FPGAs in Bioinformatics

Implementation and Evaluation of Common Bioinformatics Algorithms in Reconfigurable Logic

Dipl.-Inf. Lars Wienbrandt

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2015

Kiel Computer Science Series (KCSS) 2016/2 v1.0 dated 2016-03-15

ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

The author can be contacted via http://www.techinf.informatik.uni-kiel.de

Published by the Department of Computer Science, Kiel University

Technical Computer Science Group

Please cite as:

▻ Lars Wienbrandt. *FPGAs in Bioinformatics* Number 2016/2 in Kiel Computer Science Series. Department of Computer Science, 2016. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Wienbrandt16,
  author    = {Lars Wienbrandt},
  title     = {{FPGAs in Bioinformatics}},
  publisher = {Department of Computer Science, Kiel University},
  year      = {2016},
  number    = {2016/2},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering,
               Kiel University.}
}
```

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. rer. nat. Manfred Schimmler Technical Computer Science Group Department of Computer Science Christian-Albrechts-University of Kiel
2. Gutachter:	Prof. Dr. rer. nat. Andre Franke Genetics & Bioinformatics Group Institute of Clinical Molecular Biology Christian-Albrechts-University of Kiel
3. Gutachter:	Prof. Dr. Olaf Wolkenhauer Systems Biology & Bioinformatics Group Institute of Computer Science University of Rostock

Datum der mündlichen Prüfung: 4. März 2016

Acknowledgments

FPGA technology has fascinated me already during my study of Computer Science. Soon, I got in touch with the area of bioinformatics and saw a great potential in the combination of both. After my graduation I continued research in this area and had the opportunities to publish many of my results. I was allowed to present most of them on several conferences around the world, and some of them even on invitation. Thus, first and foremost, I owe my special thanks to my supervisor Manfred Schimmler who always supported, promoted and motivated my work and gave me the freedom I needed including the approval for travel funding. He also encouraged me to apply for project fundings together with Andre Franke, whom I owe my grateful thanks for supporting me and my work in this manner. This successful project has finally funded my research in the last three years. And I owe my true appreciation to Bertil Schmidt as well, for the numerous lucrative publications and additional motivation and support in the second successfully raised collaboration project fundings that sustained the SNP interaction analysis project and financed Jan Kässens, who turned out to be my most cooperative colleague. His and my work perfectly complemented each other. With his knowledge in writing extremely efficient host code he always squeezed out the last bits of total system performance. With him developing the host part while I was working out the hardware design we perfectly minimized the development time for our applications. I sincerely thank him for not just being an invaluable colleague but also for being my friend.

Furthermore, I would like to thank my colleagues and former colleagues for their constructive help, namely Daniel Siebert for his counterpart on the BLASTp application, Jorge González-Domínguez and Jost Bissel for their work in the SNP interaction analysis project, especially Jorge for creating the competitive solutions on GPUs, Ayman Abbas who provided me insight in the fruitful FPGA application area of cryptanalysis, and Vasco Gross-

mann, Sven Koschnicke and Christoph Starke who tested FPGA technology in stock market analysis. I also sincerely thank Matthias Hübenthal and David Ellinghaus from the ICMB for their help on the mathematics of the SHAPEIT2 genotype phasing tool.

My special thanks go to our secretary Brigitte Scheidemann. She helped me with all kinds of administration tasks especially when I again had a complicated travel cost refund application. Moreover, I truly appreciate the valuable return of results from all students whom I supervised for their theses. And thank you, Dirk Nowotka, Reinhard Koch and again Andre Franke for joining my examination committee.

Last but not least, I want to thank my mother and my sister for unquestionably supporting me in all kinds of situations, and, of course, I truly appreciate the backing of my wife. She always encouraged me to go on, when I ran into a situation that seemed to be a dead end, she always told me to keep my head up when I lost my self-confidence, and she always calmed me when I thought I was going insane. Thank you, Ilze, for being there and that I can always rely on you!

And finally I do not want to forget to mention my little daughter Alina Annija, who is my sunshine, even when it is raining, and thus always reminds me of what is most important in life.

This study makes use of data generated by the Wellcome Trust Case-Control Consortium. A full list of the investigators who contributed to the generation of the data is available from http://www.wtccc.org.uk. Funding for the project was provided by the Wellcome Trust under award 076113 and 085475.

Zusammenfassung

Das Leben. Sehr viel Aufwand wird getrieben um der Menschheit einen Einblick in dieses faszinierende und komplexe, aber fundamentale Thema zu erlauben. Um Zusammenhänge zu verstehen und Folgen ableiten zu können hat der Mensch begonnen sein Genom zu sequenzieren, d.h. seine DNA zu bestimmen um daraus Informationen, z.B. in Bezug auf Erbkrankheiten folgern zu können. Der Prozess der DNA-Sequenzierung sowie die darauffolgenden Analysen sind schon allein wegen der riesigen Datenmengen eine Herausforderung für aktuelle Rechensysteme. Laufzeiten von über einen Tag für die Analyse einfacher Datensätze sind üblich, selbst wenn der Prozess bereits auf einem Computercluster ausgeführt wird.

Diese Arbeit zeigt, wie dieses gängige Problem im Bereich der Bioinformatik mit rekonfigurierbarer Hardware, speziell FPGAs, angegangen werden kann. Es werden drei rechenintensive Themengebiete hervorgehoben: *Sequenzalignment*, *SNP-Interaktionsanalyse* und *Genotyp-Imputation*.

Beispielhaft wird im Bereich des Sequenzalignments die Software *BLASTp* für die Suche in Proteinsequenzdatenbanken vorgestellt, implementiert und evaluiert. Die SNP-Interaktionsanalyse wird mit drei Verfahren zur vollständigen Suche von Interaktionen inklusive des dazugehörigen statistischen Tests vorgestellt: die Messung der *Kullback-Leibler-Divergenz* in *BOOST*, die *ρ-Differenz* in *iLOCi* und die Messung der *Transinformation*. Alle Verfahren werden auf FPGA-Hardware implementiert und evaluiert, mit einer bestechenden Beschleunigung im dreistelligen Bereich gegenüber Standard-Rechnern.

Das letzte Gebiet der Genotyp-Imputierung ist ein zweiteiliges Verfahren bestehend aus dem *Phasing* und der eigentlichen *Imputation*. Der Schwerpunkt liegt im Phasing-Schritt, der mit dem *SHAPEIT2*-Tool adressiert wird. SHAPEIT2 wird ausführlich mit den zugrunde liegenden mathematischen Methoden diskutiert, und schließlich implementiert und evaluiert. Auch hier wird ein beachtlicher Speedup von 46 erreicht.

Abstract

Life. Much effort is taken to grant humanity a little insight in this fascinating and complex but fundamental topic. In order to understand the relations and to derive consequences humans have begun to sequence their genomes, i.e. to determine their DNA sequences to infer information, e.g. related to genetic diseases. The process of DNA sequencing as well as subsequent analysis presents a computational challenge for recent computing systems due to the large amounts of data alone. Runtimes of more than one day for analysis of simple datasets are common, even if the process is already run on a CPU cluster.

This thesis shows how this general problem in the area of bioinformatics can be tackled with reconfigurable hardware, especially FPGAs. Three compute intensive problems are highlighted: *sequence alignment*, *SNP interaction analysis* and *genotype imputation*.

In the area of sequence alignment the software *BLASTp* for protein database searches is exemplarily presented, implemented and evaluated. SNP interaction analysis is presented with three applications performing an exhaustive search for interactions including the corresponding statistical tests: *BOOST*, *iLOCi* and the *mutual information* measurement. All applications are implemented in FPGA-hardware and evaluated, resulting in an impressive speedup of more than in three orders of magnitude when compared to standard computers.

The last topic of genotype imputation presents a two-step process composed of the *phasing* step and the actual *imputation* step. The focus lies on the phasing step which is targeted by the *SHAPEIT2* application. SHAPEIT2 is discussed with its underlying mathematical methods in detail, and finally implemented and evaluated. A remarkable speedup of 46 is reached here as well.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Life. Biologists have tackled questions regarding this fascinating and complex but fundamental topic ever since. The foundations of modern biology were made by Charles Darwin already in 1859 when he published his theory in evolutionary biology [Dar59], but almost 100 years had to pass until Watson and Crick first discovered the double-helix structure of DNA in 1953 [WC53]. Today, many biological processes involving DNA are revealed and humanity is able to gather lots of information hidden in DNA. Especially one species is target of recent extensive DNA analysis – humans themselves. In 1990, the *Human Genome Project (HGP)* started with the goal to completely sequence the human genome as well as to locate all human genes [HGP]. The project officially considered the sequencing complete when the full sequence was published in April 2003, only 50 years after Watson and Crick actually discovered the structure of DNA. Today, current sequencing machines are able to sequence ten complete human genomes in less than three days [Ill15a], but the identification of all genes is still subject to follow-up projects.

Biomedical Background

The whole genome is part of every living cell. For humans, it is encoded in DNA and distributed over 22 chromosomes that occur in two copies and two sex chromosomes (one copy inherited from the father and one copy from the mother). A DNA sequence consists of the four characters `A`, `C`, `G`, and `T`, since these reflect the four possible nucleotide bases *adenine*, *cytosine*, *guanine*, and *thymine* in DNA. For RNA sequences the character `T` is replaced by `U` since RNA contains *uracil* rather than thymine. While RNA

always forms a single strand, DNA appears in a more stable double helix form consisting of two complementary strands, i.e. both strands contain redundant information and one strand forms the *reverse complement* of the other following the simple rule that always bases A and T, and bases C and G face each other. Without redundancies, the complete unique human genomic DNA sequence is more than 3 billion bases long.

Furthermore, each cell carries the mechanisms to copy a cell, i.e. also to copy the genome, and to create proteins from genes in the genome. Genes are partial sequences of the genome which carry the essential information on all organic processes in an organism, but they state only less than 2% of the whole human genome [HGSC01]. The *transcription* process copies the information encoded in the gene to a *messenger RNA (mRNA)* strand, which is then used by the *translation* process to synthesize proteins from a sequence of amino acids encoded as sequence of base triplets (*codons*) in the mRNA. Proteins in-turn serve as construction material for all kinds of cells including blood cells and hemoglobin, hair, horn, etc., and act as messenger to control biological processes in an organism.

The mechanism of transcription and translation is called *protein biosynthesis* and is a fundamental process of life. The process is controlled by the *genetic code* that is almost equal among all known organisms on Earth. It encodes which codons are translated into which amino acids, and can simply be seen as a function with all 64 possible codon triplets as input and an amino acid output. (For more details, refer to the illustration in Fig. 2.4 in Chapter 2.)

The human genome is almost equal from person to person, but little differences in our genes make us genetically different. For example our eye color or the color of our hair is encoded in our genes. It is also possible that genes are affected by mutations. Although the naturally evolved genetic code contains a lot of redundancies, i.e. different base triplets still describe the same amino acid to be inserted in a synthesized protein, some mutations of single bases may result in a completely different translation result, e.g. a mutation of the triplet AAG, describing lysine, to triplet UAG describing a stop codon leads to a premature stop of the translation process generally resulting in a non-functional or wrongly functional protein. A mutation of the start codon AUG even results in the protein not to be synthesized at all.

These gene defects may lead to certain diseases, such as cystic fibrosis or sickle-cell disease, and are hereditary.

Of course, biologists and medical scientists wish to reveal those genes responsible for certain *phenotypes* (i.e. a visible or observable characteristic) with the goal to better understand the nature of the problem such that they may be able to address it one day e.g. with personalized medicine. For the sickle-cell disease example, scientists have already found out that this illness results from a single mutation (i.e. a *Single Nucleotide Polymorphism (SNP)*) of a GAG codon describing glutamic acid to a GUG codon describing valine in the β-globin gene located on chromosome 11 [SX13]. However, this was only one of countless secrets still hidden in the human genome.

Hence, high effort is invested into all kinds of genetic analysis, which starts with gathering genetic information from a number of samples in the first place, e.g. from direct DNA sequencing or genotyping of individuals. This information can then be used for the correlation of this information to the phenotypes of the individuals. Typically, one part of the study contains those samples which are affected by the phenotype (*cases*), and the other part contains a number of unaffected samples (*controls*). This states a classical case-control scenario which can be tackled by statistical mathematical analysis, as it is demonstrated in Chapter 5.

When it came to sequencing DNA information from humans, mankind had to deal with many ethical questions on how deep science may look into one of the most private areas of a person – its DNA – with the potential to reveal all information on ancestry, potential illnesses and potential physical wealth or weaknesses. However, DNA sequence or collecting genotype information of individuals is done very extensively. Data can now be easily collected with todays available techniques and made publicly available through the Internet. This immediately raises the next important question: Is this data secure and who is allowed to access which part of the data? Todays labs have to ensure that collecting and storing genetic data in common studies conforms with laws on data security. Usually, only those information is stored which is necessary for the current study, but it is often subject to the participants which information they allow to be published. Meta-data may only be one bit of information if the phenotype is only stated as “affected” or “unaffected”, or it contains more fine-grained information

that may be all kinds of personal information, e.g. sex, origin, height, weight, eye color, nutrition information, behavior, physical wealth, etc. Preferably, studies are pseudonymous, i.e. the name, address and other information that allow a direct identification of the participants, is not collected or ideally stored in a way that it is not possible to recover it from the other data. However, further questions regarding data security are to be discussed elsewhere.

High-Performance Computing and Reconfigurable Hardware

Computer science plays a vital role in the analysis of genetic data. As already mentioned, the size of the human genome is more than 3 billion base pairs distributed over 22 chromosomes plus two sex chromosomes, and common genotyping microarrays are able to quickly type a sample at more than 900,000 markers [Aff09]. Yet, the DNA sequencing process has evolved very quickly. As the Human Genome Project once required thirteen years to completely sequence a human genome at a cost of 3 billion US-$, todays sequencing machines, such as Illumina HiSeq X Ten, are able to sequence ten whole human genomes in less than 3 days for the cost of about 1,000 US-$ per genome [Ill15a]. These numbers alone indicate that computer scientists are required to develop efficient algorithms and tools to ensure analysis to be practical and in reasonable time.

However, with this ever-growing amount of genetic data the challenge for computer scientists gets harder. The algorithmic performance is not enough to keep up with the likewise growing needs of computational power. Thus, parallel computing and the development of parallel algorithms have been established to address this problem, and bioinformatics became a new application area in high-performance computing.

Implicitly parallelizable problems can be addressed by CPU cluster architectures with low effort. Additional libraries, such as MPI [MPI] or UPC++ [ZKD+14], based on established high-level programming languages (e.g. C++), can be used to adapt existing software to run on high-performance cluster systems, such as the SNP interaction detection software BOOST on a Cray XC30 supercomputer with 12,288 cores [KGW+14]. However, parallelization on CPU clusters is an easy and flexible way to gain

more computing power, but comes with obvious disadvantages. Featuring a general purpose instruction set requires resources for each instruction to be available, even if they might not be required in the targeted application. And furthermore, a linear increase in speed comes with at least the same linear investments in equipment, space for housing, energy and maintenance, such that doubling the computational power means at least doubling the costs.

In order to address the latter drawback, runtime improvement through parallelization without significant increases in housing and energy consumption can be achieved by applying a more specialized architecture, such as a GPU. GPUs were intentionally designed for graphics processing, but with fitting the computational problem into this highly parallel architecture runtime can significantly be reduced by taking advantage of the thousands of shader cores inside such a device [GWK+15; KLL+12; LSM10; LWS13; LSM12; LSM11a; LSM11b; SSL+]. For example the Nvidia GeForce GTX Titan Z features 5,760 CUDA cores inside a single processing unit [NVIb].

However, this thesis focuses on the application of reconfigurable hardware, in particular FPGAs, to speedup computational intensive problems in bioinformatics. The advantages of FPGAs emerge when when it comes to computational intensive tasks where the same or similar process is repeated over and over again. The goal is to implement this particular process as a hardware design in a single efficient and resource optimized processing element. This addresses the first drawback of CPUs, where the general purpose instruction set occupies resources which might not be required for the application at all. Since FPGAs are freely configurable (within resource limits) it is usually possible to distribute the available resources only for tasks required for the computational problem. Thus, generally a large number of processing elements can be implemented on a single chip, such that it is very likely to outperform a single CPU with this approach. By harnessing an FPGA-cluster, such as the RIVYERA architecture [SE], the same performance as a CPU cluster can be reached, but RIVYERA only requires the housing of a simple network server and its energy consumption is comparable to a few desktop PC systems.

Three main areas of bioinformatics with compute intensive applications are presented and addressed in this thesis by high-performance comput-

ing using FPGAs: sequence analysis, detection of gene interactions and genotype imputation.

Sequence Analysis

The high throughput of todays DNA sequencing machines generates sequences in the area of several gigabases per hour. In particular, an Illumina HiSeq X Ten generates up to 1,800 gigabases per run in three days, with 10 runs processing in parallel [Ill15a]. The output sequences are referred to as *short-reads* and present short subsequences of the input DNA sample, e.g. a whole genome. For Illumina HiSeq X, the reads are typically 2×150 base pairs long, with characters over the alphabet `A`, `C`, `G`, and `T`, but 6 billion of them are generated in a single sequencer run [Ill15a]. In order to handle this amount of data efficient algorithms and hardware have to ensure data processing in-time. Raw sequence data can be de-novo assembled to create DNA sequences of the sample, e.g. whole chromosomes up to the whole genome, or it is used by short-read aligners which align the reads to a reference genome, e.g. for the detection of SNPs. Many algorithms have evolved for this purpose, e.g. Velvet [ZB08] and ABySS [SWJ+09], two popular tools for de-novo assembly. They introduce a De-Bruijn graph data structure to link all reads. Graph simplifying, loop identification and the detection of *contigs* (i.e. *contiguous sequences*) are the main tasks of these tools. Scaffolding then uses *mate-pair* information of reads to generate longer contigs, which form the output of the assembly algorithm. The *mate-pair* information is a meta-information from sequencing machines that says which two reads are sequenced together and thus form a *mate-pair* from which it is known how the approximate gap distance is.

If a reference sequence exists, e.g. in the case of humans the current commonly used reference is *GRCh38* from December 2013 [NCB13], it is much faster to align the reads against this reference instead of a new assembly. BWA [LD09], Bowtie [LTP+09] and CUSHAW [LSM12] present three popular tools for short-read alignment. For a fast lookup they use the Burrows-Wheeler transformation [BW95] and the FM-index [FM00] to find exact and inexact matches of reads in a reference sequence. Other data structures, e.g. lookup-tables [RL10], can also be used, but the runtime of

the alignments is generally much longer. Fast short-read alignment tools provide heuristic alignments, i.e. they find sufficient alignments, but do not guarantee to find the best, i.e. an *optimal*, alignment. The common way to find optimal alignments is via dynamic programming, which is done by the popular Smith-Waterman [SW81] and Needleman-Wunsch [NW70] algorithms. The drawback of finding an optimal alignment is the disproportional longer runtime. While it is possible to align 14.5 million read pairs of length 76bp in about 4 hours with BWA on a standard computer, the Smith-Waterman alignment of the same dataset would take almost two weeks even with a special high-performance architecture [Wie13; Wie14].

Another compute intensive problem related to sequence analysis are simple sequence database lookups. From the extensive sequencing throughout all species, especially human, bacterial and viral genomes, large databases have evolved. A prominent example is the still-growing UniprotKB/TrEMBL [EMB] database for protein sequences containing 52.8 million sequences with a total of 17.5 billion amino acids (in late 2015). A usual query to this database is a single or a set of other protein sequences. The goal is to find exact or similar sequences in the database to learn more about the query set by e.g. suggesting its functionality or effect. Thus, the query sequences have to be aligned to the complete database sequences. For this special task the software suite *BLAST* [AGM+90] has evolved, with *BLASTn* and *BLASTp* the most frequently used tools within this suite, maintained by the *National Center for Biotechnology Information (NCBI)* [NCBa]. BLASTp generates heuristic alignments of a set of protein sequences against a protein sequence database while BLASTn performs the same task for nucleotide sequences, i.e. DNA or RNA sequences. Other tools, such as *BLASTx*, *tBLASTn*, *tBLASTx* and *PSI-BLAST* [AMS+97], allow the search of nucleotide sequences in protein databases or the other way round by translating the query or database according to the genetic code, or to perform iterative searches.

Since the permanent growth of the databases requires more computing power to perform alignments in feasible runtimes, the application of high-performance computing is indispensable. This thesis shows how FPGA technology can be applied to tackle this task in Chapter 4.

Detecting Gene Interactions

The particular DNA sequence is not always required for certain studies. It is often faster and cheaper to determine the *genotype* of a sample at specific marker positions with the help of so-called *microarrays*, e.g. *GeneChips* by Affymetrix [Aff09]. Since the specific base information is often not required, the genotype output can be encoded as either 0 for the *homozygous wild* type, 1 for *heterozygous*, and 2 for the *heterozygous variant* type. Since each human cell contains two copies of each chromosome (humans are a *diploid* species), the terms *homozygous* and *heterozygous* refer to the equality or unequality of the DNA bases at the specific marker on both chromosomes respectively. The *wild* type is the base commonly encountered at this position, while the *variant* type means any other base.

Microarray data allows the direct identification of gene defects, if the nature of the defect is already known. However, the opposite direction cannot be followed using only a few samples. In order to detect a possible correlation of a certain phenotype, e.g. a disease, to gene mutations, a large number of study participants have to be genotyped at as many markers as possible (usually across the whole genome), thus, generating a huge amount of genetic data. These studies are referred to as *Genome-Wide Association Studies (GWAS)* and are subject to a subsequent statistical analysis. Popular GWAS are published by the *Wellcome Trust Case Control Consortium (WTCCC)* that genotyped about 2,000 people for each of seven diseases together with 3,000 shared controls at about 500,000 markers [WTCCC07].

Analysis of this data can be very compute intensive, especially if the disease is not caused by a single genetic mutation. If more than one gene is involved in the expression of a certain phenotype, a *joint genetic effect*, such as *epistasis*, can only reliably be detected by analyzing combinations of genotypes. Epistasis describes the effect that the allele of one gene is masked by the presence or absence of other genes resulting in a different phenotype. Hence, in order to detect a *second-order* or pairwise epistatic effect, all possible marker pairs of a study (also *SNP pairs*) have to be analyzed with a statistical test. For the WTCCC example, a number of $\binom{500{,}000}{2} \approx 125$ billion tests including determining correlated frequencies for all 2,000 cases for the certain disease and all 3,000 controls for each SNP

pair have to be performed. Independent of the form of the test, the sheer number indicates the computational challenge for such an analysis.

A number of tools exist which perform the association test with different statistical approaches, such as the *mutual information* measure or the *Kullback-Leibler divergence*. Among them are BOOST [WYY+10a], iLOCi [PNI+12], MDR [RHR+01] and MB-MDR [CCD+11; LJG+13] which perform the exhaustive test of all possible pairwise SNP combinations. Other tools, such as SNPRuler [WYY+10b], do some kind of pre-filtering to reduce the number of pairwise tests to be performed, but accepting the risk of losing the most significant SNP pair.

In Chapter 5 this thesis addresses the SNP interaction detection problem by implementing the exhaustive methods BOOST and iLOCi on FPGAs with a speedup of several orders of magnitude on the RIVYERA architecture when compared to a standard PC. Furthermore, the mutual information measure for third-order SNP interactions is introduced on FPGA-technology gaining a speedup of 182 on a single Kintex7 FPGA.

Genotype Imputation

Using SNP interaction detection techniques to reveal correlations between several genes is a common approach. However, genotyping ignores a person's *diplotype*, i.e. the heterozygous type does not denote which one of the two diploid chromosomes carries the variant type and which one carries the wild type. This information becomes significant especially if the person carries several variants on one chromosome. With the background of gene interactions it is important to know which variants are located on the one chromosome and which ones on the other. Studies have correlated specific haplotypes with drug response, clinical outcomes in transplantations [PMG+07] and susceptibility or resistance to diseases [JSF72]. Furthermore, large parts of genotype information might be missing, either when the quality of the microarray's results is low or especially if certain parts of the genome where not genotyped at all.

Both problems can be approached by mathematical statistics in a computational process called *genotype imputation*. This process is generally divided into two parts: *haplotype phasing* and *imputation*. Phasing determines the

phase, i.e. the missing diplotype information of each genotype typed in the study and is able to impute single unknown genotypes if they are typed by other samples. Imputation fills the gaps of unknown genotypes which are not typed in the study using a reference panel. Clearly, the imputation process may only fill those markers which are missing in the study, but are typed in the reference. Both parts commonly use a *Hidden Markov model (HMM)* as mathematical model and generate a result through various iterations. The iterations can be processed interleaved, i.e. alternating between phasing and imputation, or the imputation step takes the final result of the phasing part as input.

SHAPEIT2 [DZM13] is a recent popular tool that addresses the phasing part of genotype imputation. The following imputation using the SHAPEIT2 output can be done with Minimac2 [FAH15], Minimac3 [Abe] and IMPUTEv2 [HDM09] among others. IMPUTEv2 also supports interleaved phasing and imputation per iteration as mentioned above. Its predecessor IMPUTE [MHM+07] was able to impute missing genotypes from a reference panel without preliminary phasing. However, with the emergence of IMPUTEv2 the authors admitted that the preliminary phasing step significantly improves result quality [HDM09].

Furthermore, imputation quality clearly relies on the quality of the reference panel, i.e. previously accurately phased individuals [HMS11]. The *Haplotype Reference Consortium (HRC)* provides reference panels with so far 64,976 haplotypes at 39,235,157 SNPs, and it is still collecting further information from different studies with participants of predominantly European ancestry [HRC]. With the on-going growth of the reference the computational demands increase likewise for the genotype imputation process. Current runtimes are already in the area of at least several days or weeks for a whole-genome imputation on standard computer systems. Generally, computer clusters are addressed to speed up the process. This thesis shows in Chapter 6 how at least the phasing part can be significantly accelerated by harnessing FPGA technology while the imputation part remains for future work.

Structure of this Thesis

The complete structure of this thesis is summarized as follows.

The necessary biomedical background information is described in detail in Chapter 2. It shortly explains the fundamentals of DNA and protein biosynthesis. *Next Generation Sequencing (NGS)* as up-to-date standard method for DNA sequencing is introduced as well as the needs for sequence alignment techniques. Furthermore, terms and expressions such as *genotypes*, *haplotypes*, *alleles* and *SNPs* are explained as well as the method of genotyping with the help of *microarrays*, such as Affymetrix GeneChips [Aff09].

FPGA technology is introduced in Chapter 3. It shows the basic structure of reconfigurable hardware and the common FPGA design flow. The focus lies on Xilinx Spartan6 and 7-series FPGAs as well as the Xilinx development platforms *ISE* and *Vivado*. Furthermore, the hardware description language *VHDL* is introduced and the architectures of the applied hardware is presented, i.e. the structure and API of the RIVYERA hardware platform consisting of 128 Spartan6 FPGAs as well as the Xilinx KC705 development board containing one Kintex7 FPGA.

The series of presented FPGA applications in the area of bioinformatics begins with the topic of sequence alignment in Chapter 4. It presents an FPGA-based solution for the popular protein sequence alignment tool *BLASTp* with gapped alignment [AMS+97] on the RIVYERA architecture. This chapter contains the introduction of the BLASTp algorithm, implementation details and the evaluation of the FPGA-based solution in comparison to standard PCs. The achieved speedup of the RIVYERA is about 23.5 when compared to two Xeon quad-core processors.

The thesis continues with introducing the problem of detecting SNP interactions in Chapter 5. After explaining the necessary statistical methods for this compute intensive task, the implementations of the interaction detection tools *BOOST* [WYY+10a] and *iLOCi* [PNI+12] on the RIVYERA architecture for pairwise interactions as well as the mutual information measure on the KC705 development board for third-order interactions are presented. An evaluation of each tool compares the runtimes to standard PCs and, where applicable, to known GPU implementations. The RIVYERA architecture is able to reduce the runtime from about 19 hours on an Intel

Core i7 hexa-core processor to below 6 minutes in the case of BOOST and to below 3 minutes in the case of iLOCi for a WTCCC [WTCCC07] dataset. The speedup for the mutual information measurement on the KC705 board is about 182 when compared to the same CPU system.

