace and the `Host - Write` trace) how the buffer size affects the performance of the accelerator. Significant periods of idle time in either of the two traces imply that the throughput is not being maximised. It can be seen that if a small ($\leq$ 2 MB) buffer is used, neither the global memory bandwidth nor the FPGA utilisation are maximised, so the throughput of the accelerator is reduced. This is exemplified by the timelines for the 1 MB and 2 MB buffers which display gaps between kernel invocations and between global memory transfers indicating that the accelerator hardware is not being fully utilised.

Table 4.4 (which compares the average host-to-FPGA global memory bandwidth for the non-overlapping data transfers in Figure 4.6) demonstrates that the global memory bandwidth increases as the buffer size increases which has the effect of decreasing the



(a) Runtime vs genotype buffer size

(b) Throughput vs genotype buffer size

Figure 4.5: Plots showing the effect of the genotype buffer size on accelerator throughput and runtime when performing 200 maxT permutations (requiring 201 MVMs).

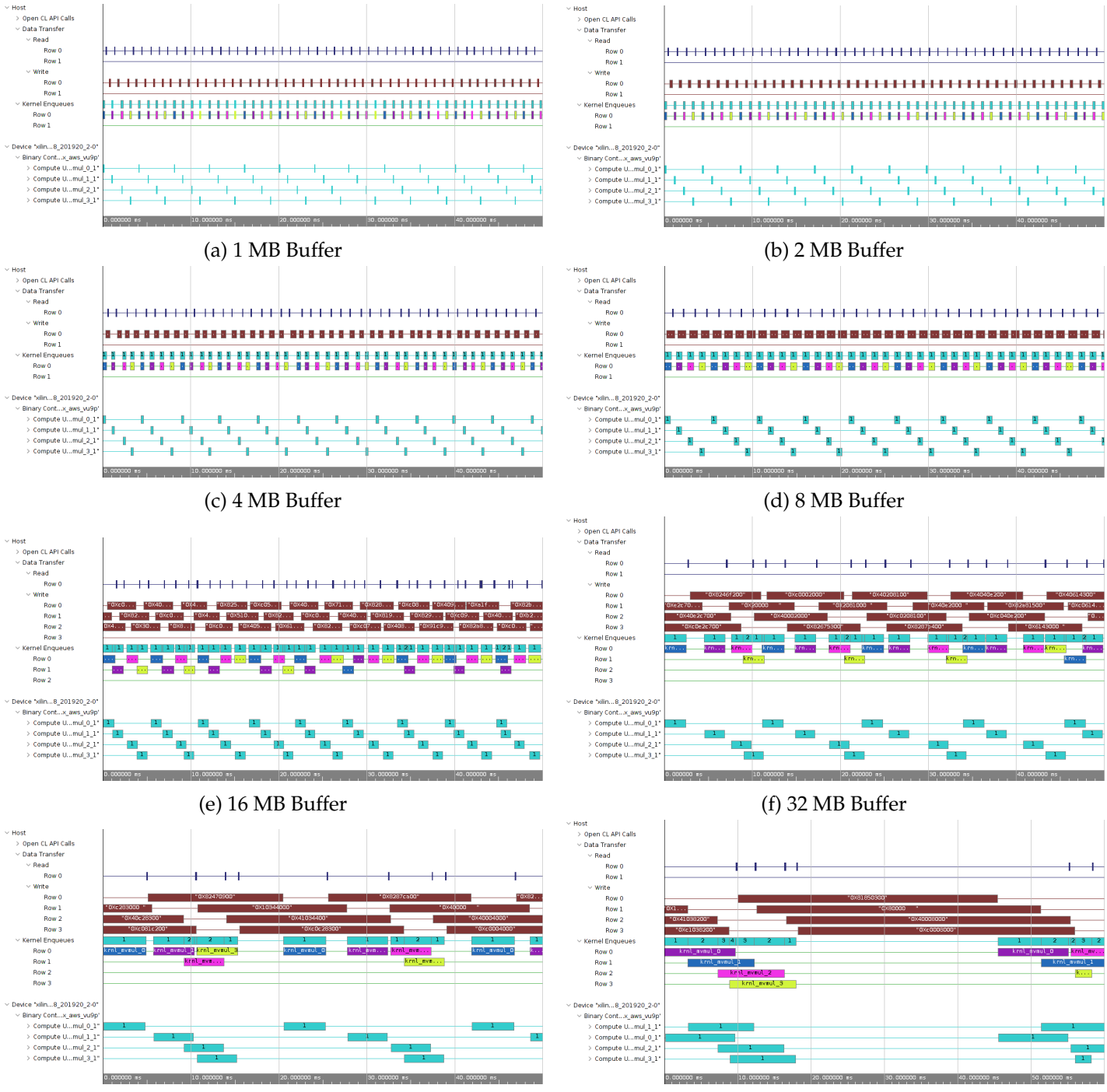
FPGA idle time. This is illustrated by the timeline for the 8 MB buffer which shows that the maximisation of the global memory bandwidth has reduced the length of the gaps between kernel invocations and, therefore, minimised the FPGA idle time.

Table 4.4: Host-to-FPGA write bandwidth for different buffer sizes.

Buffer Size (MB)	Bandwidth (GB/s)
1	5.1
2	6.6
4	7.6
8	8.7

Increasing the buffer size above 8 MB causes data transfers to overlap which, as addressed in Section 4.2, results in higher overall global memory bandwidth than non-overlapped buffer transfers. The timelines for the 16 MB and the 32 MB buffers demonstrate that overlapped memory transfers and the concurrent execution of kernels contribute to a minimisation of the FPGA idle time. The 64 MB buffer allows for more overlap of data transfers and kernel concurrency than the 32 MB buffer, but the increased buffer transfer time has the effect of slightly increasing the FPGA idle time. Although the 128 MB buffer results in a significant amount of kernel concurrency and data transfer overlap, the high data transfer time induces large stretches of FPGA idle time which results in reduced throughput.

The main conclusion that can be drawn from these results is that size of the data buffers transferred via global memory can have a significant effect on accelerator performance although a number of different buffers (8 MB, 16 MB, 32 MB and 64 MB) yield similarly high throughput due to the fact that varying the buffer size results in a trade-off between global memory bandwidth, kernel concurrency and FPGA idle time. Accelerator throughput is negatively affected, however, when the buffer size is reduced below 2 MB.



(a) 1 MB Buffer

(b) 2 MB Buffer

(c) 4 MB Buffer

(d) 8 MB Buffer

(e) 16 MB Buffer

(f) 32 MB Buffer

(g) 64 MB Buffer

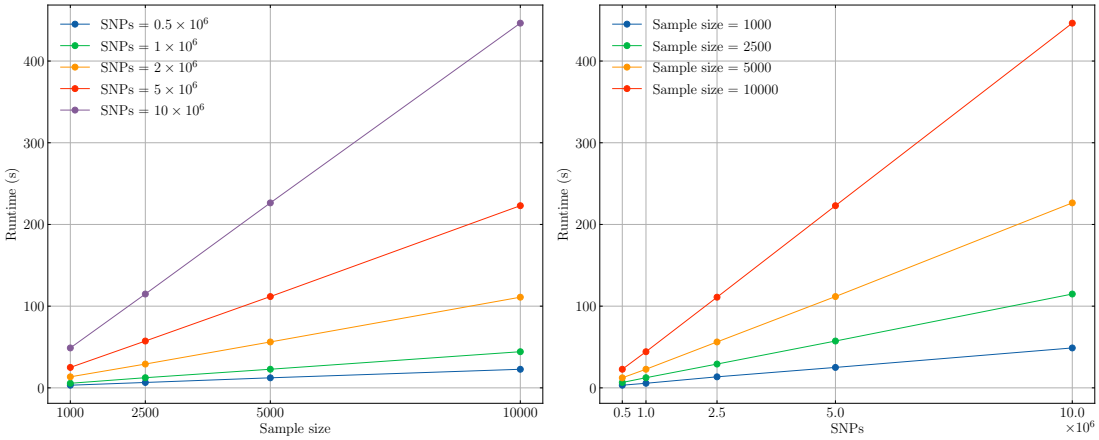
(h) 128 MB Buffer

Figure 4.6: Vitis Analyser timelines demonstrating how the genotype buffer size affects the performance of the accelerator. The aim of the buffer optimisation exercise is to reduce the amount of idle time between kernel invocations which is illustrated by the `Kernel Enqueues` trace.

4.4 Demonstrating the Effect of the Dataset Size on Performance

This section shows how the size of the GWAS dataset (in terms of both the sample size and the number of SNPs) affects the performance (i.e. the runtime and throughput) of the accelerator. Random datasets with varying amounts of individuals and SNPs were generated and the time taken to perform 200 maxT permutations (requiring 201 MVMs) was used to measure the effect of the dataset size on performance. Following the results of Section 4.3, a 32 MB buffer was used for blocking all of the datasets.

Figure 4.7, which demonstrates the effect of the dataset size on the runtime of the accelerator, shows that the runtime scales linearly with increasing sample sizes and increasing SNPs as predicted by the analysis of the permutation testing algorithm in Section 3.1.



(a) Runtime vs number of individuals

(b) Runtime vs number of SNPs

Figure 4.7: Plots showing the effect of the size of the GWAS dataset on the accelerator runtime when performing 200 maxT permutations.

Figure 4.8, which plots the average throughput against the GWAS sample size for different amounts of SNPs, shows that the throughput generally increases as the sample size increases and that the throughput appears to level out at around 11.6 GB/s for large sample sizes independent of the number of SNPs. This could be due to the fact that the MVM kernel processes each row of the genotype matrix (which represents the sampled genotype of a single SNP) sequentially, and shorter matrix rows contribute to lower throughput. It could also be due to any processing delays being averaged out over a longer runtime for larger datasets.

Figure 4.8 demonstrates that for small sample sizes, the inclusion of more SNPs results in higher throughput. When the sample size is large, however, the throughput is close to its maximum (around 11.6 GB/s) for any amount of SNPs. This is probably due to the fact that, as the genotype buffer size is fixed, the concurrency (and consequently, the throughput) of the accelerator improves as the size of the dataset increases with additional SNPs. This implies that an adaptive data blocking algorithm could improve the efficacy of the accelerator for small datasets.

4.5 Hardware Utilisation

This section evaluates the hardware utilisation of the final FPGA design which, as addressed in Section 3.2.8, consists of four MVM kernels (with each kernel connected to

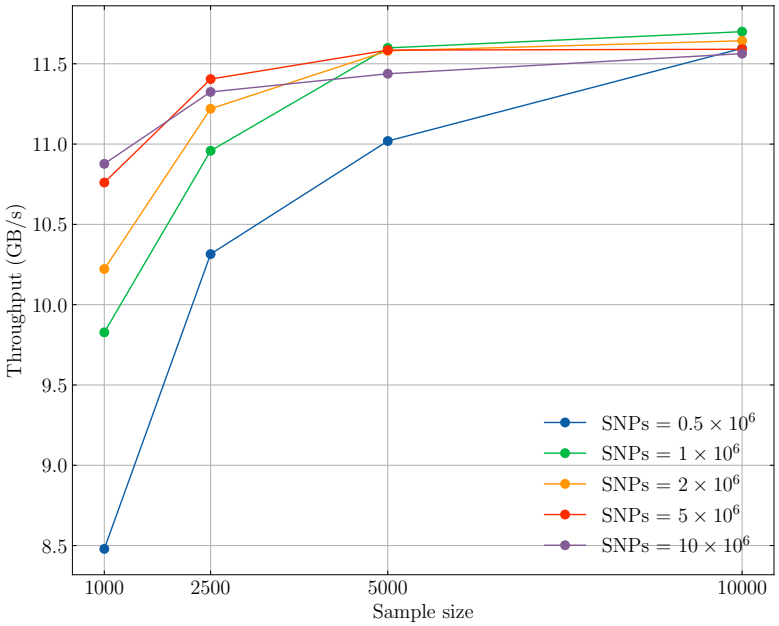


Figure 4.8: Plot demonstrating that the accelerator throughput increases as the sample size increases.

a separate global memory bank) constrained to one of the three super logic regions of the Xilinx Virtex UltraScale+ VU9P FPGA. As discussed in Section 3.2.2, the phenotype buffer for the final design was defined to hold a maximum of 262,144 elements (i.e. each kernel has a 512 kB phenotype buffer).