Chapter 6 presents the FPGA implementation of the tool *SHAPEIT2* [DZM13] for genotype phasing on the RIVYERA architecture. It begins with the presentation of the mathematical background of a *Hidden Markov model* and the *forward-backward* procedure. The evaluation concentrates on the runtime improvements of the phasing process when compared to an Intel Core i7 quad-core PC. The achieved speedup was up to 46. The chapter also gives an introduction on the imputation part of this task with the tool *IMPUTE* [MHM+07] as an example.

The thesis closes with the conclusion and an insight into subjects of future work in Chapter 7, while Appendix A finally presents a resume of the author including his publication list.

The reader might be kindly reminded that the objective of this thesis is clearly not the biological or medical interpretation of results.

Chapter 2

Biomedical Background

2.1 Introduction

This chapter briefly explains the biomedical background required for this thesis. It concentrates on the basic biological knowledge on the nature of DNA and proteins as well as the involved fundamental processes that help understanding the nature of the presented applications. It introduces basic expressions and definitions from the biomedical environment these applications belong to (see Sect. 2.2).

Furthermore, it explains how the data that is analyzed by the applications is collected in the first place. In particular, it describes how DNA sequences are digitalized with up-to-data DNA sequencing machines (see Sect. 2.3) and how common genotyping methods work with microarrays (see Sect. 2.4).

2.2 DNA and Proteins

2.2.1 DNA

Deoxyribonucleic acid (DNA) is the information carrier for the *genome* of an organism. The genome is located in every living cell, and cellular life is classified into three domains: *Archaea*, *Bacteria* and *Eukaryota*. Species from the first two domains are *prokaryotes*, i.e. their cells do not have a nucleus and their DNA is ring-shaped. In contrast, *eukaryotes* possess a nucleus that contains the complete genome in the form of chromosomes. Species of the Eukaryota domain are the only ones who are able to develop multi-cellular

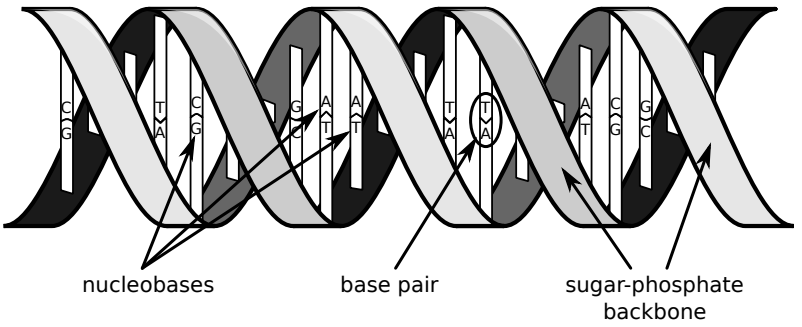


Figure 2.1. The double-helix structure of DNA.

organisms. Hence, humans belong to the Eukaryota as well.

Chemically, DNA is a *polynucleotide* composed of *nucleotides*, while each nucleotide consists of one of the four *nucleobases*, either *adenine* (`A`), *cytosine* (`C`), *guanine* (`G`) or *thymine* (`T`), as well as *deoxyribose* and a *phosphate group*. The nucleotides are bound by chemical bonds between the sugar and phosphate group of neighboring nucleotides, forming the *sugar-phosphate backbone*. The information in DNA is read from the sequence of its nucleobases, i.e. a sequence over the alphabet `A`, `C`, `G` and `T`. The direction is defined by reading from the DNA's *5′* end, having a terminal phosphate group, to its *3′* end, having a hydroxyl group from the sugar.

In general, DNA does not occur as a single strand. Instead, it forms a double helix together with its *reverse complement* (see Fig. 2.1). In this double helix structure the two bases facing each other are well-defined. It is always adenine and thymine forming a *base pair* as well as cytosine and guanine. Thus, the reverse complement contains exactly the same information as the other strand, only that the complementary bases have to be read, and from the fact that the 5′ end of the reverse complement is located where the 3′ end of the original strand is, the strand has to be read backwards to get the same nucleotide sequence as in the other strand. For example the DNA sequences in Fig. 2.2 form a reverse complement pair. It is also possible that a DNA sequence forms the reverse complement of itself. Such a sequence is called a *palindrome*, e.g. the simple sequence `ACGT` is palindromic.

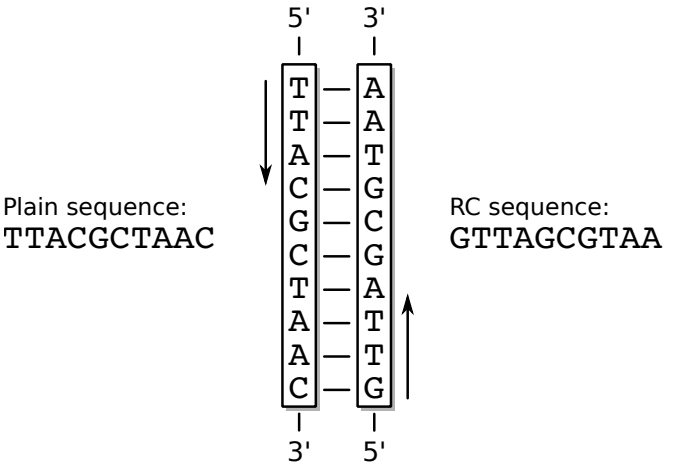


Figure 2.2. The reverse complement of a DNA sequence. The read direction is from the 5′ end to the 3′ end (indicated by arrows).

2.2.2 Protein Biosynthesis: Transcription and Translation

One fundamental biological process with DNA involved is *protein biosynthesis*. This process is also referred to as *gene expression* and is divided into two main subprocesses *transcription* and *translation*. Transcription describes the transfer of genetic information encoded in a gene from the genomic DNA into *messenger RNA (mRNA)*. RNA, just as DNA, is a polynucleotide with each nucleotide consisting of the same four DNA nucleobases with one exception, the nucleobase thymine is replaced by the similar *uracil (U)*. Furthermore, the sugar-phosphate-backbone contains *ribose* instead of deoxyribose, and RNA occurs single stranded.

The transcription process copies a gene, i.e. a part of the genomic DNA, into a complementary mRNA strand. The mRNA is able to leave the nucleus and will be further processed by *ribosomes* outside the nucleus. A ribosome is an enzyme which finally produces a protein from an mRNA strand in the translation process. For this purpose, it reads the sequence of base triplets from the mRNA that encodes a sequence of amino acid which are

then connected in a chain to actually form the protein. Proteins in-turn serve as construction material for all kinds of cells including blood cells and hemoglobin, hair, horn, etc., and act as messenger to control biological processes in an organism. Protein biosynthesis is illustrated in Fig. 2.3.

The code that correlates a nucleobase triplet, referred to as *codon*, to an amino acid is called the *genetic code*. This code is, with only a few aberrations, identical among all known species on Earth. It can simply be seen as a function from all 64 possible codons to a set of 22 amino acids plus a "stop command". The translation process always starts with the start codon `AUG` describing the amino acid *methionine*, and it stops if a "stop command" is read, i.e. one of the three stop codons `UAA`, `UAG` and `UGA` that do not describe an amino acid. The genetic code is illustrated in Fig.2.4.

2.2.3 The Human Genome

History of DNA knowledge is quite young. Friedrich Miescher identified and isolated DNA first in 1869, but Watson and Crick discovered its double helix structure not until 1953 [WC53], and only in 1990 the first attempt to sequence the human genome started with the *Human Genome Project (HGP)* [HGP]. Today, it is known that the human genome sequence contains about 3 billion base pairs, distributed over 22 regular and 2 sex chromosomes. The human species is *diploid*, i.e. each chromosome occurs in two copies with one exception regarding the sex chromosomes. A human female usually carries two *X* chromosomes while human males have one *X* and one *Y* chromosome. Hence, every living human cell carries 23 chromosome pairs resulting in a total of 46 chromosomes. An interesting fact about eukaryotic cells in general is, that the mitochondrium, a small cell organelle that converts energy from food into a format the cell can use, contains its own *mitochondrial DNA* which is about 16,600 base pairs long for human mitochondria. Furthermore, a recent observation is that the human genome contains between 20,000 and 25,000 genes encoded in only 1.5% of the genomic DNA [HGSC01; GBM+06]. The latest human reference genome (*GRCh38*) was released at December 24th in 2013 [NCB13].

The understanding of biological processes helps to deduce the nature of certain diseases from this knowledge. For example, although the genetic

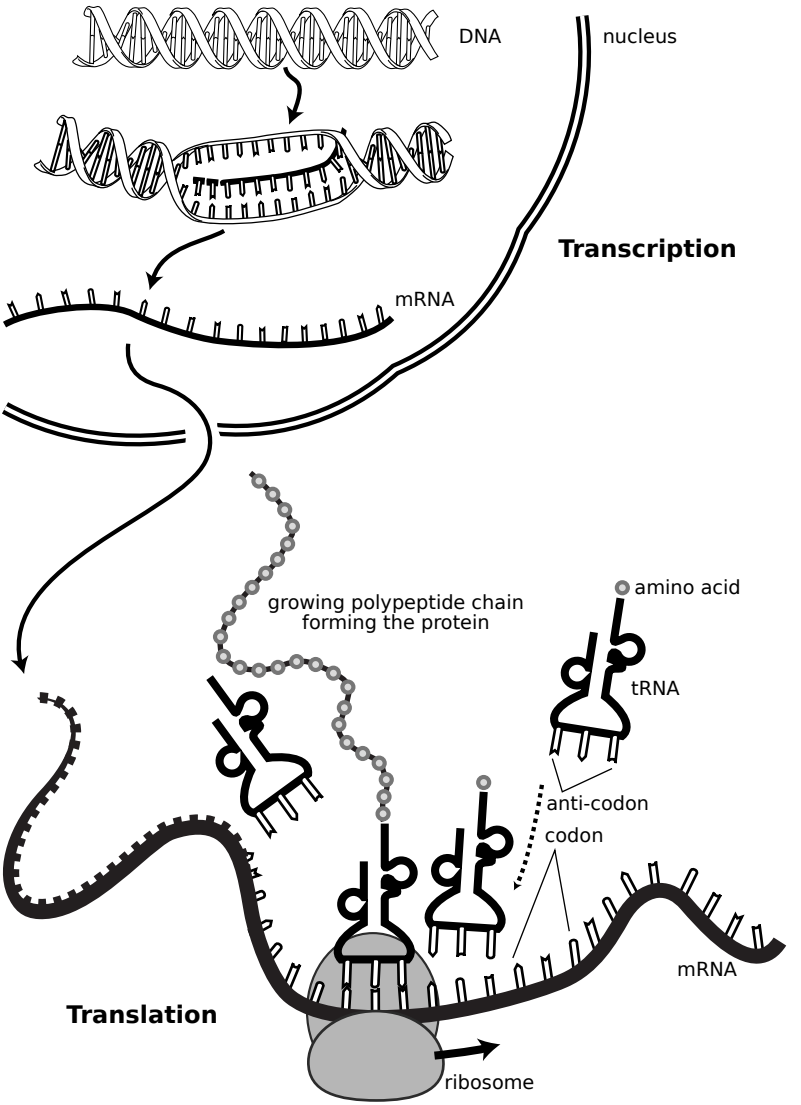


Figure 2.3. Protein biosynthesis.

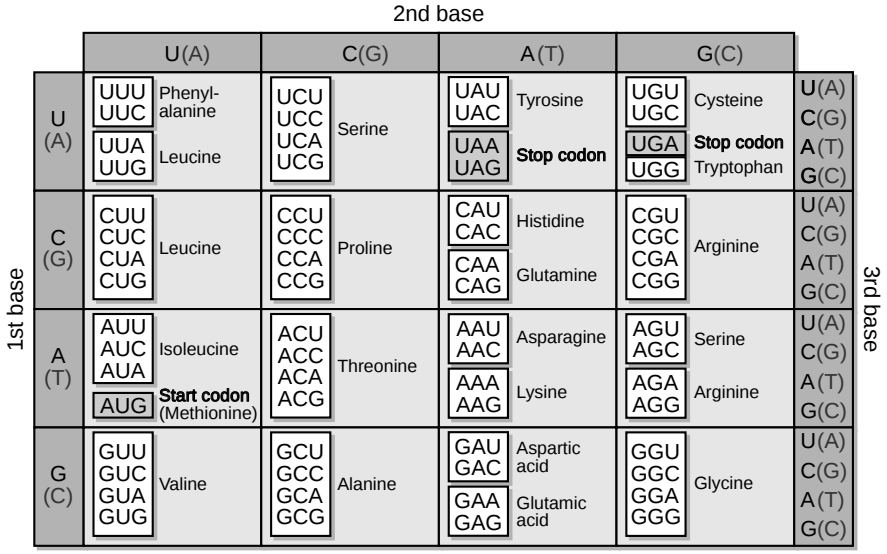


Figure 2.4. The genetic code describes which RNA codon is translated into which amino acid during protein biosynthesis.

code contains many redundancies, a single mutation of a nucleobase in the gene may result in fatal results. Highly susceptible for gene defects are mutations of the start or stop codons. If the start codon mutates to any other codon, the corresponding gene will not be expressed at all, i.e. the transcription result contains no start codon for the ribosomes to begin with the translation. In the opposite, a missing stop codon creates most likely unusable proteins that result from stopping the translation process only when reading the mRNA strand has finished. Note, that the protein encoding information only has to be a substring of the complete mRNA sequence.

However, other gene defects may result in simply replacing one amino acid by another resulting in proteins which might still be partly functional, e.g. the sickle-cell disease results from a mutation of a GAG codon describing glutamic acid to a GUG codon describing valine in the human β-globin

gene [SX13]. The result is an abnormal sickle-shaped hemoglobin protein which is still able to carry oxygen, but may stick together and block blood flow in smaller blood veins, such as capillaries, which in turn may result in a number of serious health problems.

2.2.4 Single Nucleotide Polymorphism (SNP)

A mutation of a single nucleobase is also called a *Single Nucleotide Polymorphism (SNP)*. Many of such SNPs are already known, whereby the terminus "SNP" may also refer to the genomic position, i.e. the *locus*, where it is known that such polymorphisms exist. In June 2015, the SNP database *dbSNP* [dbSNP] contained already 149,735,377 human SNPs.

A SNP may be located in a *coding* or *non-coding* region. If located in a non-coding region, the SNP does not affect a gene directly, but may still manipulate gene expression by affecting e.g. *transcription factor* binding, which is often required to start the transcription process. If a SNP is located in a coding region, it does not necessarily change the encoded amino acid due to the redundancies in the genetic code. Thus, it is categorized as *synonymous* or *non-synonymous*, whereby the latter category splits into two subgroups *missense* and *nonsense*. A missense SNP leads to the replacement of an amino acid by another in the synthesized protein, such as in the sickle-cell disease example described above. In contrast, a nonsense SNP generally leads to a premature stop codon resulting in a truncated and incomplete protein product.

Since humans are diploid, the information at a specific SNP or other genetic marker position may be different on both chromosomes of a diploid chromosome pair. The information at a marker position of one of the two chromosomes is referred to as an *allele*. The allele describes a genetic variation. Thus, it may directly describe the base pair at a specific position or, if used in the context of a complete gene, the gene's sequence. However, it may also be used in a more abstract definition, i.e. it may refer to either the *wild* or the *variant* type and is then denoted as *biallelic*. It is thus used equally with the definition of a *haplotype*, representing the same characteristic. In this definition, the wild type describes the common allele in a set of samples or in a reference, while the variant type describes any other allele.

While an allele or a haplotype describes only the information of one chromosome in a pair, the *genotype* refers to the genetic information in both chromosomes at the marker position. Thus, a genotype may be classified as *homozygous* if the alleles on both chromosomes are equal, or as *heterozygous* otherwise. Together with the biallelic definition of a haplotype, a genotype may be referred to as *homozygous wild*, *heterozygous* or *homozygous variant* type.

Any characteristic which is directly visible or observable is called a *phenotype*. For example, a phenotype of a sample may be the eye color, hair color, and also the fact if the person has a certain disease or not. Alleles may then be denoted as *dominant* or *recessive*. A dominant allele shows a certain phenotype even if the genotype is heterozygous, while a recessive allele requires a homozygous genotype for the phenotype (or at least two recessive alleles for the same phenotype). An example for a dominant allele can be found in the β-globin mutation causing the sickle-cell disease as described above. Even if only one gene is affected, blood cells contain sickle-shaped hemoglobin. However, the other gene is still working and correctly producing regular hemoglobin. This heterozygous case is also referred to as the *weak* variant, since almost half of the blood cells are still unchanged. The *strong* variant of the sickle-cell disease occurs if both genes are affected, i.e. if the genotype is the homozygous variant type.

2.3 DNA Sequencing

2.3.1 History

The first completely determined genome sequence of a species was from the RNA virus *Bacteriophage MS2* identified by Walter Fiers in 1976 [FCD+76]. It consists of only 3,569 nucleotides in single-stranded RNA. In 1977, Frederick Sanger adopted a DNA sequencing method by Ray Wu [JBP+74] to develop a faster method [SNC77] and published the identification of the first complete DNA-based genome sequence still in the same year, the single-stranded 5,386 nucleotide genome of bacteriophage *Phi X 174*. Sanger is one of to-date four persons who received the Nobel prize twice: 1958 for

determining the structure of the insulin molecule and 1980 for the invention of his DNA sequencing method.

During the years, Sanger's method has been automatized and only modified slightly. It forms a robust but exceptionally slow method for DNA sequencing until today. Large parts of the human genome sequence determined during the Human Genome Project was determined with Sanger sequencing as well. The method delivers high quality *reads*, i.e. fragments of the targeted genome sequence, with lengths from 400 up to 900 bases at 99.999% accuracy, but the throughput lies only in the range of 1,900 to 84,000 bases in 20 minutes to 3 hours, and the costs are very high in the range of 2,400 US-$ per one million bases [LLL+12]. Sanger sequencing will shortly be explained in Sect. 2.3.2.

In order to address the downsides of this first generation of DNA sequencing methods, several *Next Generation Sequencing (NGS)* methods evolved. NGS methods reach a high throughput of to-date up to 10 human genomes at 30× coverage in three days by extremely parallelizing the sequencing process with the generation of thousands up to millions of sequences concurrently while potentially reducing the costs down to only 1,000 US-$ per human genome. However, read lengths are usually smaller and the quality is a bit lower than in Sanger sequencing, e.g. 2×150 bases per read with an accuracy of $> 99.9\%$ at 75% of the bases in Illumina's HiSeq X system [Ill15a].

From many different available NGS methods, the focus is set to the *Sequencing by Synthesis (SBS)* method by *Illumina*, which will shortly be introduced in Sect. 2.3.3.

2.3.2 Sanger Sequencing

Sanger sequencing is based on a simple chemical *chain-termination* method and a separation method such as *gel electrophoresis*. The sequencing process requires a *DNA template*, i.e. a sample of the DNA sequence which is to be determined, a *DNA primer* and *DNA polymerase*, which is an enzyme that is able to copy a DNA fragment. Furthermore, the copy process requires nucleobases to form new DNA. These are added as *deoxynucleosidetriphosphates (dNTPs)*, available as *dATP*, *dCTP*, *dGTP* and *dTTP* for the four different

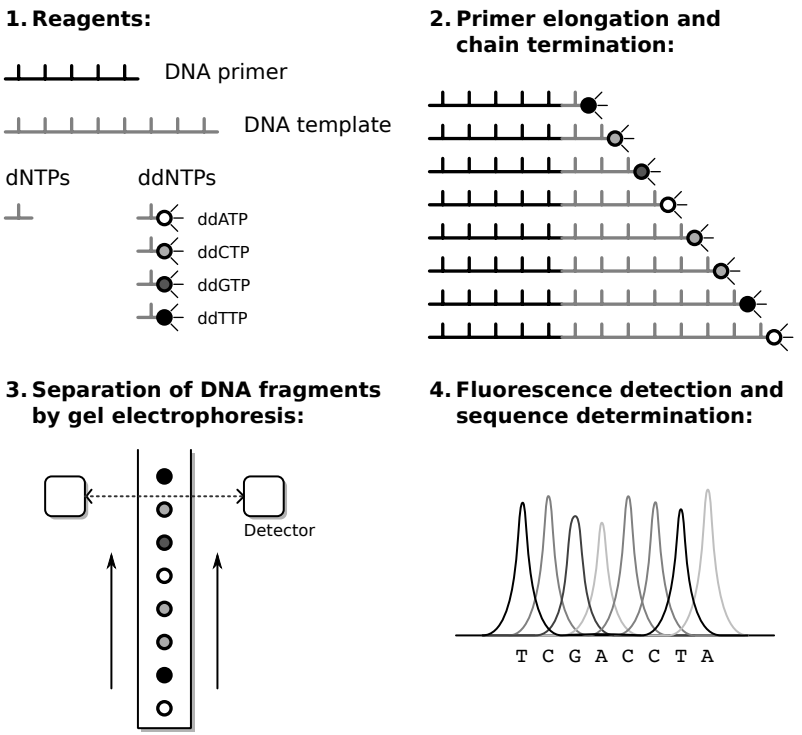


Figure 2.5. Principle of Sanger sequencing with fluorescent labeled ddNTPs.

nucleotides. The reagents in this mixture now begin to create new DNA fragments from the template bound to the DNA primer molecules. In order to generate only partial copies of different lengths, so-called *dideoxynucleosidetriphosphates (ddNTPs)* are added in a significantly lower amount than dNTPs. The DNA polymerase binds ddNTPs likewise as dNTPs to a DNA fragment, but ddNTPs are not able to form another connection to dNTPs again, thus ddNTPs act as chain terminators. If applied correctly, the chemical equilibrium now contains DNA fragments of the template in every length bound to the DNA primers. Gel electrophoresis is now able to separate the fragments of different lengths since smaller molecules move

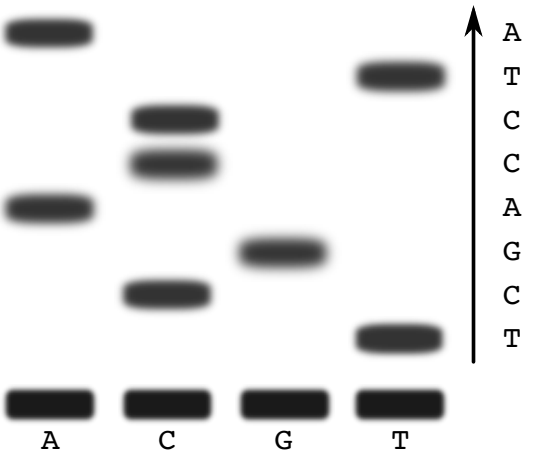


Figure 2.6. Electrophoresis profile from a Sanger sequencing run with four lanes. The revealed sequence in this example is `TCGACCTA`.

faster during electrophoresis than larger molecules.

In the original Sanger sequencing, this experiment has to be done in four different reactions in order to test for the four nucleotides `A`, `C`, `G` and `T`. Each reaction then contains only one kind of ddNTPs, namely *ddATP*, *ddCTP*, *ddGTP* or *ddTTP*. Gel electrophoresis is then performed concurrently for all four reactions resulting in a profile containing an A-lane, C-lane, G-lane and T-lane. The profile can finally be read from bottom to top revealing the DNA sequence of the template.

Modern Sanger sequencing methods use ddNTPs labeled with a fluorescent tag. With the fluorescence at different wave lengths for the four different bases, the chemical reactions can now be processed in only one experiment again. The resulting electrophoresis profile can be read with a camera, determining the different bases from the different colors of the tagged DNA fragments.

Figure 2.5 illustrates the principle of Sanger sequencing with fluorescent labeled ddNTPs, while Fig. 2.6 depicts an electrophoresis profile from a former Sanger sequencing run with four lanes.

2.3.3 Illumina Sequence by Synthesis (SBS)

The Next Generation Sequencing method *Sequence by Synthesis (SBS)* is implemented in Illumina's most recent series of DNA sequencing machines – the HiSeq X [Ill15a]. The HiSeq X Ten features a throughput of 10 human genomes in about 3 days at a 30× coverage and 2 × 150 bases read length. According to Illumina, the averaged costs for one human genome are about only 1,000 US-$.

The sequencing process requires in advance a small sample of at least 100ng *purified*, i.e. uncontaminated and "clean" DNA. Then, the first step is *tagmentation*. Enzymes called *transposomes* cut the DNA sample into small fragments at random. The fragments are *denatured*, i.e. the DNA now occurs only single-stranded.

In the second step called *Reduced Cycle Amplification*, the fragments are extended with a primer sequence, an index sequence and an adapter sequence on both ends, but for both ends different primers, adapters and indices are used. The index allows to sequence up to 96 different DNA samples in a single sequencer run since the index sequence can be determined separately and allows a later separation by software.

The extended DNA fragments enter a *flow cell* which is covered with a lawn of two types of oligonucleotides exactly matching the adapter sequences attached to the DNA fragments, i.e. they have the exact complementary adapter sequences. Thus, the fragments adapt at certain positions to the oligonucleotides at the flow cell.

The following step is called *Bridge Amplification*. Here, large clusters of about 1,000 copies of the same DNA fragments are generated. This is achieved by firstly extending the adapter from the flow cell by copying the adapted DNA strand with DNA polymerase. The original strand is washed away leaving the copy attached to the flow cell with a stable chemical bond. Now, the copy bends over to adapt with its other end to the other type of flow cell adapter. Then, polymerase creates a double-stranded bridge extending the second adapter. The bridge is denatured afterwards. The result is two complementary strands chemically bound to the flow cell. With numerous repetitions of this process clusters of the same DNA fragments and its complementary strands are created. At the end of the

Bridge Amplification process all complementary strands are cut and washed away leaving only the forward strands.

The subsequent *Sequence by Synthesis* step presents the main sequencing process. It again copies the bound DNA fragments, but this time with fluorescent labeled *reversible terminated deoxynucleosidetriphosphates (dNTPs)*. Each time a dNTP binds further replication is interrupted by the terminator in the dNTP. A photo of the flow cell identifies the bases that have bound in this step according to the color of the fluorescence. For the next step, chemicals remove the terminator from the last dNTP allowing the replication process to continue with the next dNTP, which is again terminated, thus interrupting the process again. Hence, the replication process can be documented base by base at millions of clusters in parallel, concurrently creating millions of sequencer reads.

An optional but commonly used extra step in Illumina's SBS sequencing process is *paired-end* sequencing. If the desired read-length has been reached by the SBS step, the generated copies are washed away leaving the original DNA fragment clusters as of the beginning of the SBS step. Now, the strands bend again to adapt with their open end to the other flow cell adapters, exactly as in an iteration of the Bridge Amplification step. DNA polymerase once more creates a double-stranded DNA bridge which is denatured afterwards as well. Now, the clusters contain forward and complementary backwards sequences again, but before beginning a new SBS step, this time the forward strands are cut and washed away.

This way, the complementary backwards strand of a DNA fragment is sequenced at the same position in the flow cell as the original forward strand. Hence, the new read originates from the same fragment as the previous read, only sequenced with starting from the other end. Depending on the type of transposomes used in the very first step of tagmentation, the size of the DNA fragments can be controlled, and with a known fragment size the size of the gap between both reads can be estimated. Thus, both reads form a *read pair* with known gap information, which is a very powerful information that can be used in further analysis e.g. in *short-read alignment* for recovering strongly repetitive DNA segments.

Figure 2.7 illustrates the steps for SBS used in Illumina's HiSeq X sequencers.

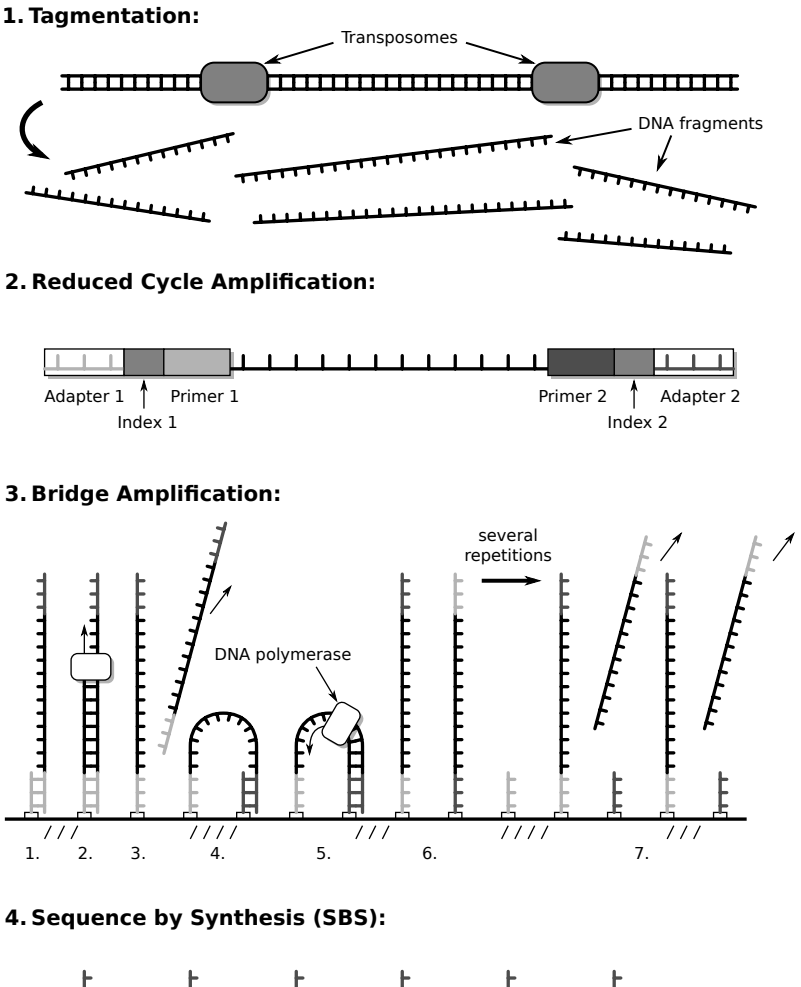


Figure 2.7. Sequence by Synthesis in Illumina's HiSeq X sequencers.

2.4 Genotyping with Microarrays

The particular DNA sequence is not always required for certain studies. It is often faster and cheaper to only determine the *genotype* of a sample at specific marker positions with the help of so-called *microarrays*, e.g. *GeneChips* by Affymetrix [Aff09] or *BeadChips* by Illumina [Ill15b] amongst others.

The genotype can either be a set of the specific nucleotides the sample carries at the predefined marker positions, or it is denoted in a more abstract way, i.e. it takes one of three different values for each position: 0 for the *homozygous wild* type, 1 for *heterozygous*, and 2 for the *homozygous variant* type. Since each human cell contains two copies of each chromosome (humans are a *diploid* species), the terms *homozygous* and *heterozygous* refer to the equality or inequality of the DNA bases at the specific marker on both chromosomes respectively. The *wild* type is the base commonly encountered at this position, while the *variant* type means any other base. The specific base information at this position is not known in this notation, though it is sufficient e.g. for classical case-control studies to know in which of the three before-mentioned categories the genotype can be classified (see Chapter 5 for more information).

2.4.1 Affymetrix GeneChip

Affymetrix and Illumina follow two slightly different approaches in order to determine genotypes with microarrays. The Affymetrix GeneChip consists of a plate with thousands of *spots*, also referred to as *features*, arranged in an array. At each spot a number of specific oligonucleotide sequences, referred to as *probes*, are attached to the plate whereby the probes are equal at one spot but differ among the other spots. A DNA sample has to be chemically prepared before being applied to the plate. First, as in gene expression, mRNA strands are created from certain locations in the DNA. The mRNA is then treated with *reverse transcriptase* to create fluorescently labeled single-stranded *complementary DNA (cDNA)*. These cDNA strands may now be able to bind to certain oligonucleotides attached to the plate. All strands that do not bind are simply washed away and the remainder can

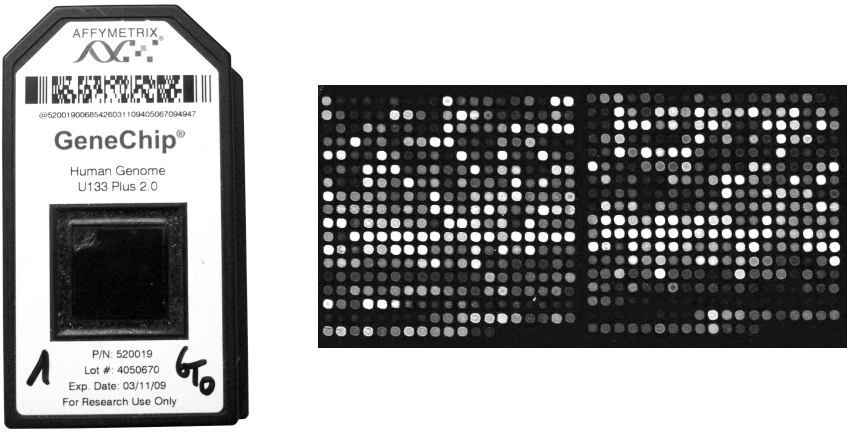


Figure 2.8. Affymetrix GeneChip (left) and the fluorescent spots of a microarray[1] (right).

be made visible with a camera due to their fluorescent labels. According to the position in the array, and the intensity and color of the fluorescence the genotype can be derived.

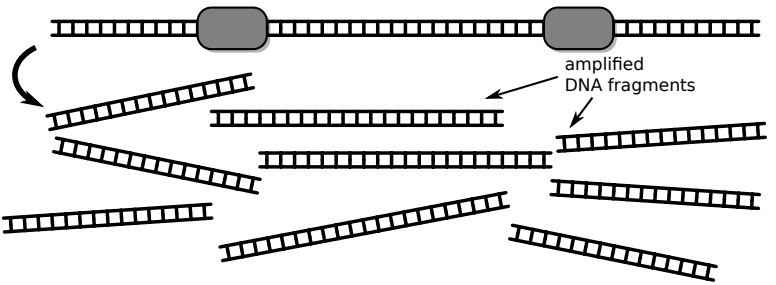
The recent *Genome-Wide Human SNP Array 6.0* by Affymetrix [Aff09] features >900,000 markers for SNPs and >900,000 markers for so-called *Copy Number Variations (CNVs)*. Figure 2.8 shows photos of an Affymetrix GeneChip (older version) and a fragment of such an array with fluorescent spots.

2.4.2 Illumina Infinium II BeadChip

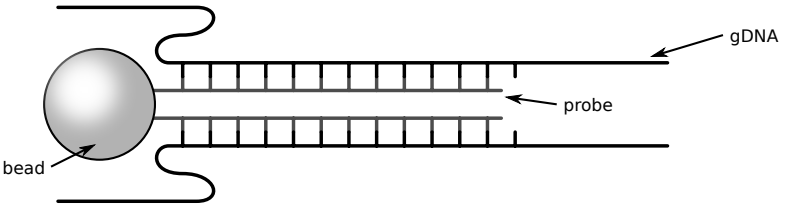
The Illumina *Infinium* approach [Ill06] differs slightly to the Affymetrix approach. Instead of creating mRNA and cDNA, the DNA sample is simply amplified and fragmented. The probes contain oligonucleotides of a precursor sequence just until the location where the genotype has to be de-

[1] "A fragment of cDNA microarray" by Mangapoco used under CC BY 2.5 (http://creativecommons.org/licenses/by/2.5), desaturated, original URL: https://upload.wikimedia.org/wikipedia/commons/f/f2/Cdnaarray.jpg

1. Fragmentation and Amplification:



2. Hybridize on BeadChip:



3. Extend probe:

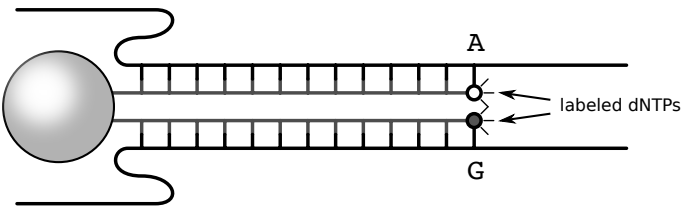


Figure 2.9. Genotyping with Illumina's Infinium II BeadChips.

termined, but not the locus itself. The denatured DNA samples, referred to as *gDNA* then, hybridize and attach to the probes if the precursor sequence matches. Now, as in the DNA sequencing process, the probes are extended by exactly one nucleotide with fluorescent labeled dNTPs, according to the attached DNA sequence. A photography of the array makes the fluorescence visible, and according to the color and the bead position in the array, the exact nucleobase can be derived for a certain location.

Illumina supports BeadChips with already a few million markers, such as the *Human Omni5 Exome* with >4.5 million markers for the human genome [Ill15b]. Figure 2.9 illustrates the step in Infinium II genotyping.

Chapter 3

FPGA Technology

3.1 Introducing FPGA Technology

A *Field Programmable Gate Array (FPGA)* basically states a device with plain resources for configurable logic. Unlike CPUs, which provide fixed resources, e.g. an *Arithmetical Logic Unit (ALU)* or *Floating Point Unit (FPU)*, with a fixed instruction set for interpreting and execution of computer programs (in particular machine instructions), an FPGA requires a configuration that defines the behavior of the chip. For this purpose, FPGAs provide components called *Configurable Logic Blocks (CLBs)*. They are able to implement and store the output of any logical operation with a limited number of input and output variables via freely configurable *Lookup Tables (LUTs)* and registers. All CLBs are connected via a *routing matrix*, such that for general operations with arbitrary in- and outputs a free number of CLBs may be used in combination. This general concept of an FPGA structure is illustrated in Fig. 3.1.

In general, CLBs feature additional logic for a fast and easy implementation of operations which are required very often, such as carry logic for adders and subtractors. A wide range of other helpful components, such as *Digital Signal Processors (DSPs)*, fast and directly accessible on-chip RAM called *Block RAM (BRAM)*, IO registers, clocking logic etc., can generally be found on FPGAs as well. Furthermore, the possibility of a free combination of the FPGA's components directly offers an inherent parallelism, i.e. different groups of FPGA components with different functionality may be executed concurrently at arbitrary times.

This flexibility features the ability to directly implement a hardware design perfectly adapted to the targeted problem. Unlike software for CPU-

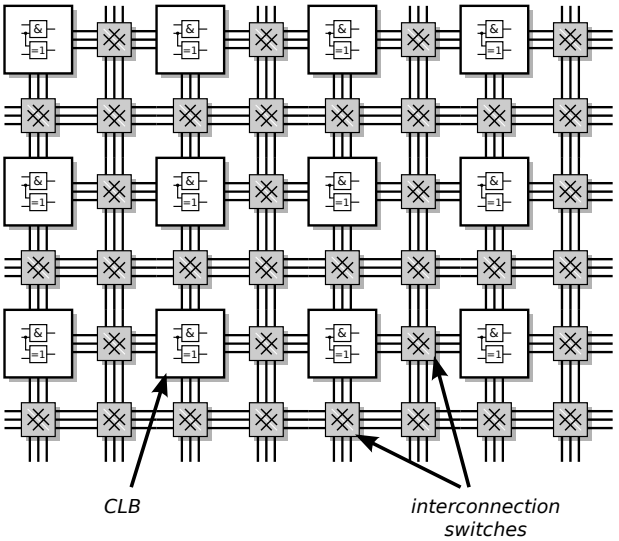


Figure 3.1. General structure of an FPGA. In order to define the functionality of the chip, CLBs may be freely configured and connected via the interconnection switches in the routing matrix.

based architectures, a hardware design can therefore be ideally optimized to equally utilize the required resources during processing time. This leads to an optimal utilization of the chip resources of an FPGA, and therefore fast execution times. In contrast, the fixed instruction set of a CPU generally prevents all resources of a CPU to be simultaneously used. Only those resources required to follow the current instruction are utilized while all others are left unused. Pipelining can improve this situation, but still only those resources from a universal instruction set can be utilized which are required to solve the targeted problem. Even multi-core processors, such as recent Intel Xeon types [Int15], have to feature the resources for the complete instruction set in each core (generally for all atomic instructions), or they have to be shared among the cores (e.g. floating-point units (FPUs)).

However, for general problems CPUs feature some main advantages, i.e. the short execution time of instructions, which might take only a few

clock cycles with a high frequency in the range of several Gigahertz, and flexibility due to a universal instruction set. Thus, CPUs are commonly used for almost any kind of application and can be found in almost any kind of technical equipment. Starting with simple desktop computers, notebooks, servers or clusters, CPUs are used in embedded systems for entertainment devices such as TVs, media players (Bluray, DVD, CD, MP3, ...), cameras, in car components such as key lock, airbags, engine, brakes, GPS, speedometer, ABS, ESP, in medical devices such as X-ray, heart monitor, MRT, CTG or as microcontrollers in watches, thermometers, smoke detectors, air conditioning, other measurement devices etc. The reason for this flexibility is the ability to solve almost any kind of computational problem by writing software which can then be compiled to a machine program based on the in-built instruction set of the underlying CPU architecture. The software is commonly written in high-level programming languages such as Java or C++ which flattens the path for an easy and fast development due to readability and easier debugging. Compilers can efficiently translate the code written in high-level language to machine code for many different kinds of CPUs. Thus, once the program is written it can be equally used for many different CPUs which underlines flexibility again. For debugging purposes, an error in a program can simply be removed by exchanging the software, but the underlying hardware stays the same. And in general, the development time for writing a sequential program is much shorter than to develop a new hardware design.

The motivation to harness FPGAs for specific solutions can have several reasons. Two reasons may be the saturation of Moore's law for frequency scaling and the arising need for parallelism. For a long time people could have relied on Moore's law, which predicts the doubling of CPU performance every 18 to 24 months. This assumption was made on the basis of the ongoing progress in building smaller and smaller transistors in a silicon chip, implying a larger amount of computing resources per area which could be clocked with higher frequencies. However, in recent years engineers encounter various problems in designing smaller transistors including voltage scaling and heat dissipation [Bos04; KAB+03]. To compensate this drawback and to keep Moore's law upright, multi-core processors were created to offer parallelism for more speed [Com05; Tan06]. The downside

of this development was that introducing parallelism into existing software is not a trivial task. Programs had to be rewritten using libraries supporting parallelism (e.g. threads), such that multiple cores could be efficiently used. *POSIX threads (pthreads)* [Bar] or *BOOST* [DAR] for the C/C++ languages are common examples. Yet with the background of Amdahl's law [Amd67], doubling the number of processing cores does not directly mean the doubling of performance. Consequently, Moore's law for frequency scaling is considered saturated, but is still working in a reduced form for transistor density scaling. This arising gap between traditionally written sequential code and increasing parallelism in the hardware offers an opportunity for niche architectures, such as GPUs or FPGAs, since they already exhibit parallelism in themselves and the problem of parallel programming has to be tackled for CPUs anyway [VB13].

The general way to offer parallelism in the area of high-performance computing is via CPU clusters. CPU cluster architectures are an easy way to improve runtime by parallelization for general applications. Simple clusters of standard PCs up to the fastest supercomputers from the *TOP 500* list [TOP], such as the Tianhe-2 in China, can be programmed with the help of additional libraries, such as MPI [MPI] or UPC++ [ZKD+14], based on established high-level programming languages (e.g. C++). We have also investigated this area by using UPC++ to implement the SNP interaction detection software BOOST [WYY+10a] on the supercomputer Cray XC30 with 12,288 cores [KGW+14]. However, the advantage of flexibility comes with obvious drawbacks. On the one hand, data transfer between the cores has to be ensured fast and reliable by the architecture's topology, and on the other hand, the more cores are combined, the more space for housing and the more energy is required for processing and infrastructure, e.g. Tianhe-2 features 16,000 computing nodes with a total of 3,120,000 cores, but consumes 17.6 megawatts of power (24 megawatts including cooling) and is housed in a room with 720 squaremeter footprint [Don13].

In order to tackle these drawbacks while still providing a high degree of parallelism, many applications are ported to general purpose graphics processing units (GPUs). GPUs feature a fixed instruction set as well, but generally a huge amount of simple parallel processing cores, e.g. Nvidias GeForce GTX Titan Z features 5,760 shader cores in one die [NVIb]. Since

GPUs are intentionally designed for graphics processing, i.e. picture processing or 3D rendering etc., data paths between the cores were optimized for this purposes. Thus, the offer of high parallelism on low space comes with the drawback of losing flexibility, since the application has to fit in this architecture highly optimized for graphics processing. However, for many applications GPUs can speedup the process signficantly compared to a CPU, and furthermore, due to the advantage to write GPU programs in high-level languages with libraries, such as CUDA [NVIa] or OpenGL [Khr], development and debugging are kept very simple. Especially in bioinformatics, GPUs have already proven their ability. Examples can be found throughout this work in the evaluation of the presented FPGA applications in comparison to CPU and GPU solutions. Additionally, our research team has an ongoing research to investigate hybrid systems of FPGAs and GPUs for applications in bioinformatics.

The advantages of FPGAs emerge when it comes to computational intensive tasks where the same or similar process has to be repeated again and again, such that a CPU architecture, e.g. a standard PC, is utilized for several hours or days. For FPGAs it is often possible to implement this particular process as a hardware design in a single processing element in a very efficient and resource optimized way. Provided that this process is parallelizable, the available resources allow the implementation of multiple, e.g. several tens or hundreds of such processing elements in parallel. With this fine-grained on-chip parallelism a single FPGA may be able to solve particular tasks as fast as if several CPUs had been connected for the same purpose. With their low requirements in energy and space, a cluster of FPGAs, such as the RIVYERA architecture (see Sect. 3.4), may outperform a CPU cluster or supercomputer while consuming only the same energy as a few desktop PC systems in the housing of a simple network server.

Since FPGAs are generally reconfigurable and do not specify a rigid architecture as in CPUs or GPUs, flexibility is still provided, and with the help of simulators debugging of hardware designs is still well practicable, but may require some extra effort when compared to software debugging. Hardware designs can be described using a hardware description language, such as VHDL or Verilog. These languages provide the most efficient way for FPGA designs and are directly supported by the SDKs provided by

the FPGA manufacturers, such as Xilinx ISE [ISE]. However, higher-level languages for FPGA designs are also available, such as SystemC [SyC], but are still under development and yet considered not so efficient in general. Thus, they shall not be part of this thesis.

Apart from solutions in the area of bioinformatics, which can be found throughout this work, our research team has already proven the ability of FPGAs in other fields such as cryptanalysis [GKN+13; ARW+12; AVW+14] or stock market analysis [SGW+12a; SGW+12b].

The following sections describe the technology of Xilinx FPGAs (see Sect. 3.2), the RIVYERA architecture consisting of several interconnected Xilinx FPGAs, which is the basis of most applications described in this work (see Sect. 3.4), and finally the Xilinx KC705 development board, since it is used for further development and enhancement of existing implementations to newer technology (see Sect. 3.5).

3.2 Xilinx FPGAs

This section particularly describes the features of FPGAs by Xilinx Inc. [Xil], currently the market leader of FPGA technology. Xilinx almost shares the complete FPGA market with only one competitor, Altera Corporation [Alt]. Among other FPGA manufacturers are Lattice Semiconductor [Lat], Microsemi [Mic], and Atmel [Atm].

Xilinx was founded in Silicon Valley in 1984 with its headquarters still in San Jose, California. Before 2010, Xilinx provided two main FPGA families: the Virtex and the Spartan series. Out of these series, this chapter mainly concentrates on the Spartan3 and the Spartan6 series where the RIVYERA architecture (see Sect. 3.4) and therefore most of the work in this thesis is based upon. Spartan3 FPGAs have a 90nm structure while Spartan6 FPGAs were manufactured with 45nm technology. With the introduction of the 28nm 7-series FPGAs, the Spartan family was replaced by the Kintex family and the low-cost Artix family. This chapter sets an additional focus on Kintex7 FPGAs which found the basis of the KC705 development board introduced in Sect. 3.5.

Xilinx latest development is the 20nm UltraScale series and the recently

Table 3.1. List of FPGA types and their resources discussed in this thesis. Specifications taken from [Xil13; Xil11a; Xil15b]

	Spartan3-5000	Spartan6-LX150	Kintex7-325T
Slices	33,280	23,038	50,950
Logic Cells	74,880	147,443	326,080
CLB Flip-Flops	33,280	184,304	407,600
max. distr. RAM (kbits)	520	1,355	4,000
Block RAM (kbits)	1,872	4,824	16,020
DCMs / CMTs	4 (DCM)	6 (CMT)	10 (CMT)
DSP Slices	-	180 (DSP48A1)	840 (DSP48E1)

introduced 16nm UltraScale+ series, providing more resources and energy efficiency than in previous FPFA series. Yet both will not be discussed in this thesis.

For all series different types of FPGAs are available, all featuring a different amount of resources. Table 3.1 shows the available resources for the three types of FPGAs discussed in this thesis.

Every Xilinx FPGA provides similar resources to implement an application described as a hardware design. The following gives a short overview of the most important components while they are explained in more detail in the following subsections.

The base components for implementing a hardware design are *Configurable Logic Blocks (CLB)*. These components basically consist of D-flip-flop registers and carry small pre-filled memories used as *Lookup Tables (LUTs)* to implement a boolean function. For the implementation of an arbitrary function, CLBs may be freely connected. Together with the initial content of the lookup tables, the interconnection of all CLBs is part of the FPGAs configuration. It is realized via setting the connection points in a huge *routing matrix* which connects all CLBs, IO pins and other components of the FPGA. In general, the configuration of an FPGA is not final, i.e. one can change the routing, LUT contents etc. infinitely. The CLBs are explained in more detail in Sect. 3.2.1.

Other important components of an FPGA include the Block RAM (BRAM), a very fast local memory consisting of several independent blocks

each holding 18kbit (Spartan series) or 36kbit (7-series) respectively. BRAM enables dual-ported write and read operations in only one clock cycle without latency. This offers the BRAM to be easily used as FIFOs. For more information on the BRAM see Sect. 3.2.2.

For clocking purposes an FPGA generally features several clock nets besides the standard data paths to ensure a uniform clocking across the whole chip. With more than one clock net available it is possible to clock a design with several different frequencies. However, an FPGA generally requires at least one clock input from outside with a known frequency. The desired frequencies for the design are generated by the *Digital Clock Manager (DCM)* component. Besides frequency synthesis the DCM is responsible for a clean jitter-free clock signal. For more information see Sect. 3.2.3.

Another feature of many FPGAs is the *Digital Signal Processor (DSP)*. While only special Spartan3 types house DSPs, they are included in every Spartan6 or 7-series types. The main functions of the DSPs are multiply, multiply-add, and accumulation operations. Additionally, the DSPs of the 7-series family feature some extra functionality. Utilizing DSP saves valuable CLB resources since arbitrary multiplications implemented in logic can be very costly. More information on the functionality of DSPs can be found in Sect. 3.2.4.

Last but not least, an FPGA requires Input-Output (IO) components for data transfer. Most IOs are general purpose IOs which can be configured as either input or output to send or receive arbitrary information. It is also possible to configure IO pins as *inout*, which means, that information can be sent or received at the same pins. This is useful when implementing a shared bus system with several transceivers. For high speed data transfers it is possible to connect IOs as pairs to allow *Low Voltage Differential Signaling (LVDS)*. Due to better electrical characteristics of LVDS signals, it is possible to reach higher data rates with an LVDS pair compared to two single-ended data signals.

Furthermore, special IOs can be used for connections to external DRAM. Beyond that, Spartan6 and 7-series FPGA feature internal DRAM controller components for an easy integration into a user's design. To ensure simple access to an FPGA in a PC environment, 7-series FPGAs feature in-built PCI Express (PCIe) cores for IO operations as well. For Spartan6 an endpoint

block for PCIe is encluded only for Spartan6-T types. These PCIe endpoints are used in the RIVYERA or KC705 API to allow fast data transfers between the FPGA design and a controller software.

FPGAs may also hold several other components up to complete CPU cores, such as ARM cores in the recent Xilinx Zynq-7000 devices or PowerPC cores in some types of the older Virtex series. Nevertheless, those components are not part of this thesis.

3.2.1 Configurable Logic Block

A *Configurable Logic Block (CLB)* in Xilinx FPGAs usually consists of two or four *Slices*. The slices carry *Lookup Tables (LUTs)* and D-flip-flop storage elements and therefore present the basic structure required for creating a reconfigurable hardware design. The structures of a slice and CLB are fixed and may only vary between the FPGA families. However, the CLBs are connected via a *switch matrix* to the routing matrix which describes the interconnection of all CLBs and other FPGA components and is configured by the user design. The user configuration also determines the contents of the lookup tables to implement an arbitrary boolean function within a CLB. For example, if the user wants to map the logic of a simple XOR operation $Y = A \oplus B$ in a 4-input LUT, the contents of the LUT have to be filled as in Table 3.2. Consequently, the inputs C and D have no effect, since they are not required for this function.

Besides the realization of logical operations, slices may feature some extra functionality dependend on their type. A common feature is the support for implementing fast adders by introducing extra carry logic, often referred to as *carry chain*. The carry logic calculates the carry bit of the summation of two input bits while the sum is calculated via the lookup table. Furthermore, the sum is corrected if a carry is propagated from the neighboring previous slice and the carry bit can be propagated to the neighboring next slice as well. Usually, the carry logic consists of an additional AND gate to calculate the carry, an XOR gate to correct the sum and some multiplexers to control the carry propagation on the additional data path to neighboring slices. The carry logic of Spartan3 slices is illustrated in the diagram of a pair of Spartan3 slices in Fig. 3.2.

Table 3.2. Example contents of a 4-input lookup table realizing the XOR function $Y = A \oplus B$, inputs C and D are not required and therefore have no effect on the output.

D	C	B	A	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Other extra functionality of FPGA slices may be logic to support the implementation of shift registers and *distributed RAM* which uses the capacity of the lookup tables as memory.

In the following, the details of the CLBs from different FPGA families are discussed.

Spartan3 CLB

The Spartan3 CLB [Xil13] consists of four slices, two in the *Left-Hand SLICEM* and two in the *Right-Hand SLICEL*. Each slice contains one 4-input LUT and one D-flip-flop. Both pairs of slices support the carry logic, but only the SLICEM supports distributed RAM and shift register functionality.

To directly support boolean functions with up to 8 inputs, Xilinx has

introduced so called *F5*, *F6*, *F7*, and *F8* multiplexers. For example, if a 5-input boolean function had to be realized, one LUT describes the function of the first four inputs when the fifth input is set, and a second LUT describes the function when the fifth input is not set. The F5 multiplexer now selects the output from both LUTs depending on the fifth input. Once a 5-input function is realized this way, 6-, 7-, and 8-input functions are implemented following the same scheme.

A diagram depicting the Left-Hand SLICEM can be seen in Fig. 3.2.

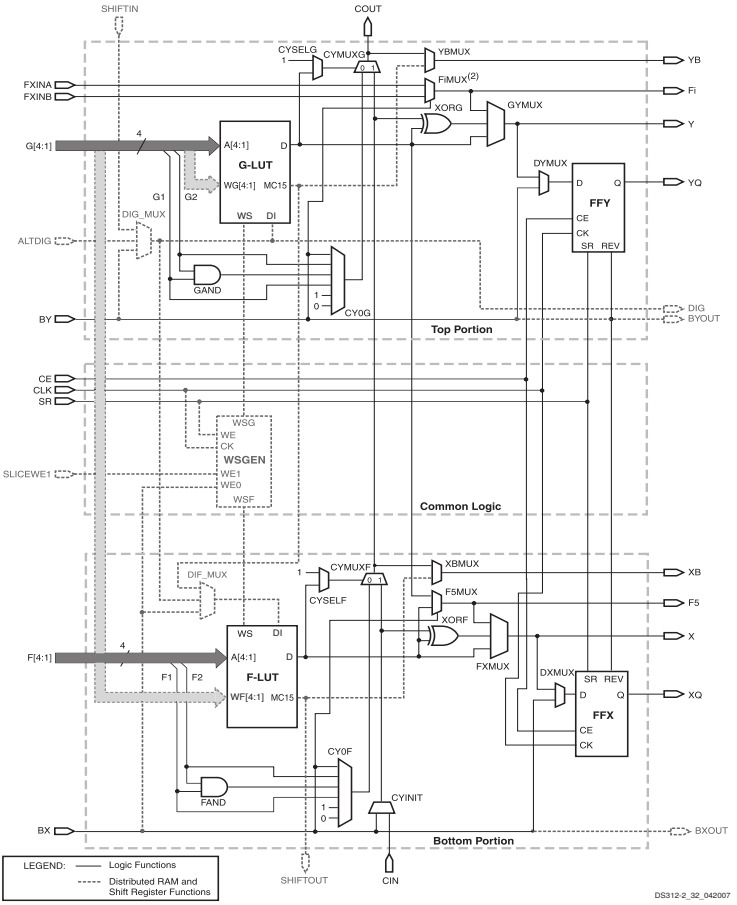
Spartan6 CLB

In contrast to a Spartan3 CLB, a Spartan6 CLB [Xil10] consists of only two slices, though one slice now contains four 6-input lookup tables and eight D-flip-flops. There are three types of slices, SLICEX, SLICEL, and SLICEM. SLICEX is the basic slice which only contains the aforementioned LUTs and D-flip-flops. SLICEL additionally features carry logic and wide-function multiplexers similar to those described for the Spartan3 CLB, and SLICEM supports carry logic, wide-function multiplexer, shift register and distributed RAM functionality. The slices are distributed in the following way. Each column of CLBs consists of two columns of slices whereby one column is a SLICEX column and the other column alternates between SLICEL and SLICEM. Therefore, approximately 50% of a Spartan6 FPGAs slices are SLICEX, 25% are SLICEL and 25% are SLICEM. This design probably intended as a feature turned out to be a drawback for some of the applications described in this thesis. Since not all slices can be used to implement fast adders, designs strongly based on extensive calculations or even counting may be hindered to utilize the complete resources of the FPGA.

Figures 3.3, 3.4, and 3.5 illustrate the three different types of Spartan6 slices.

Kintex7 CLB

Compared to Spartan6 CLBs, Kintex7 FPGAs contain only SLICEL or SLICEM slices [Xil14b], as depicted in Figs. 3.4 and 3.5. SLICEX slices



Notes:

1. Options to invert signal polarity as well as other options that enable lines for various functions are not shown.
2. The index i can be 6, 7, or 8, depending on the slice. In this position, the upper right-hand slice has an F8MUX, and the upper left-hand slice has an F7MUX. The lower right-hand and left-hand slices both have an F6MUX.

Figure 3.2. Simplified diagram of the Left-Hand SLICEM in Spartan3 FPGAs. (Source: [Xil13], page 24, figure 12)

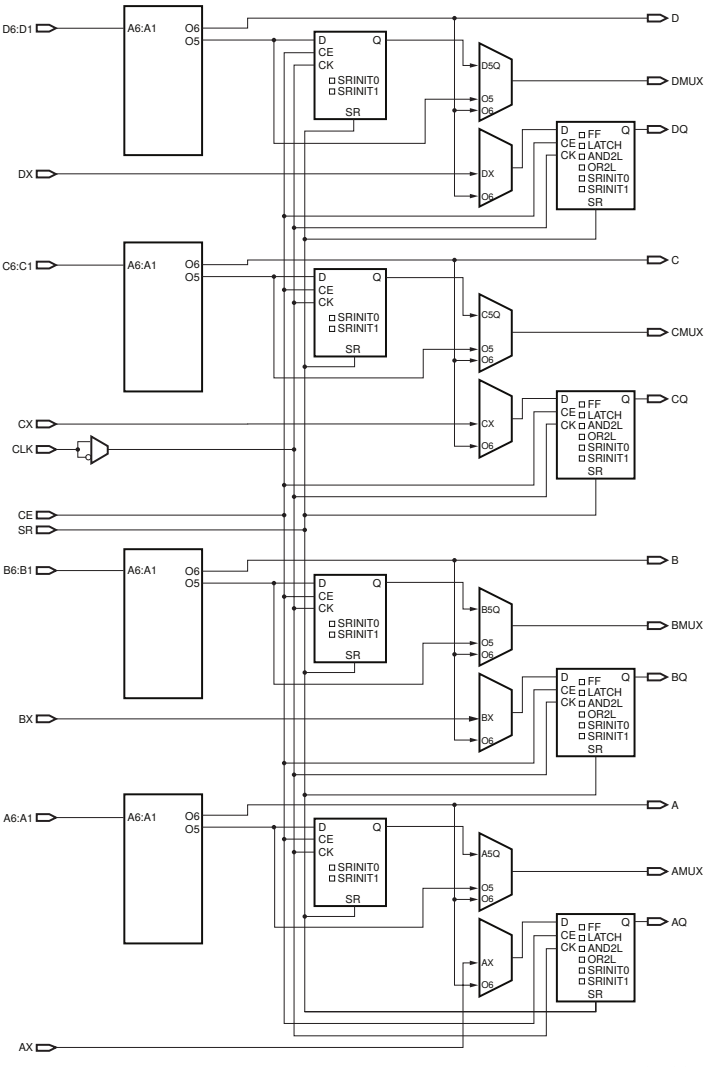


Figure 3.3. Diagram of SLICEX in Spartan6 FPGAs. (Source: [Xil10], page 11, figure 5)

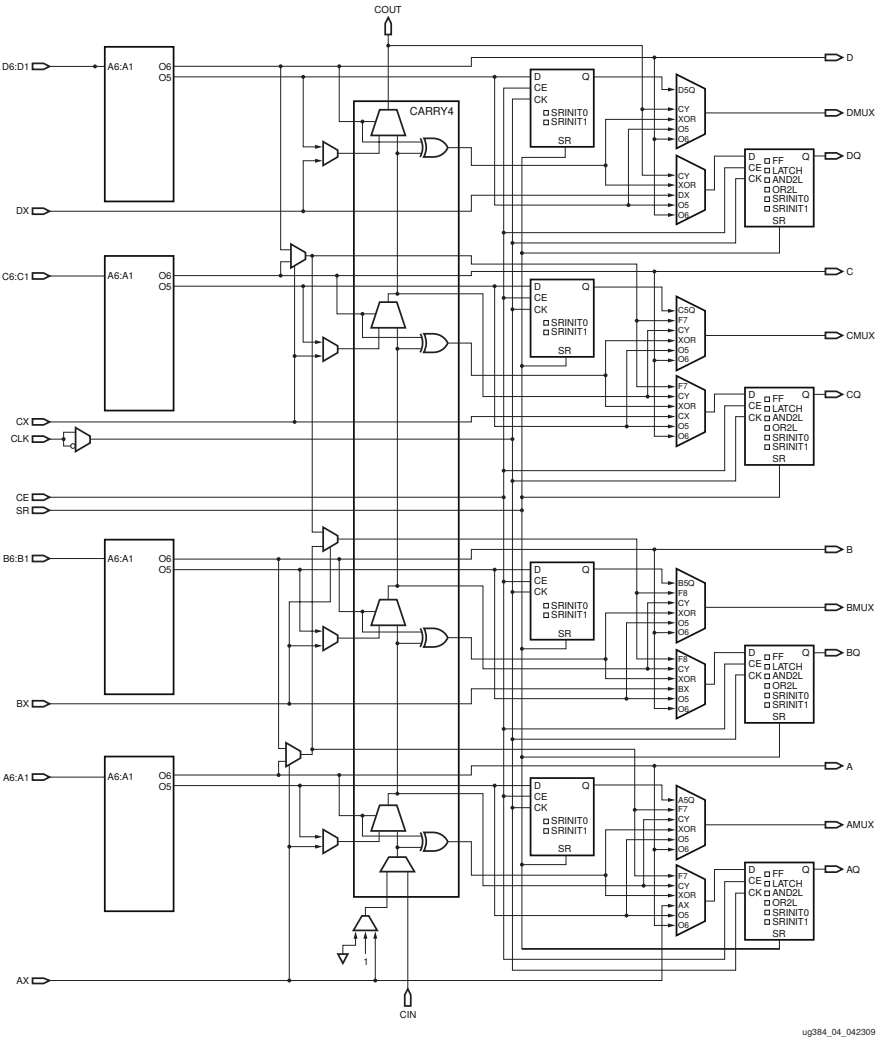


Figure 3.4. Diagram of SLICEL in Spartan6 FPGAs. (Source: [Xil10], page 10, figure 4)

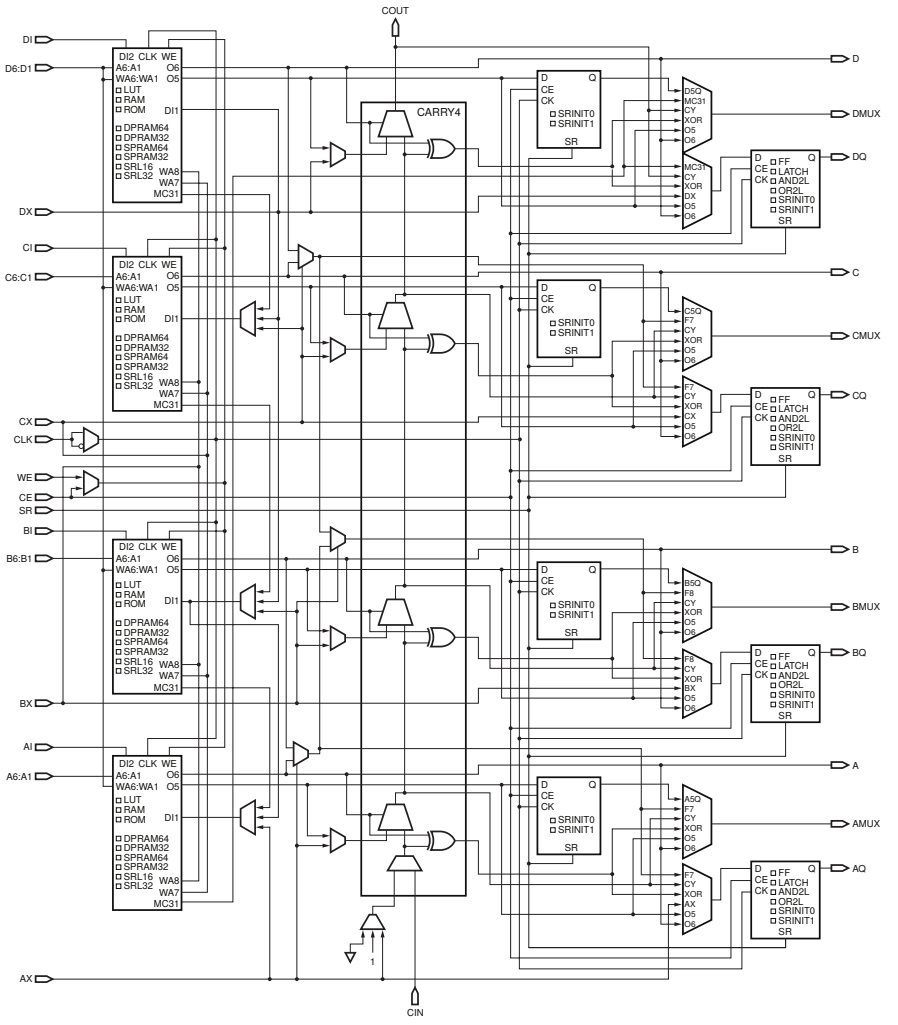


Figure 3.5. Diagram of SLICEM in Spartan6 FPGAs. (Source: [Xil10], page 9, figure 3)

have been abandoned in all 7-series FPGAs such that all slices may implement fast adders now. Each column of CLBs may contain two columns of SLICEL slices or one SLICEL column and one SLICEM column. For all Kintex7 devices there are about two times more SLICEL slices than SLICEM slices.

Another modification has been applied to the 6-input lookup tables. In 7-series FPGAs each 6-input LUT can be configured as two 5-input LUTs with separate outputs. Additionally, one of the two D-flip-flops after each LUT may be configured as a D-latch, with the precondition, that the other flip-flop is not used.

Since Spartan6 slices and Kintex7 slices are structurally almost equal, diagrams of the modified SLICEL and SLICEM are omitted here.

3.2.2 Block RAM and FIFOs

Block RAM

Block RAM (BRAM) is a fast and directly accessible memory inside the FPGA. The complete BRAM resources are distributed over a number of BRAM blocks, depending on the type of the FPGA. BRAM blocks are usually organized in several columns throughout the FPGA's area to ensure uniform access from the CLBs, e.g. for most Spartan3 types there are four BRAM columns on each device.

Each BRAM block can be used independent of others. For Spartan3 and Spartan6 devices each block provides 18kbit of memory, whereby for Spartan6 devices one block can be configured as two independent 9kbit blocks [Xil11b]. For 7-series FPGAs the size of a BRAM block has been doubled, i.e. 36kbit per BRAM block which can be configured as two independent 18kbit blocks as well [Xil14c].

Access to a BRAM block depends on its configuration. Generally, the data port width can be configured as 1, 2, 4, 9, 18, 36 (only 18kbit blocks) or 72 bit (only 36kbit blocks). Three access modes are possible: *single-port*, *simple dual-port (SDP)* and *true dual-port (TDP)*. The single-port configuration allows only sequential read or write operations to the memory, while the dual-port configurations allows concurrent access to the BRAM on two

ports. In SDP mode both ports are limited to one port exclusively for read operations and one for write operations. In contrast, TDP mode allows both operations on both ports, but the data width cannot be larger than 18 bit for 18kbit blocks or 36 bit for 36kbit blocks respectively. Another feature is the possibility to operate both ports at different frequencies, and additionally, the data port width does not need to be equal on both ports as well.

BRAM is a fast and directly accessible memory since each operation only takes one clock cycle, i.e. for write operations, data at the data input are stored in the memory at the next clock cycle when activated. The same applies for read operations, one clock cycle after enabling the read command the data is ready at the data output of the BRAM.

Furthermore, there are three write modes for ports with read/write configuration which describe the behaviour of the data output after a write operation. In *READ_FIRST* mode, the output states the contents of the RAM cell before it is written to. This is the default behaviour. In *WRITE_FIRST* mode, the data output holds the same data, which has just been written to the RAM, and in *NO_CHANGE* mode, the contents of the data output are not changed.

Access to a shared memory via two ports introduces the possibility of a collision, i.e. if the same memory cell is accessed by both ports at the same time. If a read operation is performed concurrently on both ports at the same cell, this collision is not problematic since both data outputs will hold the same data. For all other collision types, i.e. both ports write at the same cell at the same time or one port writes while the other port reads the same cell, Xilinx predicts undefined data outputs. At a write-write collision the state of the memory cell is undefined as well unless both ports write the same data. However, the write operation at a read-write collision is guaranteed to succeed, only with the data output of the read port being undefined. There is one exception, if the write port is configured in READ_FIRST mode and both ports are synchronously clocked, the data outputs of both ports will hold the correct data at a read-write collision.

Since user designs may claim different amounts of memory at different occasions, BRAM blocks can be arbitrarily connected to a larger memory fitting the users requirements. For this purpose Xilinx provides tools, such as *Xilinx Coregen* (referred to as *IP Catalog* in Vivado) (see Sect. 3.3 for more

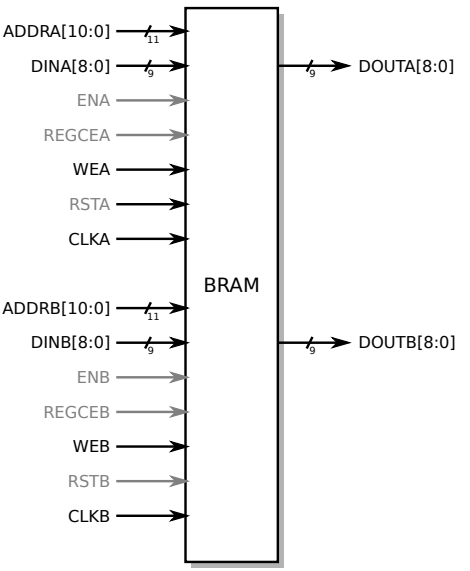


Figure 3.6. Interface for an 18kbit BRAM component created with Xilinx Coregen. The data port width is 9 bit and the BRAM is configured in true dual-port mode. Signals colored in gray are optional.

information), to comfortably create memory components with user-defined size, port width and access mode. Using these tools to configure BRAM memory, the interface can be as simple as depicted in Fig. 3.6.

FIFOs

A common use-case for Block RAM is the implementation of *FIFOs*, i.e. a simple queue with *First-In-First-Out (FIFO)* behaviour. The core logic uses a read-pointer, a write-pointer and the BRAM for storage. Anyway, the user does not need to implement the core logic himself. The Xilinx Coregen allows the direct configuration of FIFOs. The FIFO interface provides data input, data output, write enable, read enable, and reset pins as well as flags for the full or empty state of the FIFO. Optional outputs are data counts or

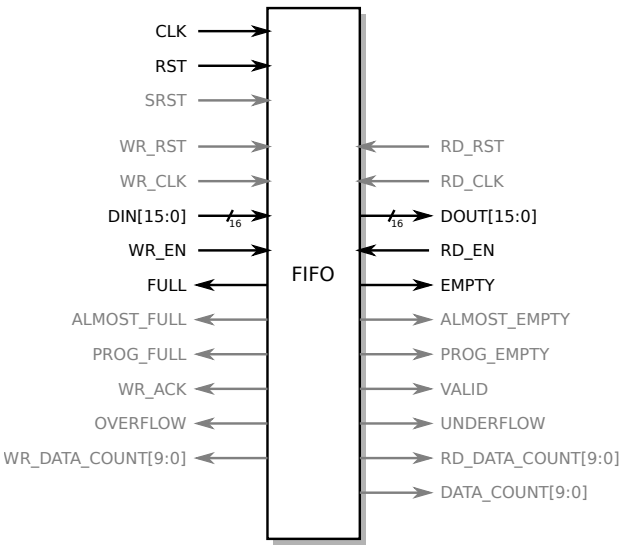


Figure 3.7. Interface for a simple FIFO component created with Xilinx Coregen. The data port width is 16 bit, the size is 1024 words. Signals colored in gray are optional.

flags for almost full, write overflow and read underrun. A simple interface of a FIFO is shown in Fig. 3.7.

Additionally, FIFOs are commonly used to reliably transfer data between two clock domains. For this purpose, the Coregen allows the write and read port of the FIFO to be configured with different clocks, and if the storage capacity does not need to be large, distributed RAM may be used as underlying memory. Another feature is to provide read and write port with different port widths, with the restriction that the ratio between read port width and write port width is 8:1, 4:1, 2:1, 1:2, 1:4 or 1:8. Then, the word order for the smaller port is always MSB first, i.e. for write operations, the larger word will be filled beginning from the higher (most significant) bits, or for read operations, the larger word will be read beginning from the higher bits.

3.2.3 Clocking

All FPGAs discussed in this work feature several independent clock nets to support different clock frequencies throughout the complete chip. Besides the common data paths, clock nets are routed directly to the clock inputs of the FPGA components, such as for the D-flip-flops in the CLBs. To support high frequencies they ensure a clock signal which has low skew, low duty cycle distortion, low voltage and safe jitter tolerance. The topology of the clock nets in Xilinx FPGAs are a mixture of an H-tree and comb. An H-tree would ensure the paths from the clock source to each destination component to be the same. However, since an exact H-tree is difficult to design for this many endpoints (as there are alone for the flip-flops in the CLBs), Xilinx has found other ways for an almost equal clock distribution.

Spartan6 Clocking

For Spartan6 devices, firstly, the clock sources are routed to almost the center of the FPGA containing clock buffer multiplexers and a *Switch Box*. Here, external clock inputs as well as internal clocks are distributed. The clocks are routed through a vertical spine to so called *HCLK Rows*. At the bottom of the HCLK Rows multiplexers select the required clocks for the components maintained by the row. Each HCLK Row features 16 clock nets which are connected to a certain number of primitives, i.e. 16 CLBs, 12 Block RAM blocks and 4 DSPs amongst others [Xil15c].

At the bottom of the HCLK Rows the *Digital Clock Managers (DCMs)* are located as well. The DCM is part of a *Clock Management Tile (CMT)* that contains a *Phase-Locked Loop (PLL)* and two DCMs each. The DCM provides a comfortable way to generate desired output clocks from an input clocks. It consists of four distinct functions: *Delay-Locked Loop (DLL)*, *Skew Adjustment*, *Digital Frequency Synthesizer (DFS)*, and *Variable Phase Shift*.

The DLL provides clean, low-jitter clock outputs and eliminates delay from the external clock input to individual clock loads within the device. Clock skew is eliminated by the global clock network. Outputs of the DLL are `CLK0`, `CLK90`, `CLK180`, `CLK270`, `CLK2X`, `CLK2X180`, and `CLKDV`, providing the unshifted input clock, the input shifted by 90, 180, and 270 degrees, multiplied by two (unshifted and shifted by 180 degress) and divided by an

arbitrary divider between 1.5 and 16. All generated clocks have a 50% duty cycle besides the CLKDV output when configured with a non-integer divider.

Skew Adjustment can be used to minimize the clock skew by providing a suitable clock feedback signal.

The DFS can synthesize a wide-range of clock frequencies based on a configured multiplier and divider for the input clock. The multiplier supports integer values in the range of 2 to 32 while the divider value has to be in the range of 1 to 32. The CLKFX and CLKFX180 outputs hold the synthesized clock unshifted and shifted by 180 degrees respectively.

With an activated Variable Phase Shift the phase of the input clock of the DCM is shifted by an arbitrary value. As a result, all outputs are shifted as well. When used in combination with other DCMs, clock signal pairs with an arbitrary phase shift can be generated.

7-series Clocking

For 7-series FPGAs the clock network is similar to the Spartan6 network with a few architectural abbreviations [Xil15a]. The clock backbone is not located directly in the center of the device and the CMTs are restricted to drive either the top half of the device or the bottom half. Furthermore, the device is divided into clock regions each containing 50 CLBs amongst other primitives. Each clock region can have up to 12 global clock domains. Additionally, the clock nets may now drive not only clock pins, but control pins as well, such as CE for chip enable or SR for a shift command.

The CMTs in the 7-series FPGAs contain one PLL and one *Mixed-Mode Clock Manager (MMCM)*. The MMCM inherits the complete functionality of a DCM extended by a couple of new features which will not be discussed in detail here. The new features include seven arbitrarily configurable clock outputs (CLKOUT0 to CLKOUT6) whereby the first four are provide as negated clock output as well (CLKOUT0B to CLKOUT3B), a dynamic phase shifter for varying the phase during runtime, and the possibility to cascade clocks to allow a wider range for frequency synthesis. Additionally, the multiplier and divider for the DFS support a wider range of configuration values as well, i.e. 1, or 2.000 to 128.000 in increments of 0.125 for the divider, and 2.000 to 128.000 in increments of 0.125 for the multplier.

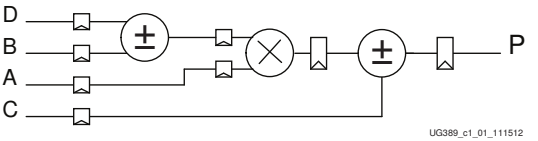


Figure 3.8. Simplified DSP48A1 Slice with Pre-Adder. (Source: [Xil14d], page 8, figure 1-1)

3.2.4 Digital Signal Processor

Mainly *Digital Signal Processors (DSPs)* provide dedicated circuits for multiply or multiply-add operations. However, the Spartan6 DSP48A1 slice supports additional features, such as accumulation, multiply-accumulation, pre-adder/subtractor followed by multiply-accumulation, magnitude comparison, wide bus multiplexer etc. Furthermore, the DSP48A1 slice allows the connection of several DSP slices to support wide math functions. Besides the direct usage of DSPs for previously mentioned functions, the main applications for DSPs are *Finite Impulse Response (FIR)* filters. These filters are extensively used in applications including Wireless Communications or Multimedia Applications such as Image Processing or Video Filtering [Xil14d].

In particular, a Spartan6 DSP slice basically consists of a pipeline of a two-input 18bit pre-adder, an 18x18bit two's complement multiplier with 36bit result, sign extended to 48bits, and a two-input 48bit post-adder with optional registered accumulation feedback. Each pipeline step may optionally be registered, and additional components can be used for cascading several DSPs. Figure 3.8 shows a simplified diagram of a Spartan6 DSP48A1 slice.

The 7-series DSP slices are referred to as DSP48E1 slices. These slices support the same functionality as the DSP48A1 slices, but with significant enhancements. The port width of the pre-adder ports is increased to 25bit and 30bit respectively, the multiplier performs 25x18bit two's complement multiplications with an optional dynamic bypass, and the 48bit post-adder is replaced by an *Arithmetic Logic Unit (ALU)* supporting ten different logic functions of the two operands. Furthermore, a pattern detector and 17bit

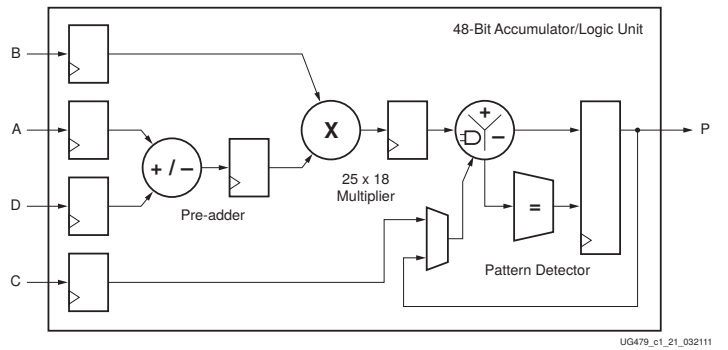


Figure 3.9. Basic DSP48E1 Slice Functionality. (Source: [Xil14a], page 9, figure 1-1)

shifter are included, and the ALU can be operated in *Single-Instruction-Multiple-Data (SIMD)* mode supporting dual 24bit or quad 12bit addition, subtraction or accumulation [Xil14a]. A simplified diagram of a 7-series DSP48E1 slice can be seen in Fig. 3.9.

Spartan3 FPGAs generally do not feature DSP support besides the Spartan3A DSP series, which will not be discussed in this work.

3.3 FPGA Design Flow

The development of an FPGA-based application is generally divided into two parts, firstly, the creation of an FPGA-design, and secondly, the construction of a host software controlling the hardware and data transfer. Usually, both parts are connected via some communication interface. While the software development can be done using any programming language as long as the communication interface can be accessed, which is usually provided by some kind of *Application Programming Interface (API)* of the underlying architecture, the creation of the FPGA-design is restricted to a development kit provided by the FPGA manufacturer. For Xilinx FPGAs, the universal development software has been the *Integrated System Environment (ISE)*. Unfortunately, support for this software has been cancelled in October 2013 with the latest version being 14.7. It was replaced by the *Vivado Design*

Suite software. Vivado has been developed by Xilinx for 7-series and newer FPGAs. With a new GUI front-end and better multi-core support, it mainly performs the same tasks as ISE though, but it does not support older devices than 7-series FPGAs such as Spartan3 or Spartan6. Anyway, the main focus of this thesis lies on the development of FPGA designs, mainly for Spartan6 FPGAs. Thus, the toolchain required for the creation of an FPGA design will be explained according to the ISE toolchain. General software development will not be discussed any further here, although a description of the software API for the RIVYERA architecture and the KC705 development board can be found in Sect. 3.4.2 and Sect. 3.5.1 respectively.

The first step in developing an FPGA application is the creation of a hardware design using a hardware description language. The first compilation step is referred to as *synthesis*, followed by *translate* and the generally very time consuming steps *map* and *place-and-route*. The output is a netlist which is quickly converted to a binary configuration file, also referred to as *bitfile*, in a last step. The bitfile contains all the information required by the FPGA to configure the interconnection of the internal components, such as the CLBs, BRAM, DSPs etc. via the routing matrix, as well as the configuration of primitives, such as the contents of the LUTs in the CLBs, the multiply and divide constants for the clock frequency synthesizers in the DCMs or the direction and voltage characteristics of IOs. The configuration can be loaded via reserved pins of the FPGA die or a JTAG interface (mostly used in the development phase or for debugging purposes). Very often, the FPGA configuration is stored in a directly attached flash memory which is read directly after powerup. This procedure is often necessary for embedded systems since FPGAs generally do not keep their configuration after powerdown. The next paragraphs describe the compilation steps in more detail.

3.3.1 Hardware Description with VHDL

To describe the hardware in an FPGA design two popular hardware description languages are supported by ISE and Vivado. The first is *Verilog* and the second is *VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language)*. ISE and Vivado also support the creation of an FPGA

design by drawing block diagrams using the schematic editor. However, this thesis concentrates on hardware description using VHDL. Since VHDL allows plenty of language constructs, only a very basic instruction into the VHDL language will be given here. It is referred to e.g. [Zwo03] or [AL06] for more information on this versatile hardware description language.

An FPGA hardware description generally consists of a couple of *entities* forming the complete design. The *top-level-entity* includes all other entities and describes the IO interface of the FPGA. The interface of an entity is referred to as *port* and the functionality is described in its *architecture*. The architecture may contain direct signal assignments which will not be buffered. For the description of registers the developer may use the *process* construct where conditions on clock signals can comfortably be expressed. Other entities or FPGA primitives may directly be instantiated in the architecture as well. Each architecture may contain local signal, type, constant or entity declarations at the beginning. Furthermore, process constructs allow the use of *variables*. *Procedure* or *function* calls are also available within VHDL.

One of the most used data types is `std_logic` which mainly describes a binary value (`'0'` or `'1'`) of a signal. However, the `std_logic` datatype allows more values which can be very helpful in simulation, i.e. `'U'` for *undefined* or `'X'` for a strong conflict amongst others. Signals of `std_logic` type can also be combined to *vectors* for signals with a signal width of more than one bit, i.e. `std_logic_vector`. Vectors may also be regarded and directly declared as `unsigned` or `signed` data types, depending on whether the signal is used as a signed or unsigned integer value. *Libraries* allow the direct use of implicit functions, such as the addition ("+") for signed or unsigned data types. Last but not least, VHDL is case insensitive, so expressions written in upper or lower case depend on the developers style and make no difference in behaviour.

Listing 3.1 shows the description of a full adder in VHDL.

If a half adder component is available, it could be instantiated and directly used within the full adder architecture. The architecture description may look like the example in Listing 3.2.

The `generic` construct can be used to describe entities with variable settings, e.g. the port width of some signals. The generic constants must

Listing 3.1. Full adder in VHDL.