Vitis 2020.2 was used to compile the final FPGA design using the the build directives defined in Table 3.4. It should be noted that new Vitis releases typically include optimisations which help to generate more efficient hardware designs, so a design built with Vitis 2020.2 may differ slightly from a design built with a different version of Vitis.

Table 4.5, which summarises the hardware utilisation of the four kernel design, indicates that the hardware resources provided by the Xilinx Virtex UltraScale+ VU9P can easily support the constrained four kernel design at the target frequency of 250 MHz, and the low RAM utilisation suggests that it should be possible to generate a design with a larger phenotype buffer if required.

As a reference, Table 4.6 estimates the utilisation of the (less expensive, but less powerful) Xilinx Zynq UltraScale+ ZU9EG FPGA using the four kernel design generated for the VU9P. Although Vitis will perform a number of logic optimisations in an attempt to reduce the logic utilisation, Table 4.6 suggests that Vitis will struggle to generate a usable design for the ZU9EG due to the high utilisation of CLB LUTs and CLB registers. This implies that, if an FPGA with fewer hardware resources than the VU9P is used to implement the permutation testing accelerator, the number of MVM kernels should be reduced and/or the size of the phenotype buffer should be reduced.

To show how the number of MVM kernels affects the hardware utilisation, Table 4.7 summarises the hardware utilisation of the eight kernel design used in Section 4.2. As discussed in Section 4.2, the phenotype buffer size was reduced to 131,072 and the kernels were not constrained to SLRs in order to generate a design which could operate at 250 MHz. Furthermore, the eight kernel design was compiled with Vitis 2020.1, so

Table 4.5: Xilinx Virtex UltraScale+ VU9P hardware utilisation for the constrained four kernel design.

(a) Hardware Resource Utilisation

	Used	Available	% Utilisation
CLBs	76,533	147,623	52
CLB LUTs as Logic	329,279	1,180,984	28
CLB LUTs as memory	47,507	591,440	8
CLB Registers/Flip-flops	520,728	2,361,968	22
BRAM	432	2,160	20
UltraRAM	299	960	31
DSPs	1,056	6,840	15

(b) SLR % Utilisation

	SLR 0	SLR 1	SLR 2
CLBs	61	70	25
CLB LUTs as Logic	37	34	12
CLB LUTs as memory	9	10	5
CLB Registers/Flip-flops	26	30	10
BRAM	30	22	8
UltraRAM	20	53	20
DSPs	12	23	12

Table 4.6: Xilinx Zynq UltraScale+ ZU9EG estimated hardware utilisation for the constrained four kernel design.

	Used (VU9P)	Available (ZU9EG)	% Estimated Utilisation
CLB LUTs as Logic	329,279	274,000	> 100
CLB LUTs as memory	47,507	144,180	33
CLB Registers/Flip-flops	520,728	548,000	95
BRAM	432	920	47
UltraRAM	299	0	N/A
DSPs	1,056	2,520	42

Vitis 2020.2 may be able generate a more efficient design (in terms of the consumed hardware resources).

Table 4.7 demonstrates that the eight kernel design requires significantly more hardware resources than the four kernel design in spite of the fact that the size of the phenotype buffer is half the size of the buffer used in four kernel design. This suggests that the number of MVM kernels has a more significant effect on the resource utilisation than the size of the phenotype buffer.

Table 4.7: Xilinx Virtex UltraScale+ VU9P hardware utilisation for the unconstrained eight kernel design.

	Used	Available	% Utilisation
CLB LUTs as Logic	528,361	1,180,984	45
CLB LUTs as memory	66,062	591,440	11
CLB Registers/Flip-flops	691,306	2,361,968	29
BRAM	1,493	2,160	69
UltraRAM	43	960	5
DSPs	2,228	6,840	33

4.6 Summary

This chapter presented the results of a number of experiments they were performed during the course of the accelerator design process. Section 4.1 showed that the actual FPGA execution time is a fraction of the overall OpenCL runtime which underscores the need for multiple kernels as well as the concurrent execution of kernels. It also demonstrated the effect of connecting multiple kernel interface ports to the same global memory bank. Section 4.2 demonstrated that four MVM kernels (with each kernel connected one global memory bank) maximise the global memory bandwidth and that the use of more than four MVM kernels does not lead to an increase in accelerator throughput. Section 4.3 showed that the size of the buffer used to transfer data between the host CPU and the FPGA can have a significant effect on global memory bandwidth and that varying the buffer size results in a trade-off between global memory bandwidth, kernel concurrency and FPGA idle time. It also demonstrated that the efficiency of the accelerator is reduced when small ($\leq$ 1 MB) buffers are used to transfer data to the FPGA kernels via global memory. Section 4.4 demonstrated that the accelerator runtime scales linearly with the GWAS sample size and number of SNPs. It also showed that the accelerator throughput is affected by the size of the dataset and that the throughput typically improves, up to an apparent limit, for larger datasets. Section 4.5 showed how the FPGA hardware utilisation is affected by the number of kernels and the phenotype buffer size.

5 Results

This chapter compares the speed and accuracy of the FPGA-accelerated permutation algorithms to PLINK — a popular C/C++ command line tool that implements both maxT and adaptive permutation testing. Additionally, the cost and power consumption of the FPGA-based permutation testing accelerator are analysed and compared to CPU-based permutation testing.

PLINK was used as a performance reference as it is one of the most widely used GWAS tools. The FPGA-accelerated permutation algorithms were tested on an AWS EC2 f1.2xlarge instance and PLINK testing was done on a CPU cluster with two Intel Xeon Silver 4114 CPUs (each consisting of twenty 2.2 GHz hyper-threaded cores) per node. The CPU cluster was used to conduct the performance comparison as it is commonly used to perform GWAS analyses using PLINK. All runtime measurements were performed with the Linux `time` command.

Two datasets were used to conduct the performance comparison:

- **Dataset 1** — A simulated dataset of 1,898,052 SNPs sampled from 2,018 individuals;
- **Dataset 2** — A real, fully imputed dataset of 13,742,528 SNPs sampled from 3,652 individuals.

It should be noted that the sample sizes of modern GWAS datasets can range anywhere from 2,000 to more than 100,000 individuals, so the sample sizes of the test datasets are relatively low. As addressed in Section 4.4, however, the accelerator runtime should scale linearly with increasing sample sizes.

Section 5.1 compares the speed and accuracy of the FPGA-accelerated permutation testing algorithms to PLINK, Section 5.2 analyses the power consumption of the accelerator and Section 5.3 analyses the cost of FPGA-based GWAS permutation testing using AWS EC2 FPGA instances.

5.1 Comparison Against PLINK

This section compares the speed and the accuracy of the FPGA-accelerated maxT and adaptive permutation algorithms to PLINK 1.9 [14].

Although PLINK uses multithreading to parallelise maxT permutation on a multi-core CPU, it does not natively support parallelisation on a CPU cluster, so a Nextflow script was developed to split the maxT computation over a specified number of cluster nodes by triggering an instance of PLINK on each of the specified nodes (where each PLINK instance is required to perform n_{perms}/n_{nodes} maxT permutations) and combining the results.

The data dependencies of the adaptive permutation algorithm limited the opportunity for parallelisation over multiple CPU nodes, although PLINK parallelises the algorithm using multithreaded processing on a single node. In the course of testing the PLINK permutation algorithms, it was found that PLINK's `threads` parameter (which is ostensibly used to set the number of cores that PLINK uses to parallelise

the algorithms) has a significant effect on the runtime of the adaptive permutation algorithm, but has no effect on the runtime of the maxT algorithm. As PLINK uses a multithreaded linear algebra library that is not affected by the `threads` parameter, it uses the maximum number of available cores irrespective of the `threads` setting when performing association testing. It was found, however, that `threads=8` (as opposed to `threads=40`) significantly improved the speed of adaptive permutation testing for Dataset 1. This may be caused by the fact that, in the later stages of the adaptive permutation testing algorithm when most SNPs have been dropped, having the `threads` parameter lower than the number of cores allows PLINK to parallelise the algorithm more effectively. Time constraints prevented further analysis and optimisation of PLINK's runtime, however, so `threads=8` was used for PLINK adaptive permutation testing while `threads=40` was used for PLINK maxT permutation testing.

The accuracy of the FPGA accelerator was compared to PLINK by comparing the permuted p-values to determine whether the same significant SNPs are identified with the same accuracy. As permutation testing is an inherently random process, some variation in the results is expected, but the p-values of the most significant SNPs should be very similar (if not identical). Permuted p-values were compared by ranking the PLINK p-values from lowest to highest and determining the FPGA-generated p-values of the corresponding SNPs. To simulate a realistic GWAS permutation testing scenario, the p-value comparison was conducted with maxT p-values generated with 1,000 permutations and adaptive p-values generated with at least 20 million permutations.

It should be noted that these tests were performed without covariates (i.e. the phenotypes were pre-adjusted with covariate data) as, if a covariate file is included, PLINK recalculates the phenotypes for each permutation which significantly reduces the speed of the permutation testing procedure. The FPGA-based algorithms perform the covariate adjustment once at the start of the permutation procedure, so the inclusion of covariates can be expected to have a minimal effect on the runtime of the FPGA-based algorithms.

5.1.1 Runtime Comparison

This section uses PLINK as a reference with which to compare the speed of the FPGA-accelerated permutation testing algorithms. To quantify the level of acceleration provided by the FPGA, this section determines the FPGA speedup for the two permutation testing algorithms where

$$\text{FPGA Speedup} = \frac{\text{PLINK Runtime}}{\text{FPGA Runtime}} \tag{5.1}$$

maxT Permutation Testing

The runtime of the FPGA-based maxT permutation algorithm was analysed and compared to the runtime of PLINK running on an Intel Xeon CPU cluster. The effect of large-scale parallelisation of PLINK's maxT implementation was determined by splitting the PLINK maxT permutation testing algorithm over multiple nodes of the CPU cluster (resulting in parallelisation over 40, 200 or 400 CPU cores). The multi-node PLINK tests were only run with the smaller, simulated dataset (Dataset 1), however, as it was not possible to get access to multiple nodes of the CPU cluster for the time required to process the fully imputed, real dataset (Dataset 2).

To show how the runtime of the FPGA accelerator is affected by the number of permutations, Figure 5.1 plots the maxT permutation testing runtime against varying numbers of permutations. It can be seen that the runtime scales linearly with the

number of permutations which was predicted by the analysis of the permutation testing algorithm in Section 3.1. This result, together with the results of Section 4.4, proves that the runtime of maxT permutation testing scales linearly with increasing sample sizes, SNPs and permutations.

The constant offsets displayed by the fitted linear models in Figure 5.1 are caused by the data preprocessing and results output steps of the maxT permutation testing algorithm which are not FPGA-accelerated and are constant (i.e. independent of the number of permutations) for a given dataset.