```
library ieee;
use ieee.std_logic_1164.all;

entity FullAdder is
port (
  a  : in std_logic;
  b  : in std_logic;
  ci : in std_logic;
  s  : out std_logic;
  co : out std_logic
);

architecture Behavioral of FullAdder is
begin
  s <= a xor b xor ci;
  co <= (a and b) or (a and ci) or (b and ci);
end Behavioural;
```

be known at compile time. Listing 3.3 describes a clocked two-input adder with a generic input width (default is 32bit) and registered output using the `numeric_std` library for the definition of the addition "+". The carry bit will be omitted in this example.

3.3.2 Implementation

Synthesis

The *synthesis* step in the ISE toolchain gives a first assumption of which and how many resources of the target FPGA will be required for the implementation of the hardware design. After a syntax check including a consistency check for port widths, the synthesis performs a detailed analysis of the VHDL code and recognizes basic components in the description such as adders, subtractors, multipliers, multiplexers, state machines, shifters etc. It also chooses the optimal encoding for state machine states and estimates a possible clock frequency for the input clock based on the interconnection

Listing 3.2. Architecture of a full adder with available half adder components.

```
architecture Behavioral2 of FullAdder is
  component HalfAdder is
  port (
    a  : in std_logic;
    b  : in std_logic;
    s  : out std_logic;
    co : out std_logic
  );
  signal ha1_s : std_logic;
  signal ha1_co : std_logic;
  signal ha2_s : std_logic;
  signal ha2_co : std_logic;
begin
  ha1 : HalfAdder
  port map (
    a  => a,
    b  => b,
    s  => ha1_s,
    co => ha1_co
  );
  ha2 : HalfAdder
  port map (
    a  => ha1_s,
    b  => ci,
    s  => ha2_s,
    co => ha2_co
  );
  s <= ha2_s;
  co <= ha1_co or ha2_co;
end Behavioral2;
```

Listing 3.3. Clocked adder with generic input width.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity RegisteredAdd is
generic (
  PORT_WIDTH : integer := 32
);
port (
  clk : in std_logic;
  a   : in unsigned(PORT_WIDTH-1 downto 0);
  b   : in unsigned(PORT_WIDTH-1 downto 0);
  s   : out unsigned(PORT_WIDTH-1 downto 0)
);

architecture Behavioral of RegisteredAdd is
begin
  add_p: process
  begin
    -- signals are only updated at the positive clock edge
    wait until rising_edge(clk);
    s <= a + b;
  end process add_p;
end Behavioral;
```

of these components. The resource usage is estimated by combining the known resource requirements for the single components. For example, if the synthesis recognizes a 32bit 2-to-1 multiplexer in the design, it already knows the number of CLBs required to implement such a multiplexer and adds this number to the total number of currently required CLBs. The synthesis also runs a constant clearing, i.e. constant signal values are recognized and evaluated throughout the whole design.

After the synthesis process, the developer gets a rough estimate of resource usage on the target FPGA and a decision if there is a chance that the complete implementation succeeds. For example, if his design contains

more BRAM primitives than the device supports, he will be notified that the following map and place-and-route processes will fail. However, even if the synthesis succeeds, the results are only estimates. The precise resource requirements will be determined during the map and place-and-route processes, and thus, the final decision if the design is implementable or not will be certain after finishing these processes.

Translate

The *translate* process applies user constraints defined in *User Constraints Files (UCFs)* as well as system-wide constraints to the synthesized user design. These constraints include the voltage configuration of input/output ports, their location on the die, and special attributes for internal signals, such as *Timing Ignore (TIG)*. It also tests if the chosen locations and configurations of the ports are feasible, e.g. a clock input can only be located at special clock pins and may have restricted voltage specifications. Finally, clock period constraints are applied to the clock nets, which will be required by the map and place-and-route processes to ensure that timing will be met at the end. The clock frequency prescribes the maximum delay between two registers and includes the maximum length of a path between two components.

Map

The *map* process takes the synthesis results and maps the recognized basic components into available FPGA primitives. It also identifies unused signals and components which may result in resource requirements less than the synthesis process has estimated before. Furthermore, resources may be shared if the same functionality with exactly the same inputs is required more than one time, or components can be doubled if the fan-out from one component would be too high. The output of the map process is an abstract network of connected primitives of the target FPGA implementing the user design. The locations of the components is not yet fixed. This will be accomplished in the next step. If the map process fails, it is mostly due to a lack of available resources.

Place-and-Route

Place-and-Route describes the process of placing the required components from the network on the target FPGA, i.e. mapping a primitive from the network to a particular primitive on the target FPGA, and the routing of the connections, i.e. connecting the primitives on the target FPGA via the routing matrix. Sometimes resource requirements may increase during this process, e.g. if there are not enough routing channels provided by the routing matrix and the routing has to be done through additional CLBs, or if resources have to be multiplied for the same reasons. Furthermore, place-and-route obeys the constraints defined in the translate process to meet timing for the desired clock frequencies. The place-and-route output is a constrained netlist which particularly describes the behaviour of each primitive on the target FPGA. This netlist can quickly be converted to a binary configuration file used to configure the FPGA. The last part in place-and-route is a timing analysis which helps the developer to identify long paths in his design in the case the timing has failed.

Unfortunately, the problem of finding the optimal solution for the map and place-and-route processes is np-hard. To compensate long computation times, Xilinx has implemented an approximation using cost tables and different implementation strategies. However, large designs with tight timings may still require computation times of several hours, as for some of the presented applications in this thesis. Furthermore, the user does generally not know which strategy or cost tables are optimal for his design, or even lead to a successful implementation. That Xilinx keeps a secret of the particular implementation of its strategies and the contents of the cost tables does neither help in this case. Anyway, Xilinx provides the *SmartXplorer* tool for ISE which tries out several different user-definable strategies. For Linux systems SmartXplorer features multi-host support such that different strategies can be run on different hosts at the same time.

3.4 RIVYERA

The FPGA-computer *RIVYERA* was introduced in 2009 as *COPACOBANA 5000* with the primary intention to target bioinformatics ap-

plications [PBS+09]. In contrast to its predecessor *COPACOBANA 1000*, which was developed in 2006 to break the DES cipher in less than one week [KPP+06], the new architecture should reduce the data transfer and storage bottleneck by improving the architecture and speed of the underlying bus system as well as providing directly attached DRAM to the in-built FPGAs. While COPACOBANA 1000 consists of 120 Xilinx Spartan3-1000 FPGAs connected via a single-master multiple-slave shared bus, the two available RIVYERA types RIVYERA S3-5000 and RIVYERA S6-LX150 contain up to 128 Xilinx Spartan3-5000 FPGAs or 256 Spartan6-LX150 FPGAs respectively. Each FPGA in the RIVYERA S3-5000 has 32MB of directly attached SDRAM, while each RIVYERA S6-LX150 FPGA features 2x256MB of DDR3-RAM. The basic structure for the RIVYERA bus system is a ring with point-to-point connections. See Sect. 3.4.1 for details. RIVYERA is now developed and distributed by SciEngines GmbH [SE].

The first bioinformatics applications presented on the new architecture were the Smith-Waterman optimal alignment algorithm and a DNA motif search algorithm called BMA [WBB+10; SWG+10]. Other applications such as BLASTp (Sect. 4.3), BOOST (Sect. 5.3), iLOCi (Sect. 5.4) and SHAPEIT2 (Sect. 6.3) followed. The RIVYERA has also been successful in the area of cryptanalysis by performing a dictionary attack on the TrueCrypt encryption software [ARW+12; AVW+14] and for stock market prediction [SGW+12a; SGW+12b].

3.4.1 RIVYERA Architecture

RIVYERA S3-5000 and RIVYERA S6-LX150 share a common architecture and bus structure. The design is divided into two main parts, a common server grade mainboard with peripherals acting as host, and the FPGA computer. The first part, the host, can be equipped and configured as any standard server system. The systems used for this thesis contain an Intel Core i7-930 quad-core processor with 2.8GHz and 12GB of DRAM for the RIVYERA S3-5000 and two Intel Xeon E5-2620 hexa-core processors with 2GHz and 128GB of DRAM for the RIVYERA S6-LX150. The second part, the FPGA computer, consists of a backplane with 16 slots for FPGA cards. Each FPGA card consists of up to 8 user FPGAs, either Spartan3-5000 or

Spartan6-LX150, and one controller FPGA. In the case of S6-LX150 cards, it is possible to equip both sides of the backplane such that 16 FPGAs per card are possible. However, the systems used for this thesis both contain 128 user FPGAs each, distributed over 16 cards.

The user FPGAs on an FPGA card together with the controller FPGA are each connected with two neighbors via fast point-to-point connections forming a ring. The controller FPGA handles the communication link to the backplane slot. Each backplane slot is again connected via point-to-point connections to its neighboring slots. Thus, the backplane slots form an array. Additionally, at least one FPGA card hosts a cable connection to a PCIe controller card in the host system, which states the communication link between host and FPGA computer. Theoretically, more than one controller uplink is possible to accelerate data transfer, i.e. each FPGA card may connect to a separate PCIe controller. However, it is not available for the systems used in this thesis. The communication between the components is realized by the RIVYERA API described in Sect. 3.4.2.

Furthermore, each Spartan3 user FPGA has an attached memory module with 32MB of SDRAM. For the Spartan6 there are two modules of 256MB DDR3-RAM attached to each FPGA. However, the interface to the RAM module must be defined by the developer itself since there is no memory interface controller. At least for Spartan6, the developer can use the *Memory Interface Generator (MIG)* provided by Xilinx to connect to the DRAM.

RIVYERA allows small packaging, e.g. RIVYERA S6-LX150 is packed in a rack-mountable standard server housing using 4HUs. Figure 3.10 shows the RIVYERA architectural design and a photo of a RIVYERA system.

3.4.2 RIVYERA API

To ensure usability, an intelligent routing scheme for the bus system has been implemented in the RIVYERA API. The API includes the communication interface for software and hardware to provide a transparent connection for the developer between host and FPGA or any two FPGAs by an automatic routing of data packets. The API provides broadcast facilities and methods for configuring the user FPGAs during system runtime. Broadcasting can be done in three different ways: slot, FPGA or total broadcast.

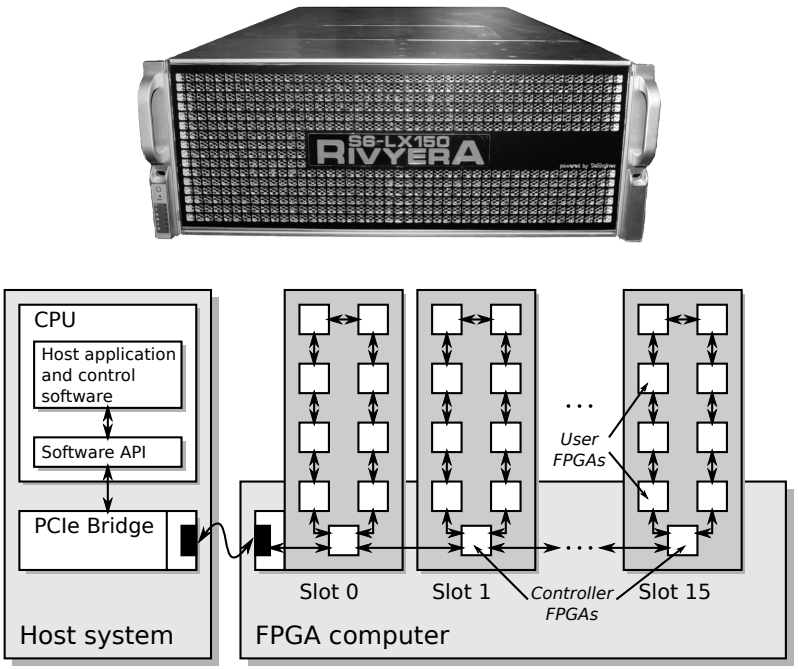


Figure 3.10. Picture and concept of the RIVYERA architecture.

Slot broadcast addresses all FPGAs on one slot, FPGA broadcast addresses one FPGA on all available slots, and total broadcast selects all available FPGAs [Sci13a; Sci13b]. In the following, the interface for software and hardware development will be described in more detail.

Software API

The software interface is written in the programming language C, but an interface for Java using the *Java Native Interface (JNI)* is provided by the manufacturer as well. In this thesis, exclusively the C-interface has been used. The following list of API methods is incomplete but states all necessary methods for an efficient communication. All data transferring

methods include a `timeout` field, which can be arbitrarily set to a numerical value or to `SE_TIMEOUT_INFINITE` for an infinite timeout. The return value of each function is an enumeration value and should always be `SeApiSuccess`, otherwise an operation has failed, which can happen by a timeout as well.

▷
```
SE_STATUS se_allocMachine(se_machine_t       machine,
                          const SE_OPTIONS_T *pOptions);
```

Before the RIVYERA can be used, the user has to allocate the system. This prevents other potential users from intervening in a running application. `machine` is the machine index and typically zero.

▷
```
SE_STATUS se_freeMachine(se_machine_t machine);
```

This method releases the system and allows the allocation by other users.

▷
```
SE_STATUS se_program(se_machine_t  machine,
                     const SE_ADDR *pAddr,
                     const char    *pFilename,
                     se_time_t     timeout);
```

Configurates the addressed FPGA with the bitstream file provided by `*pFilename`. Although the configuration of a single FPGA is possible, the user should program all FPGAs by using a broadcast address first, such that all FPGAs contain the API core to enable communication. `timeout` should be greater than 5ms, otherwise the configuration might fail.

▷
```
SE_STATUS se_write(se_machine_t     machine,
                   const SE_ADDR    *pAddr,
                   const __uint64_t *pPayload,
                   size_t           size,
                   size_t           *pCount,
                   se_time_t        timeout);
```

Data in `*pPayload` is written to the specified FPGA. `size` determines the number of words (not bytes!) to be transmitted. The address may be a broadcast address. The method blocks until all data is written or

the timeout is exceeded. *pCount holds the number of truly transmitted words.

```
▷ SE_STATUS se_read(se_machine_t  machine,
                    const SE_ADDR *pAddr,
                    __uint64_t    *pPayload,
                    size_t        size,
                    SE_READMODE   mode,
                    size_t        *pCount,
                    se_time_t     timeout);
```

Requests or awaits data from the specified FPGA. size determines the number of requested words (not bytes!). There are three available read modes specified by mode. Firstly, SeReadActive sends a read request to the specified FPGA and awaits the reply. Secondly, SeReadPassive awaits data without sending a request. Both modes cannot be used with a broadcast address. The third mode, SeReadRequest, only sends a read request, but may do this to a broadcast address. The user can fetch the data from the selected FPGAs by a passive read then. The method blocks until the requested data is received or the timeout is exceeded. *pCount holds the number of truly received words.

```
▷ SE_STATUS se_waitForData(se_machine_t machine,
                           se_contr_t   controller,
                           SE_ADDR      *pAddr,
                           size_t       *pCount,
                           se_time_t    timeout);
```

This method waits for data on the specified controller, which is zero for systems with only one communication controller between host and FPGA computer. This method blocks until data is available or the timeout is exceeded. After returning, *pAddr holds the address of the FPGA where data is available from and *pCount specifies the number of available words.

```
▻ SE_STATUS se_getSlotCount(se_machine_t machine,
                            se_slot_t    *pSlotCount);

▻ SE_STATUS se_getFPGACount(se_machine_t machine,
                            se_slot_t    slot,
                            se_fpga_t    *pFPGACount);
```

To determine the system topology, these methods are useful to identify the number of available slots, and for each slot the number of available FPGAs.

Hardware API

The RIVYERA hardware API consists of a pre-compiled core which has to be used by the developer to enable the communication facilities of the RIVYERA FPGA. The interface of this core is provided to the developer and is described in the following. The 50MHz clock input and the reset pin as well as a user LED output is not listed here.

1. Address information ports:

```
▻ api_self_contr_in      : in    seFlag_type;
```

Indicates if the FPGA is located at a slot with a communication controller to the host.

```
▻ api_next_contr_in      : in    seSlotAddr_type;
  api_prev_contr_in      : in    seSlotAddr_type;
```

Indicate the slot addresses of the next slot in ascending or descending direction respectively without the own slot address. The own slot address will only be provided if the only controller in the system is located here.

```
▻ api_self_slot_in       : in    seSlotAddr_type;
  api_self_fpga_in       : in    seFpgaAddr_type;
```

Indicates the own FPGA address.

2. Direction from host to FPGA:

▷ `api_i_clk_out : out std_logic;`

The clock signal where all output port signals are synchronized with. This is useful when the clock frequency of the user design differs from the API clock frequency.

▷ `api_i_src_slot_in : in seSlotAddr_type;`
`api_i_src_fpga_in : in seFpgaAddr_type;`
`api_i_src_reg_in : in seRegAddr_type;`

The source address of incoming data.

▷ `api_i_src_cmd_in : in seCmd_type;`

The command which was used to send the data. In most cases this should be `CMD_WR`. This can be used as an additional data flag in FPGA to FPGA communication.

▷ `api_i_tgt_reg_in : in seRegAddr_type;`

The register address this data is intended for. Note that if the input datum is a read request this value has to be set as source register for the reply.

▷ `api_i_tgt_cmd_in : in seCmd_type;`

The command for the target FPGA. This could be either `CMD_WR` if this is simple user data, or `CMD_RD` if this is a read request. To prevent blocking, a read request should be answered immediately.

▷ `api_i_data_in : in seData_type;`

The 64bit input data word.

▷ `api_i_empty_in : in seFlag_type;`
`api_i_am_empty_in : in seFlag_type;`

Indicates an empty or almost empty (i.e. only one word is left) input FIFO respectively.

```
▷ api_i_rd_en_out        : out   seFlag_type     := '0';
```

Reads one data word from the input FIFO. The input FIFO has *First Word Fall Through (FWFT)* behaviour, i.e. incoming data is updated at the input ports as soon as it is available, and will be acknowledged and removed in the clock cycle after asserting this signal.

3. Direction from FPGA to host:

```
▷ api_o_clk_out          : out   std_logic;
```

The clock signal where all input port signals are synchronized with. This is useful when the clock frequency of the user design differs from the API clock frequency.

```
▷ api_o_rfd_in           : in    seFlag_type;
```

This signal indicates *ready-for-data*, i.e. the output FIFO is not full and data can be sent via the API.

```
▷ api_o_tgt_slot_out     : out   seSlotAddr_type := (others => '0');
  api_o_tgt_fpga_out     : out   seFpgaAddr_type := (others => '0');
  api_o_tgt_reg_out      : out   seRegAddr_type  := (others => '0');
```

The target address for the data.

```
▷ api_o_tgt_cmd_out      : out   seCmd_type      := CMD_WR;
```

The target command. This can be either `CMD_WR` for simple user data, or `CMD_RD` if this is a read request to another FPGA. Note that read requests to the host are ignored.

```
▷ api_o_src_reg_out      : out   seRegAddr_type  := (others => '0');
```

The source register of the data. This value must match the register address specified in a read request.

```
▷ api_o_src_cmd_out      : out   seCmd_type      := CMD_WR;
```

The command used to send this data. In most cases, this should be `CMD_WR`. This can be used as an additional data flag in FPGA to FPGA communication.

```
▷ api_o_data_out         : out   seData_type     := (others => '0');
```

The 64bit data word to be written.

```
▷ api_o_wr_en_out        : out   seFlag_type     := '0';
```

Signalizes the pending data word to be inserted in the output FIFO. If `api_o_rfd` is not asserted when this signal is asserted, the insertion will not be performed.

3.5 The KC705 Development Board

The RIVYERA architecture had been developed in the first place to tackle the bottleneck of a low bus transfer speed and low data storage capabilities of the COPACOBANA system. The situation has been improved, but with the growing needs of a fast and reliable handling of big amounts of data, the same parts became a bottleneck again. In order to benefit from newer technology features in the 7-series FPGAs and direct PCI-Express support, this thesis examines the abilities of a single Kintex7-325T FPGA on the Xilinx KC705 Connectivity Kit development board on the basis of a 3rd-order SNP interaction method using the *Mutual Information (MI)* measurement (see Sect. 5.5). The measured net data rate of full-duplex 16Gbit/s with the KC705 API (see Sect. 3.5.1 significantly widens the old bottleneck for transfer speed. Additionally, the board is equipped with 1 GB SODIMM DDR3 memory, i.e. two times more than the attached memory of a RIVYERA S6-LX150 FPGA. Furthermore, the SODIMM RAM-controller provides a 512bit interface while the attached DRAM on the RIVYERA can only be accessed with an interface size of maximal 128bit. This ensures faster access to the FPGA externally stored data.

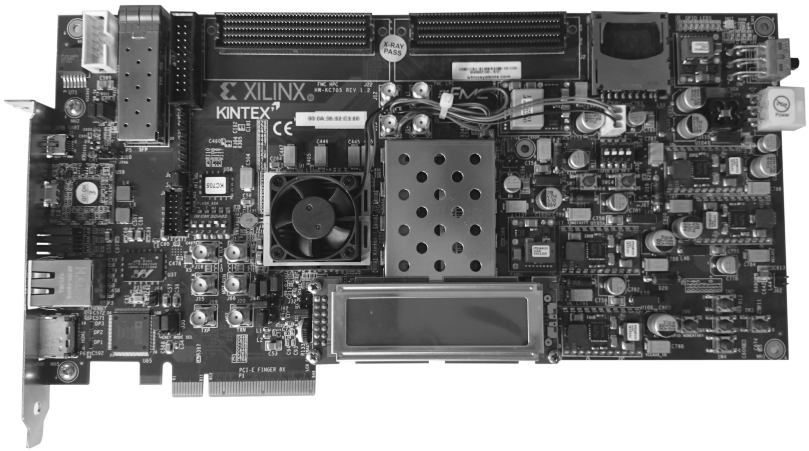


Figure 3.11. The KC705 development board.

The backplane of the KC705 development board is designed as a PCIe expansion card for standard server or desktop PC systems. Besides the already discussed features, the KC705 provides several other memory and communication components which can be used for general hardware development. However, these are not required for this thesis. Figure 3.11 shows a picture of the KC705 development board.

3.5.1 KC705 API

The KC705 API has been developed by Jan Kässens at the Department of Technical Computer Science at the CAU Kiel. It provides a comfortable interface for data transfer on the software and on the hardware side which is described in the following. Unfortunately, the API does not support the configuration of the FPGA. This has to be done via the integrated JTAG interface using e.g. the Xilinx *hw_server* software running on the PC system equipped with the development board and Vivado on the development system.

The API provides a full-duplex data transfer via PCIe with a net data

rate of up to 16Gbit/s. It can be accessed in two different modes, the synchronous and the asynchronous mode, whereby the full data rate can only be achieved in asynchronous mode.

In synchronous mode the provided buffer will either be written or read in blocking mode, while the asynchronous mode is more complex but also more powerful. It features a buffer queue which can be filled by the user with several buffers. The read or write methods are non-blocking and the transfer process will be done in a seperate thread using the provided buffers in the queue and a user definable callback method.

Furthermore, each transfer block supports 64bit of metadata which can be used for status flags or debugging purposes. Details of the hardware and software interface will be described in the following.

Software API

The software API is written in the programming language C. The following list of available methods explains the communication interface for host-side software development.

```
▷ int kc705_open(const char          *xdma_device,
                 const char          *data_device,
                 struct kc705_handle **new_handle,
                 FILE                *log)
```

Before using the communication interface the device has to be opened and a device handle to be created using this method. The return code is `0` on success or a Linux error code on failure.

```
▷ void  kc705_close(struct kc705_handle *handle)
```

Releases the previously created device handle.

```
▻ int kc705_write(struct kc705_handle *hdl,
                  void                *buffer,
                  size_t              buffer_length,
                  size_t              *bytes_written,
                  unsigned long       metadata)
```

Writing in synchronous mode. The contents of the provided buffer will be written to the device together with the specified metadata. The method does not return before all bytes were written or an error occurs. After return `*bytes_written` contains the number of truly written bytes to the device. The return code is `0` on success or a Linux error code on failure.

```
▻ int kc705_read(struct kc705_handle *hdl,
                 void                *buffer,
                 size_t              buffer_length,
                 size_t              *bytes_read,
                 unsigned long       *metadata)
```

Reading in synchronous mode. The method blocks until the requested amount of data has been read from the device or an error occurs. `*metadata` contains the metadata for this transfer and `*bytes_read` specifies the number of truly read bytes from the device. The return code is `0` on success or a Linux error code on failure.

```
▻ int kc705_write_async(
        struct kc705_handle *hdl,
        void                *buffer,
        size_t              buffer_length,
        unsigned long       metadata,
        void (*callback)    (void *buffer, size_t bytes_written,
                             void *user_data, unsigned long metadata),
        void                *user_data)
```

Writing in asynchronous mode. This method returns immediately after queuing the content buffer including the metadata in the internal buffer queue. After the data has been written from a separate thread, the specified callback function is called. Here, the number of truly written bytes

is provided together with the original buffer and metadata. Additionally, the user may specify some `*user_data` that is directly passed by this method to the callback function. The return code is `0` on success or a Linux error code on failure.

```
▻ int kc705_read_async(
        struct kc705_handle *hdl,
        void                *buffer,
        size_t              buffer_length,
        void (*callback)    (void *buffer, size_t bytes_read,
                             void *user_data, unsigned long metadata),
        void                *user_data)
```

Reading in asynchronous mode. This method returns immediately after enqueuing the provided buffer in the internal buffer queue. After the requested data has been written to this buffer, the specified callback function is called providing the buffer, the number of truly read bytes and metadata. Analogue to the asynchronous write function, the user may specify some `*user_data` which is directly passed to the callback function. The return code is `0` on success or a Linux error code on failure.

Hardware API

The following list explains the PCIe interface signals of the KC705 top-level entity in VHDL. The interface size of the data port at the hardware side is 128bit. Firstly, all signals required to read data from the host are explained. Secondly, the signals required for data sending are listed afterwards. The 250MHz clock input and the synchronous reset pin are not listed.

1. Direction from host to FPGA:

```
▻ s2c_data : in  std_logic_vector(127 downto 0);
```

128bit data input.

▷ `s2c_keep : in  std_logic_vector(15 downto 0);`

Each byte of the input data is marked with a *keep*-bit. `'1'` indicates that the data byte is valid, `'0'` if not. Usually, the keep-bits are always set to `'1'`.

▷ `s2c_meta : in  std_logic_vector(63 downto 0);`

Contains the metadata of the current transfer. It is updated with the first datum of a transfer block.

▷ `s2c_valid : in  std_logic;`

Indicates valid input data with `'1'`. This signal is only valid for one clock cycle, i.e. data at the data port must be used within this cycle.

▷ `s2c_last : in  std_logic;`

Indicates the last data word of a transfer block with `'1'`.

▷ `s2c_ready : out std_logic := '0';`

Signalizes the PCIe core with `'1'` that the design is ready to receive data. If set to zero, the PCIe core does not provide new data. Incoming data is buffered and a transfer is finally blocked if the buffer is full.

2. Direction from FPGA to host:

▷ `c2s_data : out std_logic_vector(127 downto 0);`

128 bit data output.

▷ `c2s_keep : out std_logic_vector(15 downto 0);`

Each byte of the output data is marked with a *keep*-bit, analogue to the keep-input. These bits should usually be constantly set to `'1'`;

▷ `c2s_meta : out std_logic_vector(63 downto 0);`

Provides the metadata for the current transfer block. It is sampled only with the first datum of a block.

▷ `c2s_valid : out std_logic := '0';`

Signalizes new output data to the PCIe core with `'1'` and is considered valid only for this cycle. Each asserted cycle marks a new output data word.

▷ `c2s_last : out std_logic := '0';`

Signalizes the last output data word for this transfer block to the PCIe core with `'1'`.

▷ `c2s_ready : in  std_logic;`

Indicates if the PCIe core is ready with `'1'`. If the *valid*-flag is set while `c2s_ready` is not asserted, data at the data output port will be ignored.

Chapter 4

Sequence Alignment

This chapter contains excerpts from [WBB+11; WSS12; Wie13; Wie14] which represent the original publications of this work presented here.

4.1 The Alignment Problem

Sequence alignment is a very popular application area in bioinformatics. It is required mainly for the digitalization and reconstruction of genetic information from a DNA sample and it is used in a sub-process in many other application fields such as short read alignment and biological database search.

Nowadays, the exponential growth of biological sequence data becomes a severe problem if processed on standard general-purpose PCs, e.g. the size of the UniProtKB/TrEMBL [EMB] protein database doubles every 1.5 to 2 years. It grew from ∼2 million sequences in 2005 to ∼80 million sequences in 2014. A picture underlining the exponential growth can be seen in Fig. 4.1. However, when the contents reached about 95 million sequences in early 2015, the operators cleaned up the database and removed about half of the sequences, but in late 2015 the number of entries in the database already reached 52.8 million again with a total of about 17.5 billion amino acids [EMB].

Tackling this problem with large computing clusters is a widely accepted solution, although acquaintance and maintenance as well as space and energy requirements may introduce significant costs. This chapter shows that this problem can be addressed by reconfigurable computing with the FPGA-based computing platform RIVYERA (see Sect. 3.4). Although only a

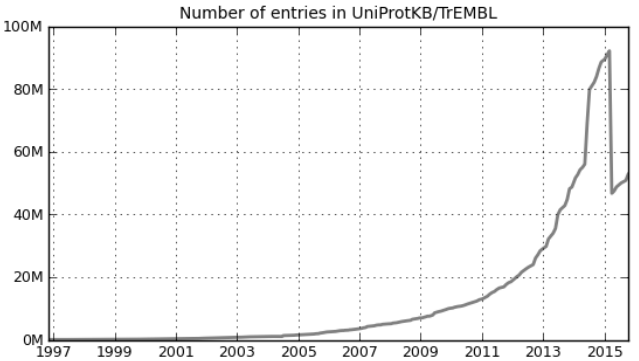


Figure 4.1. Number of protein sequences in UniProtKB/TrEMBL database over recent years showing an exponential growth until early 2015 [EMB].

constant speedup is expected, many applications already benefit from faster alignments to generate more significant results.

In the area of sequence analysis, the implementation of the common protein database search tool BLASTp is presented as an example (see Sect. 4.3). Prior to that the terms *optimal* and *heuristic sequence alignment* are defined and the two well-known optimal sequence alignment algorithms Needleman-Wunsch [NW70] and Smith-Waterman [SW81] are explained.

Whenever the term *alignment* is used within this thesis, it is referred to as *pair-wise sequence alignment* unless stated otherwise. Although *multiple sequence alignment* is also a famous compute intensive problem, it is not being discussed here.

Definition of *pair-wise (global) sequence alignment*: Let $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_m$ be sequences over an alphabet Σ, and let "$-$" denote the gap symbol with $- \notin \Sigma$. The pair of the two equally long sequences $\overline{A} = \overline{a_1}\,\overline{a_2} \dots \overline{a_t}$ and $\overline{B} = \overline{b_1}\,\overline{b_2} \dots \overline{b_t}$ over the alphabet $\overline{\Sigma} = \Sigma \cup \{-\}$ form a *pair-wise sequence alignment* $(\overline{A}, \overline{B})$ if the following two conditions are given:

1. Removing all gaps from $\overline{A}$ produces A and removing all gaps from $\overline{B}$ produces B.
2. For each i: $\overline{a_i} \neq \{-\}$ or $\overline{b_i} \neq \{-\}$.

Refinement definition of *global*, *semi-global* and *local* alignments: An alignment can be either *global*, *semi-global* or *local*. Let $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_m$ be sequences over an alphabet Σ. The definition for a *global* sequence alignment is directly conform with the general definition of an alignment above. For a *semi-global* and *local* alignment, condition 1. has to be weakened. The pair of the two equally long sequences $\overline{A} = \overline{a_1}\ \overline{a_2} \dots \overline{a_t}$ and $\overline{B} = \overline{b_1}\ \overline{b_2} \dots \overline{b_t}$ over the alphabet $\overline{\Sigma} = \Sigma \cup \{-\}$ form a *global*, *semi-global* or *local* sequence alignment $(\overline{A}, \overline{B})$ respectively if the following two conditions are given:

1. (a) *global:* Removing all gaps from $\overline{A}$ produces A and removing all gaps from $\overline{B}$ produces B.
   (b) *semi-global:* Removing all gaps from $\overline{A}$ produces A and removing all gaps from $\overline{B}$ produces a subsequence of B.
   (c) *local:* Removing all gaps from $\overline{A}$ produces a subsequence of A and removing all gaps from $\overline{B}$ produces a subsequence of B.
2. For each i: $\overline{a_i} \neq \{-\}$ or $\overline{b_i} \neq \{-\}$.

Definition of a *scoring matrix*: Let $a, b \in \overline{\Sigma}$. A *(linear) scoring matrix* S defines a score for each pair of opposing characters (including the gap symbol) in an alignment. Thus the scoring matrix can be seen as a function:

$$S(a, b) : S(\overline{\Sigma} \times \overline{\Sigma}) \to \mathbb{Z}. \tag{4.1.1}$$

Commonly, the score for two equal characters is higher than for two different characters. The former is referred to as *match* while the latter is referred to as *mismatch*. There is the possibility to define *affine* gap penalties which is generally used to distinguish a *gap opening* from a *gap extension*. For simplicity, by defining two different gap symbols for opening "$-_o$" and extension "$-_e$" the definition of the linear scoring matrix can be used.

However, with this modification the definition of an alignment needs to be refined formally.

Refinement definition of alignments for use with affine gap penalties: Let $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_m$ be sequences over an alphabet Σ. The pair of the two equally long sequences $\overline{A} = \overline{a_1}\ \overline{a_2} \dots \overline{a_t}$ and $\overline{B} = \overline{b_1}\ \overline{b_2} \dots \overline{b_t}$ over the alphabet $\overline{\Sigma} = \Sigma \cup \{-_o, -_e\}$ form a *global*, *semi-global* or *local* sequence alignment $(\overline{A}, \overline{B})$ respectively if the following conditions are given:

1. (a) *global:* Removing all gaps from $\overline{A}$ produces A and removing all gaps from $\overline{B}$ produces B.
   (b) *semi-global:* Removing all gaps from $\overline{A}$ produces A and removing all gaps from $\overline{B}$ produces a subsequence of B.
   (c) *local:* Removing all gaps from $\overline{A}$ produces a subsequence of A and removing all gaps from $\overline{B}$ produces a subsequence of B.
2. For all i: $\overline{a_i} \notin \{-_o, -_e\}$ or $\overline{b_i} \notin \{-_o, -_e\}$.
3. For $i = 1$: $\overline{a_1} \neq \{-_e\}$ and $\overline{b_1} \neq \{-_e\}$
4. $\forall i > 1$: $\overline{a_i} = \{-_e\} \Rightarrow \overline{a_{i-1}} \in \{-_o, -_e\}$ and $\overline{b_i} = \{-_e\} \Rightarrow \overline{b_{i-1}} \in \{-_o, -_e\}$
5. $\forall i > 1$: $\overline{a_i} = \{-_o\} \Rightarrow \overline{a_{i-1}} \notin \{-_o, -_e\}$ and $\overline{b_i} = \{-_o\} \Rightarrow \overline{b_{i-1}} \notin \{-_o, -_e\}$

Conditions 3. to 5. describe formally that a gap extension must follow a gap opening and that a gap opening cannot follow another gap opening or extension.

Definition of the *score* of an alignment: The *score* s of an alignment is defined by the sum of the corresponding entries in the scoring matrix for each of the opposing character pairs in an alignment. Let (A, B) be an alignment and n its length. The score of the alignment using the scoring matrix S is:

$$s(A, B) = \sum_{i=1}^{n} S(a_i, b_i) \tag{4.1.2}$$

	A	T	G	C	N
A	5	−4	−4	−4	−2
T	−4	5	−4	−4	−2
G	−4	−4	5	−4	−2
C	−4	−4	−4	5	−2
N	−2	−2	−2	−2	−1

Gap open penalty: −10
Gap extension penalty: −1

Figure 4.2. Extract of the NUC.4.4 scoring matrix [NCBb] for the basic nucleotides.

```
ACGCTTTGAATACAC
|||||xxxx||xxxx
ACGCTGAATATTCAC
```

Alignment score = 30

```
ACGCTTTGAATA--CAC
|||||  |||||  |||
ACGCT--GAATATTCAC
```

Alignment score = 43

Figure 4.3. Two example alignments of the same sequences leading to different scores. This reflects the necissity of introducing gaps.

To underline the necessity to introduce gaps Fig. 4.3 shows an example of two alignments of the same nucleotide sequences, one without and one with gaps. The scores are calculated using the NUC.4.4 scoring matrix supplied by the NCBI [NCBb] (see Fig. 4.2). Although paying a high *gap penalty*, i.e. a high negative score for gaps, the alignment with the gap reaches a higher score than the alignment without gaps due to multiple mismatches. Depending on which sequence a gap is placed in, it may be referred to as an *insertion* or *deletion*. The differentiation of both terms may be confounding because it may be unclear in which of both sequences something is inserted or deleted and what is inserted or deleted, a character or a gap. Since scoring functions are generally symmetric, and therefore do not distinguish between both terms, insertions and deletions are uniformly referred to as gaps in this thesis.

In conclusion, an alignment is per definition an opposition of two arbitrary sequences of the same alphabet including gaps and length, weighted by its score. However, the usage of the terms “find an alignment” or “to align one sequence to another” is mostly referred to finding an alignment with a high score, e.g. if a read sequence from a sequencer is “to be aligned” to a reference sequence, the user is interested in finding the part or parts

of the reference sequence where the consensus is maximal, i.e. the score is maximized. Such alignments are considered *optimal* and can only be predictably found at higher computational costs (see Sect. 4.2). Thus, many alignment tools provide a tradeoff between quality and runtime savings, such as BLAST (see Sect. 4.3).They don't guarantee to find the optimal alignment, but do so in most cases, which is sufficient if there are enough sequences to be aligned. Their big advantage is that they generally outperform optimal alignment algorithms by far in terms of computing time. Such algorithms that do not necessarily find the optimal alignment are often referred to as *heuristic*.

4.2 Optimal Sequence Alignment

A sequence alignment $(\overline{A}, \overline{B})$ is considered *optimal* if the score $s(\overline{A}, \overline{B})$ is maximal, i.e.

$$s_o(\overline{A}, \overline{B}) = \max\left\{s(\overline{A}, \overline{B}) | (\overline{A}, \overline{B}) \text{ is alignment of sequences } A \text{ and } B\right\} \tag{4.2.1}$$

There exist two algorithms which identify global optimal and local optimal sequence alignments, the Needleman-Wunsch [NW70] algorithm for global alignments and the very similar Smith-Waterman [SW81] algorithm for local alignments. Obviously, an optimal alignment does not need to be unique, but both algorithms are able to find the complete number of optimal alignments.

Although providing the best quality with an optimal alignment, the runtime and memory requirements of both algorithms are of quadratic complexity, in particular $\mathcal{O}(n \cdot m)$ if n and m are the lengths of the sequences to be aligned. Thus, directly calculating the optimal alignment for real-world alignment problems is rather unusual. However, many heuristic alignment algorithms calculate the optimal alignment for shorter subsequences of the original sequences often determined by seeding, e.g. BWA [LD09], Bowtie [LTP+09], GASSST [RL10], SOAP2 [LYL+09], CUSHAW [LSM12] and others. BLAST (see Sect. 4.3) uses optimal alignments in its post-processing as well. Thus, the two available algorithms are explained in more detail in

the following.

4.2.1 Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm was already introduced in 1970 by Saul B. Needleman and Christian D. Wunsch [NW70]. It finds optimal *global* alignments of two input sequences A and B and consists of two main steps. Firstly, in order to find the optima out of every possible alignment of the two input sequences, an alignment matrix $H \in \mathbb{Z}^{(n+1)} \times \mathbb{Z}^{(m+1)}$ is calculated, whereby n and m are the corresponding lengths. The process uses a simple scoring function distinguishing between a match or mismatch of two characters and the insertion of a gap. The score for match or mismatch is taken from the scoring matrix S and let $g < 0$ denote a linear gap penalty. Then, the alignment matrix is calculated as follows (for affine gap penalties this has to be adapted accordingly with different gap penalties for gap opening and extension). For all $i, j > 0$:

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S(a_i, b_j) & \text{match/mismatch} \\ H_{i-1,j} + g & \text{gap opening/extension (insertion)} \\ H_{i,j-1} + g & \text{gap opening/extension (deletion)} \end{cases} \tag{4.2.2}$$

Furthermore, let the upper and left border values be $H_{i,0} = i \cdot g$ and $H_{0,j} = j \cdot g$ for all $i \geqslant 0$.

After the calculation of the alignment matrix, a backtracking step is performed to generate the final alignment. Summarized, the backtracking starts at the lower right corner $H_{n,m}$ of the alignment matrix. The cell entry already states the score of the final alignment. The backtracking follows the path through the alignment matrix which reflects the chain of matrix cells whose values were taken for the maximum calculation in Eq. 4.2.2. For each chosen direction, either *up*, *left*, or *up-left*, the corresponding character or gap is inserted into the final alignment. The backtracking stops if the upper left corner $H_{0,0}$ is reached. Since the backtracking path is not necessarily unique, each path defines another optimal alignment for the two input sequences. For a detailed description of the backtracking step, the original publication [NW70] is referred to.

Clearly the calculation of the alignment matrix is of quadratic runtime complexity $\mathcal{O}(n \cdot m)$ and the backtracking for one alignment is of linear runtime complexity $\mathcal{O}(n + m)$. For applications where many different sequences have to be aligned to filter only a subset as candidates for good alignments, the algorithm may be used to only determine the alignment score without the backtracking. In this case, the memory requirements are quite low since it is not necessary to store the complete alignment matrix. For the calculation it is sufficient to store only one column or one row. Hence the memory complexity is linear in $\mathcal{O}(\min\{n, m\})$. In the general case, when a backtracking is required, the complete alignment matrix has to be stored and the memory complexity is quadratic in $\mathcal{O}(n \cdot m)$ as the runtime complexity.

Figure 4.4 illustrates the alignment matrix, the backtracking step and the final two optimal Needleman-Wunsch alignments of the sequences from the example in Fig. 4.3.

4.2.2 Smith-Waterman Algorithm

The Smith-Waterman algorithm, introduced by Temple F. Smith and Michael S. Waterman in 1981 [SW81], as a simple modification of the Needleman-Wunsch algorithm (see Sect. 4.2.1), finds the optimal *local* alignments of two input sequences A and B. The basic procedure is the same as for the Needleman-Wunsch algorithm. Firstly, an alignment matrix H is calculated, and secondly, a backtracking step is performed. However, for Smith-Waterman the alignment matrix is defined as $H \in \mathbb{N}_0^{(n+1)} \times \mathbb{N}_0^{(m+1)}$ with a modified calculation rule to exclude negative values. Let S be a scoring matrix and $g < 0$ denote a linear gap penalty (again, an adaption is necessary if affine gap penalties have to be considered). For all $i, j > 0$:

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S(a_i, b_j) & \text{match/mismatch} \\ H_{i-1,j} + g & \text{gap opening/extension (insertion)} \\ H_{i,j-1} + g & \text{gap opening/extension (deletion)} \\ 0 \end{cases} \tag{4.2.3}$$

		A	**C**	**G**	**C**	**T**	**T**	**T**	**G**	**A**	**A**	**T**	**A**	**C**	**A**	**C**
	0	−10	−11	−12	−13	−14	−15	−16	−17	−18	−19	−20	−21	−22	−23	−24
A	−10	5	−5	−6	−7	−8	−9	−10	−11	−12	−13	−14	−15	−16	−17	−18
C	−11	−5	10	0	−1	−2	−3	−4	−5	−6	−7	−8	−9	−10	−11	−12
G	−12	−6	0	15	5	4	3	2	1	0	−1	−2	−3	−4	−5	−6
C	−13	−7	−1	5	20	10	9	8	7	6	5	4	3	2	1	0
T	−14	−8	−2	4	10	25	15	14	13	12	11	10	9	8	7	6
G	−15	−9	−3	3	9	15	21	11	19	9	8	7	6	5	4	3
A	−16	−10	−4	2	8	14	11	17	9	24	14	13	12	11	10	9
A	−17	−11	−5	1	7	13	10	7	13	14	29	19	18	17	16	15
T	−18	−12	−6	0	6	12	18	15	5	13	19	34	24	23	22	21
A	−19	−13	−7	−1	5	11	8	14	11	12	18	24	39	29	28	27
T	−20	−14	−8	−2	4	10	16	13	10	11	17	23	29	35	25	24
T	−21	−15	−9	−3	3	9	15	21	11	10	16	22	28	25	31	21
C	−22	−16	−10	−4	2	8	5	11	17	9	15	21	27	33	23	36
A	−23	−17	−11	−5	1	7	4	10	7	22	14	20	26	23	38	28
C	−24	−18	−12	−6	0	6	3	9	6	12	18	19	25	31	28	43

Alignment 1

```
ACGCTTTGAATA--CAC
|||||  |||||  |||
ACGCT--GAATATTCAC
```

Alignment 2

```
ACGCTTTGAATA--CAC
||||  ||||||  |||
ACGC--TGAATATTCAC
```

Figure 4.4. Needleman-Wunsch alignment matrix and backtracking for the two possible optimal global alignments of the sequences `ACGCTTTGAATACAC` and `ACGCTGAATATTCAC` using the NUC.4.4 scoring matrix (see example in Fig. 4.3).

Furthermore, the upper and left border values are filled with zeroes: $H_{i,0} = H_{0,j} = 0$ for all $i \geqslant 0$.

Now, the backtracking does not necessarily start at the lower right corner and proceeds to the upper left, but starts at the cell with the highest value ($H_{i,j} = \max\{H_{p,q} | p, q \geqslant 0\}$) and continues until the first cell with a zero value ($H_{i,j} = 0$) is reached. The result is an optimal local alignment. Again, the backtracking path is not necessarily unique and in addition, the maximum cell value is not necessarily unique as well. So, each possible backtracking path describes another optimal local alignment.

For Smith-Waterman the same considerations on runtime and memory complexity apply as for Needleman-Wunsch. The calculation of the alignment matrix is of quadratic runtime complexity $\mathcal{O}(n \cdot m)$ and the backtracking for one alignment is of linear runtime complexity $\mathcal{O}(n + m)$. Memory complexity is linear in $\mathcal{O}(\min\{n, m\})$ if backtracking is not required and quadratic in $\mathcal{O}(n \cdot m)$ otherwise in the general case.

Figure 4.5 illustrates the alignment matrix, the backtracking step and the final Smith-Waterman alignments of the sequences `AACGCTTGAATACTC` and `CTTGCACTCGCTGAA` with four optimal results.

For the direct application of the Smith-Waterman algorithm on e.g. DNA sequence data there exists an implementation for the two RIVYERA systems S3-5000 and S6-LX150 provided by SciEngines GmbH [SE]. Table 4.1 shows the performance of the two systems compared to another commercially available Smith-Waterman alignment tool by CLCbio [CLC] on two different PC systems. The values are determined by aligning one million 100bp Illumina paired-end reads against the human genome *hg19* with a size of about 3.2Gbp and are taken from [Wie13] and [Wie14]. Note that the speed to perform a Smith-Waterman alignment is measured in *GCUPS (Giga Cell Updates Per Second)*, i.e. how many billion cells of the alignment matrix are calculated per second.

		A	A	C	G	C	T	T	G	A	A	T	A	C	T	C
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	5	0	5	0	0	0	0	0	0	0	5	0	5
T	0	0	0	0	1	0	10	5	0	0	0	5	0	0	10	0
T	0	0	0	0	0	0	5	15	5	4	3	5	1	0	5	6
G	0	0	0	0	5	0	0	5	20	10	9	8	7	6	5	4
C	0	0	0	5	0	10	0	4	10	16	6	5	4	12	2	10
A	0	5	5	0	1	0	6	3	9	15	21	11	10	9	8	7
C	0	0	1	10	0	6	0	2	8	5	11	17	7	15	5	13
T	0	0	0	0	6	0	11	5	7	4	10	16	13	5	20	10
C	0	0	0	5	0	11	1	7	6	3	9	6	12	18	10	25
G	0	0	0	0	10	1	7	0	12	2	8	5	2	8	14	15
C	0	0	0	5	0	15	5	4	3	8	7	4	1	7	4	19
T	0	0	0	0	1	5	20	10	9	8	7	12	2	6	12	9
G	0	0	0	0	5	4	10	16	15	5	4	3	8	5	2	8
A	0	5	5	0	0	3	9	6	12	20	10	9	8	7	6	7
A	0	5	10	1	0	2	8	5	2	17	25	15	14	13	12	11

Alignment 1

```
AACGCTTGAATACTC
||||X  ||||
CTTGC--ACTCGCTGAA
```

Alignment 2

```
AACGCTTGAATACTC
||||X|  |||
CTTGCA--CTCGCTGAA
```

Alignment 3

```
AACGCTTGAATACTC
|||  ||||
CTTGCACTCGC-TGAA
```

Alignment 4

```
AACGCTTGAATACTC
||||  |||
CTTGCACTCGCT-GAA
```

Figure 4.5. Smith-Waterman alignment matrix and backtracking for the optimal local alignments of the sequences AACGCTTGAATACTC and CTTGCACTCGCTGAA with four results using the NUC.4.4 scoring matrix.

Table 4.1. Smith-Waterman performance for DNA sequence alignment. 1 million 100 bp reads (plus their reverse complements) are aligned against the human genome *hg19*.

Architecture	Time	Speed (GCUPS)
RIVYERA S6-LX150	29 h 01 m	6,020
RIVYERA S3-5000	57 h 16 m	3,050
CLCbio 2x Xeon X3210 @ 2.13 GHz (2x4 cores)	∼162 d	45
CLCbio Core2Duo @ 2.17 GHz (2 cores)	∼560 d	13

4.3 BLAST

The *Basic Local Alignment Search Tool (BLAST)* [AGM+90] is a popular tool for microbiologists, medical scientists, and bioinformaticians to search for sequence similarities of a query sequence in DNA or protein sequence databases. For this purpose BLAST performs quick heuristic sequence alignments of the query sequence against all database sequences. The BLAST software is now provided and maintained by the NCBI [NCBa] and comes with several enhancements compared to the original publication, such as the *two-hit method* and gapped alignment to increase quality and speed [AMS+97].

BLAST is available as *BLASTn* for nucleotide (DNA) queries in nucleotide databases or as *BLASTp* for protein queries in protein sequence databases. Both tools work identically but with adapted parameters and scoring matrices. They are explained in detail in Sect. 4.3.1. Additionally, based on the original core algorithm, BLAST also comes in other variations such as *BLASTx*, *tBLASTn*, *tBLASTx* or *PSI-BLAST* to perform iterative or translated queries, e.g. nucleotide searches in protein databases or the other way round.

However, with the primary focus on BLASTp, todays exponential growth of sequence databases (as already shown in Fig. 4.1 in Sect. 4.1) challenges standard PC architectures to reach their limits when an extensive analysis with several hundred or thousands of query sequences has to be performed in reasonable time. Recent development already addresses alternative ar-

chitectures, e.g. CUDA-BLASTp [LSM11a] utilizing general purpose GPUs with a speedup of up to 6 on an nVidia GeForce GTX 280 graphics card compared to a single CPU-thread of an Intel Core i7-920. Others provide single FPGA-based implementations such as Kasap et al. [KBL08]. They had already gained a significant speedup with the implementation in the Handel-C language on a single Xilinx Virtex-4 LX160 FPGA. Later, Kasap et al. presented a hardware implementation of PSI-BLAST [AMS+97; KBL09]. Sotiriades et al. [SD07] published a hardware implementation of BLAST as well, with the advantage to be freely configurable to use with BLASTn, BLASTp, BLASTx, tBLASTn and tBLASTx. However, it misses superior advantages by omitting the usage of gapped BLAST or the two-hit method. Mahram and Herbordt followed another approach [MH10]. They used a single Altera Stratix-III FPGA connected to 4.5 GB DRAM to speed up the process of the ungapped and gapped extension. Recently, Guo et al. published an FPGA-based prototype implementation using a multi-hits detection strategy [GWD11].

Two approaches are available for multiple FPGAs using the RIVYERA S3-5000 architecture [WBB+11; WSS12]. The first one is related to the BLASTp pipeline presented by Kasap et al. while the second and more recent approach follows some ideas presented by Jacob et al. in Mercury BLASTp [JLB+08]. It has been developed to remove several bottlenecks detected in the dataflow of the first design, including the replacement of a quadratic two-hit method to one with linear complexity as well as the enhancement with a gapped extension filter. This approach will be described in Sect. 4.3.2.

The design was tested with three different query sets on the NCBI *RefSeq* database [NCBc]. A total speedup of up to 24 was reached against a system equipped with two quad-core Intel Xeon E5520 at 2.26 GHz and 48 GB RAM. For this measurement the system was utilized with all 16 available threads using multi-threading. Regarding GPUs, a speedup of more than 22 was gained, compared to author related results on an nVidia GeForce GTX280 graphics card running CUDA-BLASTp.

The BLASTp FPGA implementation has also been adapted to the RIVYERA S6-LX150 system and is provided by SciEngines GmbH [SE]. A performance evaluation taken from [Wie14] reveals a speedup of about 19 when

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	−1	−2	−2	0	−1	−1	0	−2	−1	−1	−1	−1	−2	−1	1	0	−3	−2	0
R	−1	5	0	−2	−3	1	0	−2	0	−3	−2	2	−1	−3	−2	−1	−1	−3	−2	−3
N	−2	0	6	1	−3	0	0	0	1	−3	−3	0	−2	−3	−2	1	0	−4	−2	−3
D	−2	−2	1	6	−3	0	2	−1	−1	−3	−4	−1	−3	−3	−1	0	−1	−4	−3	−3
C	0	−3	−3	−3	9	−3	−4	−3	−3	−1	−1	−3	−1	−2	−3	−1	−1	−2	−2	−1
Q	−1	1	0	0	−3	5	2	−2	0	−3	−2	1	0	−3	−1	0	−1	−2	−1	−2
E	−1	0	0	2	−4	2	5	−2	0	−3	−3	1	−2	−3	−1	0	−1	−3	−2	−2
G	0	−2	0	−1	−3	−2	−2	6	−2	−4	−4	−2	−3	−3	−2	0	−2	−2	−3	−3
H	−2	0	1	−1	−3	0	0	−2	8	−3	−3	−1	−2	−1	−2	−1	−2	−2	2	−3
I	−1	−3	−3	−3	−1	−3	−3	−4	−3	4	2	−3	1	0	−3	−2	−1	−3	−1	3
L	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4	−2	2	0	−3	−2	−1	−2	−1	1
K	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5	−1	−3	−1	0	−1	−3	−2	−2
M	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5	0	−2	−1	−1	−1	−1	1
F	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6	−4	−2	−2	1	3	−1
P	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7	−1	−1	−4	−3	−2
S	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4	1	−3	−2	−2
T	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5	−2	−2	0
W	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11	2	−3
Y	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	−1
V	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	−1	−1	−2	−2	0	−3	−1	4

Figure 4.6. The BLOSUM62 scoring matrix for protein sequence alignment [NCBb].

compared to a more recent quad-core Intel Core i7-950 PC system using 8 threads (multi-threading) as well.

4.3.1 BLASTn and BLASTp Algorithms

Query Pre-processing / Neighborhood

The BLAST algorithm is organized in several steps. In the first step, the query sequence is being pre-processed to identify its *neighborhood*. The neighborhood contains a list of short sequences of size k (k-mers) which are similar to k-mers of the query sequence. The value k is fixed, but different for either BLASTn ($k = 11$) or BLASTp ($k = 3$). A k-mer is declared similar to a k-mer of the query sequence if the score of a direct comparison, calculated according to a scoring matrix (such as BLOSUM62 [NCBb], see Fig. 4.6), exceeds a predefined threshold value. An example in Fig. 4.7 illustrates the generation of a list of similar k-mers to an input k-mer of a query sequence.

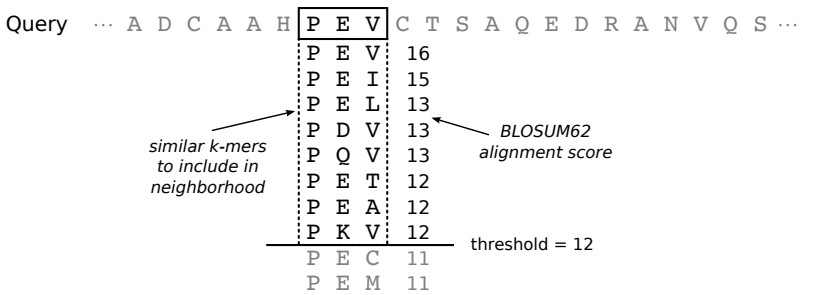


Figure 4.7. Example for the generation of the neighborhood of a query sequence according to the BLOSUM62 scoring matrix.

Two-Hit Method

Afterwards, exact matches (*hits*) of the neighborhood in the database sequences (further referred to as *subject*) are located. The two-hit method analyzes each pair of hits to hold the same distance to each other in the query sequence and the subject sequence. The pairs satisfying this restriction are referred to as *two-hits*.

To save runtime and memory the distance between the hits in a pair is bounded to a certain parameter A. Overlapping hits are omitted by applying the value k as lower bound ($k = 3$ for BLASTp). The following equation shows the condition for a two-hit whereby s_0 and s_1 state the location of two hits in the subject and q_0 and q_1 their locations in the query respectively:

$$k \leqslant q_1 - q_0 = s_1 - s_0 < A \tag{4.3.1}$$

Ungapped Extension / X-Drop Mechanism

Each two-hit is being further examined by an *ungapped extension* process. Both hits of a hit pair are extended forwards and backwards by calculating a similarity score of the current part of the subject and query sequence. In detail, following the NCBI implementation, the similarity score is firstly calculated for the hit pair itself and the gap between it. Then, the calculation of the score is extended residue by residue from the first hit of the pair to

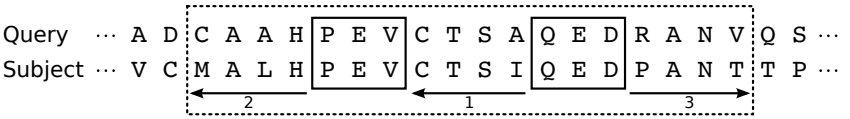


Figure 4.8. Example for the ungapped extension of a two-hit in the NCBI BLAST implementation. The solid rectangles mark the hit pair, the dashed an extension. Arrows indicate the direction and the attached numbers the order of the extensions.

the left and afterwards from the second hit to the right (in positional order). The calculation stops for each direction if the score declines a certain cut-off distance below the so far reached maximum. This method is referred to as *X-drop* mechanism. The result of this process, i.e. the *high-scoring pair (HSP)* of this extension, refers to the two positions where the maximum score has been reached for each direction. An example of the ungapped extension process is shown in Fig. 4.8.

Gapped Extension

The last step in the BLAST algorithm is the *gapped extension*. To introduce gapped alignment, HSPs are being analyzed by a slightly modified version of the Needleman-Wunsch algorithm [NW70] (see Sect. 4.2.1). First, the alignment is bound to the positional range of the ungapped extension, and second, in contrast to the original Needleman-Wunsch algorithm, the score of the alignment is stated by the maximum cell value rather than the value of the lower right corner of the alignment matrix. If a traceback is required to complete the final alignment (depending on the alignment score), it starts at the matrix cell with the calculated maximum as well. Additionally, runtime is being reduced by applying the X-drop mechanism again, i.e. omitting the calculation of matrix cell values where the score declines below a certain cut-off distance from the so far calculated maximum cell value.

4.3.2 FPGA-based BLASTp

The following description focuses on the implementation of the BLASTp application on the RIVYERA S3-5000 system. The adaption to the RIVYERA

S6-LX150 provided by SciEngines GmbH [SE] may differ in the number of pipelines per FPGA, the maximum query size or any other detail presented here.

Overview

The implementation of BLASTp for the RIVYERA architecture is divided into two parts: the software part, which is responsible for data pre- and post-processing, and the hardware part accelerating the most computational intensive parts of the BLAST application. Fortunately, the design could be completely integrated into the original NCBI BLASTp software version 2.2.25+. Therefore, it can be executed transparent as any usual NCBI BLASTp query and delivers the accustomed result format. Additionally, software routines are available which split large queries exceeding the maximum query size for the FPGAs to smaller subqueries. Hence, the splitting can be done transparent to the user. The maximum query size for this implementation is $1024 - A$, whereby A is a user definable parameter for the maximum size of a two-hit (see explanation of the *TwoHitFinder* below).

The components of the BLASTp algorithm are distributed among hardware and software part as follows. The query pre-processing including the generation of the neighborhood and query splitting is performed by the original NCBI BLASTp software routines on the host system. The hardware part consists of a number of BLASTp pipelines for each FPGA. One pipeline contains processing elements performing the location of hits (*HitFinder*), the two-hit method (*TwoHitFinder*) and the ungapped extension process (*UngappedExtender*). A Spartan3-5000 FPGA of the RIVYERA S3-5000 system is able to carry two of these pipelines. Additionally, a gapped extension filter (*GappedExtender*) is implemented on each FPGA which serves the two pipelines together.

After the pre-processing the queries are distributed among the available BLASTp pipelines on the FPGAs of the RIVYERA system, i.e. each FPGA contains two plain-text queries and the corresponding neighborhoods. The subject is streamed afterwards via broadcast to all FPGAs. Thus, the BLASTp pipelines perform the analysis of their queries in parallel. Only High-Scoring Pairs (HSPs) which pass the gapped extension filter are reported to the host.

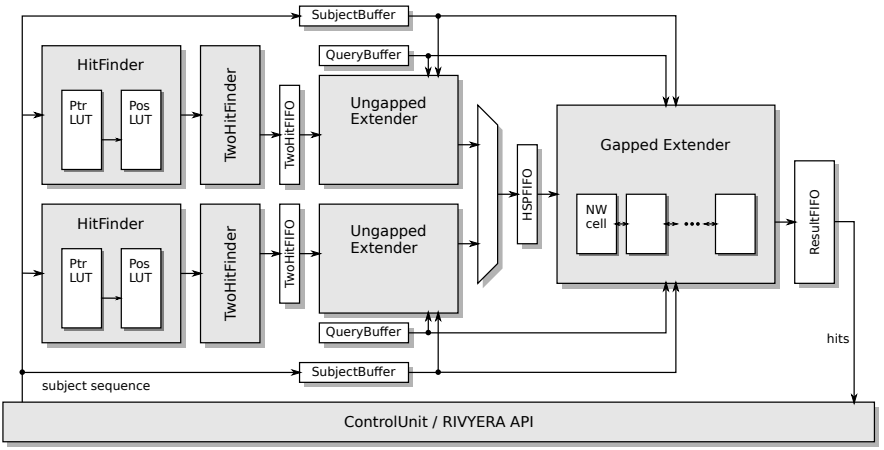


Figure 4.9. Structure of two BLASTp hardware pipelines sharing one GappedExtender component.

The host system now performs a detailed Needleman-Wunsch alignment of the query and subject part which contain the HSP and reports this as a result.

An overview of two BLASTp pipelines sharing one GappedExtender component can be seen in Fig. 4.9. It follows the description of the single components and a performance evaluation.

HitFinder Component

The HitFinder searches for occurences of k-mers of the subject sequence in the neighborhood. Each k-mer of the subject is simply looked up in a table containing each k-mer in the neighborhood and their corresponding positions in the query. This table is generated and initialized by the host software. It is organized in two lookup tables. The first (*PtrLUT*) is directly addressed by the k-mer, i.e. each k-mer is interpreted as an address. It provides pointers to a variable memory space implemented in the second lookup table (*PosLUT*). If occurences of a k-mer exist in the query sequence, the return values are *hits*, i.e. pairs of valid positions (s, q_i) whereby s

denotes the current subject position and q_i are the positions in the query where the current k-mer (or similar) occurs. These hits will be routed to the TwoHitFinder component. Otherwise, if the k-mer does not exist in the neighborhood, it is ignored for further processing.

TwoHitFinder Component

Testing all possible pairs of hits to hold the condition for a two-hit (s. Eq. 4.3.1) results intuitively in a quadratic runtime complexity. In the following, a strategy is presented to reduce the runtime complexity to find two-hits to a linear complexity. The idea is similar to the one published for Mercury BLASTp [JLB+08].

First, an array of length $l = 1024$ is required. This corresponds to the maximum query length plus the parameter A for the bounds of (4.3.1). For each hit (s_i, q_i) in the queue, this array stores at index p the most recent subject position s_i to the corresponding query position q_i. The index p is calculated from the following equation:

$$p \quad = \quad (s_i - q_i) \bmod 1024 \tag{4.3.2}$$

The insertions are done subsequently, but before inserting a new position, the content of the array cell is read. If this cell contains a valid subject position s_j, it holds $s_i > s_j$ and:

$$s_i - q_i \quad = \quad s_j - q_j \qquad (\text{mod } 1024) \tag{4.3.3}$$

$$\Leftrightarrow \qquad s_i - s_j \quad = \quad q_i - q_j \qquad (\text{mod } 1024) \tag{4.3.4}$$

Assuming $s_i - s_j < A$, it follows from (4.3.4):

$$s_i - s_j \quad = \quad \begin{cases} q_i - q_j & \text{if } q_i \geqslant q_j \\ 1024 - (q_j - q_i) & \text{if } q_i < q_j \end{cases} \tag{4.3.5}$$

The second case (assuming $q_i < q_j$) results in:

$$A \quad > \quad 1024 - (q_j - q_i) \tag{4.3.6}$$

$$\Leftrightarrow \qquad q_j - q_i \quad > \quad 1024 - A \tag{4.3.7}$$

This stays in contradiction to the bounds of the query length which is $l = 1024 - A$. Hence, if $s_i - s_j < A$ it directly follows $s_i - s_j = q_i - q_j$, and if (4.3.1) holds, i.e. $k \leqslant s_i - s_j$, this result is reported as a two-hit, i.e. a pair of hits, and buffered in a FIFO before being further processed by the UngappedExtender component.

This method might be problematic if hits arrived unordered, i.e. if $s_j > s_i$. However, this possibility can be counted out since the HitFinder provides hits only in ascending query positions, followed by an ascending order of the subject positions.

All resulting two-hits are buffered in a FIFO before being processed further by the UngappedExtender component.

UngappedExtender Component

The ungapped extension process conforms to the order indicated in Fig. 4.8 and the description in Sect. 4.3.1. The process continously stores the current score, the so-far reached maximum score of this extension and its corresponding position. Firstly, the extension is directed left, starting with the right hit of the hit pair. In every clock cycle the score of a pair of collated residues from the query and the subject sequence is calculated using a scoring matrix, e.g. BLOSUM62, implemented in a dual-ported ROM. The determined score is accumulated continuously to the current score. The X-drop mechanism is implemented by checking the new calculated score in every clock cycle. If it drops a predefined cut-off distance below the so far calculated maximum score, the process stops. The position corresponding to the maximum score is stored as the left position of the high-scoring pair (HSP) which will be reported as result.

To determine the right position of the HSP, the extension continues directed right from the right hit of the hit pair, but only if the gap between the two hits has already been crossed in the left extension before. The right extension performs exactly as the previously described left extension and stops according to the X-drop mechanism as well. The previously stored left position and the current right position where the maximum score has been reached, form the complete HSP, which is reported to the GappedExtender via a FIFO buffer only if its score exceeds another predefined threshold

value.

The UngappedExtender component contains a feedback path, controlling the elements stored in the preceding FIFO. According to the current progress, a pending two-hit may already be included in the running extension process. The UngappedExtender is able to remove such two-hits in advance to prevent the same extension with different starting points being processed several times.

GappedExtender Component

The GappedExtender component basically performs a modified Needleman-Wunsch alignment with a banded matrix and a HSP at its center. In contrast to Mercury BLASTp [JLB+08] which performs a Smith-Waterman alignment with a banded matrix of fixed size, this implementation is kept close to the one in NCBI BLASTp using the X-drop mechanism to stop the extension process. However, the width of the matrix band is fixed to $\omega = 64$, but the length of the matrix band stays variable.

In order to create the alignment matrix with the HSP at its center it is necessary to do the calculation in two steps. The alignment starts at the center of the HSP and firstly, extends backwards, using the reverse sequences for Needleman-Wunsch. Afterwards, a forward directed alignment, starting from the center of the HSP as well, is performed in the same way. The original HSP is being reported to the host software if the sum of both alignment scores exceeds a predefined report threshold. The structure of this process is illustrated in Fig. 4.10.

The subcomponents of the GappedExtender component basically consist of ω *NWcells* connected in a chain. Since the calculation is restricted to a banded matrix, a pre-initialization of the chain with the query sequence is impossible. Instead, the part of the query sequence, which is to be analyzed, is inserted from the one end of the chain, while the corresponding part of the subject sequence is inserted from the other end. With every clock cycle one residue of either the query or the subject is inserted.

The calculation of the score of a cell $H_{i,j}$ in the alignment matrix corresponds to Eq. 4.2.2 in Sect. 4.2.1, but using an affine gap penalty. As easily can be seen, the score of each matrix cell is dependent on the scores of its

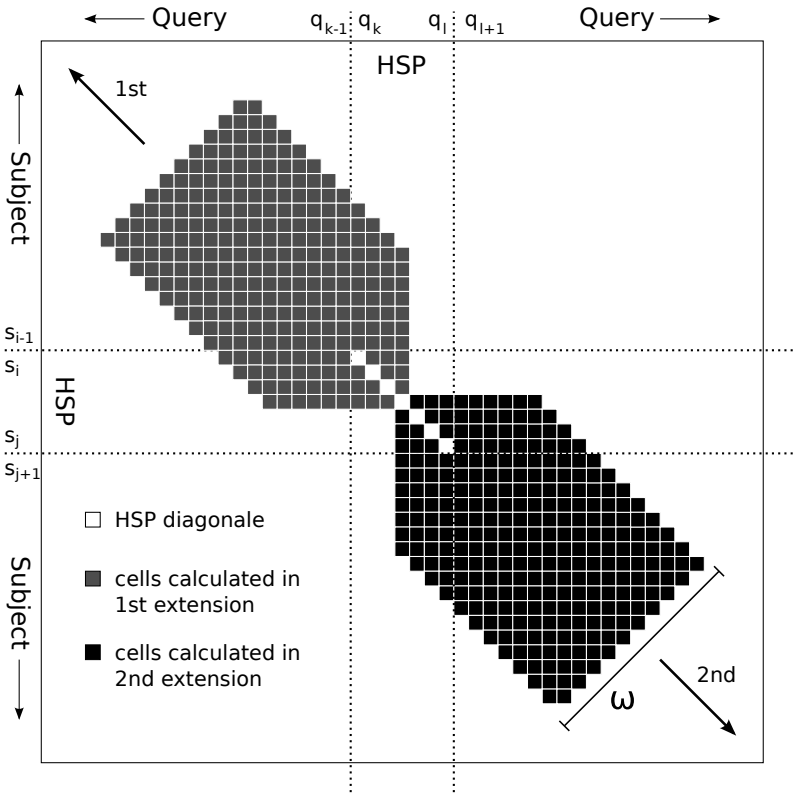


Figure 4.10. Principle of the gapped extension of a high-scoring pair (HSP). The Needleman Wunsch matrix is only calculated at the highlighted matrix band with width ω. The first extension is done with subject and query directed backwards, starting at the center of the HSP. The second extension starts at the center of the HSP as well, but is directed forwards. The X-drop mechanism is used to stop each extension.