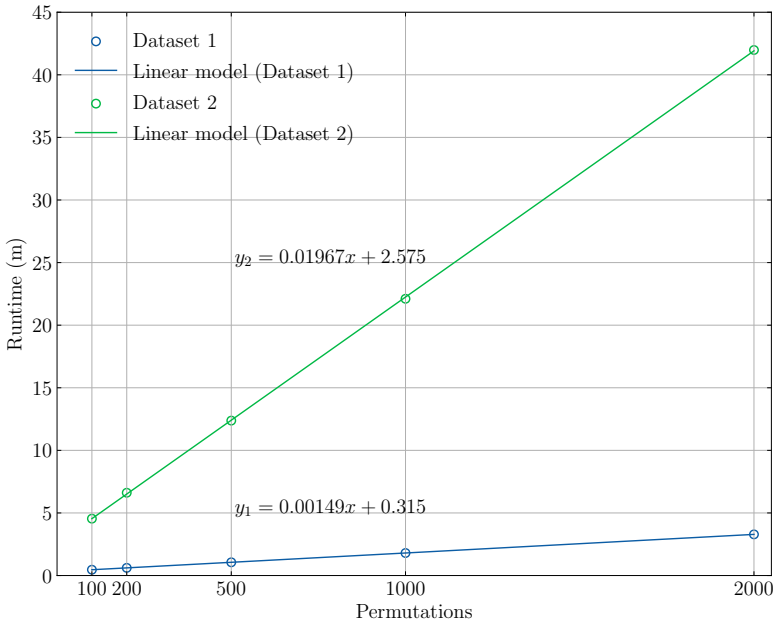


Figure 5.1: Plot of the FPGA-accelerated maxT runtime against the number of maxT permutations (together with fitted linear models) showing that the accelerator runtime scales linearly with number of permutations.

Figure 5.2 plots the accelerator throughput against the number of maxT permutations. It can be seen that the throughput of the maxT accelerator increases as the number of permutations increases which is probably due to any processing delays being averaged out over a longer runtime. The fact that the throughput achieved for Dataset 2 is higher than that achieved for Dataset 1 reinforces the conclusion from Section 4.4 that higher sample sizes yield higher throughput. The rate of throughput increase slows down (and, in the case of Dataset 2, the throughput can be observed to decrease) as more permutations are performed. This implies that, although each dataset has a different maximum achievable throughput which is based on the size of the dataset and the number of samples, the throughput of the FPGA-based maxT accelerator appears to be close to its maximum for a particular dataset when performing more than 500 maxT permutations.

Figure 5.3 compares the runtime of the FPGA-accelerated maxT algorithm to PLINK's maxT runtime. It can be seen that the FPGA-accelerated maxT algorithm is at least two orders of magnitude faster than PLINK running on 40 CPU cores for any amount of permutations. Furthermore, the runtime of the FPGA-based maxT accelerator is more than an order of magnitude faster than highly parallelised PLINK.

Figure 5.3b demonstrates that the efficiency of multi-node PLINK decreases as more nodes are used. In other words, there is not a direct linear relationship between the

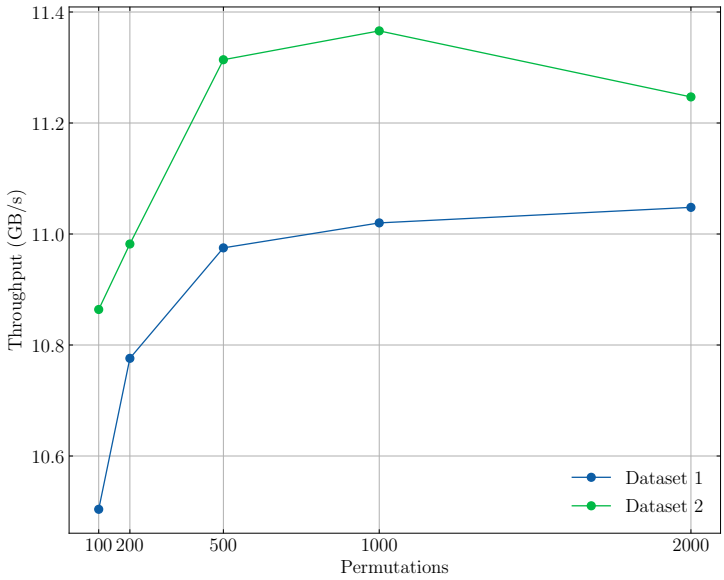


Figure 5.2: Plot of the accelerator throughput against the number of maxT permutations demonstrating that the throughput typically increases as the number of permutations increases.

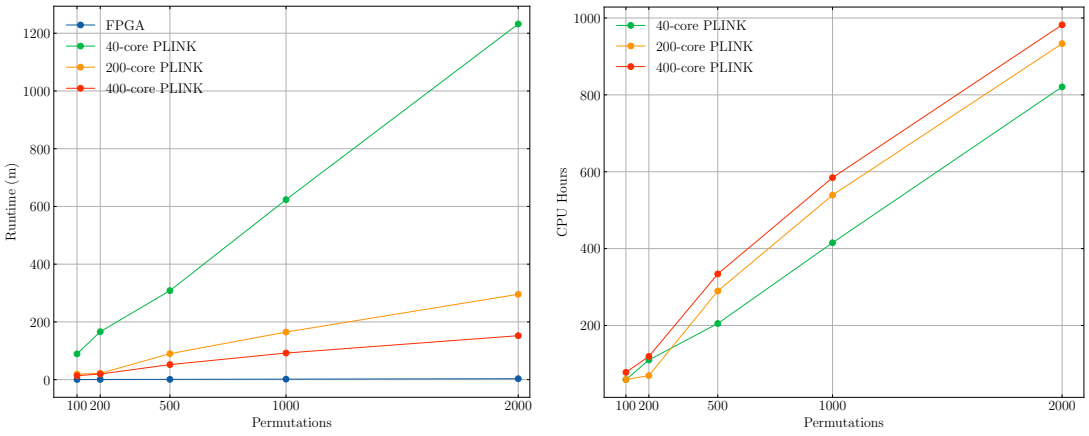
number of cluster nodes and PLINK's maxT permutation runtime. This is probably due to the fact that PLINK was parallelised by running one instance of PLINK on each node, so the constant-time parts of PLINK's maxT implementation (e.g. reading data from input files, calculating the observed p-values) are replicated on each node.

To quantify the level of acceleration provided by the FPGA, Table 5.1 summarises the recorded maxT runtimes, while Figure 5.4 plots the FPGA speedup against the number of maxT permutations. The FPGA runtimes in Table 5.1a are averaged over 5 runs. As the runtime variance was low (typically less than one second), multiple runs were not done for Dataset 2 or for the adaptive permutation tests.

Figure 5.4 demonstrates that the FPGA speedup increases as the number of maxT permutations increases. This is due to:

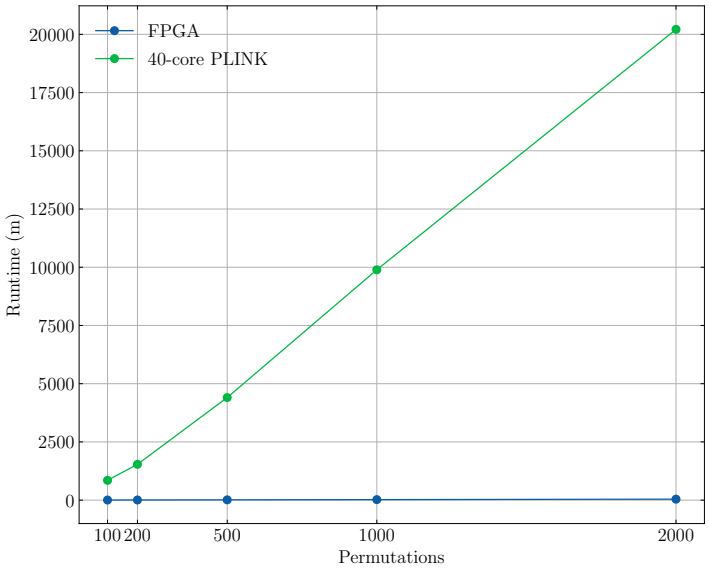
- The constant-time, CPU-only parts of the FPGA-based maxT algorithm which constitute the majority of the runtime when few permutations are performed but become less influential on the overall runtime as the number of permutations increases.
- The fact that the FPGA throughput increases as the number of permutations increases

To give an indication of the magnitude of the acceleration provided by the FPGA, the FPGA-based maxT algorithm can complete 2,000 permutations of Dataset 2 significantly faster than PLINK (running on 40 CPU cores) can complete 100 permutations of Dataset 1. For 1,000 maxT permutations (which, as stated in Section 2.3, provides effective control of the false positive rate for any number of SNPs) the FPGA-based algorithm is 346 times faster than 40-core PLINK for Dataset 1 and 450 times faster than 40-core PLINK for Dataset 2. This suggests that the FPGA-based maxT accelerator is more effective when operating on larger datasets.



(a) Dataset 1

(b) CPU hours used for Dataset 1



(c) Dataset 2

Figure 5.3: Plots comparing the runtime of the FPGA-based maxT accelerator to the runtime of PLINK for both test datasets. It can be seen that the FPGA runtime is at least two orders of magnitude faster than PLINK for any amount of maxT permutations.

Adaptive Permutation Testing

The speed of the FPGA-based adaptive permutation algorithm was tested and compared to PLINK using the recommendations from Che et al. [8] for the minimum number of permutations for each SNP (see Section 2.4). To determine how the minimum permutations per SNP affects the runtime of the algorithm, Dataset 1 testing was carried out with two minimum permutation values (121 and 36), although time constraints limited Dataset 2 testing to a single minimum permutation value (121).

The following (default) values were used for PLINK's adaptive permutation parameters (see Section 2.4 for an explanation of how these parameters affect the permutation procedure):

- $\beta = 0.0001$
- $\alpha = 0$
- $b = 1$
- $a = 0.001$

Table 5.1: Table comparing the runtime of the FPGA-based maxT implementation to PLINK together with the associated FPGA speedup.

(a) Dataset 1

Permutations	FPGA (m)	40-core PLINK		200-core PLINK		400-core PLINK	
		Runtime (m)	Speedup	Runtime (m)	Speedup	Runtime (m)	Speedup
100	0.5	89	178	19	38	14	28
200	0.6	166	277	22	37	20	33
500	1.1	308	280	90	82	52	47
1,000	1.8	624	346	165	92	92	51
2,000	3.3	1,232	373	296	90	152	46

(b) Dataset 2

Permutations	FPGA (m)	40-core PLINK	
		Runtime (m)	Speedup
100	5	851	170
200	7	1,538	220
500	12	4,404	367
1,000	22	9,891	450
2,000	42	20,214	481

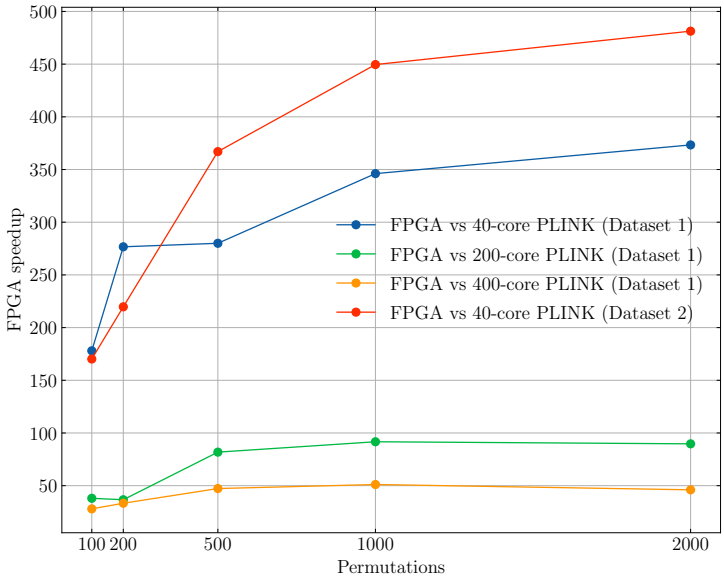


Figure 5.4: Plot of the FPGA speedup against the number of maxT permutations showing that the speedup increases as the number of maxT permutations increases.

No runtime optimisation was done on these parameters so it is not known if they minimise the runtime of PLINK's adaptive permutation algorithm or if they can be optimised for different datasets.