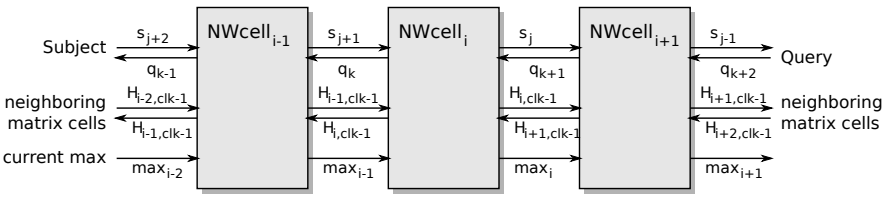


Figure 4.11. Structure of the NWcell chain implemented in the GappedExtender component. s_j denotes the subject symbol at position j, q_k the query symbol at position k, $H_{i,clk}$ is the current score of cell i.

upper, left and upper-left neighboring cells.

Since residues are inserted from both ends of the NWcell chain, all scores of the matrix cells are calculated within an anti-diagonal of the matrix band of width ω in one clock cycle, but due to the alternating insertion of residues, the current anti-diagonal "moves" alternating rightward and downward in each clock cycle. Hence, each NWcell requires access to the scores of *both* neighboring cells in the chain, calculated in the previous clock cycle.

However, the information for the current maximum score is passed through the chain only in one direction. Again, as for the ungapped extension, each NWcell accesses the scoring matrix via a dual-ported ROM. The structure of an NWcell chain is depicted in Fig. 4.11.

The gapped extension step in hardware acts as an additional filter to keep the number of reports small, since the post-processing step to create the final alignment remains in the software part. If a high-scoring pair passes the gapped extension filter, the exact alignment is generated including the backtracking by the original NCBI routines on the host. This way, valuable software runtime is saved by filtering nearly every HSP in advance in hardware, which would be omitted by the gapped extension of the host software anyway.

Before being fetched by the host software, the reports for each FPGA are collected in the attached DRAM. This way, the number of communication interruptions for the submission of reports during the core process can be kept small.

Performance Evaluation

The BLASTp pipeline implementation described above is written in the VHDL programming language with the Xilinx ISE 13.2 development environment and targets a Xilinx Spartan3-5000 FPGA. The software part is written in C++ and integrated into the core of NCBI BLASTp version 2.2.25+. One Spartan3-5000 hosts the resources for two pipelines sharing one GappedExtender component. Disregarding the GappedExtender, block RAM utilization was the limitative factor for the device utilization. The width of the matrix band in the GappedExtender was set to $\omega = 64$, as in Mercury BLASTp.

The total device utilization, including the RIVYERA API, is 25,499 slices (76%), 24,848 slice flipflops (37%), 32,654 LUTs (49%) and 93 BRAMs (89%). The base clock frequency of the implementation is 50 MHz, although all pipeline components, excluding the GappedExtender, are clocked at 100 MHz.

The reference system was a PC system equipped with two Intel Xeon E5520 CPUs, each containing 4 cores (8 threads with multi-threading) running at 2.26 GHz, 48 GB DDR3-RAM, and a 64 bit Linux OS. The comparison was made against NCBI BLASTp version 2.2.25+ with default parameters, BLOSUM62 scoring matrix and a varying number of threads (BLASTp option "`-num_threads`) with the FPGA implementation utilizing a fully equipped RIVYERA machine (128 FPGAs), and several partly equipped configurations (64, 32, and 16 FPGAs) downto one single FPGA card (8 FPGAs).

Three different query sets were chosen (proteomes of *Arabidopsis thaliana*, *Populus trichocarpa*, and human (*Homo sapiens*) from *SUPERFAMILY* database [Sup]), randomly reduced to 2,335, 3,151, and 1,990 sequences respectively, such that each set contains about 1 million residues. Each query set was aligned against the first part of the NCBI *RefSeq* BLAST database, release 50 [NCBc] containing 2,996,372 sequences ($\approx$ 1 billion residues).

All results are stated in Table 4.2. It shows that a fully equipped RIVYERA S3-5000 outperforms the reference with a speedup of up to 23.5. Hence, the runtime performance of one single FPGA conforms to about 1.5 CPU cores. Additionally, Fig. 4.12 illustrates the speedups of RIVYERA with a

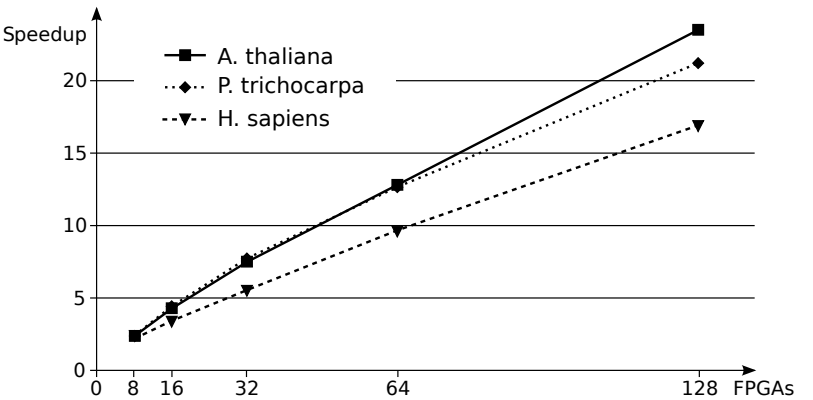


Figure 4.12. BLASTp speedups of RIVYERA S3-5000 with different number of utilized FPGAs vs. 2x Xeon E5520 (16 threads).

Table 4.2. BLASTp runtimes (in seconds) on the RIVYERA S3-5000 system of three randomly reduced query sets against part one of the NCBI *RefSeq* database [NCBc]. The 2x Xeon E5520 reference system runs NCBI BLASTp v. 2.2.25+. The marked (*) runtimes are estimations calculated from published runtimes extrapolated to the changed database and query set.

Query set	RIVYERA (*n* FPGAs)			2x Xeon E5520		Mercury	CUDA
	128	32	8	16 thr.	8 thr.	BLASTp	BLASTp
A. thaliana	353	1,106	3,531	8,301	9,995	3,780*	7,780*
P. trichoc.	482	1,323	4,210	10,226	12,506	5,161*	9,615*
H. sapiens	561	1,723	4,409	9,464	11,602	6,007*	8,026*

different number of utilized FPGAs versus the reference system, showing an approximately linear increase of speed with an increasing number of FPGAs.

The stated runtimes of Mercury BLASTp [JLB+08] and CUDA-BLASTp v2.0 [LSM11a] have been linearly extrapolated from the best results in the respective publications. Due to the lack of hardware for Mercury BLASTp and a non-functional CUDA-BLASTp on a test system with an nVidia GeForce GTX480 GPU, no real measurements could be made. Since the

Table 4.3. BLASTp energy consumption on RIVYERA S3-5000 of three randomly reduced query sets against first part of the NCBI *RefSeq* database [NCBc]. The 2x Xeon E5520 reference system runs NCBI BLASTp v2.2.25+.

Query set	RIVYERA 128 FPGAs (525 W)	2x Xeon E5520 16 thr. (290 W)	$\frac{\text{RIVYERA}}{\text{2x Xeon}}$
A. thaliana	51.5 Wh	668.7 Wh	7.7%
P. trichocarpa	70.3 Wh	823.8 Wh	8.5%
Homo sapiens	81.8 Wh	762.4 Wh	10.7%

runtime is extremely dependent on the quality of the query, these results are only to be seen as a rough estimate. However, regarding these estimations, RIVYERA still outperforms these solutions as well.

Table 4.3 shows the energy consumption for the query sets measured with a customary power measurement device. The measured energy consumption of a fully-equipped RIVYERA is only 590 W. Regarding the energy consumption of 290 W by the reference system, up to 92.3% can be saved, even when compared to a PC cluster containing 24 times the reference system to have the same performance.

Regarding quality analysis, a detailed view on a smaller query subset (109 sequences, 28,483 residues) showed 21,918 hits from RIVYERA while NCBI found 22,167 hits. A one-by-one comparison revealed 63 hits (0.29%) were additional results not found by NCBI, and 312 hits (1.41%) from the NCBI results are not found by the RIVYERA implementation. Another 24 hits (0.11%) in both sets were differing only in their alignment positions for the same query and subject sequence. This indicates that the alignment quality almost equals to the NCBI software. However, since BLAST is heuristic, small discrepancies in the alignments do not necessarily imply a difference in quality.

Summarized, due to the ability of processing 256 queries at once, the parallelization of BLASTp benefits especially from large query sets. Regarding a permanent occupation of the machine, more than 92% of the required energy can be saved while keeping almost the same alignment quality as in

Table 4.4. BLASTp runtimes (in seconds) and energy consumption (in Wh) on RIVYERA S6-LX150 of three query sets and part one of the NCBI *RefSeq* database [NCBc]. The Intel Core i7 reference system runs NCBI BLASTp v. 2.2.25+.

Query set		RIVYERA S6-LX150 (780 W)		Intel Core i7-950 (8 threads, 130 W)		Speedup
	#queries	time (s)	energy (Wh)	time (s)	energy (Wh)	
Human	10,000	3,340	723.7	61,345	2,215.2	18.4
Mouse	10,000	3,213	696.2	56,635	2,045.2	17.6
Rat	10,000	3,347	725.2	63,183	2,281.6	18.9

NCBI BLASTp.

A more recent performance analysis of the BLASTp adaption to the RIVYERA S6-LX150 has been made in [Wie14]. Here, two BLASTp pipelines were implemented on each Spartan6-LX150 FPGA as well. The reference PC system was equipped with an Intel Core i7-950 CPU (4 cores / 8 threads @ 3.07 GHz) and 12 GB RAM running NCBI BLASTp v2.2.25+ [NCBa] on 8 threads.

Three different query sets with proteoms of human, mouse and rat have been tested, each set reduced to exactly 10,000 queries. As reference, the first part of the NCBI *RefSeq* BLAST database, release 50, has been taken, containing 2,996,372 sequences ($\approx$ 1 billion residues) [NCBc].

The runtimes for the three datasets as well as the energy consumption are listed in Table 4.4. It shows that RIVYERA S6-LX150 still outperforms the more recent PC system by a factor of about 19. The power consumption of RIVYERA is measured with 780 W while for the PC system only the Thermal Design Power of the CPU with 130 W is considered without peripherals etc. This alone results in energy savings of about 70%.

Chapter 5

SNP Interaction Detection

This chapter contains excerpts from [WKG+14; Wie14; GWK+15; KWG+15] which represent the original publications of this work presented here.

5.1 Background

Todays fast genotyping methods allow the determination of genome-wide genetic markers of people. Together with a list of associated characteristics of these individuals, databases can be created providing a correlation of genetic information (genotypes) to personal traits (phenotypes) on a large scale. These *Genome-Wide Association Studies (GWAS)* can be used e.g. for a statistical analysis of genetic effects on human diseases. For this purpose this thesis concentrates on *binary traits*, i.e. in the case of diseases, the binary trait is the characteristic of a person indicating if it is affected by that disease or not.

Popular GWAS for diseases are the ones published by the *Wellcome Trust Case Control Consortium (WTCCC)* that genotyped about 2,000 people for each of the following seven diseases of major public health importance together with 3,000 shared controls at about 500,000 markers: bipolar disorder (BD), coronary artery disease (CAD), Crohn's disease (CD), hypertension (HT), rheumatoid arthritis (RA), type 1 diabetes (T1D), and type 2 diabetes (T2D) [WTCCC07].

The categorization in *case* and *control* group according to the presence (case) or absence (control) of a disease allow a statistical evaluation of the obtained data to analyze a potential genetic effect. Sometimes, single genetic markers cannot be reliably associated to a particular disease, such that it is assumed that *joint genetic effects* might influence the expression of that

Table 5.1. Example of phenotypes (hair color of mice) obtained from genotypes of two epistatic loci.

Genotype at locus B	Genotype at locus G		
	g/g	*g/G*	*G/G*
b/b	white	gray	gray
b/B	black	gray	gray
B/B	black	gray	gray

disease. A joint genetic effect generally describes the influence of one or more genes to others, i.e. the existence of one allele on a locus A might hinder or promote the expression of another gene dependent on the allele on locus B.

A special case of a joint genetic effect is *epistasis*. Epistasis describes the effect that the allele of one gene is masked by the presence or absence of other genes resulting in a different phenotype. Then, the other genes are said to have an *epistatic effect* on the first one. For example, supposed there are two loci, B and G, that influence a trait such as the hair color in mice. Locus B has the possible alleles *B* and *b*, locus G has *G* and *g*. The possible phenotypes to the corresponding combinations of genotypes are listed in Table 5.1. It can be seen that the allele *G* is dominant to *g* since all mice with this allele have gray hair. *B* is also dominant to *b* causing black hair, but the effect is masked by the presence of allele *G*, i.e. mice with allele *B* may have black hair only if allele *G* is not present. Hence, the gene at locus G is *epistatic* to the gene at locus B, or more specifically, the allele *G* at locus G is epistatic to allele *B* at locus B [Cor02].

The association of a disease to a genetic marker might be analyzed by observing the frequencies of the genotypes of the samples at this marker position in the case group in comparison to the control group. A statistically significant difference between the frequencies of case and control group might indicate a relation. However, in many cases this method is not powerful enough [Mah08; MAW10]. Thus, in order to detect joint-genetic effects, interactions between markers have to be considered. These interactions are often referred to as *SNP interactions* since most of the marker positions

are regarded as SNPs (see Chap. 2) and may act as an indicator for *gene interactions* since typically more than a single marker is involved in a gene.

A common approach is the test for pairwise interactions for which reason many tools exist [Ste11; WLF+11], such as BOOST [WYY+10a; WYY+13], iLOCi [PNI+12], MDR [RHR+01] and MB-MDR [CCD+11; LJG+13]. Many of these tools perform an *exhaustive* test for pairwise SNP interactions. For that purpose each possible pair of markers from a dataset (e.g. GWAS) is analyzed by a statistical test resulting in a number of tests which is quadratic to the number of markers. This typically results in long runtimes, such as 60 hours for BOOST on a 3 GHz desktop computer for a dataset of 360,000 SNPs and 5,000 samples [WYY+10a], or 19 hours for iLOCi on a 2×2.4 GHz workstation for a dataset with 500,000 SNPs and 5,000 samples [PNI+12]. Thus, tools exist, such as SIXPAC [PP12], SNPRuler [WYY+10b], SNPHarvester [YHW+09], TEAM [ZHZ+10] and Screen and Clean [WDR+10], which reduce the amount of SNP pairs to be tested by applying some pre-filtering methods. However, since SNP pairs may reveal a statistical significance only in combination and are completely inconspicuous when observed alone, these tools gather runtime for the cost of quality by potentially losing significant SNP pairs in the results. Therefore, Sects. 5.3 and 5.4 show how to make the process of an exhaustive analysis feasible by reducing the runtime of BOOST and iLOCi to only a few minutes with the help of FPGA technology. Other methods use GPU technology to significantly speed up the process, for instance GBOOST [YYW+11], GWIS [GRW+13], EpistSearch [GSK+14], EpiGPU [HTW+11] and SHEsisEPI [HLZ+10]. Among these, some of them have been included in the evaluation of the implementation of FPGA-based BOOST in Sect. 5.3.2.

Sometimes pairwise tests may not be powerful enough [CHW+13]. In this case higher-order interaction tests may reveal interactions between multiple markers, such as third-order tests analyzing marker triples. Since the number of tests is cubic to the number of markers in the dataset, this analysis is even more time consuming and computational challenging than pairwise interaction tests. Anyway, in Sect. 5.5 it is demonstrated how FPGAs help to realize acceptable runtimes for datasets which are infeasible for standard PCs.

Preliminary to the FPGA implementations, Sect. 5.2 introduces mathe-

cases ($k=1$)		SNP A w	SNP A h	SNP A v
SNP B	w	n_{001}	n_{011}	n_{021}
	h	n_{101}	n_{111}	n_{121}
	v	n_{201}	n_{211}	n_{221}

controls ($k=0$)		SNP A w	SNP A h	SNP A v
SNP B	w	n_{000}	n_{010}	n_{020}
	h	n_{100}	n_{110}	n_{120}
	v	n_{200}	n_{210}	n_{220}

Figure 5.1. Second-order contingency tables for cases and controls. n_{ijk} reflect the number of occurrences for the corresponding genotype combination in a given SNP pair.

matical methods commonly used together with interaction measurement. Thereupon, the FPGA designs mentioned above are described and evaluated in comparison to their originals. The evaluations use the GWAS by the WTCCC [WTCCC07] for performance analysis of the implemented tools, further referred to as *WTCCC dataset*. The evaluations strictly do not present any particular biological or medical results and are not interpreting them in any way. The result quality is given only in relation to the original software, i.e. it is verified that the results are equal for the FPGA implementation and the corresponding original CPU implementation for the tested datasets.

5.2 Mathematical Methods

5.2.1 Contingency Tables

A typical GWAS dataset consists of two groups of samples (cases and controls) which are genotyped at a set of marker positions, such as SNPs. This work considers biallelic markers for diploid organisms which is the common use case, i.e. genotypes may appear as *homozygous wild (w)*, *heterozygous (h)* or *homozygous variant (v)* types.

Contingency tables in interaction tests with biallelic markers are created for each possible combination of SNPs (according to the interaction order to be tested) separately for case and control group. For pairwise interactions (second-order), their dimension is 3×3, one entry for each possible combination of genotypes. The entries n_{ijk} reflect the number of occurrences each combination of genotypes appears in the dataset for the corresponding

cases ($l=1$)			SNP A			controls ($l=0$)			SNP A		
			w	h	v				w	h	v
SNP B	w	SNP C w	n_{0001}	n_{0101}	n_{0201}	SNP B	w	SNP C w	n_{0000}	n_{0100}	n_{0200}
		SNP C h	n_{0011}	n_{0111}	n_{0211}			SNP C h	n_{0010}	n_{0110}	n_{0210}
		SNP C v	n_{0021}	n_{0121}	n_{0221}			SNP C v	n_{0020}	n_{0120}	n_{0220}
	h	SNP C w	n_{1001}	n_{1101}	n_{1201}		h	SNP C w	n_{1000}	n_{1100}	n_{1200}
		SNP C h	n_{1011}	n_{1111}	n_{1211}			SNP C h	n_{1010}	n_{1110}	n_{1210}
		SNP C v	n_{1021}	n_{1121}	n_{1221}			SNP C v	n_{1020}	n_{1120}	n_{1220}
	v	SNP C w	n_{2001}	n_{2101}	n_{2201}		v	SNP C w	n_{2000}	n_{2100}	n_{2200}
		SNP C h	n_{2011}	n_{2111}	n_{2211}			SNP C h	n_{2010}	n_{2110}	n_{2210}
		SNP C v	n_{2021}	n_{2121}	n_{2221}			SNP C v	n_{2020}	n_{2120}	n_{2220}

Figure 5.2. Third-order contingency tables for cases and controls. n_{ijkl} reflect the number of occurrences for the corresponding genotype combination in a given SNP triple.

SNP pair in either case ($k = 1$) or control ($k = 0$) group. See Fig. 5.1 for an example.

Therefore, with n denoting the total number of SNPs, and since a SNP pair can be viewed as symmetric, $\binom{n}{2} = \frac{n(n-1)}{2}$ tables have to be created for a complete pairwise analysis. For datasets, such as the WTCCC dataset with about 500,000 SNPs, this implies about 125 billion (i.e. 1.25×10^{11}) tables.

Generally, the order of SNPs is irrelevant when treated in combination. Hence, for third-order interactions, $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$ tables have to be created. For the WTCCC dataset, this implies about 2.08×10^{16} tables now. Even a subset with e.g. 10,000 SNPs still implies about 1.67×10^{12} tables, underlining the computational effort required to perform an exhaustive interaction analysis on standard datasets. Fig. 5.2 shows an example for a third-order contingency table.

5.2.2 Mutual Information

The *mutual information* or *transinformation* of two random variables X and Y is a measure for the mutual dependency of both variables and is defined as follows:

$$I(X;Y) = \sum_{x \in X, y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{5.2.1}$$

It can be seen as the information both variables share or how much the knowledge of one variable reduces the uncertainty (or *entropy*) of the other. The entropy of a random variable X is defined as:

$$H(X) = -\sum_{x \in X} p(x) \log p(x) \tag{5.2.2}$$

For example, the mutual information is zero if X and Y are independent. At the other extreme, if X is a deterministic function of Y and vice versa, the mutual information is equal to the entropy of X (or Y). Therefore, mutual information can be equivalently expressed as:

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) && (5.2.3)\\ &= H(Y) - H(Y|X) && (5.2.4)\\ &= H(X) + H(Y) - H(X,Y) && (5.2.5)\\ &= H(X,Y) - H(X|Y) - H(Y|X) && (5.2.6) \end{aligned}$$

Here, $H(X)$ and $H(Y)$ describe the *marginal entropies* of X and Y respectively, $H(X|Y)$ and $H(Y|X)$ are the *conditional entropies*, and $H(X,Y)$ states the *joint entropy* of both variables. Figure 5.3 shows an illustration of Eqs. 5.2.3 to 5.2.6 from an example with correlated random variables X and Y.

Relation of a Disease to a Single SNP

Related to the detection of correlations of SNPs to certain diseases, the random variable X may describe the distribution of the genotypes of a particular SNP among all samples in a GWAS. Then, Y describes the distribution of the samples according to whether they have a certain disease or not. Y is also referred to as the *disease state*. By counting and classifying all samples according to their genotype (either homozygous wild, heterozygous or homozygous variant) at the particular SNP marker, the marginal entropy $H(X)$ can be calculated. The marginal entropy $H(Y)$ can be calculated by counting the number of cases and controls, and the determination of the genotype distribution again, but this time separately for cases and controls, directly reflects the joint entropy $H(X,Y)$. Now, using Eq. 5.2.5, the mutual

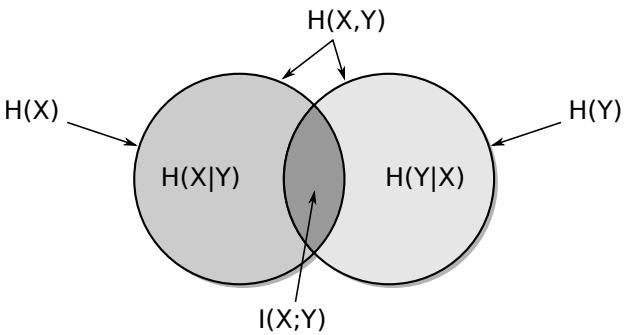


Figure 5.3. Entropy diagram illustrating the mutual information $I(X;Y)$ of two correlated random variables X and Y. The left circle indicates the entropy $H(X)$, the right circle $H(Y)$. The mutual information $I(X;Y)$ is shown as the intersection of both (highlighted in dark gray). Additionally, the joint entropy $H(X,Y)$ is indicated as the combination of both circles, and the gray area in the left illustrates the conditional entropy $H(X|Y)$ while the gray area in the right illustrates $H(Y|X)$.

information provides a simple measure for the correlation of a single SNP marker to the tested disease.

Pairwise and Higher-Order Interactions

For pairwise or higher-order SNP combinations, the definition of mutual information has to be enhanced to be used with more than one random variable correlated to a SNP. For an arbitrary order k the uncertainty of the variable X from the previous example expands to the joint entropy of all variables $X_1, \ldots, X_k$ describing the distributions of k different SNPs from the GWAS respectively. Now, the goal is to measure the correlation of the combination of these SNPs to the disease state Y. In other words, the mutual information of the random variables $X_1, \ldots, X_k$ to the random variable Y has to be calculated, which is denoted as $I(X_1, \ldots, X_k; Y)$. In practice, a generalization of Eq. 5.2.5 can be used:

$$I(X_1, \ldots, X_k; Y) = H(X_1, \ldots, X_k) + H(Y) - H(X_1, \ldots, X_k, Y) \qquad (5.2.7)$$

Example: Mutual Information for Pairwise Interactions

In order to calculate the mutual information for pairwise interactions Eq. 5.2.7 reduces to the following, whereby X_1 and X_2 describe the distributions of two SNPs from a dataset and Y is the disease state:

$$I(X_1, X_2; Y) = H(X_1, X_2) + H(Y) - H(X_1, X_2, Y) \tag{5.2.8}$$

Let π_{ijk} be a shortcut for a joint distribution of X_1, X_2 and Y with $x_1 \in X_1$, $x_2 \in X_2$ and $y \in Y$:

$$\pi_{ijk} = p(x_1 = i, x_2 = j, y = k) \tag{5.2.9}$$

With n describing the total number of samples from a dataset, n_1 the number of cases, n_0 the number of controls, and n_{ijk} the entries of a second-order contingency table described in Sect. 5.2.1, the required entropies can be directly calculated as follows:

$$H(X_1, X_2) = -\sum_{i \in X_1, j \in X_2} p(x_1 = i, x_2 = j) \log p(x_1 = i, x_2 = j) \tag{5.2.10}$$

$$= -\sum_{i \in X_1, j \in X_2} (\pi_{ij0} + \pi_{ij1}) \log(\pi_{ij0} + \pi_{ij1}) \tag{5.2.11}$$

$$= -\sum_{i \in X_1, j \in X_2} \frac{n_{ij0} + n_{ij1}}{n} \log \frac{n_{ij0} + n_{ij1}}{n} \tag{5.2.12}$$

$$H(Y) = -\sum_{k \in Y} p(y = k) \log p(y = k) \tag{5.2.13}$$

$$= -\frac{n_0}{n} \log \frac{n_0}{n} - \frac{n_1}{n} \log \frac{n_1}{n} \tag{5.2.14}$$

$$H(X_1, X_2, Y) = -\sum_{i \in X_1, j \in X_2, k \in Y} \pi_{ijk} \log \pi_{ijk} \tag{5.2.15}$$

$$= -\sum_{i \in X_1, j \in X_2, k \in Y} \frac{n_{ijk}}{n} \log \frac{n_{ijk}}{n} \tag{5.2.16}$$

The mutual information can directly be gained by inserting these calcu-

lated values in Eq. 5.2.8.

Kullback-Leibler Divergence

Mutual information can also be expressed as *Kullback-Leibler divergence* of the product of the marginal distributions $p(x)$ and $p(y)$ from the joint distribution $p(x,y)$ of two random variables X and Y.

$$I(X;Y) = D_{\mathrm{KL}}\left(p(x,y)||p(x)p(y)\right) \tag{5.2.17}$$

In general, the Kullback-Leibler divergence (or *information gain*) $D_{\mathrm{KL}}(P||Q)$ of Q from P describes a measure of the difference between two probability distributions P and Q of a random variable X. More specifically, it measures the information loss when Q is used as a model to approximate P. The general definition of the Kullback-Leibler divergence of discrete probability distributions is as follows:

$$D_{\mathrm{KL}}(P||Q) = \sum_{x \in X} P(x) \ln \frac{P(x)}{Q(x)} \tag{5.2.18}$$

Related to SNP interactions, the Kullback-Leibler divergence can be used as a measure to indicate the relation of a SNP pair to a disease when a dataset, e.g. GWAS, is modeled by assuming that there is no interaction at all. This measure is used by e.g. the BOOST software described in Sect. 5.3.

5.2.3 p-Value

In order to quantify the significance of a result, the *p-value* is a common measure. The p-value describes the probability of the observed or a "more extreme" result under the assumption of a *null hypothesis* H_0. Depending on the nature of the experiment, "more extreme" can mean different things. Let X be a random variable and x the observed result, a *right tail* event is classified by $\{X \geqslant x\}$. Respectively, a *left tail* event is given by $\{X \leqslant x\}$. Last but not least there may be the chance of a *double tail* event. Then, both

inequalities have to be tested. Formally, the p-value is expressed as:

$$P(X \geqslant x|H_0) \text{ (right tail event)} \tag{5.2.19}$$

$$P(X \leqslant x|H_0) \text{ (left tail event)} \tag{5.2.20}$$

$$2\min\{P(X \geqslant x|H_0), P(X \leqslant x|H_0)\} \text{ (double tail event)} \tag{5.2.21}$$

A low p-value means that there is a low probability to have the result observed purely by chance under the assumption of the null hypothesis. Usually, if the p-value falls below a predefined threshold, the null hypothesis has to be rejected. In the literature thresholds of 5%, 1% or even 0.1% are common [GN96].

Regarding SNP interactions, the null hypothesis can be best described in words. It is usually equivalent to the statement that there is no interaction at all. Thus, it is assumed that the distribution of each random variable follows a normal distribution. Now, for the determination of the p-value the value for x is taken from a test statistic. A popular statistical test is the *Pearson chi-square test* (χ^2-test), which is described in detail below. This test quantifies the deviation of the observation from a χ^2-distribution, which would result from the combination of several random variables with normal distribution. Since the χ^2-test only delivers values $\geqslant 0$, the p-value is determined according to Eq. 5.2.19 for a right tail event.

Let X^2 be a chi-square test statistic, X_0^2 the obtained value from the chi-square test for the observed result and χ_c^2 a chi-square distributed random variable with c denoting the necessary *degrees of freedom* (see below for more details). Then, X_0^2 is approximately chi-square distributed and the p-value can be approximated by [Agr12]:

$$P(X^2 \geqslant X_0^2|H_0) \approx P(\chi_c^2 \geqslant X_0^2) \tag{5.2.22}$$

Pearson Chi-Square Test

Karl Pearson introduced this hypothesis test already in 1900 with the focus on describing associations [Pea00; Agr12]. It is defined by

$$X^2 = \sum_{j=0}^{c-1} \frac{\left(n_j - \mu_j\right)^2}{\mu_j} \tag{5.2.23}$$

whereby c denotes the number of categories or classes a random variable X can be divided into, n_j the observed frequency in the corresponding class or category and μ_j the estimated frequency under the null hypothesis. For large samples, this test statistic has approximately a chi-squared distribution with $(c-1)$ *degrees of freedom*.

The higher the value of the χ^2-test the more it indicates a deviation from the null hypothesis, i.e. a high value of this test would lead to a rejection of the null hypothesis. However, in contrast to the p-value, the value of this test depends on the degrees of freedom which leads to different threshold values for a rejection. Thus, the calculation of the p-value according to Eq. 5.2.22 after this test is preferred.

Projected on the problem of detecting SNP interactions, a simple chi-square test for the association of a single SNP to a disease would extend Eq. 5.2.23 to the following:

$$X^2 = \sum_{i=0}^{2} \sum_{j=0}^{1} \frac{\left(n_{ij} - \hat{\mu}_{ij}\right)^2}{\hat{\mu}_{ij}} \tag{5.2.24}$$

Here, n_{ij} state the entries of a contingency table reflecting the observed frequencies of homozygous wild, heterozygous and homozygous variant typed samples for a specific SNP categorized in case and control group. Note that μ_{ij} has been replaced by $\hat{\mu}_{ij}$. μ_{ij} is the expected value of n_{ij} under the null hypothesis and typically calculated as

$$\mu_{ij} = E\{n_{ij}\} = n\pi_{i\bullet}\pi_{\bullet j} \tag{5.2.25}$$

with

$$\pi_{i\bullet} = \sum_j \pi_{ij} \tag{5.2.26}$$

and $\pi_{\bullet j}$ analogue. Unfortunately, $\pi_{i\bullet}$ and $\pi_{\bullet j}$ are generally unknown, but their maximum likelihood estimates can be calculated by

$$\hat{\pi}_{i\bullet} = n_{i\bullet}/n \tag{5.2.27}$$

$$\text{and} \qquad \hat{\pi}_{\bullet j} = n_{\bullet j}/n \tag{5.2.28}$$

with n denoting the total number of samples. Therefore, the expected frequencies can be estimated as

$$\hat{\mu}_{ij} = n\hat{\pi}_{i\bullet}\hat{\pi}_{\bullet j} = \frac{n_{i\bullet}n_{\bullet j}}{n}. \tag{5.2.29}$$

Since the expected frequencies have to be estimated, the statistic described by Eq. 5.2.24 is approximately chi-square distributed with $(2 \cdot 1) = 2$ degrees of freedom.

However, according to [Agr12], problems with this relatively simple test occur if the underlying contingency table is sparse or contains small values leading to small and moderately large $\hat{\mu}_{ij}$. In such cases, the test statistic can deviate far from a chi-squared distribution. In [GN96] the $\hat{\mu}_{ij}$ are recommended not to be less than 5 except for one, if the rejection threshold for the p-value is 5%. The error increases for lower p-values, i.e. the conditions get worse if the rejection limit is chosen to be less than 5%, and for increasing degrees of freedom. Unfortunately, the test for pairwise or higher-order SNP interactions gets imprecise since the sparseness of contingency tables clearly grows with the order of interaction. Contingency tables with zero entries occur already for pairwise interaction tests of the WTCCC datasets.

The definitions of the p-value and the chi-square test now help to understand the different approaches other authors followed in order to detect SNP interactions. Among these, Wan et al. introduced BOOST [WYY+10a] using log-linear models, which is described in Sect. 5.3, and Piriyapongsa et al. introduced iLOCi [PNI+12] using ρ-values, a self-created measure for

pairwise interactions described in Sect. 5.4.

5.3 Detecting Pair-wise SNP Interactions with BOOST

According to the authors, BOOST provides a fast approach for detecting pairwise SNP-interactions in GWAS [WYY+10a]. However, the complete evaluation of all SNP pairs from a filtered WTCCC dataset [WTCCC07] with about 360,000 remaining SNPs took about 60 hours on a standard desktop PC system. With the help of GPUs [YYW+11; GSK+14; GKW+15; GWK+15] or CPU clusters [KGW+14] the runtime could significantly be reduced. This section describes and evaluates the acceleration of BOOST with the help of FPGA technology, which has been published in [GWK+15]. On the RIVYERA S6-LX150 system with 128 Spartan6 FPGAs the runtime for a complete WTCCC dataset with 500,000 SNPs reduces to only 5 m 20 s. Furthermore, the design has been adapted to the Kintex7 FPGA of the KC705 development board. Here, the runtime is only 50 m 39 s for the same dataset on a single FPGA.

5.3.1 BOOST Algorithm

Measuring Interaction via Log-Linear Models

BOOST measures pairwise interactions of markers via log-linear models. For this purpose, the authors define two logistic regression models describing the interaction, as in [Cor02]. The *main effect model* with only main effects has the form

$$log\frac{P\left(Y=0|X_p=i,X_q=j\right)}{P\left(Y=1|X_p=i,X_q=j\right)}=\beta_0+\beta_i^{X_p}+\beta_j^{X_q} \tag{5.3.1}$$

while the *full model* with main effects and interaction effects has the form

$$log\frac{P\left(Y=0|X_p=i,X_q=j\right)}{P\left(Y=1|X_p=i,X_q=j\right)}=\beta_0+\beta_i^{X_p}+\beta_j^{X_q}+\beta_{ij}^{X_pX_q}. \tag{5.3.2}$$

Here, Y is a random variable describing the disease state, and X_p and X_q are the random variables describing the two SNPs of a pair in the three possible categories homozygous wild, heterozygous and homozygous variant. β_0 states the null model, $\beta_i^{X_p}$ and $\beta_j^{X_q}$ describe the coefficients for X_p and X_q respectively at category i and j, and $\beta_{ij}^{X_pX_q}$ represent the coefficients for all the combinations of the categories of both SNPs. Note that $\beta_i^{X_p}$ only represents two coefficients since one category is used as reference. This applies for $\beta_j^{X_q}$ and $\beta_{ij}^{X_pX_q}$ analogue such that Eq. 5.3.1 has five coefficients and Eq. 5.3.2 has nine coefficients.

The equivalent log-linear models for the two logistic regression models are further referred to as the *homogeneous association model* M_H and the *saturated model* M_S. With L_H and L_S denoting the log-likelihood of M_H and M_S respectively, the interaction effect can be quantified by the difference of the maximum likelihood estimates of both log-likelihoods [Cor09]. Let n_{ijk} denote the entries of a second-order contingency table for SNP pairs, as it is depicted in Fig. 5.1 in Sect. 5.2.1. The log-likelihood function is then defined by

$$L(\mu_{ijk}) = \sum_{ijk} \Big(n_{ijk} \log(\mu_{ijk}) - \mu_{ijk} - \log(n_{ijk}!) \Big) \tag{5.3.3}$$

with μ_{ijk} describing the means of the table entries.

The maximum likelihood estimate of μ_{ijk} under the saturated model is the number of observations itself:

$$\hat{\mu}_{ijk}^S = n_{ijk} \tag{5.3.4}$$

Thus, the maximum log-likelihood of M_S is

$$\hat{L}_S = L_S(\hat{\mu}_{ijk}^S) = \sum_{ijk} \Big(n_{ijk} \log(n_{ijk}) - n_{ijk} - \log(n_{ijk}!) \Big) . \tag{5.3.5}$$

Unfortunately, no closed form exists to calculate the maximum likelihood estimate $\hat{\mu}_{ijk}^H$ under the homogeneous model, but the solution exists and is

unique [Agr12]. Therefore, $\hat{L}_H$ can be denoted as

$$\hat{L}_H = L_H(\hat{\mu}_{ijk}^H) = \max_{\mu_{ijk}} L_H(\mu_{ijk}). \tag{5.3.6}$$

The difference of the maximum likelihood estimates can now be used to measure the interaction effect:

$$\hat{L}_S - \hat{L}_H = \sum_{ijk} \left(n_{ijk} \log \frac{n_{ijk}}{\hat{\mu}_{ijk}^H} - n_{ijk} + \hat{\mu}_{ijk}^H \right) \tag{5.3.7}$$

It is easy to see that the sum of all means of the contingency table cells must evaluate to the total sum of all samples:

$$\sum_{ijk} \hat{\mu}_{ijk}^H = n \tag{5.3.8}$$

Hence, Eq. 5.3.7 can be further reduced to:

$$\hat{L}_S - \hat{L}_H = \sum_{ijk} \left(n_{ijk} \log \frac{n_{ijk}}{\hat{\mu}_{ijk}^H} \right) \tag{5.3.9}$$

Let π_{ijk} denote the joint distribution obtained under the saturated model M_S and $\hat{p}_i jk$ the distribution obtained under the homogeneous model M_H. Then, Eq. 5.3.9 can be evaluated to:

$$\hat{L}_S - \hat{L}_H = n \sum_{ijk} \left(\frac{n_{ijk}}{n} \log \frac{\frac{n_{ijk}}{n}}{\frac{\hat{\mu}_{ijk}^H}{n}} \right) \tag{5.3.10}$$

$$= n \sum_{ijk} \left(\pi_{ijk} \log \frac{\pi_{ijk}}{\hat{p}_{ijk}} \right) \tag{5.3.11}$$

$$= n D_{\mathrm{KL}} \left(\pi_{ijk} || \hat{p}_{ijk} \right) \tag{5.3.12}$$

This proportional correlation to the Kullback-Leibler divergence (see also Sect. 5.2.2) lets the authors interpret the interaction effect as information

contained only in the saturated model, but not in the model with only homogeneous associations. Furthermore, this shows that if no interaction effect exists, the observed joint distribution can well be explained by the lower-order homogeneous model.

Kirkwood Superposition Approximation

It was already mentioned that no closed form exists to calculate $\hat{\mu}_{ijk}^{H}$ and thus $\hat{p}_{ijk}$. Commonly, this is done using iterative methods. However, in order to deal with the massive amount of the necessary pairwise tests for a complete GWAS, the authors of BOOST introduced the utilization of the *Kirkwood Superposition Approximation (KSA)* for this problem. The KSA for $\hat{p}_{ijk}$ is defined by

$$\hat{p}_{ijk}^{K} = \frac{1}{\eta} \frac{\pi_{ij\bullet}\pi_{i\bullet k}\pi_{\bullet jk}}{\pi_{i\bullet\bullet}\pi_{\bullet j\bullet}\pi_{\bullet\bullet k}} \tag{5.3.13}$$

whereby

$$\eta = \sum_{ijk} \frac{\pi_{ij\bullet}\pi_{i\bullet k}\pi_{\bullet jk}}{\pi_{i\bullet\bullet}\pi_{\bullet j\bullet}\pi_{\bullet\bullet k}} \tag{5.3.14}$$

$$= \sum_{ijk} \frac{n_{ij\bullet}n_{i\bullet k}n_{\bullet jk}}{n_{i\bullet\bullet}n_{\bullet j\bullet}n_{\bullet\bullet k}} \tag{5.3.15}$$

is a normalization term. The representation uses the dot notation already introduced in Sect. 5.2.3, i.e.

$$\pi_{ij\bullet} = \sum_{k} \pi_{ijk} \tag{5.3.16}$$

and analogue for others.

With this definition applied to Eq. 5.3.11, the interaction measure can be approximated with a non-iterative method. The authors show that the new measure $\hat{L}_S - \hat{L}_{KSA}$ is a tight upper bound of $\hat{L}_S - \hat{L}_H$, i.e.

$$\hat{L}_S - \hat{L}_H \leqslant \hat{L}_S - \hat{L}_{KSA} \tag{5.3.17}$$

and if the joint distribution is $\hat{p}_{ijk}$ the equality holds.

Screening and Testing

Taking advantage of these findings, the algorithm BOOST consists of two steps, *screening* and *testing*. In the screening phase all possible SNP pairs of the incoming dataset are analyzed with the proposed method. A threshold is applied to the KSA measure to filter out insignificant SNP pairs. The authors propose a threshold of $\tau = 15$ (in fact, they propose a threshold of $\tau = 30$ applied to the measure of $2(\hat{L}_S - \hat{L}_{KSA})$, which is factually the same), and they used e as the base for the logarithm, i.e. they calculate the natural logarithm "ln" in their equations. However, the threshold can be user-defined.

In the testing phase, each SNP pair passing the KSA filter is further analyzed by the likelihood ratio statistic (or *log-linear test*) according to Eq. 5.3.11 with iteratively fitted models M_S and M_H. After that, the chi-square test with four degrees of freedom is applied with a final calculation of the p-value (see also Sect. 5.2.3) to test whether the result is significant or not.

According to the authors, from about 12.5 billion SNP pairs to be analyzed in a WTCCC dataset only about 300,000 to 600,000 had to be analyzed in the testing phase. Therefore, the FPGA implementation in the next section concentrates only on the screening phase.

5.3.2 BOOST on FPGAs

Overview

The implementation of BOOST on FPGAs has to be divided into two parts. Firstly, the software part is responsible for reading and preparation of the input data, e.g. a GWAS. It handles the data distribution over several FPGAs and, if necessary, over several FPGA runs. Furthermore, it collects the resulting SNP pairs passing the KSA filter and calculates the closing log-linear test. Since these tasks are implemented relative straight forward, the focus of this section lies on the second part, the hardware implementation. It is responsible for generating the contingency tables for each possible

SNP pair and applying the KSA filter to these tables. The result will be a collection of SNP pair IDs passing this filter. However, a closer look will be given to the software part in terms of data distribution over several FPGAs and runs. The hardware implementation mainly targets a Spartan6-LX150 FPGA of the RIVYERA S6-LX150 architecture. Furthermore, the design has been adapted to a Kintex7-325T FPGA on the KC705 development board, providing about 13.5 times the performance of the Spartan6.

Generating Contingency Tables

The main computational task to perform a pairwise interaction test is usually not the calculation of the test statistic or filter, but the collection of data for the contingency table. This observation results from the generally large amount of input data where, e.g. in the case of a WTCCC dataset, for each SNP pair the genotype data for each of the 5,000 samples has to be collected at two marker positions. Therefore, the input data for the FPGA is organized by marker IDs and not by sample ID, i.e. each "line" contains the genotypes for one marker position. Furthermore, each line of genotypes is grouped by cases and controls with the cases first. If the user's raw data is not organized in this format, the software part may perform the transposition step in advance. However, it is preferable to directly load a transposed dataset (which could be generated by converting the original dataset with an external tool, e.g. PLINK [PNT+07]).

From the complete input data each FPGA processes two intervals of markers, which are loaded into the external DRAM of the FPGA. From these intervals all pairwise combinations of the first interval and all pairs containing one marker from the first and one from the second interval are analyzed. (For more details see the paragraph on *Parallelization and Data Distribution* below.) In order to do so, a chain of processing elements (PEs) with a systolic-like data flow is implemented on each FPGA. These processing elements concurrently generate contingency tables for different SNP pairs and provide the data via a simple data transfer mechanism to a unit applying the subsequent KSA filter. See Fig. 5.4 for an overview.

Each PE contains three main components, a local RAM (implemented in Block RAM) to store the genotypes of one SNP, the required counters of one

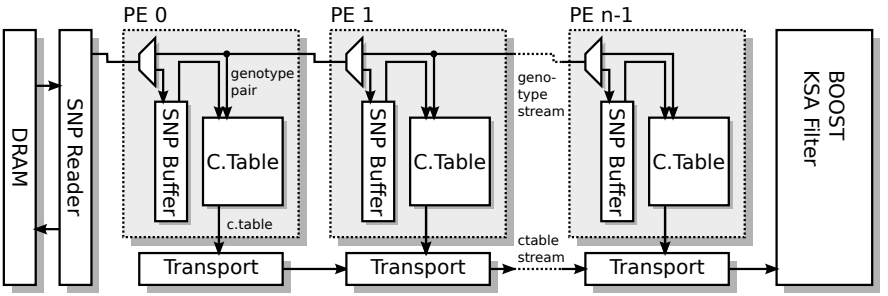


Figure 5.4. Overview of the chain of processing elements for contingency table creation in BOOST.

contingency table, and a bus transfer unit to send a completed contingency table to the unit applying the filter.

The SNP data is streamed in several iterations from the external DRAM to the first PE in the chain. The first SNP arriving at each processing element is simply stored in the local RAM. Any further SNPs are streamed to the next element in the chain. This way all l PEs contain the first l SNPs in their local RAM whereby l denotes the total number of PEs. The contingency table in each PE is created while streaming SNP data to the next PE. Each genotype of the data in the stream is compared to the corresponding genotype of the SNP stored in the local RAM and the appropriate counter of the contingency table is incremented.

When all genotypes of either cases or controls of one SNP have been streamed, the contents of the contingency table are provided via the bus transfer unit to a FIFO buffer collecting all tables from all PEs, and the local counters are reset. To save valuable FPGA resources, it is not necessary to implement all nine required counters for each table. Since the total numbers of cases and controls are known in advance, it is possible to reconstruct one counter value if all others are already known. In this case, the counter value n_{22k} is reconstructed, whereby k denotes either case or control group.

$$n_{22k} = n_{\bullet\bullet k} - \sum_{ij \neq 22} n_{ijk} \tag{5.3.18}$$

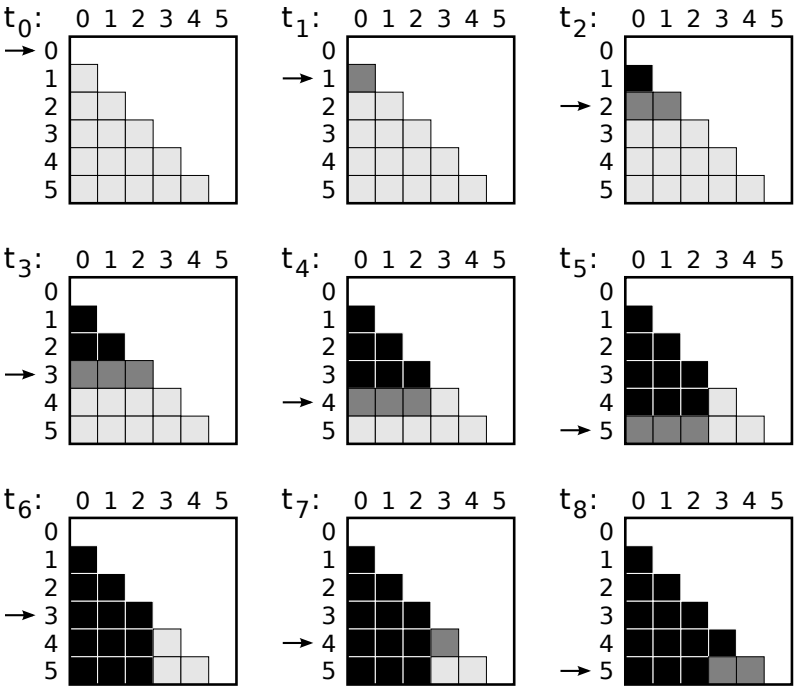


Figure 5.5. Sequence of the parallel creation of SNP pairs from an example dataset of six SNPs with a chain of three PEs in nine time steps. Light gray squares indicate SNP pair combinations to be processed while black squares indicate already processed pairs. Dark gray squares are currently being processed while an arrow on the vertical axis indicates the currently streamed SNP.

The reconstruction is performed by a single unit for all PEs at the end of the PE chain. The calculation of the KSA value starts the moment the tables for cases and controls of a SNP pair are complete (see paragraph on *Implementation of the KSA Filter* below).

After all SNP data from both intervals have been streamed, the next iteration begins. This way, the first iteration has already generated all contingency tables corresponding to the SNP combinations of the first l SNPs to all others in both intervals. Thus, each further iteration starts

streaming all SNPs from both intervals omitting the first l SNPs of the previous iteration and all SNPs which were omitted before as well. The process is finished when all SNPs of the first interval have to be omitted. This way the partitioning of SNP data into two intervals for each FPGA creates the base for an efficient distribution of all SNPs among all FPGAs. Each FPGA computes the statistics for all possible SNP pairings in the first interval and all pairings between the first and the second interval.

Figure 5.5 illustrates how this process works for a PE chain with three PEs on a small dataset with six SNPs. The distribution of SNPs among FPGAs is discussed in the paragraph on *Parallelization and Data Distribution* below.

Implementation of the KSA Filter

The FPGA implementation of the KSA filter is designed in a completely unrolled pipeline allowing to get the result of one test every 18 clock cycles (i.e. the number of entries in both contingency tables) after a certain latency. In order to achieve an efficient FPGA implementation, we have transformed the calculation of $\hat{L}_S - \hat{L}_{KSA}$ according to Eq. 5.3.11 using the natural logarithm as follows:

$$\hat{L}_S - \hat{L}_{KSA} = n \sum_{ijk} \left(\pi_{ijk} \ln \frac{\pi_{ijk}}{\hat{p}^K_{ijk}} \right) \tag{5.3.19}$$

$$= n \sum_{ijk} \left(\frac{n_{ijk}}{n} \ln \frac{\frac{n_{ijk}}{n}}{\frac{1}{\eta} \frac{\pi_{ij\bullet}\pi_{i\bullet k}\pi_{\bullet jk}}{\pi_{i\bullet\bullet}\pi_{\bullet j\bullet}\pi_{\bullet\bullet k}}} \right) \tag{5.3.20}$$

$$= \sum_{ijk} \left(n_{ijk} \ln \frac{n_{ijk}}{\frac{n}{\eta} \frac{n_{ij\bullet}n_{i\bullet k}n_{\bullet jk}}{n_{i\bullet\bullet}n_{\bullet j\bullet}n_{\bullet\bullet k}}} \right) \tag{5.3.21}$$

$$= \sum_{ijk} \left(n_{ijk} \left(\ln \frac{n_{ijk}n_{i\bullet\bullet}n_{\bullet j\bullet}n_{\bullet\bullet k}}{n_{ij\bullet}n_{i\bullet k}n_{\bullet jk}} + \ln \frac{\eta}{n} \right) \right) \tag{5.3.22}$$

$$= \sum_{ijk} \left(n_{ijk} \ln \frac{n_{ijk} n_{i\bullet\bullet} n_{\bullet j\bullet} n_{\bullet\bullet k}}{n_{ij\bullet} n_{i\bullet k} n_{\bullet jk}} \right) + n \left(\ln \eta - \ln n \right) \qquad (5.3.23)$$

Equation (5.3.23) is directly implemented into the hardware description of the FPGA with the following optimizations. Firstly, the accumulations $n_{\bullet jk}$ and $n_{i\bullet k}$ are calculated on-the-fly and stored into separate FIFOs while receiving n_{ijk} from the PE chain. The sums $n_{ij\bullet}$, $n_{\bullet j\bullet}$ and $n_{i\bullet\bullet}$ can then be calculated using a simple adder resource each at the corresponding FIFO outputs. Secondly, a divider resource is required to calculate η. It is easy to see, that each fraction in (5.3.15) is between zero and one (if none of the factors in the denominator is zero). Hence, an efficient divider unit is implemented which generates a fixed point format with only 32 fraction bits. The special case, if one of the factors in the denominator is zero, directly triggers a division-by-zero error and regards the calculated value as insignificant.

To save further resources, only two logarithm units are implemented. Both units iteratively generate a fixed point format with 8 signed integer and 56 fraction bits. The first one directly calculates the logarithm of a fraction (i.e. $\ln \frac{a}{b}$) to save another divider unit. The second calculates $\ln \eta$. Since $\ln n$ is a constant throughout the whole process, it is calculated by the same unit by selecting n as input as long as η is not ready. Since the Xilinx Coregen does not provide a library for Spartan6 FPGAs to calculate the logarithm, an own implementation is presented in the paragraph on *Implementation of the Logarithm Units* below. Nonetheless, the same implementation has been used in the adaption to the Kintex7 device as well.

Due to the implementation as an unrolled pipeline, each clock cycle produces after several latency cycles one summand of the left part in (5.3.23) and one summand of η in (5.3.15). Within 18 cycles the accumulations are complete and the result of the whole equation is obtained and compared to a user definable threshold. Of each significance value passing the threshold, the identifier to the corresponding SNP pair is stored in a FIFO. Due to the pipeline nature of this design, the calculation of each significance value can be started every 18 clock cycles. There is no need to ever block the data flow, resulting in very fine-grained concurrent processing and therefore very efficient utilization of FPGA resources.

Since the log-linear filter still has to be applied to the SNP pairs passing the KSA filter, the CPU concurrently fetches the corresponding IDs from the FPGA FIFOs and stores them in a local buffer. The items in the buffer are submitted to an arbitrary number of worker threads calculating the necessary log-linear tests. In most cases, if the threshold is not set too low (as for normal processing, see paragraph on *Evaluation* below), the calculation of the tests can be finished in time with the FPGAs, i.e. the log-linear tests do not significantly increase the runtime.

The complete KSA filter pipeline is depicted in Fig. 5.6. The calculations are performed in a fixed-point format with a precision continuously adapted in the pipeline resulting from the unsigned 16 bit input precision for the contingency table entries (and accumulations) and the precision of the divider and logarithm units, but never exceeding a total width of 64 bits. The output precision for the KSA value is signed 27.37 bit which is sufficient if the total number of samples does not exceed $2^{16} - 1 = 65,535$. See the illustration in Fig. 5.6 again for the details of the precision of signals in the filter.

Implementation of the Logarithm Units

Two different logarithm units were implemented on the Spartan6 FPGA for the KSA filter. One computes the natural logarithm of a number in unsigned 32.32 bit fixed-point representation, the other computes the logarithm of the fraction $\frac{a}{b}$ with a and b being unsigned 64 bit integer numbers. This implementation is necessary since the Xilinx Coregen does not provide a ready-to-use library for Spartan6 FPGAs. Both units share the same principle taken from [Owe12]. It avoids the usage of multiplications and divisions since these operations are expensive on an FPGA in terms of resource requirements. Instead, the algorithm requires only addition, comparison and shift operations in a fixed number of iterations and can therefore be efficiently implemented on the Spartan6.

The algorithm uses two temporary variables x and y. In the implementation, x is interpreted as an unsigned 1.63 bit fixed-point number while y is interpreted as signed 8.56 bit fixed-point. Both variables are marked with an index to indicate the corresponding iteration. The input is denoted with

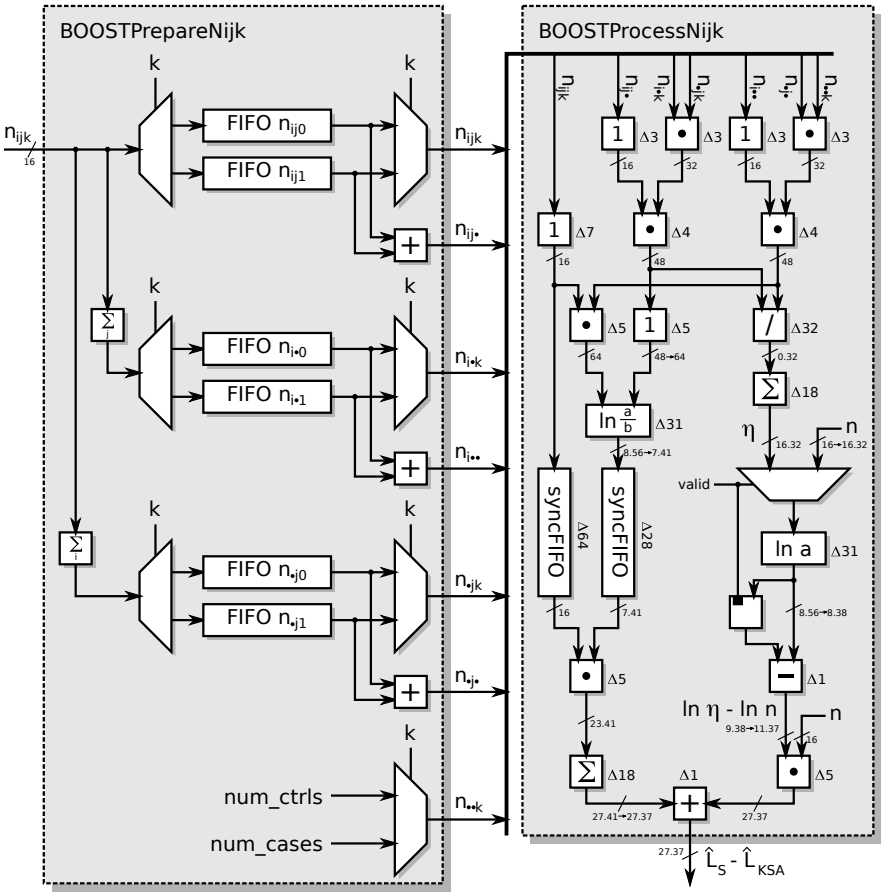


Figure 5.6. Illustration of the KSA filter pipeline in the implementation of BOOST. Δ denotes the delay of a component in clock cycles. The precision of a signal is denoted as a single number *a* if the signal is interpreted as an unsigned integer without fraction, or as *a*.*b* if the signal is interpreted as a signed fixed-point number with *a* integer and *b* fraction bits. An arrow indicates an implicit precision conversion.

a, interpreted as unsigned 32.32 bit fixed-point and has to be $0 < a < 2^{31}$ for the algorithm to work correctly. In each iteration of the algorithm, the following invariant is maintained:

$$y = \ln \frac{a}{x} \tag{5.3.24}$$

Thus, the start values of x and y are set to the following:

$$x_0 = \frac{a}{2^{31}} \tag{5.3.25}$$

$$y_0 = \ln 2^{31} \tag{5.3.26}$$

The realization of x_0 is simply achieved by reinterpreting the input a as 1.63 bit instead of 32.32 bit fixed-point number. The value for y_0 is pre-calculated.

The goal is to get the value of x close to one, i.e. $x \to 1$, while maintaining the invariant. Clearly, it follows $y \to \ln a$ which will be returned as the result.

The algorithm is divided into two parts. The first part "normalizes" the value of x to ensure $0.5 \leqslant x \leqslant 1$. This is done by shifting x as long as the most significant bit (MSB) does not become 1 while subtracting the logarithm of the corresponding power of two from y to maintain the invariant (i denotes the number of shifts):

$$y - \ln 2^i = \ln \frac{a}{x} - \ln 2^i \tag{5.3.27}$$

$$= \ln \frac{a}{x2^i} \tag{5.3.28}$$

The second part approximates x to one by multiplying x with so-called *nice numbers* of the form $1 + 2^{-i}$ with $i \in \mathbb{N}$. Again, the logarithm of this number is subtracted from y in each step to maintain the invariant. i reflects the index of the current iteration starting with $i = 1$. Since $\ln x$ gets closer to zero, the precision is increased with each step. To enable a practical pipelined solution, a fixed number of steps of 24 is chosen in the second part. The absolute error to the true result in each step is $\ln x$:

$$y + \ln x = \ln \frac{a}{x} + \ln x \tag{5.3.29}$$

$$= \ln a \tag{5.3.30}$$

Thus, after 24 steps, the error of the result is less than $\ln(1 + 2^{-24}) \approx 5.96 \cdot 10^{-8}$ which is already sufficient. However, since for t with $t \approx 1$, $\ln t$ is approximately $t - 1$. Adding the value of $(x - 1)$ to the final result in the last step leads to a further improvement of the precision:

$$y + (x - 1) = \ln \frac{a}{x} + (x - 1) \tag{5.3.31}$$

$$\approx \ln \frac{a}{x} + \ln x \tag{5.3.32}$$

$$= \ln a \tag{5.3.33}$$

Now, the absolute error of the result is less than $\ln x + (x - 1) = \ln(1 + 2^{-24}) + ((1 + 2^{-24}) - 1) \approx 1.78 \cdot 10^{-15}$. The complete pseudocode of the implemented algorithm can be seen in Listing 5.1. Note that the multiplications with powers of two are implemented as shift operations and the values for $\ln 2^{2^{(6-i)}}$ and $\ln(1 + 2^{-i})$ are precalculated. The multiplication $x \cdot (1 + 2^{-i})$ is the same as adding the by i bits right-shifted x to x. In conclusion, the algorithm calculates the natural logarithm of the input a with a precision of at least 14 digits in 31 steps requiring only subtractions, shifts, comparisons and 30 precalculated values, which makes it ideal for a pipelined FPGA implementation.

The unit directly calculating the natural logarithm of a fraction $\ln \frac{a}{b}$ uses the same strategy, but never calculates the fraction $\frac{a}{b}$ directly. Instead, it makes use of the relation $\ln \frac{a}{b} = \ln a - \ln b$ and handles the nominator a and denominator b separately and similar to x in the previous algorithm. In fact, it calculates $\ln a$ and $\ln b$ separately, but with interleaved iteration steps and a common variable for the intermediate result y. The temporary variables corresponding to x are further denoted as p and q. The algorithm evaluates $p \to 1$ and $q \to 1$ while maintaining the following invariant:

$$y = \ln \frac{aq}{bp} \tag{5.3.34}$$

Here, the input variables a and b can be interpreted as any fixed-point

Listing 5.1. Pseudocode for calculating the natural logarithm $\ln a$

```
// initial values
x = a / 2^31
y = ln 2^31

// first part: "normalization"
for i in 1 to 6 do
  if x < 2^-2^(6-i) then
    x = x * 2^2^(6-i)
    y = y - ln 2^2^(6-i)
  end if
end for

// second part: evaluate x → 1
for i in 1 to 24 do
  t = x * (1 + 2^-i)
  if t < 1 then
    x = t
    y = y - ln(1 + 2^-i)
  end if
end for

// increase precision
y = y + (x - 1)
return y
```

number as long as the interpretation is equal for both. This is clearly to see since any multiplicative factor is canceled out in the fraction. Formally, p and q are interpreted as 1.63 unsigned fixed-point and y is in 8.56 signed fixed-point format as before. Initially, p and q are set to a and b respectively, but interpreted as 1.63 unsigned fixed-point. In order to maintain the invariant, the start value of y has to be zero. Listing 5.2 shows the adapted pseudocode.

Listing 5.2. Pseudocode for calculating the natural logarithm of a fraction $\ln \frac{a}{b}$

```
// initial values
p = a // reinterpreted as 1.63 unsigned fixed-point
q = b // reinterpreted as 1.63 unsigned fixed-point
y = 0

// first part: "normalization"
for i in 1 to 6 do
  if p < 2^-2^(6-i) then
    p = p * 2^2^(6-i)
    y = y - ln 2^2^(6-i)
  end if
  if q < 2^-2^(6-i) then
    q = q * 2^2^(6-i)
    y = y + ln 2^2^(6-i)
  end if
end for

// second part: evaluate p → 1 and q → 1
for i in 1 to 24 do
  t = p * (1 + 2^-i)
  if t < 1 then
    p = t
    y = y - ln(1 + 2^-i)
  end if
  t = q * (1 + 2^-i)
  if t < 1 then
    q = t
    y = y + ln(1 + 2^-i)
  end if
end for

// increase precision
y = y + (p - 1) - (q - 1)
return y
```

Multiple PE Chains

As described in the paragraph on *Generating Contingency Tables* the processing elements are fed with data from a genotype stream. Since the streaming of genotypes is a runtime critical task, efforts have been made to perform this task as fast as possible. This includes streaming of multiple genotypes in one clock cycle. For the Spartan6 FPGA only 2 genotypes could be streamed per clock cycle while for the Kintex7 FPGA streaming of 8 genotypes per clock cycle was possible, resulting in four times higher throughput. (The effect on the total runtime can be seen in the paragraph on *Evaluation* below).

This performance gain comes with a drawback. The more genotypes can be streamed in a clock cycle and the more process elements are implemented in a chain, the sooner it comes to an overload of the transport system for the contingency tables, depending on the number of samples in the dataset. Since the speed of the transport system is directly correlated to the maximum throughput of the KSA filter, which is 18 clock cycles per contingency table, it cannot be accelerated. In fact, 9 clock cycles are available for each PE to transfer half a contingency table (for either cases or controls). With l denoting the number of process elements in the PE chain, $9l$ is the time in clock cycles required to fetch all tables from all process elements. Now, let g denote the number of genotypes streamed per clock cycle. The minimum number of samples for either case or control group $n_{\min}$ can be calculated as follows.

$$n_{\min} = 9lg \tag{5.3.35}$$

The focus of the designs presented here lies on the ability to handle at least a dataset of the size of a WTCCC dataset, i.e. 500,000 SNPs of 2,000 cases and 3,000 controls. Thus, $n_{\min}$ must be at least 2,000. With a streaming of $g = 2$ genotypes per clock cycle this results in a maximum chain length of $l = 111$ processing elements. For the Spartan6 this does not present a problem at all. Due to the lack of resources the maximum number of processing elements for implementation had been 80 (see *Evaluation*).

In contrast, the Kintex7 implementation is able to stream $g = 8$ genotypes per clock cycle. Thus, the maximum chain length is only $l = 27$, which

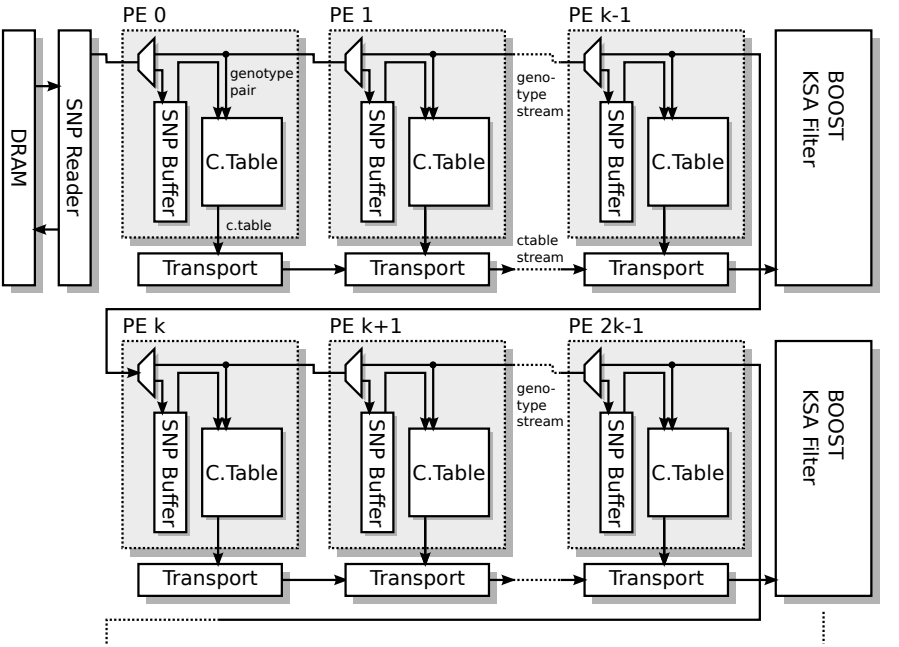


Figure 5.7. Example for a BOOST implementation with multiple PE chains of chain length k.

is significantly below the resource capabilities of the device. Therefore, multiple PE chains with multiple KSA-filter units have been introduced. The genotype streaming is done as before, i.e. as if all chains were connected subsequently. However, the transport systems of each chain are independent now. Each system is connected to its own KSA filter unit processing only SNP pairs assigned to its chain. The results are collected in a small independent buffer as well before being collected to a common buffer and provided to the host system. This way arbitrary many PE chains can be implemented where merely each chain alone is restricted to Eq. 5.3.35. See Fig. 5.7 for an example of multiple PE chains.

Data Distribution

The number of SNP-pairs each FPGA processes directly results from the sizes of the two intervals of SNPs which are assigned to each FPGA. Let the size of the first interval of FPGA i be denoted as a_i. (Against the usual way computer scientists are counting, the first FPGA has to be indexed with $i = 1$.) The size of the second interval is then calculated by $\sum_{j=0}^{i-1} a_j$ starting with $a_0 = 0$. To calculate a_i the total number of SNP pairings $n(n-1)/2$ is equally distributed over all FPGAs resulting in $n(n-1)/2t$ SNP pairs to be processed by each FPGA (with t denoting the number of available FPGAs). Thus, the following equations hold for $i > 0$:

$$\frac{n(n-1)}{2t} = \frac{a_i^2}{2} + a_i \sum_{j=0}^{i-1} a_j \tag{5.3.36}$$

$$\Leftrightarrow \quad 0 = a_i^2 + 2a_i \sum_{j=0}^{i-1} a_j - \frac{n(n-1)}{t} \tag{5.3.37}$$

$$\Leftrightarrow \quad a_i = -\sum_{j=0}^{i-1} a_j + \sqrt{\left(\sum_{j=0}^{i-1} a_j\right)^2 + \frac{n(n-1)}{t}} \tag{5.3.38}$$

It is likely for larger datasets, that for some FPGAs the calculated SNP intervals do not fit into the attached DRAM memory of the FPGA. In this case, the second interval is split over multiple FPGAs and the interval sizes are recalculated considering the adjusted size of the second interval. Unfortunately, it has to be taken into account that the SNP pairs from the first interval are then processed multiple times. This amount of redundant calculations has to be considered for the total SNP distribution. Furthermore, redundant results need to be filtered.

The particular SNPs are now mapped directly to the calculated interval sizes and are assigned to the corresponding FPGAs for an equal workload. Figure 5.8 illustrates an example distribution over 8 FPGAs.

If the total number of SNPs exceeds the maximum possible number of SNPs that can be distributed among the FPGAs, the distribution process

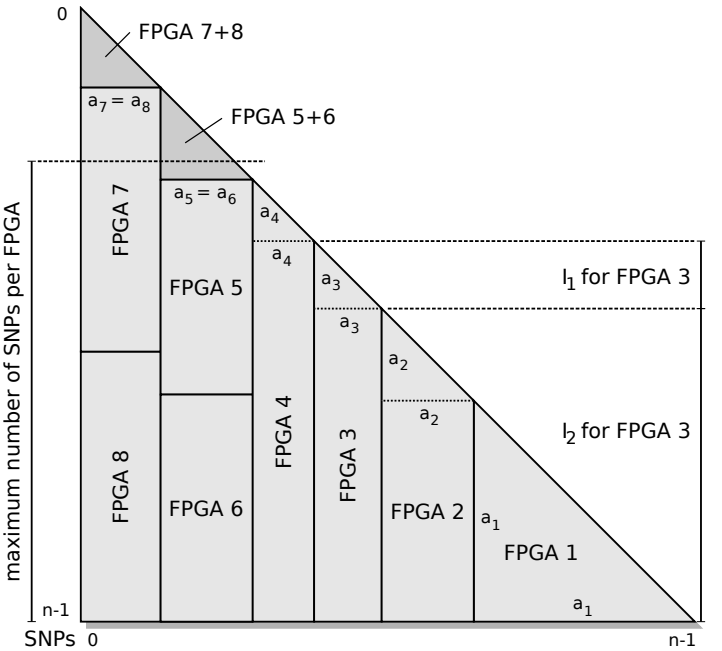


Figure 5.8. Example SNP distribution among 8 FPGAs. The two intervals I_1 and I_2 for SNP processing of FPGA 3 are exemplarily marked.

is executed with a multiple r of the number of available FPGAs t. Then, r represents the number of rounds to be performed such that each FPGA successively processes r intervals from this distribution. Especially, this applies if only one FPGA is available, as in the case of the KC705 implementation.

Evaluation

BOOST has been implemented on the Spartan6-LX150 FPGA for the RIVYERA S6-LX150 architecture with 80 processing elements running at 100 MHz. This leads to a total of $128 \times 80 = 10,240$ PEs on the complete RIVYERA.

In each clock cycle two genotypes are streamed through the PE chain. According to Eq. 5.3.35 the minimum number of samples in case and control

group has to be $n_{\min} = 1,440$ which is sufficient for a WTCCC dataset. The total runtime for such a dataset is only 5 m 20 s on the RIVYERA leading to a speedup of 1,890 to the original implementation.

Furthermore, the design has been adapted to the Kintex7-325T FPGA on the KC705 development board. Here, in each clock cycle 8 genotypes are streamed through the PE chain. According to the $n_{\min}$ calculation, the maximum number of processing elements in the chain is only 27 to be able to handle a WTCCC dataset. Thus, the design for the Kintex7 has been implemented with multiple PE chains. In particular, 4 chains with each 27 processing elements have been implemented resulting in 108 PEs in total. The design is clocked with a frequency of 250 MHz such that the total runtime to analyze a WTCCC dataset is only 50 m 39 s. This is about 13.5 times faster than a single Spartan6-LX150 FPGA.