Figure 5.5, which plots the runtime of the FPGA-based adaptive permutation algorithm against the number of permutations, demonstrates that, unlike the maxT algorithm, the relationship between the runtime and the number of adaptive permutations is not constant. In the early stages of the algorithm, insignificant SNPs are frequently dropped from the permutation procedure causing the average time per permutation to decrease which results in a non-linear relationship between runtime and the number of permutations. In the later stages of the algorithm, only significant SNPs remain, so the rate at which SNPs are dropped decreases and the average time per permutation becomes effectively constant, therefore resulting in a linear relationship between runtime and the number of permutations.

Figure 5.5a shows that the minimum permutations per SNP has a relatively small effect on the runtime of the FPGA-accelerated adaptive permutation algorithm — it results in an almost constant runtime difference over the range of measurement indicating that it has the largest effect on the runtime early in the permutation procedure when many insignificant SNPs are undergoing permutation. Figure 5.6a, which compares the FPGA runtime to PLINK's runtime, indicates that the FPGA-based algorithm appears to be less affected by the minimum permutations per SNP — the additional processing time that is incurred by increasing the minimum permutations per SNP is smaller for the FPGA-based algorithm than it is for PLINK.

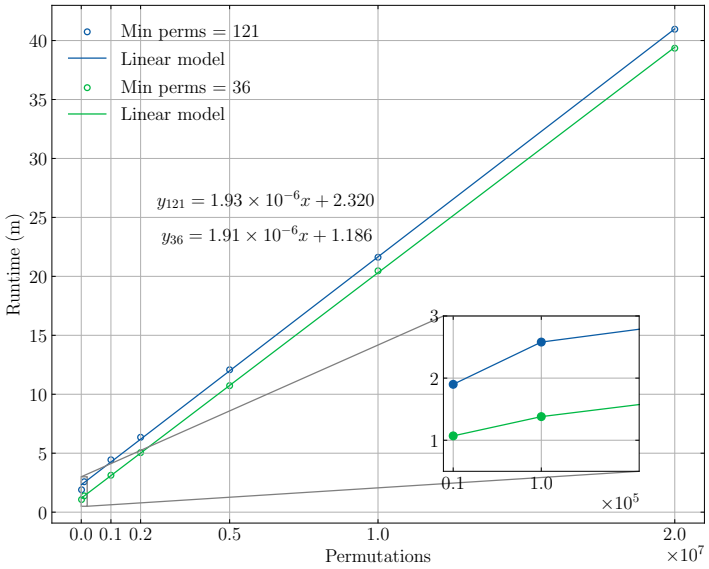
Figure 5.6 compares the runtime of the FPGA-accelerated adaptive permutation algorithm to PLINK by plotting the respective runtimes against the maximum permutations per SNP. It can be seen that the FPGA-accelerated algorithm is at least one order of magnitude faster than PLINK for any number of adaptive permutations. Moreover, Figure 5.6 shows that the FPGA runtime graphs become linear earlier than the PLINK graphs which indicates that the FPGA-based adaptive permutation algorithm minimises the average permutation time faster than the PLINK implementation.

In order to evaluate the level of acceleration provided by the FPGA-based algorithm, Figure 5.7 plots the FPGA speedup against the number of adaptive permutations for the two test datasets. It is immediately apparent that the speedup achieved for the adaptive permutation algorithm is lower than the speedup achieved for the maxT algorithm. This is caused by two aspects of the adaptive permutation algorithm:

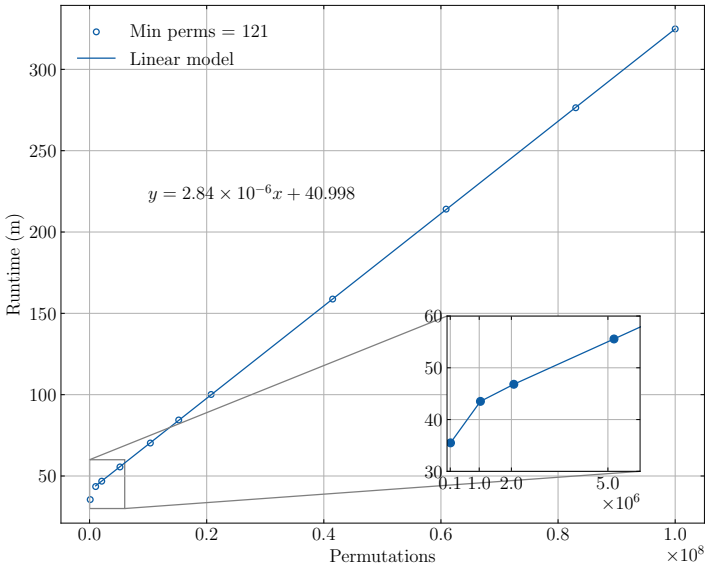
- The adaptive algorithm is much more CPU-dependent than the maxT algorithm — the host application must periodically identify insignificant SNPs and regenerate the data buffers to remove the insignificant SNPs from the permutation procedure.
- In the later stages of the algorithm, the FPGA accelerator is not operating at peak efficiency due to the use of a suboptimal buffer size as addressed in Section 4.3.

In spite of this, the FPGA-accelerated adaptive permutation testing algorithm is faster than PLINK for any number of permutations.

Figure 5.7 shows that the FPGA speedup decreases as the number of adaptive permutations increases which is due to a reduction in the efficiency of the FPGA accelerator caused by the use of a suboptimal buffer size when most of the SNPs have been dropped. In spite of the reduction of efficiency, the FPGA-based accelerator retains a speed advantage over PLINK in the later stages of the algorithm — particularly in the



(a) Dataset 1



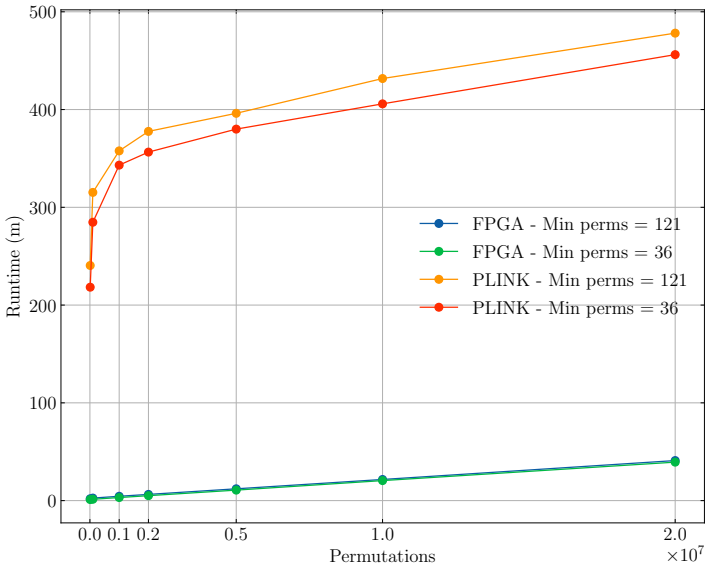
(b) Dataset 2

Figure 5.5: Plots of the FPGA-accelerated adaptive permutation runtime against the maximum number of adaptive permutations per SNP with fitted linear models showing the average time per permutation in the later stages of the algorithm.

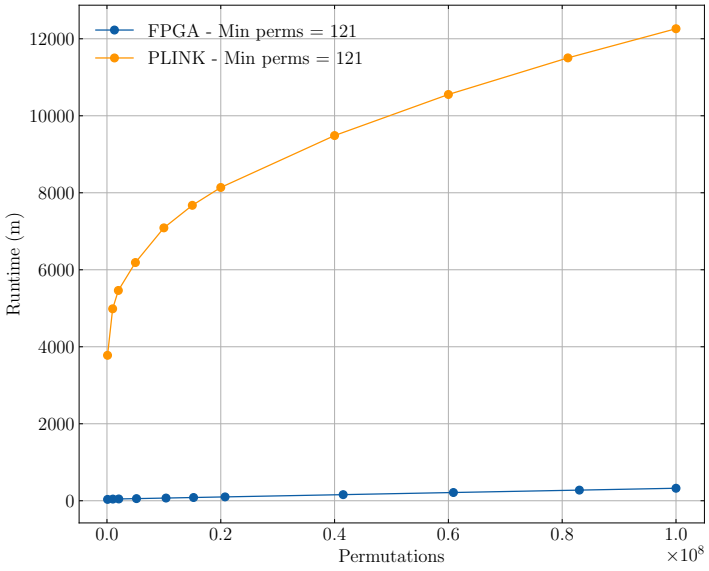
case of Dataset 2. The improved efficacy for Dataset 2 is due to:

- The fact that Dataset 2 has more significant SNPs than Dataset 1 (for example, after 20 million permutations, 21 SNPs remained for Dataset 1 compared to 300 for Dataset 2), so the FPGA accelerator is able to make more effective use of the system memory bandwidth for a higher proportion of the runtime.
- The improved throughput when dealing with larger sample sizes as addressed in Section 4.4.

Figure 5.7a demonstrates that, although the minimum permutations per SNP has a significant effect on the FPGA speedup when few permutations are performed, this effect becomes negligible as the number of permutations increases. This is due to that fact that the minimum permutations per SNP only has a significant effect on the runtime



(a) Dataset 1

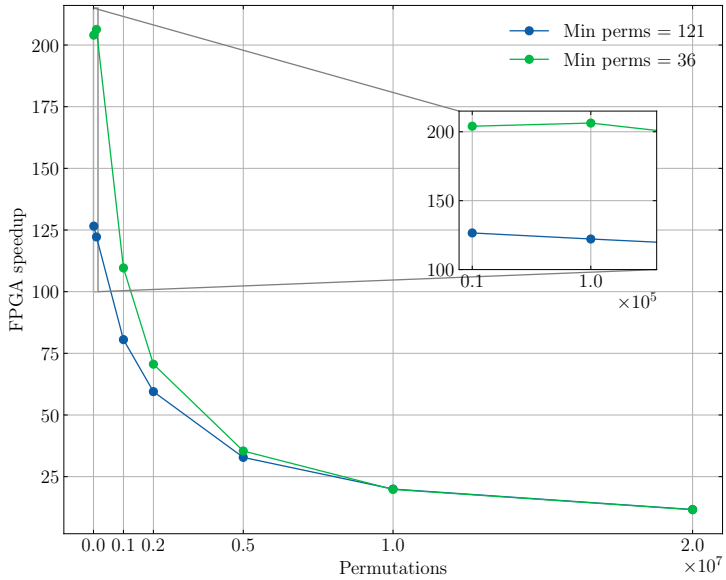


(b) Dataset 2

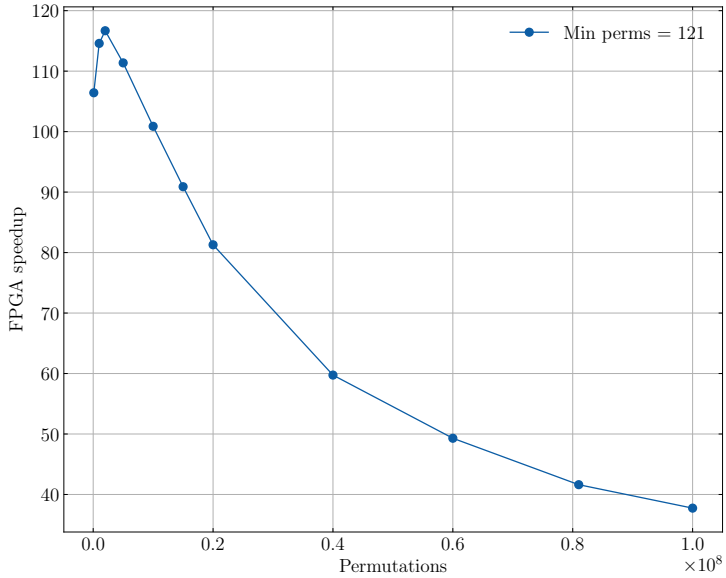
Figure 5.6: Plots comparing the runtime of the FPGA-based adaptive permutation testing accelerator to the runtime of PLINK for both test datasets.

when many SNPs are undergoing permutation. When most of the insignificant SNPs have been dropped, the additional processing time that was incurred by increasing the minimum permutations per SNP is a negligible part of the total algorithm runtime so the speedup difference becomes very small.

Table 5.2 summarises the relative FPGA speedup over different stages of the adaptive permutation algorithm to show how the speedup changes over the course of the algorithm. It can be seen that, although the FPGA speedup decreases as the number of adaptive permutations increases, the FPGA-based algorithm retains a speed advantage over PLINK during the later stages of the algorithm in spite of the reduced efficacy of the FPGA accelerator. Furthermore, the fact that the FPGA speedup for Dataset 2 decreases at a slower rate than the speedup for Dataset 1 indicates that the accelerator is more effective when operating on larger-sized datasets.



(a) Dataset 1



(b) Dataset 2

Figure 5.7: Plot of adaptive permutation speedup against the number of permutations showing that, although the FPGA speedup decreases as the number of permutations increases, the FPGA-based algorithm is significantly faster than PLINK for any number of adaptive permutations.

To prove that the FPGA accelerator can effectively speed up GWAS permutation testing, a run of 700 million adaptive permutations (an almost unfeasible workload for PLINK running on a 40-core CPU) was performed on Dataset 2. Figure 5.8, which plots the runtime of the FPGA-accelerated algorithm over the range of 100 million to 700 million adaptive permutations, demonstrates that the accelerator can complete 700 million permutations of Dataset 2 significantly faster (about 14.5 hours less) than PLINK can complete 10,000 permutations of the same dataset on a 40-core CPU. In fact, using the fitted linear model in Figure 5.8, the FPGA-based algorithm can be estimated to complete one billion permutations of Dataset 2 in the same amount of time that PLINK requires to complete 10,000 permutations (about 48 hours).

Table 5.2: Table summarising the relative FPGA speedup over different stages of the adaptive permutation algorithm to demonstrate how the speedup changes over the course of the algorithm.

(a) Dataset 1

Perm interval	Num perms	Min perms = 36			Min perms = 121		
		FPGA (m)	CPU (m)	Speedup	FPGA (m)	CPU (m)	Speedup
0 to 1×10^4	1×10^4	1.1	218	204	1.9	241	127
1×10^4 to 1×10^5	9×10^4	0.3	66	214	0.7	75	110
1×10^5 to 1×10^6	9×10^5	1.8	58	33	1.9	42	23
1×10^6 to 2×10^6	1×10^6	1.9	13	7	1.9	20	10
2×10^6 to 5×10^6	3×10^6	5.7	24	4	5.7	19	3
5×10^6 to 1×10^7	5×10^6	9.7	26	3	9.6	36	4
1×10^7 to 2×10^7	1×10^7	18.8	50	3	19.3	46	2
0 to 2×10^7	2×10^7	39.4	456	12	41.0	478	12

(b) Dataset 2

Perm interval	Num perms	Min perms = 121		
		FPGA (m)	CPU (m)	Speedup
0 to 1×10^5	1×10^5	35.5	3,778	106
1×10^5 to 1×10^6	9×10^5	8.0	1,208	151
1×10^6 to 2×10^6	1×10^6	3.3	476	144
2×10^6 to 5×10^6	3×10^6	8.8	726	83
5×10^6 to 1×10^7	5×10^6	14.7	900	61
1×10^7 to 2×10^7	1×10^7	29.9	1,050	35
2×10^7 to 4×10^7	2×10^7	58.7	1,348	23
4×10^7 to 6×10^7	2×10^7	55.3	1,067	19
6×10^7 to 1×10^8	4×10^7	110.8	1,708	15
0 to 1×10^8	1×10^8	324.9	12,261	38

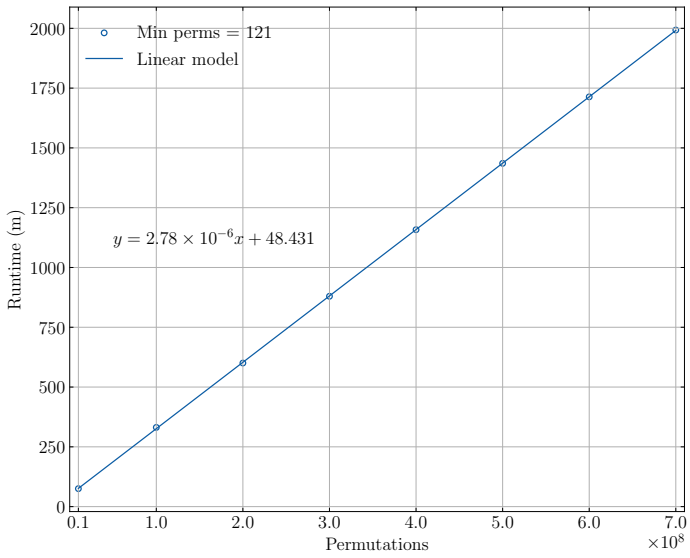


Figure 5.8: Plot of the FPGA-accelerated adaptive permutation runtime against the number of adaptive permutations together with a fitted linear model.

5.1.2 Accuracy Comparison

This section compares the accuracy of the FPGA-based permutation algorithms to PLINK by comparing the p-values of the most significant SNPs (which are selected based on an arbitrary cut-off value) in terms of the difference between the p-values (i.e. the absolute difference and the relative difference) and the FPGA accuracy with respect to PLINK (i.e. the number of false positive and false negative associations). As explained earlier, approximately correct permutation p-values are typically sufficient for estimating statistical significance in a GWAS that aims to identify a few potentially associated SNPs from a large collection of sampled (and possibly imputed) SNPs. For this reason, p-value cut-off values often vary between studies based on the required accuracy of the results. Furthermore, as permutation testing is a random process, some minor variation between p-values is expected. The accuracy comparison focuses on p-values generated with a realistic number of permutations (1,000 maxT permutations and at least 20 million adaptive permutations with 121 minimum permutations per SNP).

It should be noted that this work does not seek to draw any conclusions from the results of the GWAS. It merely aims to compare the accuracy of the FPGA-based algorithms to PLINK by comparing the p-values of corresponding SNPs. For this reason, the SNP IDs and the permutation p-values are not reported (in lieu of the difference between the p-values).

The relative difference used in this section can be defined as:

$$\frac{|p_{PLINK} - p_{FPGA}|}{\max(|p_{PLINK}|, |p_{FPGA}|)} \tag{5.2}$$

maxT Permutation Testing

This section compares the significant SNPs (p_{PLINK} or $p_{FPGA} \leq 0.05$) identified by both implementations of the maxT permutation testing algorithm. Table 5.3 compares the accuracy of the FPGA-accelerated maxT algorithm to PLINK, while Table 5.4 compares p-values of the significant SNPs. Using the $p \leq 0.05$ cutoff, eight significant SNPs were identified from Dataset 1, while 30 significant SNPs were found in Dataset 2. It should be noted that the SNPs in Table 5.4 are ordered in terms of increasing p-values (i.e. SNPs are ordered from most significant to least significant) as calculated by PLINK.

Table 5.3: FPGA-based maxT permutation testing accuracy with respect to PLINK.

	False positives	False negatives
Dataset 1	0	0
Dataset 2	0	0

Table 5.3 demonstrates that the same significant SNPs are identified by both implementations of the maxT algorithm, and Table 5.4 shows that there is very little variation between the maxT p-values of the significant SNPs. As the p-values being compared are very small, a slight difference would have little effect on the significance of the SNP. The accuracy of FPGA-based accelerator is, therefore, acceptable for maxT permutation testing.

Adaptive Permutation Testing

The accuracy of the FPGA-based adaptive permutation algorithm was compared to PLINK by comparing the p-values of the significant SNPs (p_{PLINK} or $p_{FPGA} \leq 1 \times 10^{-7}$)

Table 5.4: Table comparing the p-values of the significant SNPs identified by maxT permutation testing. The SNPs are ordered from most significant to least significant as computed by PLINK.

(a) p-values of the 8 significant SNPs from Dataset 1.

SNP	Absolute Difference	Relative Difference
SNP1	0	0
SNP2	0	0
SNP3	0	0
SNP4	0	0
SNP5	0	0
SNP6	0	0
SNP7	0.002997	0.75
SNP8	0	0

(b) p-values of the 30 significant SNPs from Dataset 2.

SNP	Absolute Difference	Relative Difference	SNP	Absolute Difference	Relative Difference
SNP1	0	0	SNP16	0	0
SNP2	0	0	SNP17	0.000999	0.25
SNP3	0	0	SNP18	0.000999	0.17
SNP4	0	0	SNP19	0.004995	0.36
SNP5	0	0	SNP20	0.003996	0.29
SNP6	0	0	SNP21	0.000999	0.05
SNP7	0	0	SNP22	0	0
SNP8	0	0	SNP23	0	0
SNP9	0	0	SNP24	0	0
SNP10	0	0	SNP25	0	0
SNP11	0	0	SNP26	0	0
SNP12	0	0	SNP27	0.001998	0.07
SNP13	0	0	SNP28	0.006993	0.17
SNP14	0	0	SNP29	0.003996	0.08
SNP15	0	0	SNP30	0	0

generated with 20 million permutations of Dataset 1 and 100 million permutations of Dataset 2. Table 5.5 compares the accuracy of the FPGA-based algorithm with respect to PLINK, while Table 5.6 compares the p-values of the significant SNPs. As in the maxT comparison, the SNPs in Table 5.6 are ordered from most significant to least significant.

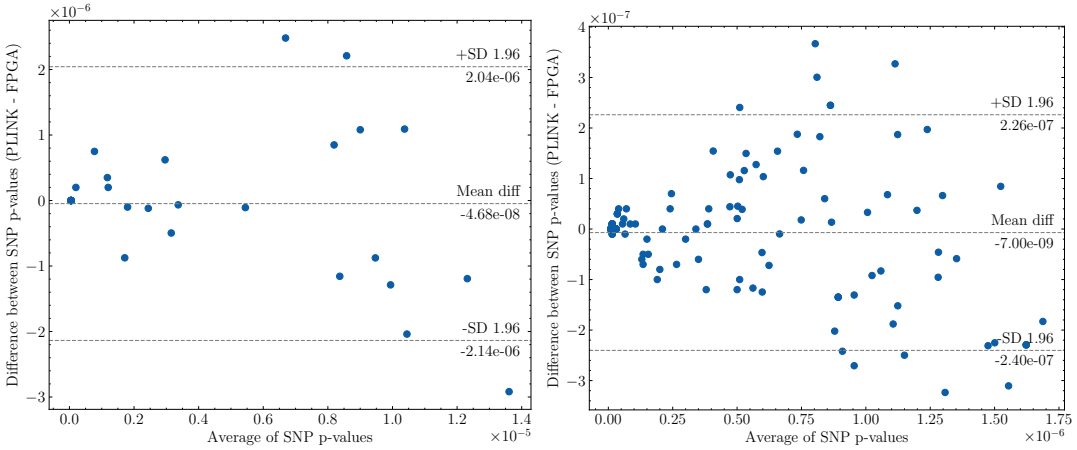
Table 5.5 shows that both adaptive permutation implementations identified the same significant SNPs for Dataset 2, but one false positive occurred for Dataset 1. This can be explained by the random nature of the adaptive permutation testing algorithm and the fact that more permutations were performed for Dataset 2 so the p-values calculated for Dataset 2 are more accurate than those generated for Dataset 1. Table 5.6 shows that the variation between the adaptive p-values is minimal. As very small p-values are being compared, a small amount of variation (which was expected due to the inherent randomness of the algorithm) has little effect on the significance of a given SNP.

Therefore, the accuracy of the FPGA-based adaptive permutation algorithm is sufficient for identifying significant SNPs.

Table 5.5: FPGA-based adaptive permutation testing accuracy with respect to PLINK.

	False positives	False negatives
Dataset 1	1	0
Dataset 2	0	0

Figure 5.9 shows Tukey mean-difference plots comparing the difference between the significant p-values. Figure 5.9 demonstrates that the p-values of the most significant SNPs (i.e. the SNPs found near $(0, 0)$) are very similar. It also shows that the difference between p-values is typically higher for less significant SNPs which is due to the inherent randomness of the permutation procedure. Furthermore, it can be seen that no systematic bias is present in the results for either dataset as the differences are scattered relatively uniformly above and below $x = 0$.



(a) Tukey mean-difference plot of the 30 most significant SNPs from Dataset 1.

(b) Tukey mean-difference plot of the 130 most significant SNPs from Dataset 2.

Figure 5.9: Tukey mean-difference plots comparing the adaptive permutation p-values of the significant SNPs.

5.2 Power Consumption

This section compares the power consumption of the FPGA-based permutation testing accelerator to CPU-based permutation testing using the results of the Vivado power analysis tool. Table 5.7 summarises the estimated power consumption of an f1.2xlarge instance running the four MVM kernel design.

The power consumption of the FPGA-based accelerator running on an f1.2xlarge instance is much less than that of a 40-core node of an Intel Xeon Silver 4114 cluster (340 W) or of a GPU-based accelerator (generally greater than 300 W). FPGA-based acceleration can, therefore, significantly reduce the energy consumption of GWAS permutation testing. As an example, for $1,000$ maxT permutations of Dataset 2, the FPGA accelerator uses $118.1 \times 0.368 = 44$ Wh of energy, while PLINK running on one 40-core node of the CPU cluster consumes $340 \times 164.84 = 56,046$ Wh, so the use of the FPGA accelerator results in a 99.92% reduction in energy consumption.

Table 5.6: Table comparing the p-values of the significant SNPs identified by adaptive permutation. The SNPs are ordered from most significant to least significant as computed by PLINK.

(a) Adaptive p-values of the 9 significant SNPs from Dataset 1.

SNP	Absolute Difference	Relative Difference
SNP1	0	0
SNP2	0	0
SNP3	0	0
SNP4	0	0
SNP5	0	0
SNP6	0	0
SNP7	0	0
SNP8	0	0
SNP9	2.00×10^{-7}	0.67

(b) Adaptive p-values of the 50 significant SNPs from Dataset 2.

SNP	Absolute Difference	Relative Difference	SNP	Absolute Difference	Relative Difference
SNP1	0	0	SNP26	1.00×10^{-8}	0.5
SNP2	0	0	SNP27	1.00×10^{-8}	0.5
SNP3	0	0	SNP28	1.00×10^{-8}	0.5
SNP4	0	0	SNP29	1.00×10^{-8}	0.5
SNP5	0	0	SNP30	1.00×10^{-8}	0.5
SNP6	0	0	SNP31	1.00×10^{-8}	0.5
SNP7	0	0	SNP32	1.00×10^{-8}	0.5
SNP8	0	0	SNP33	0	0
SNP9	0	0	SNP34	0	0
SNP10	0	0	SNP35	0	0
SNP11	0	0	SNP36	0	0
SNP12	0	0	SNP37	0	0
SNP13	0	0	SNP38	0	0
SNP14	0	0	SNP39	0	0
SNP15	0	0	SNP40	0	0
SNP16	0	0	SNP41	0	0
SNP17	0	0	SNP42	3.00×10^{-8}	0.6
SNP18	0	0	SNP43	3.00×10^{-8}	0.6
SNP19	0	0	SNP44	3.00×10^{-8}	0.6
SNP20	0	0	SNP45	1.00×10^{-8}	0.14
SNP21	0	0	SNP46	4.00×10^{-8}	0.67
SNP22	0	0	SNP47	1.00×10^{-8}	0.17
SNP23	1.00×10^{-8}	0.5	SNP48	2.00×10^{-8}	0.29
SNP24	1.00×10^{-8}	0.5	SNP49	1.00×10^{-8}	0.11
SNP25	1.00×10^{-8}	0.5	SNP50	4.00×10^{-8}	0.44

Table 5.7: Table summarising the results of the Vivado power analysis tool to estimate the power consumption of an f1.2xlarge instance running the four MVM kernel design.

	Power consumption (W)
Static FPGA power consumption	3.964
Clocks	3.784
Clock manager	1.384
Signals	17.790
Logic	12.140
BRAM	3.289
URAM	1.500
DSP	1.659
PCIe	0.391
I/O	3.243
Data transceivers	4.360
System monitor	0.014
FPGA Total	53.650
Intel Xeon E5-2686 v4 (8 cores)	64.440
Instance total	118.090

5.3 Cost Analysis

This section analyses the cost of cloud-based acceleration of GWAS permutation testing using AWS EC2 f1.2xlarge FPGA instances. The cost of an f1.2xlarge instance is $1.65 per hour, while the cost of a spot instance is $0.495 per hour. Spot instances are EC2 instances that AWS offers at a discount depending on excess instance capacity. The use of spot instances can significantly reduce the cost of AWS EC2 instances, but they may not always be available as spot instance availability is based on spare capacity. Furthermore, an in-use spot instance can be terminated by AWS whenever instance demand is high.

The cost of the Xilinx Virtex UltraScale+ VU9P development board (the VCU118) is $7,000 so, to give an indication of how the cost of the f1.2xlarge instance compares to the cost of the FPGA hardware, an f1.2xlarge instance must run constantly for about 177 days (or a spot instance for about 590 days) before the total cost of the instance becomes higher than the cost of the hardware. This suggests that, unless the FPGA hardware is utilised for a significant amount of time, the use of a cloud-based FPGA instance is more cost-effective than the purchase of FPGA hardware.

The compute-optimised 36-core c5.12xlarge CPU instance (with an instance cost of $2.239/hr and a spot cost of $0.6099/hr) is used as a reference in order to compare the cost of FPGA-based permutation testing to the cost of PLINK/CPU-based permutation testing using AWS EC2. Table 5.8 compares the cost of the FPGA accelerator running on an f1.2xlarge instance to the estimated cost of PLINK running on a c5.12xlarge instance for 1,000 maxT permutations of Dataset 2. As the CPU of the c5.12xlarge instance is 60% faster than the Intel Xeon Silver 4114 CPU used to test PLINK's permutation testing runtime (3.5 GHz vs 2.2 GHz), PLINK's processing time using a c5.12xlarge instance was estimated to be 113 hours (as opposed to the measured runtime of 164 hours).

Table 5.8: Table comparing the cost of FPGA-accelerated permutation testing using an EC2 f1.2xlarge FPGA instance to the estimated cost of PLINK running on an EC2 compute-optimised c5.12xlarge CPU instance for 1,000 maxT permutations of Dataset 2.

Instance	Runtime (h)	Instance cost ($)	Spot instance cost ($)
f1.2xlarge	0.37	0.61	0.18
c5.12xlarge	113.00	253.00	68.92

Table 5.8 shows that FPGA accelerator significantly reduces the cost of GWAS permutation testing on AWS — the acceleration provided by the FPGA helps to minimise the instance rental time which, in turn, minimises the cost of the instance. For 1,000 maxT permutations of Dataset 2, the cost of the FPGA instance is about 415 times less than the estimated cost of the CPU instance. Table 5.8 also indicates that spot instances can be used to further reduce the cost of FPGA-based permutation testing using EC2.

It should be noted that, for maxT permutation testing, a given dataset size will result in very similar runtimes (and, therefore, costs) for different datasets, whereas the runtime (and cost) of adaptive permutation testing for similarly sized datasets could vary slightly based on the number of significant SNPs.

5.4 Code Availability

All of the code developed for this research is stored in a GitHub repository (`https://github.com/witseie/fpgaperm`) together with instructions for users. The codebase is split into a hardware component (the Vitis HLS code used to compile the FPGA design together with various configuration files used to direct the Vivado build tools during compilation) and a software component (the C++ host application and a Python script which calculates phenotype residuals based on the presence of covariate data and runs the host application).

A Makefile is provided to automate the compilation of both the FPGA design (using Vitis HLS and Vivado) and the C++ host application (using GCC). A Linux CentOS AMI (Amazon Machine Image) is also provided in order to simplify the setup of an AWS FPGA instance. The AMI has all of the required software and libraries installed and a setup script is provided to initialise the environment variables required for the use of Xilinx runtime.

6 Conclusion

6.1 Summary

To prove that FPGAs can be used to accelerate bioinformatics workloads, this work studied the efficacy of FPGAs when applied to GWAS permutation testing. An FPGA-based accelerator was developed to accelerate two algorithms that are commonly used to reduce the computational burden of GWAS permutation testing: adaptive permutation testing and maxT permutation testing.

The accelerator is designed to run on an AWS EC2 f1.2xlarge instance consisting of a Xilinx Virtex UltraScale+ VU9P FPGA together with an 8-core Intel Xeon host CPU, so the permutation testing algorithm was split into a hardware component and a software component. As matrix-vector multiplication is the most computationally expensive part of the permutation testing algorithm, the FPGA was tasked only with accelerating the MVM computation, while the host CPU was used to handle data preprocessing, manage the FPGA execution as well as data transfers to/from the FPGA using the OpenCL framework and to implement the permutation testing algorithms.

This work showed that the adaptability of FPGAs can be utilised to develop highly optimised, massively parallel architectures to efficiently handle specific workloads. In the case of GWAS permutation testing, the compressed nature of genotypic data was exploited to increase the level of parallelism (allowing for the design of an MVM kernel which can perform 256 concurrent multiply-accumulate operations every clock cycle) and to reduce the data transfer volume between the host CPU and the FPGA. In order to maximise both the parallelism of the FPGA design and the global memory bandwidth, an FPGA design consisting of four identical MVM kernels was built using Xilinx Vivado 2020.2. When the default build settings were used, however, Vivado was unable to generate a design which could operate at the maximum frequency of 250 MHz due to high levels of congestion, so Vivado compiler directives were used to reduce congestion during all phases of the build process, which resulted in a final FPGA design which could operate at 250 MHz.

PLINK, a popular CPU-based tool that implements both maxT and adaptive permutation testing, was used as a reference with which to compare the speed and accuracy of the FPGA-based accelerator. For a real GWAS dataset of about 13.7 million genetic variants sampled from 3,652 individuals, PLINK running on 40 Intel Xeon 4114 CPU cores requires 164 hours (or approximately 7 days) for 1,000 maxT permutations and 204 hours (or around 8.5 days) for 100 million adaptive permutations. Conversely, the FPGA-based accelerator requires 22 minutes for 1,000 maxT permutations and 325 minutes for 100 million adaptive permutations resulting in speedups of 447 and 38 respectively. Furthermore, for 700 million adaptive permutations of the same dataset (an almost computationally unfeasible workload for PLINK running on a 40-core CPU) the runtime of the FPGA-based algorithm is 33.2 hours which is significantly lower than PLINK's runtime for 100 million adaptive permutations.

This work demonstrated that high-level synthesis can generate effective FPGA designs which means that, although some knowledge of the underlying hardware architecture is required, FPGA development can begin to be framed as a software design exercise rather

than a hardware design exercise. This may not be applicable to more complex workloads however, which may require an RTL-based design to achieve acceptable performance.

To conclude, this dissertation showed that FPGAs can significantly reduce the runtime of two GWAS permutation testing algorithms with little loss of accuracy (i.e. the associated genetic variants identified by PLINK were also shown to be identified by the FPGA-accelerated algorithms). FPGA acceleration reduced the runtime of GWAS permutation testing to the point where it can be measured in hours (or minutes) rather than in days or weeks. Furthermore, the use of FPGA acceleration enables the handling of workloads which are almost unfeasible for current CPU-based methods and perhaps most significantly, drastically reduces the energy consumption of GWAS permutation testing. This proves that FPGAs can be effectively employed as bioinformatics accelerators (at relatively low cost) in order to handle the large sample sizes of modern biological datasets thereby opening up new possibilities for bioinformatics research.

6.2 Future Work

While this work has demonstrated that FPGAs can effectively accelerate GWAS permutation testing for continuous phenotypes using simple linear regression, work can be done both to improve the functionality of the accelerator and to improve the performance.

As discussed in Section 3.2.2, the accelerator cannot currently handle sample sizes of more than 262,144 individuals due to the fixed phenotype buffer size. The phenotype buffer size could be increased, but this will result in an increase in circuit congestion which could limit the operating frequency of the design. A useful optimisation exercise would be to determine the largest phenotype buffer size that does not affect the operating frequency. The host application could also be adapted to perform partial dot products for large sample sizes that cannot be processed in a single invocation of an MVM kernel but this will increase the complexity of the data blocking algorithms.

Although significant speedup of adaptive permutation testing was achieved, this dissertation demonstrated that the efficacy of the FPGA-based adaptive permutation implementation is lower than that of the maxT implementation. Further work could be done to improve the efficacy of the FPGA-based adaptive permutation testing algorithm, but this will probably require a redesign of the FPGA part of the accelerator to process multiple permutations in a single invocation of a kernel.

Although this work touched on the use of GPUs as high-throughput accelerators, the effect of GPU acceleration on GWAS permutation testing has not been addressed. A comparison between GPU and FPGA-based acceleration of GWAS permutation testing should be undertaken to determine whether comparable speedup can be achieved with GPU acceleration. Additional work could include the analysis of the combined effect of FPGA and GPU acceleration.

Work can be done to investigate the use of multiple FPGAs to provide more acceleration. While this work showed that a single FPGA provides a significant amount of acceleration, AWS F1 instances with two and eight FPGAs are available. To investigate the effect of multiple FPGAs, the host application will need to be reworked to split the MVM calculation over different FPGA platforms.

The accelerator could be updated to support the analysis of case/control phenotypes in addition to continuous phenotypes. To handle case/control phenotypes in hardware

however, would require the development of a novel FPGA architecture as case/control phenotypes are analysed using logistic regression (or Fisher's exact test) rather than linear regression. Logistic regression is more complex than linear regression, so FPGA acceleration may not yield as high a speedup as that demonstrated in this work for the analysis of continuous phenotypes.

The use of FPGA acceleration for linear mixed model GWAS permutation testing should be investigated. LMM regression is used to account for genetically related individuals (which can bias the results of a GWAS) and the fact that LMM regression is significantly slower than simple linear regression means that there is a need for a linear mixed model permutation testing accelerator.

Bibliography

[1] W. S. Bush and J. H. Moore, "Genome-Wide Association Studies," in *PLoS Computational Biology*, 2012, vol. 8, no. 12, ch. 11, p. e1002822.

[2] Health Education England, "Genomics Education Programme," 2021. [Online]. Available: https://www.genomicseducation.hee.nhs.uk/image-library/

[3] A. Choudhury, S. Aron, L. R. Botigué, D. Sengupta, G. Botha, T. Bensellak, G. Wells, J. Kumuthini, D. Shriner, Y. J. Fakim, A. W. Ghoorah, E. Dareng, T. Odia, O. Falola, E. Adebiyi, S. Hazelhurst, G. Mazandu, O. A. Nyangiri, M. Mbiyavanga, A. Benkahla, S. K. Kassim, N. Mulder, S. N. Adebamowo, E. R. Chimusa, D. Muzny, G. Metcalf, R. A. Gibbs, C. Rotimi, M. Ramsay, A. A. Adeyemo, Z. Lombard, and N. A. Hanchard, "High-depth African genomes inform human migration and health," *Nature*, vol. 586, no. 7831, pp. 741–748, Oct. 2020.

[4] Xilinx, "UltraScale Architecture and Product Overview (DS890)," Tech. Rep., 2015. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[5] Xilinx, "UltraScale Architecture: Configurable Logic Block (UG574)," Tech. Rep., 2017. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[6] Xilinx, "UltraScale Architecture: DSP Slice User Guide (UG579)," Tech. Rep., 2020. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

[7] Xilinx, "Vitis High-Level Synthesis User Guide (UG1399)," Tech. Rep., 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf

[8] R. Che, J. R. Jack, A. A. Motsinger-Reif, and C. C. Brown, "An adaptive permutation approach for genome-wide association study: Evaluation and recommendations for use," *BioData Mining*, vol. 7, no. 1, pp. 1–13, Jun. 2014.

[9] A. Xue et al., "Genome-wide association analyses identify 143 risk variants and putative regulatory mechanisms for type 2 diabetes," *Nature Communications*, vol. 9, no. 1, pp. 1–14, Dec. 2018.

[10] A. Sud, B. Kinnersley, and R. S. Houlston, "Genome-wide association studies of cancer: Current insights and future perspectives," *Nature Reviews Cancer*, vol. 17, no. 11, pp. 692–704, Oct. 2017.

[11] B. Brachi, G. P. Morris, and J. O. Borevitz, "Genome-wide association studies in plants: The missing heritability is in the field," *Genome Biology*, vol. 12, no. 10, pp. 1–8, Oct. 2011.

[12] B. Hayes and M. Goddard, "Genome-wide association and genomic selection in animal breeding," *Genome*, vol. 53, no. 11, pp. 876–883, 2010.

[13] S. Dudoit and M. J. van der Laan, *Multiple testing procedures with applications to genomics*, ser. Springer Series in Statistics. Springer, 2008.

[14] C. C. Chang, C. C. Chow, L. C. Tellier, S. Vattikuti, S. M. Purcell, and J. J. Lee, "Second-generation PLINK: Rising to the challenge of larger and richer datasets," *GigaScience*, vol. 4, no. 1, 2015.

[15] X. Zhou and M. Stephens, "Genome-wide efficient mixed-model analysis for association studies," *Nature Genetics*, vol. 44, no. 7, pp. 821–824, 2012.

[16] C. Lippert, J. Listgarten, Y. Liu, C. M. Kadie, R. I. Davidson, and D. Heckerman, "FaST linear mixed models for genome-wide association studies," *Nature Methods*, vol. 8, no. 10, pp. 833–835, 2011.

[17] K. Sikorska, E. Lesaffre, P. F. Groenen, and P. H. Eilers, "GWAS on your notebook: Fast semi-parallel linear and logistic regression for genome-wide association studies," *BMC Bioinformatics*, vol. 14, no. 1, p. 166, 2013.

[18] N. T. Weeks, G. R. Luecke, B. M. Groth, M. Kraeva, L. Ma, L. M. Kramer, J. E. Koltes, and J. M. Reecy, "High-performance epistasis detection in quantitative trait GWAS," *International Journal of High Performance Computing Applications*, vol. 32, no. 3, pp. 321–336, 2018.

[19] L. Beyer and P. Bientinesi, "GWAS on GPUs: Streaming data from HDD for sustained performance," in *European Conference on Parallel Processing*. Springer, 2013, pp. 788–799.

[20] J. Gonzalez-Dominguez, L. Wienbrandt, J. C. Kaassens, D. Ellinghaus, M. Schimmler, and B. Schmidt, "Parallelizing epistasis detection in GWAS on FPGA and GPU-accelerated computing systems," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 12, no. 5, pp. 982–994, 2015.

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

[22] L. Wienbrandt, J. C. Kässens, M. Hübenthal, and D. Ellinghaus, "1000x faster than PLINK: Combined FPGA and GPU accelerators for logistic regression-based detection of epistasis," *Journal of Computational Science*, vol. 30, pp. 183–193, 2018.

[23] J. C. Kassens, L. Wienbrandt, M. Schimmler, J. Gonzalez-Dominguez, and B. Schmidt, "Combining GPU and FPGA technology for efficient exhaustive interaction analysis in GWAS," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors*, 2016, pp. 170–175.

[24] J. González-Domínguez, B. Schmidt, J. C. Kässens, and L. Wienbrandt, "Hybrid CPU/GPU acceleration of detection of 2-SNP epistatic interactions in GWAS," in *European Conference on Parallel Processing*. Springer, 2014, pp. 680–691.

[25] S. Gundlach, J. C. Kässens, and L. Wienbrandt, "Genome-wide association interaction studies with MB-MDR and maxT multiple testing correction on FPGAs," in *Procedia Computer Science*, vol. 80. Elsevier, Jan. 2016, pp. 639–649.

[26] G. Yang, W. Jiang, Q. Yang, and W. Yu, "PBOOST: A GPU-based tool for parallel permutation tests in genome-wide association studies," *Bioinformatics*, vol. 31,

no. 9, pp. 1460–1462, May 2015.

[27] A. Terada, H. Kim, and J. Sese, "High-speed Westfall-Young permutation procedure for genome-wide association studies," in *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, 2015, pp. 17–26.

[28] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[29] J. Teubner and L. Woods, "Data processing on fpgas," *Synthesis Lectures on Data Management*, vol. 5, no. 2, pp. 1–118, 2013.

[30] R. J. Klein, C. Zeiss, E. Y. Chew, J. Y. Tsai, R. S. Sackler, C. Haynes, A. K. Henning, J. P. SanGiovanni, S. M. Mane, S. T. Mayne, M. B. Bracken, F. L. Ferris, J. Ott, C. Barnstable, and J. Hoh, "Complement factor H polymorphism in age-related macular degeneration," *Science*, vol. 308, no. 5720, pp. 385–389, Apr. 2005.

[31] A. Buniello, J. A. Macarthur, M. Cerezo, L. W. Harris, J. Hayhurst, C. Malangone, A. McMahon, J. Morales, E. Mountjoy, E. Sollis, D. Suveges, O. Vrousgou, P. L. Whetzel, R. Amode, J. A. Guillen, H. S. Riat, S. J. Trevanion, P. Hall, H. Junkins, P. Flicek, T. Burdett, L. A. Hindorff, F. Cunningham, and H. Parkinson, "The NHGRI-EBI GWAS Catalog of published genome-wide association studies, targeted arrays and summary statistics 2019," *Nucleic Acids Research*, vol. 47, no. D1, pp. D1005–D1012, Jan. 2019.

[32] B. W. Kunkle et al., "Genetic meta-analysis of diagnosed Alzheimer's disease identifies new risk loci and implicates Aβ, tau, immunity and lipid processing," *Nature Genetics*, vol. 51, no. 3, pp. 414–430, Mar. 2019.

[33] A. R. Wood et al., "Defining the role of common variation in the genomic and biological architecture of adult human height," *Nature Genetics*, vol. 46, no. 11, pp. 1173–1186, Nov. 2014.

[34] A. E. Locke et al., "Genetic studies of body mass index yield new insights for obesity biology," *Nature*, vol. 518, no. 7538, pp. 197–206, Feb. 2015.

[35] J. E. Savage et al., "Genome-wide association meta-analysis in 269,867 individuals identifies new genetic and functional links to intelligence," *Nature Genetics*, vol. 50, no. 7, pp. 912–919, Jul. 2018.

[36] H. Lodish, A. Berk, C. A. Kaiser, M. Krieger, M. P. Scott, A. Bretscher, H. Ploegh, P. Matsudaira, and Others, *Molecular Cell Biology*. Macmillan, 2008.

[37] National Human Genome Research Institute, "Introduction to Genomics," 2019. [Online]. Available: https://www.genome.gov/About-Genomics/Introduction-to-Genomics

[38] A. Tan, G. R. Abecasis, and H. M. Kang, "Unified representation of genetic variants," *Bioinformatics*, vol. 31, no. 13, pp. 2202–2204, Jul. 2015.

[39] V. Tam, N. Patel, M. Turcotte, Y. Bossé, G. Paré, and D. Meyre, "Benefits and limitations of genome-wide association studies," *Nature Reviews Genetics*, vol. 20, no. 8, pp. 467–484, Aug. 2019.

[40] A. Korte and A. Farlow, "The advantages and limitations of trait analysis with gwas: a review," *Plant methods*, vol. 9, no. 1, pp. 1–9, 2013.

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

[42] A. Vignal, D. Milan, M. SanCristobal, and A. Eggen, "A review on snp and other types of molecular markers and their use in animal genetics," *Genetics selection evolution*, vol. 34, no. 3, pp. 275–305, 2002.

[43] S. T. Sherry, M. H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin, "dbSNP: The NCBI database of genetic variation," *Nucleic Acids Research*, vol. 29, no. 1, pp. 308–311, 2001.

[44] A. Auton et al., "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, Sep. 2015.

[45] D. Malhotra et al., "High frequencies of de novo CNVs in bipolar disorder and schizophrenia," *Neuron*, vol. 72, no. 6, pp. 951–963, Dec. 2011.

[46] E. Wheeler et al., "Genome-wide SNP and CNV analysis identifies common and low-frequency variants associated with severe early-onset obesity," *Nature Genetics*, vol. 45, no. 5, pp. 513–517, May 2013.

[47] J. W. Trampush et al., "GWAS meta-analysis reveals novel loci and genetic correlates for general cognitive function: A report from the COGENT consortium," *Molecular Psychiatry*, vol. 22, no. 3, pp. 336–345, Mar. 2017.

[48] A. Okbay et al., "Genetic variants associated with subjective well-being, depressive symptoms, and neuroticism identified through genome-wide analyses," *Nature Genetics*, vol. 48, no. 6, pp. 624–633, Jun. 2016.

[49] T. J. Hoffmann et al., "A large multiethnic genome-wide association study of prostate cancer identifies novel risk variants and substantial ethnic differences," *Cancer Discovery*, vol. 5, no. 8, pp. 878–891, Aug. 2015.

[50] J. Dai et al., "Genome-wide association study of INDELs identified four novel susceptibility loci associated with lung cancer risk," *International Journal of Cancer*, vol. 146, no. 10, pp. 2855–2864, May 2020.

[51] M. D. Ritchie and K. Van Steen, "The search for gene-gene interactions in genome-wide association studies: challenges in abundance of methods, practical considerations, and biological interpretation," *Annals of Translational Medicine*, vol. 6, no. 8, 2018.

[52] H3ABioNet, "H3Africa SNP Microarray," 2021. [Online]. Available: https://chipinfo.h3abionet.org/

[53] L. Yun, C. Willer, S. Sanna, and G. Abecasis, "Genotype imputation," *Annual Review of Genomics and Human Genetics*, vol. 10, pp. 387–406, Sep. 2009.

[54] G. M. Clarke, C. A. Anderson, F. H. Pettersson, L. R. Cardon, A. P. Morris, and K. T. Zondervan, "Basic statistical analysis in genetic case-control studies," *Nature protocols*, vol. 6, no. 2, pp. 121–133, 2011.

[55] R. M. Cantor, K. Lange, and J. S. Sinsheimer, "Prioritizing gwas results: a review of statistical methods and recommendations for their application," *The American Journal of Human Genetics*, vol. 86, no. 1, pp. 6–22, 2010.

[56] C. Tian, P. K. Gregersen, and M. F. Seldin, "Accounting for ancestry: Population substructure and genome-wide association studies," *Human Molecular Genetics*, vol. 17, no. R2, pp. R143–R150, Oct. 2008.

[57] P. Armitage and G. Berry, *Statistical Methods in Medical Research*. Wiley, 2002.

[58] Y. S. Aulchenko, D. J. De Koning, and C. Haley, "Genomewide rapid association using mixed model and regression: A fast and simple method for genomewide pedigree-based quantitative trait loci association analysis," *Genetics*, vol. 177, no. 1, pp. 577–585, Sep. 2007.

[59] H. M. Kang, J. H. Sul, S. K. Service, N. A. Zaitlen, S. Y. Kong, N. B. Freimer, C. Sabatti, and E. Eskin, "Variance component model to account for sample structure in genome-wide association studies," *Nature Genetics*, vol. 42, no. 4, pp. 348–354, 2010.

[60] M. K. Hyun, N. A. Zaitlen, C. M. Wade, A. Kirby, D. Heckerman, M. J. Daly, and E. Eskin, "Efficient control of population structure in model organism association mapping," *Genetics*, vol. 178, no. 3, pp. 1709–1723, 2008.

[61] J. Yang, B. Benyamin, B. P. McEvoy, S. Gordon, A. K. Henders, D. R. Nyholt, P. A. Madden, A. C. Heath, N. G. Martin, G. W. Montgomery, M. E. Goddard, and P. M. Visscher, "Common SNPs explain a large proportion of the heritability for human height," *Nature Genetics*, vol. 42, no. 7, pp. 565–569, 2010.

[62] A. E. Hoerl and R. W. Kennard, "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[63] P. Waldmann, G. Mészáros, B. Gredler, C. Fuerst, and J. Sölkner, "Evaluation of the lasso and the elastic net in genome-wide association studies," *Frontiers in Genetics*, vol. 4, no. 12, p. 270, Dec. 2013.

[64] R. De Vlaming and P. J. Groenen, "The current and future use of ridge regression for prediction in quantitative genetics," *BioMed Research International*, vol. 2015, 2015.

[65] R. Tibshirani, "Regression Shrinkage and Selection Via the Lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[66] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," Tech. Rep. 2, 2005.

[67] X. Zhou, P. Carbonetto, and M. Stephens, "Polygenic Modeling with Bayesian Sparse Linear Mixed Models," *PLoS Genetics*, vol. 9, no. 2, p. e1003264, 2013.

[68] S. Dudoit, J. P. Shaffer, and J. C. Boldrick, "Multiple hypothesis testing in microarray experiments," Tech. Rep. 1, 2003.

[69] S. Greenland, S. J. Senn, K. J. Rothman, J. B. Carlin, C. Poole, S. N. Goodman, and D. G. Altman, "Statistical tests, P values, confidence intervals, and power: a guide to misinterpretations," *European Journal of Epidemiology*, vol. 31, no. 4, pp. 337–350,

Apr. 2016.

[70] N. M. Laird and C. Lang, *The fundamentals of modern statistical genetics*. Springer Science & Business Media, 2010.

[71] Y. Benjamini and Y. Hochberg, "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[72] J. D. Storey, "A direct approach to false discovery rates," *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, vol. 64, no. 3, pp. 479–498, Aug. 2002.

[73] J. J. Goeman and A. Solari, "Multiple hypothesis testing in genomics," *Statistics in Medicine*, vol. 33, no. 11, pp. 1946–1978, 2014.

[74] I. Pe'er, R. Yelensky, D. Altshuler, and M. J. Daly, "Estimation of the multiple testing burden for genomewide association studies of nearly all common variants," *Genetic Epidemiology*, vol. 32, no. 4, pp. 381–385, May 2008.

[75] Sture Holm, "A Simple Sequentially Rejective Multiple Test Procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[76] Y. Hochberg, "A Sharper Bonferroni Procedure for Multiple Tests of Significance," *Biometrika*, vol. 75, no. 4, pp. 800–802, Dec. 1988.

[77] R. E. Blakesley, S. Mazumdar, M. A. Dew, P. R. Houck, G. Tang, C. F. R. Iii, and M. A. Butters, "Supplemental Material for Comparisons of Methods for Multiple Hypothesis Testing in Neuropsychological Research," *Neuropsychology*, vol. 23, no. 2, pp. 255–264, 2009.

[78] Y. Benjamini and D. Yekutieli, "The control of the false discovery rate in multiple testing under dependency," *Annals of Statistics*, vol. 29, no. 4, pp. 1165–1188, Aug. 2001.

[79] J. D. Storey, "A direct approach to false discovery rates," *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, vol. 64, no. 3, pp. 479–498, Aug. 2002.

[80] K. Korthauer, P. K. Kimes, C. Duvallet, A. Reyes, A. Subramanian, M. Teng, C. Shukla, E. J. Alm, and S. C. Hicks, "A practical guide to methods controlling false discoveries in computational biology," *Genome Biology*, vol. 20, no. 1, pp. 1–21, 2019.

[81] X. Zhang, S. Huang, W. Sun, and W. Wang, "Rapid and robust resampling-based multiple-testing correction with application in a genome-wide expression quantitative trait loci study," *Genetics*, vol. 190, no. 4, pp. 1511–1520, Apr. 2012.

[82] P. H. Westfall and S. S. Young, *Resampling-based multiple testing: Examples and methods for p-value adjustment*, 279th ed. John Wiley and Sons, Ltd, 1993.

[83] R. Pahl and H. Schäfer, "PERMORY: An LD-exploiting permutation test algorithm for powerful genome-wide association testing," *Bioinformatics*, vol. 26, no. 17, pp. 2093–2100, 2010.

[84] B. L. Browning, "PRESTO: Rapid calculation of order statistic distributions and multiple-testing adjusted P-values via permutation for one and two-stage genetic association studies," *BMC Bioinformatics*, vol. 9, 2008.

[85] J. A. Freudenthal, M. J. Ankenbrand, D. G. Grimm, and A. Korte, "GWAS-Flow: A GPU accelerated framework for efficient permutation based genome-wide association studies," *bioRxiv*, p. 783100, 2019.

[86] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2016. [Online]. Available: https://www.tensorflow.org/

[87] M. J. Zhang, J. Zou, and D. Tse, "Adaptive Monte Carlo multiple testing via multi-armed bandits," in *International Conference on Machine Learning*, 2019, pp. 7512–7522.

[88] I. Kuon, R. Tessier, and J. Rose, *FPGA Architecture: Survey and Challenges*. Now Publishers Inc, 2008.

[89] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *arXiv:1805.03648*, 2018.

[90] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming models for hybrid FPGA-CPU computational components: A missing link," *IEEE Micro*, vol. 24, no. 4, pp. 42–53, Jul. 2004.

[91] D. J. Smith, "VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C," in *33rd Design Automation Conference Proceedings*. IEEE, 1996, pp. 771–776.

[92] ARM, "DesignStart FPGA," 2020. [Online]. Available: https://www.arm.com/resources/designstart/designstart-fpga

[93] Xilinx, "Zynq-7000 SoC," pp. 1–25, 2018. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[94] Xilinx, "Alveo U280 Data Center Acceleration Card," 2019. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html

[95] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal FPGA matrix-vector multiplication architecture," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 9–16.

[96] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, New York, New York, USA, 2014, pp. 161–170.

[97] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A massively parallel FPGA-based coprocessor for support vector

machines," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 2009, pp. 115–122.

[98] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 2009, pp. 53–60.

[99] C. Kyrkou, C. S. Bouganis, T. Theocharides, and M. M. Polycarpou, "Embedded Hardware-Efficient Real-Time Classification With Cascade Support Vector Machines," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 1, pp. 99–112, 2016.

[100] M. B. Rabieah and C.-S. Bouganis, "FPGASVM: A Framework for Accelerating Kernelized Support Vector Machine," in *Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*. PMLR, 2016, pp. 68–84.

[101] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "FPGA-accelerated dense linear machine learning: A precision-convergence trade-off," in *2017 25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. Napa, CA: IEEE, Jun. 2017, pp. 160–167.

[102] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A customizable matrix multiplication framework for the intel HARPv2 Xeon+FPGA platform a deep learning case study," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, New York, NY, USA, Feb. 2018, pp. 107–116.

[103] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, "FPGASW: Accelerating Large-Scale Smith-Waterman Sequence Alignment Application with Backtracking on FPGA Linear Systolic Array," *Interdisciplinary Sciences: Computational Life Sciences*, vol. 10, no. 1, pp. 176–188, Mar. 2018.

[104] F. Xia, Y. Dou, G. Lei, and Y. Tan, "FPGA accelerator for protein secondary structure prediction based on the GOR algorithm," *BMC Bioinformatics*, vol. 12, no. 1, pp. 1–9, 2011.

[105] L. Wienbrandt, J. C. Kässens, and D. Ellinghaus, "Reference-based haplotype phasing with FPGAs," in *International Conference on Computational Science*, vol. 12139. Springer, 2020, pp. 481–495.

[106] Illumina, "Illumina DRAGEN Bio-IT Platform," 2021. [Online]. Available: https://emea.illumina.com/products/by-type/informatics-products/dragen-bio-it-platform.html