It has been verified that the reported SNP pairs with a potential interaction are the same as from the original publication. Nevertheless, due to the imprecise calculation using fixed-point arithmetics on the FPGAs, a slight deviation of the results from the original using double precision floating-point arithmetics has been expected. However, the total deviation of the KSA filter score has never been greater than 0.01. After application of the exact log-linear test on the CPU, the output list of SNP-pairs with potential epistasis from the FPGA implementations is identical to the one returned by the original BOOST.

For further comparison, results from GPU implementations of BOOST have been taken from [GWK+15]. These include runtimes of a multi GPU implementation on 4 Nvidia GeForce GTX Titan and 4 Nvidia Tesla K20m GPUs as well as runtimes for the original GBOOST and BOOST. The CPU version of BOOST has been reimplemented with *pthreads* to take advantage from multiple CPU-cores. The runtime for the CPU versions have been extrapolated from the runtime of a smaller dataset containing 40,000 SNPs and 5,000 samples assuming a quadratic increase.

All runtimes together with the energy consumption for the different architectures are presented in Tab. 5.2. The table also includes the runtimes of BOOST on a computer cluster with 32 Intel Xeon E5-2660 octa-core processors (referred to as *Pluton*) and the Cray XC30 supercomputer *Edison* with 5,576 nodes each containing two Intel Xeon E5 *Ivy Bridge* 12-core

processors taken from [KGW+14].

The energy consumption for the RIVYERA implementation has been measured directly from the power supply with the internal IPMI interface. The continuous total power drain was about 700 W on full load. The energy consumptions of the CPU and GPU implementations are estimated from the *thermal design power (TDP)* specifications only and do not include the power consumed by the host PC (GPU only) and peripherals. Thus, the power drain of the Intel Core i7 CPU is assumed with 130 W, the Xeon E5-2660 with 95 W, the Xeon E5 *Ivy Bridge* with 115 W and the GTX Titan GPU with 250 W. Similarly, the energy consumption was estimated for the KC705 board using the estimated power drain of 12 W from the Vivado Power Report of the Kintex7 device only.

Additionally, the runtime for a simulated dataset containing 2 million SNPs for 10,000 samples has been evaluated for the FPGA designs compared to the multi-GPU implementations. These results are presented in Tab. 5.3. The runtimes for the original BOOST software on such a dataset has been extrapolated from the results in Tab. 5.2 assuming a linear increase in the number of samples and a quadratic increase in the number of SNPs.

The results show that the RIVYERA implementation clearly outperforms all other architectures by far except for the Cray XC30 supercomputer cluster. Anyway, it is only 7 times slower using only 128 low cost FPGAs against 11,152 high-performance Intel Xeon 12-core processors, and it is still 8.4 times faster than a cluster of 32 Intel Xeon 8-core processors (Pluton cluster). Furthermore, the runtime of the single FPGA solution on the Kintex7 is in the same order of magnitude as the GPU solutions and the Pluton cluster, but consuming significantly less power.

The device utilization for the BOOST implementation on the Spartan6 FPGA can be seen in Tab. 5.4. It shows a nearly fully utilized device with 87% occupied slices. However, the implementation of more processing elements was not possible since the routing process could not meet the timing requirements anymore. The same applies for the device utilization on the Kintex7 FPGA shown in Tab. 5.5. (Note that Vivado does not report the number of occupied slices anymore.)

Table 5.2. Performance and energy consumption of BOOST in various designs on different architectures analyzing the WTCCC dataset. The results for the CPU designs were estimated from a smaller dataset (*). The power consumption of the Kintex7 is FPGA-only and calculated according to the Vivado Power Report (**).

Design	Architecture	Runtime	Power (kWh)
BOOST (FPGA)	RIVYERA S6-LX150 (128× Spartan6-LX150)	5 m 20 s	0.06
BOOST (FPGA)	KC705 (1× Kintex7-325T)	50 m 39 s	0.01**
BOOST (Cray XC30)	11,152× Intel Xeon E5 *Ivy Bridge* (133,824 cores)	45 s	16.03
BOOST (Pluton)	32× Intel Xeon E5-2660 (256 cores)	45 m	2.28
BOOST (multi-GPU)	4× GTX Titan	9 m	0.15
BOOST (multi-GPU)	4× Tesla K20m	12 m	0.18
BOOST (multi-GPU)	1× GTX Titan	35 m	0.15
BOOST (multi-GPU)	1× Tesla K20m	47 m	0.18
GBOOST	1× GTX Titan	1 h 15 m	0.31
GBOOST	1× Tesla K20m	1 h 26 m	0.32
BOOST* (pthreads)	Intel Core i7-3930K (6 cores)	19 h	2.47
BOOST* (original)	Intel Core i7-3930K (1 core)	7 d	21.84

Table 5.3. Performance and energy consumption of BOOST in various designs on different architectures analyzing a simulated dataset with 2M SNPs and 10,000 samples. The results for the CPU designs were estimated from a smaller dataset (*).The power consumption of the Kintex7 is FPGA-only and calculated according to the Vivado Power Report (**).

Design	Architecture	Runtime	Power (kWh)
BOOST (FPGA)	RIVYERA S6-LX150 (128× Spartan6-LX150)	2 h 51 m	1.99
BOOST (FPGA)	KC705 (1× Kintex7-325T)	27 h 00 m	0.33**
BOOST (multi-GPU)	4× GTX Titan	4 h 08 m	4.13
BOOST (multi-GPU)	4× Tesla K20m	4 h 43 m	4.25
BOOST (multi-GPU)	1× GTX Titan	16 h 02 m	4.01
BOOST (multi-GPU)	1× Tesla K20m	18 h 39 m	4.20
BOOST* (pthreads)	Intel Core i7-3930K (6 cores)	25 d 08 h	79.04
BOOST* (original)	Intel Core i7-3930K (1 core)	224 d	698.88

Table 5.4. Device utilization of the BOOST implementation on a Spartan6-LX150.

	Occupied Slices	Slice Registers	Slice LUTs	Block RAM (18k)	DSP48A1
Used	20,060	53,381	67,551	204	17
Available	23,038	184,304	92,152	268	180
Utilization	87%	28%	73%	76%	9%

Table 5.5. Device utilization of the BOOST implementation on a Kintex7-325T.

	Slice Registers	Slice LUTs	Block RAM (36k)	DSP48E1
Used	149,713	170,299	387.5	61
Available	407,600	203,800	445	840
Utilization	37%	84%	87%	7%

5.4 Detecting Pair-wise SNP Interactions with iLOCi

iLOCi [PNI+12] uses a novel and simpler significance measurement than BOOST (see Sect. 5.3) to test for pairwise interactions. Nonetheless, the authors claim to tackle the problem other tools might have to find significant interactions with their method. The runtime on CPU architectures is yet comparable (19 hours on a MacPro workstation with two 2.4 GHz quad-core Intel Xeon CPUs for a complete WTCCC dataset with about 500,000 SNPs and 5,000 samples). However, the acceleration with FPGA technology described here leads to a runtime of less than 2.5 minutes on the RIVYERA S6-LX150 architecture for the same dataset, which is almost two times faster than the runtime for BOOST on the same architecture.

5.4.1 iLOCi Algorithm

ρ_{diff}-Value

For each possible marker pair in the input dataset iLOCi calculates the so-called ρ_{diff}-value. This is effectively the difference of two disease related values referred to as ρ_{case} and ρ_{ctrl}:

$$\rho_{\text{diff}} = |\rho_{\text{case}} - \rho_{\text{ctrl}}| \tag{5.4.1}$$

According to the authors, these two values are proven to be concordant with *linkage disequilibrium (LD) values*. Linkage disequilibrium is an

allelic deviation from a *Hardy-Weinberg Equilibrium* model which assumes that the frequencies of alleles and genotypes in a population remain constant from one generation to the next. For more details, the publication of iLOCi [PNI+12] is referred to. However, the authors note, that they do not calculate the exact LD values, since it is computationally too demanding. Thus, the only information captured by the ρ values is the correlation between markers, which is in fact the intention of this tool.

The calculation of ρ_{case} and ρ_{ctrl} is based on contingency tables (see Sect. 5.2.1). Let k denote either case or control group. Using the same notations as in Sect. 5.3 for BOOST, ρ_k evaluates to:

$$\rho_k = \frac{\pi_{00k} - \pi_{02k} - \pi_{20k} + \pi_{22k}}{\sqrt{(\pi_{0\bullet k} + \pi_{2\bullet k})\,(\pi_{\bullet 0k} + \pi_{\bullet 2k})}} \tag{5.4.2}$$

$$= \frac{\frac{1}{n}\,(n_{00k} - n_{02k} - n_{20k} + n_{22k})}{\sqrt{\frac{1}{n}\,(n_{0\bullet k} + n_{2\bullet k})\,\frac{1}{n}\,(n_{\bullet 0k} + n_{\bullet 2k})}} \tag{5.4.3}$$

$$= \frac{n_{00k} - n_{02k} - n_{20k} + n_{22k}}{\sqrt{(n_{0\bullet k} + n_{2\bullet k})\,(n_{\bullet 0k} + n_{\bullet 2k})}} \tag{5.4.4}$$

The significance value ρ_{diff} is then calculated according to Eq. 5.4.1 for each possible marker pair. Instead of defining a threshold τ, the results are ranked according to the ρ_{diff} value and the t best pairs with the highest values are provided as output and can be used for further analysis.

5.4.2 iLOCi on FPGAs

Overview

The hardware implementation of iLOCi is closely related to the one presented for BOOST in Sect. 5.3.2. Only few differences regard firstly, the representation of a contingency table, and secondly, obviously the calculation of the significance value which is ρ_{diff} in this case. Furthermore, iLOCi requires the storage of the t-best SNP pairs before returning, which is more difficult than simply buffering all SNP pairs exceeding a threshold as in BOOST.

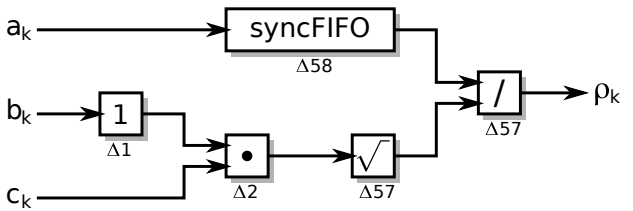


Figure 5.9. Calculation of ρ_k for iLOCi. Δ indicates the delay of each components in clock cycles. Note, that the order of the inputs is with each clock cycle b_k, then c_k and at last a_k.

Generating Contingency Tables and Calculating ρ_{diff}

For Eq. 5.4.4 the entries of the contingency tables are required in three different summations. Thus, the representation of a contingency table can be reduced to only three values for cases and control group respectively (denoted by k) according to the following definitions.

$$a_k = n_{00k} - n_{02k} - n_{20k} + n_{22k} \tag{5.4.5}$$

$$b_k = n_{0\bullet k} + n_{2\bullet k} \tag{5.4.6}$$

$$c_k = n_{\bullet 0k} + n_{\bullet 2k} \tag{5.4.7}$$

These values are directly calculated on-the-fly by the PEs instead of determining all entries of the table first. The calculation of ρ_k then reduces to:

$$\rho_k = \frac{a_k}{\sqrt{b_k c_k}} \tag{5.4.8}$$

As the significance calculation in BOOST, Eq. 5.4.8 is implemented as a completely unrolled pipeline requiring the resources of a multiplier, divider and a square root extractor. The multiplication is performed in integer arithmetic while the division and square root extraction require double precision floating point arithmetic. The value a_k is delayed with the help of a small FIFO for the time the multiplication and square root extraction endures. A streaming diagram of the calculation of ρ_k is depicted in Fig. 5.9.

Since all case samples are streamed before the control samples, ρ_k can

directly be calculated as soon as a half-table for either cases or controls is ready. In order to calculate ρ_{diff} according to Eq. 5.4.1 only ρ_{case} has to be buffered for all PEs. This is contrary to BOOST where all entries of the contingency tables had to be buffered before the calculation of the significance had been possible. Furthermore, it is possible to start a new calculation of ρ_k every three clock cycles since only three values are required for the computation.

Storing the t-best Results

In order to keep the t-best results, a priority queue with a maximum of $t = 512$ entries has been implemented. Thus, the queue holds the 512 best SNP pairs with the highest ρ_{diff}-values while rejecting all lower scored pairs.

Since it is not resource efficient to implement a comparator between each storage cell, the priority queue has been implemented in block RAM with the drawback that inserting an item is not possible in constant time. Anyway, it is likely that two results may have to be inserted with a distance of only three clock cycles, resulting from the calculation time for ρ_{ctrl}. (Note that ρ_{case} has already been calculated and buffered before such that each finished calculation of ρ_{ctrl} triggers the calculation of ρ_{diff}).

This problem is tackled by a preliminary comparison to a small *base threshold* τ, rejecting most results in advance. All results passing this base threshold are first stored in a buffer the same size as the queue before finally being inserted into the priority queue.

The insertion process is naively implemented to run in linear time. At first, the datum to be inserted is compared to the smallest value in the queue. If it is smaller, it will be rejected if the queue is full, or directly inserted after the smallest datum if the queue is not full. In the case the datum is not smaller, the smallest value will be deleted if the queue is full, or else shifted to the subsequent cell. The datum to be inserted is then compared to the next smallest value and the cell which has become free is filled with the smaller value of this comparison. The process continues until the datum to be inserted is stored or the top of the queue is reached. Hence, the worst case runtime for this process is linear in the number of items stored in the queue.

The base threshold is initially set to $\tau = 0.1$ which has been found out to be sufficient for most problems. However, this does not ensure that the insertion buffer encounters an overflow or that at least t values pass the base threshold at all. In the first case, a flag signalizing an overflow of the buffer is provided in the FPGA status. The user may then decide if he increases the base threshold for a second run. In the second case, the lack of results might suggest a decrease of the base threshold for another run.

Data Distribution and Evaluation

In contrast to the data distribution method described in [WKG+14] the same method as in BOOST (see Sect. 5.3.2) has been applied here. However, resulting from the few values which are required for the calculation of the ρ_{diff} value, the processing elements for iLOCi are much smaller when compared to BOOST. This leads to an implementation of 123 processing elements in the PE chain running at 150 MHz. As in BOOST, two genotypes per clock cycle are streamed through the PE chain. Analogue to Eq. 5.3.35 the calculation of the minimum number of samples in the dataset for case and control group $n_{\min}$ is as follows.

$$n_{\min} = 3lg \tag{5.4.9}$$

Thus, at least only 738 samples need to be in the case and control group, indicating that the design is able to handle a WTCCC dataset with 500,000 SNPs of 2,000 cases and 3,000 controls. The runtime for such a dataset is measured with only 2 m 19 s on the RIVYERA S6-LX150 architecture with 128 Spartan6-LX150 FPGAs. The analysis of the simulated dataset with 2,000,000 SNPs and 10,000 samples took about 1 h 14 m on the same architecture. An evaluation of the runtimes can be found in Table 5.6.

The device utilization of the Spartan6 FPGA is listed in Tab. 5.7. It shows that only three quarters of the device is utilized in terms of slices, but no block RAM is available for more processing elements. Furthermore, since only 50% of the slices provide fast adder functionality, it would be unlikely to be able to implement more processing elements even if there was more block RAM available. Thus, the Spartan6 device may be regarded as full

Table 5.6. Runtime performance of iLOCi analyzing the WTCCC dataset and a simulated dataset with 2M SNPs and 10,000 samples. The CPU results are obtained from the iLOCi publication [PNI+12] and extrapolated for the simulated dataset.

Design	**Architecture**	**Runtime** WTCCC data	**Runtime** simulated data
iLOCi (FPGA)	RIVYERA S6-LX150 (128× Spartan6-LX150)	2 m 19 s	1 h 14 m
iLOCi (original)	2× Intel Xeon 2.4 GHz (8 cores)	19 h	25 d 08 h

Table 5.7. Device utilization of the iLOCi implementation on a Spartan6-LX150.

	Occupied Slices	Slice Registers	Slice LUTs	Block RAM (18k)	DSP48A1
Used	17,962	49,170	50,009	267	3
Available	23,038	184,304	92,152	268	180
Utilization	77%	26%	54%	100%	1%

and optimally utilized.

Regarding quality, no differences in the results of the original iLOCi algorithm compared to the FPGA version can be found. For the WTCCC datasets, leaving the default base threshold of $\tau = 0.1$ did not cause a buffer overflow or the lack of results. The interpretation of the particular results found during the tests is not part of this thesis.

5.5 Detecting Third-order SNP Interactions with Mutual Information

From the previous sections it can be seen that the detection of pairwise SNP interactions is yet feasible but still computationally challenging. However, recent research results suggest that higher order epistasis can still provide significant findings [CHW+13], but due to the high computational complex-

ity only very few tools exist, such as the method by Guo, et. al. [GMY+14]. The main computational task when searching for SNP interactions in case-control studies arises from the large number of tests that have to be performed. For detecting third-order interactions in an exhaustive search, $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$ statistical tests have to be performed with n denoting the number of SNPs. Clearly, this implies cubic runtime, and it can easily be seen that third-order interaction analyses on a whole WTCCC dataset with 500,000 SNPs are not viable as more than 2.08×10^{16} statistical tests would have to be computed. In fact, some authors assume that exhaustive search for interaction on all the third-order combinations is only feasible for small datasets with hundreds of SNPs [LJL+14].

Thus, most approaches discard a large number of triples during the process which might be insignificant by some probability. For instance, BEAM [ZL07] and its extension epi-MODE [TWJ+09] use Markov Chain Monte Carlo (MCMC) to calculate the probability of a SNP being part of a combination associated to the disease. Another approach consists of stepwise algorithms that only analyze those combinations in a subset of SNPs that are selected at the beginning [FHW+12]. Non-exhaustive approaches based on the clustering of relatively frequent items, such as EDCF [XLJ12] or the method by Leem, et. al. [LJL+14], and on machine learning techniques, such as MegaSNPHunter [WYY+09] and SNPRuler [WYY+10b], are also becoming popular. Unfortunately, these non-exhaustive tools may discard significant SNP-triples since they create a bias on those SNPs which already reveal some significance either alone or as a SNP pair. Despite being highly time-consuming, there also exist exhaustive-search strategies that analyze all the possible triples. Some examples are the Combinatorial Partitioning Method (CPM) [NKF+01], the Restriction Partition Method (RPM) [Cul07] and the Multifactor Dimensionality Reduction (MDR) method [RHR+01] (and its extensions MB-MDR [CCD+11; LJG+13] or RMDR [GAA+11]). However, their use is limited to small datasets due to their high execution times.

In contrast, this section shows that an exhaustive third-order SNP interaction analysis can be feasible for datasets containing a few thousand SNPs with the help of FPGA technology. Based on third-order contingency tables (see Fig. 5.2 in Sect. 5.2.1) and the mutual information measurement

(see Sect. 5.2.2), the exhaustive analysis of a dataset with 10,000 SNPs and 5,000 samples took only 1 h 08 m on the Kintex7-325T FPGA of the KC705 development board.

5.5.1 Third-order MI Measurement on FPGAs

Overview

The implementation presented here is kept very similar to the one presented for BOOST (see Sect. 5.3.2) or iLOCi (see Sect. 5.4.2). Processing elements organized in a systolic chain create third-order contingency tables from a stream of genotypes. These tables are provided to the unit calculating the mutual information via a specific transfer system. The results are compared to a threshold τ and those exceeding the threshold are reported to the host. Alternatively, storing the t-best MI values similar to iLOCi would also be applicable.

Consequently, in order to handle third-order contingency tables, some modifications had to be applied to the design when compared to the pairwise tests in BOOST or iLOCi. The concept of using multiple PE chains from the Kintex7 implementation of BOOST has also been applied here. An overview of the complete structure of the design can be seen in Fig. 5.10.

Generating Third-Order Contingency Tables

Third-order contingency tables require the storage of $3 \times 3 \times 3 = 27$ values for cases and controls each, as opposed to the 9-value tables for pairwise analysis. Additionally, each processing element has to store a SNP pair instead of a single SNP in the local RAM in order to count genotype triples created from the stored pair and the currently streamed SNP data. This is solved by implementing an additional SNP buffer in each PE.

Furthermore, the streaming unit for genotypes has to be modified to ensure that every possible combination of three SNPs of the input dataset will be analyzed. Thus, it keeps track of which SNP pairs have already been stored in the PEs as the basis to form a SNP triple. This is achieved by streaming the genotypes of the SNPs in two nested loops. The first one controlling the stored SNP in the first buffer of each PE, the second one

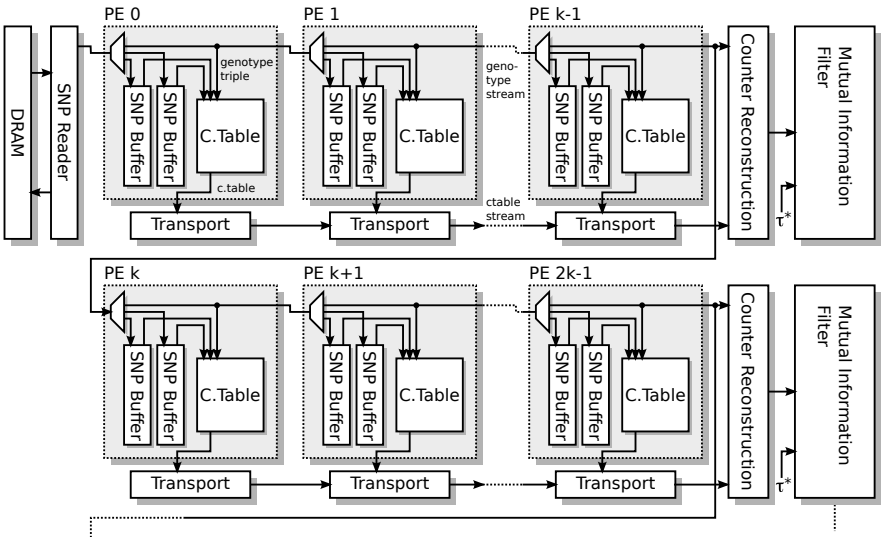


Figure 5.10. Overview of the processing element chain structure for third-order SNP interactions with subsequent counter reconstruction and mutual information filter.

controlling the stored SNP in the second buffer. Two reset signals in the stream indicate whether to reset only the second buffer or both buffers.

The filling of the buffers is performed similar to the pairwise analysis. On an empty chain, the first SNP arriving at each PE is stored in the first buffer and not streamed to the next PE, while the second SNP arriving is stored in the second buffer and also provided to the next PE which stores it in its first buffer. Then, the third SNP can be used to form a SNP triple by the first PE. The second PE stores it in its second buffer while for the third PE it is the first SNP for the first buffer etc. The process continues until all SNPs were streamed. Then, a reset only for the second buffer is provided. The next iteration therefore fills only the second buffer of each PE, one after the other. With l denoting the length of the PE chain, this process continues until all SNP pairs with one element from the first l SNPs have been stored in the buffers of the PEs. Following that, the next iteration of the outer of the two nested loops begins, starting with a reset of both buffers cleaning the

complete chain. The process continues as from the beginning, but leaving out the first l SNPs. The process has finished if there are less than three SNPs available for the outer loop since it is impossible to form another SNP triple then. An example with 6 SNPs and 3 PEs is presented in Fig. 5.11 showing a graphical representation of the created SNP triples in each step.

Counter Reconstruction

Since 27 counters for the entries of the contingency table in each processing element require a lot of resources, only 20 selected counters are implemented while the missing 7 will be restored at the end of each PE chain. The only information which is required in advance is the total number of cases and controls (i.e. n_0 and n_1), and the number of genotypes with homozygous wild type and heterozygous type in each SNP of the corresponding triple, i.e. $n_{0\bullet\bullet l}$, $n_{1\bullet\bullet l}$, $n_{\bullet 0\bullet l}$, $n_{\bullet 1\bullet l}$, $n_{\bullet\bullet 0 l}$, and $n_{\bullet\bullet 1 l}$, whereby l denotes either case or control group.

This information can easily be gathered by counting the corresponding types at the end of the genotype stream in each chain. The number of cases and controls is given directly with the dataset and has already been used for reconstruction of a single counter in the previous implementations for the pairwise tests. Here, it is only required to determine the value of $n_{2\bullet\bullet l}$:

$$n_{2\bullet\bullet l} = n_l - n_{0\bullet\bullet l} - n_{1\bullet\bullet l} \tag{5.5.1}$$

Now, the counters n_{000l}, n_{100l}, n_{110l}, n_{111l}, n_{211l}, n_{221l}, and n_{222l} are reconstructed by implementing the following equations. Note that the reconstruction is done in a pipeline one value after the other, since their reconstruction generally depends on restored values of the previous steps.

$$n_{000l} = n_{0\bullet\bullet l} - (n_{001l} + n_{002l} + n_{010l} + n_{011l} + n_{012l} + n_{020l} + n_{021l} + n_{022l}) \tag{5.5.2}$$

$$n_{100l} = n_{\bullet 0\bullet l} - (n_{000l} + n_{001l} + n_{002l} + n_{101l} + n_{102l} + n_{200l} + n_{201l} + n_{202l}) \tag{5.5.3}$$

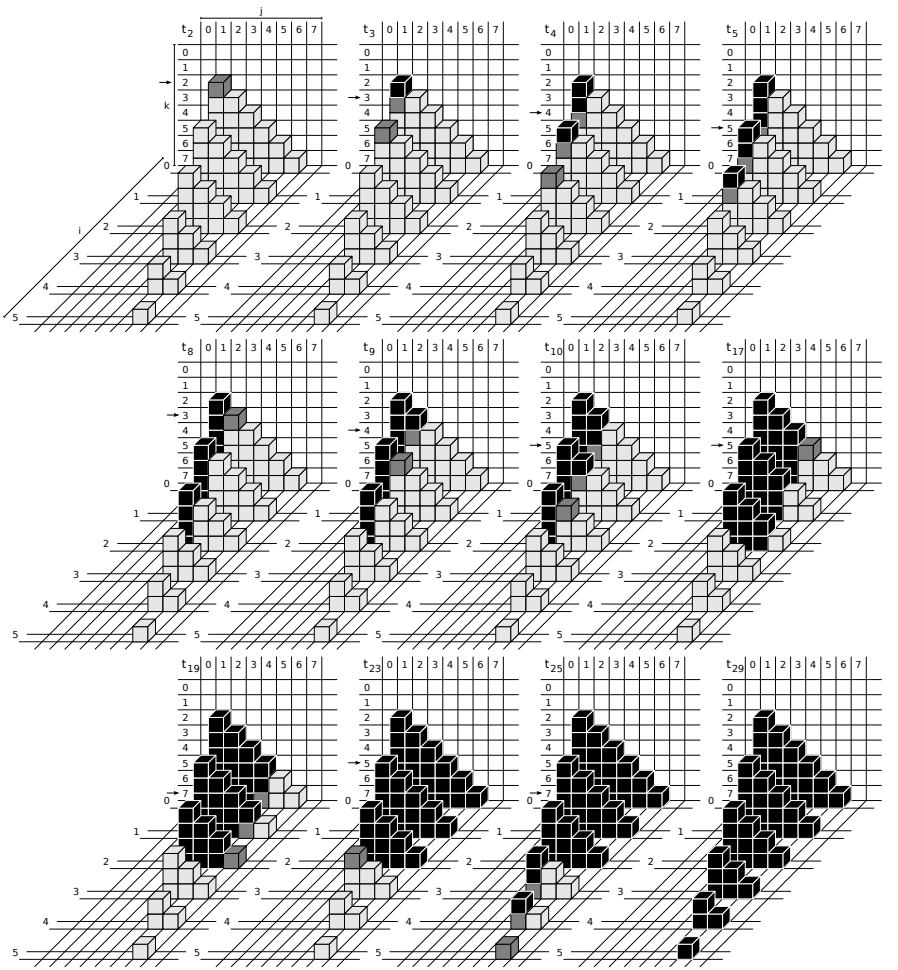


Figure 5.11. Sequence of creating all SNP triples from an example dataset of eight SNPs with a chain of three PEs in 30 time steps (whereof 12 are exemplarily shown). Light gray cubes indicate SNP triples to be processed while black cubes indicate already processed triples. Dark gray cubes are currently being processed. The currently streamed SNP is indicated by an arrow on the *k*-axis.

$$n_{110l} = n_{\bullet\bullet 0l} - (n_{000l} + n_{010l} + n_{020l} + n_{100l} + n_{120l} + n_{200l} + n_{210l} + n_{220l}) \tag{5.5.4}$$

$$n_{111l} = n_{1\bullet\bullet l} - (n_{100l} + n_{101l} + n_{102l} + n_{110l} + n_{112l} + n_{120l} + n_{121l} + n_{122l}) \tag{5.5.5}$$

$$n_{211l} = n_{\bullet 1\bullet l} - (n_{010l} + n_{011l} + n_{012l} + n_{110l} + n_{111l} + n_{112l} + n_{210l} + n_{212l}) \tag{5.5.6}$$

$$n_{221l} = n_{\bullet\bullet 1l} - (n_{001l} + n_{011l} + n_{021l} + n_{101l} + n_{111l} + n_{121l} + n_{201l} + n_{211l}) \tag{5.5.7}$$

$$n_{222l} = n_{2\bullet\bullet l} - (n_{200l} + n_{201l} + n_{202l} + n_{210l} + n_{211l} + n_{212l} + n_{220l} + n_{221l}) \tag{5.5.8}$$

Calculation of Mutual Information

According to Eq. 5.2.7 in Sect. 5.2.2 the mutual information of a third-order contingency table can be calculated as

$$I(X_1, X_2, X_3; Y) = H(X_1, X_2, X_3) + H(Y) - H(X_1, X_2, X_3, Y) \tag{5.5.9}$$

whereby $X_{\{1,2,3\}}$ denote the random variables for each SNP of the triple and Y denotes the random variable for the sample type being either case or control. Using the information from the contingency table, the entropies in this equation follow the following conditions. n_{ijkl} denote the entries of the contingency tables, n_0, n_1 and n represent the number of controls, cases and the total number of samples respectively.

$$H(X_1, X_2, X_3) = -\sum_{ijk} (\pi_{ijk0} + \pi_{ijk1}) \log(\pi_{ijk0} + \pi_{ijk1}) \tag{5.5.10}$$

$$= -\sum_{ijk} \frac{n_{ijk0} + n_{ijk1}}{n} \log \frac{n_{ijk0} + n_{ijk1}}{n} \tag{5.5.11}$$

$$H(Y) = -\frac{n_0}{n} \log \frac{n_0}{n} - \frac{n_1}{n} \log \frac{n_1}{n} \tag{5.5.12}$$

$$H(X_1, X_2, X_3, Y) = -\sum_{ijkl} \pi_{ijk} \log \pi_{ijk} \tag{5.5.13}$$

$$= -\sum_{ijkl} \frac{n_{ijk}}{n} \log \frac{n_{ijk}}{n} \tag{5.5.14}$$

In order to filter only significant results, a user defined threshold τ is applied to the information measure. Only SNP triples which pass this threshold will be reported. Let $H = H(X_1, X_2, X_3)$ denote the joint entropy of $X_{\{1,2,3\}}$ and $H' = H(X_1, X_2, X_3, Y)$ the joint entropy of all random variables. Because n_0, n_1, and n are constant in each dataset, it is sufficient to implement the calculation of the value $n(H - H')$ in order to save resources.

$$n(H - H') = n \sum_{ijk} \Bigg[\frac{n_{ijk0}}{n} \log \frac{n_{ijk0}}{n} + \frac{n_{ijk1}}{n} \log \frac{n_{ijk1}}{n} - \frac{n_{ijk0} + n_{ijk1}}{n} \log \frac{n_{ijk0} + n_{ijk1}}{n} \Bigg] \tag{5.5.15}$$

$$= \sum_{ijk} \Bigg[n_{ijk0} \Big(\log n_{ijk0} - \log n \Big) + n_{ijk1} \Big(\log n_{ijk1} - \log n \Big) - (n_{ijk0} + n_{ijk1}) \Big(\log(n_{ijk0} + n_{ijk1}) - \log n \Big) \Bigg] \tag{5.5.16}$$

$$= \sum_{ijk} \Bigg[n_{ijk0} \log n_{ijk0} + n_{ijk1} \log n_{ijk1} - (n_{ijk0} + n_{ijk1}) \log(n_{ijk0} + n_{ijk1}) \Bigg] \tag{5.5.17}$$

Then, a modified threshold τ^* is defined and pre-calculated on the host:

$$\tau^* = n(\tau - H(Y)) \tag{5.5.18}$$

$$= n \left(\tau + \frac{n_0}{n} \log \frac{n_0}{n} + \frac{n_1}{n} \log \frac{n_1}{n} \right) \tag{5.5.19}$$

It follows:

$$n(H - H') \geqslant \tau^* \tag{5.5.20}$$
$$\Leftrightarrow \quad n(H - H') \geqslant n(\tau - H(Y)) \tag{5.5.21}$$
$$\Leftrightarrow \quad H - H' + H(Y) \geqslant \tau \tag{5.5.22}$$
$$\Leftrightarrow \quad I(X_1, X_2, X_3; Y) \geqslant \tau \tag{5.5.23}$$

Following this observation, the calculation of $n(H - H')$ with a subsequent comparison to the modified threshold τ^* is the same as applying the threshold τ directly to the information measure. Alternatively, storing the t-best results as in the iLOCi implementation (see Sect. 5.4.2) would also be possible instead of applying a threshold to results. This might be useful because the expected information value of a significant SNP is generally not known in advance. Thus, the wrong decision for a threshold might turn out in having too many or no results at all. However, for demonstration purposes, the implementation has been left with the threshold comparison yet.

Regarding the implementation of Eq. 5.5.17 it can be seen that the three logarithm calculations can be processed independently. Thus, three logarithm units were implemented in parallel, each with a subsequent multiplier. For one logarithm unit the two inputs n_{ijk0} and n_{ijk1} have to be added up in advance. The three concurrently calculated values are then combined via an addition and subtraction step before being accumulated over all 27 possible combinations of i, j, and k. The result $n(H - H')$ is then compared to the modified threshold τ^* which decides whether the underlying SNP triple will be reported or not.

All operations are designed as a large pipeline. This allows to provide an input every clock cycle although the calculation itself takes several cycles. Therefore, the time required to calculate $n(H - H')$ from one case and one control contingency table is 27 clock cycles plus latency. The scheme of the unit calculating $n(H - H')$ for the Mutual Information measure is depicted in Fig. 5.12.

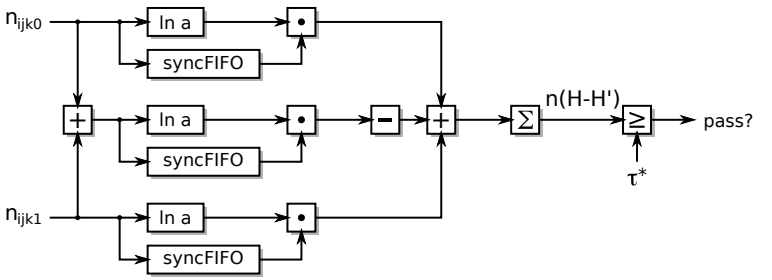


Figure 5.12. Scheme for calculating $n(H - H')$ for the Mutual Information with downstream threshold comparison.

Evaluation

The design has been implemented on the Kintex7-325T FPGA on the KC705 development board. Each entry of the contingency table is implemented as a 14 bit integer. The calculation of the logarithm is done in single precision floating point format. Afterwards, the precision is set to 6.26 fixed-point format which is also the precision of the final result. The logarithm and multiplication units are generated using the Vivado IP Catalog with full DSP support.

As in the Kintex7 implementation for BOOST (see Sect. 5.3.2) eight genotypes are streamed via the genotype bus in each clock cycle. Furthermore, since the calculation of the information value takes 27 clock cycles and requires a complete table for cases and controls, the table transport system could be optimized by sending two values in one cycle. This leads to only 14 clock cycles to fetch a case or control table from a processing element. Thus, the $n_{\min}$ calculation (analogue to Eq. 5.3.35 in Sect. 5.3.2) in this case evaluates to:

$$n_{\min} = 14lg \tag{5.5.24}$$

In accordance with this observation, the maximum number of processing elements in a PE chain unfortunately evaluates to only $l = 17$ to support a WTCCC dataset (or at least a subset of this) with about 2,000 cases and

Table 5.8. Device utilization of the third-order Mutual Information implementation on a Kintex7-325T.

	Slice Registers	Slice LUTs	Block RAM (36k)	DSP48E1
Used	163,564	185,137	290	271
Available	407,600	203,800	445	840
Utilization	40%	91%	65%	32%

Table 5.9. Performance analysis of exhaustive third-order MI analysis of datasets with 5,000, 10,000, and 20,000 SNPs and 5,000 samples as well as the complete WTCCC dataset with about 500,000 SNPs. Runtimes marked with an asterisk (*) are extrapolated.

Runtimes		**Architecture**		
		Virtex7-690T	Kintex7-325T	Intel Core i7 (6 cores @ 3.2 GHz)
Dataset	5k SNPs	4 m 16 s	8 m 30 s	1 d 01 h 48 m
	10k SNPs	34 m 02 s	1 h 08 m	8 d 14 h 26 m*
	20k SNPs	4 h 33 m	9 h 05 m	68 d 19 h 43 m*
	500k SNPs	8 y 35 d*	16 y 70 d*	2,947 years*

3,000 controls. Thus, 6 chains with 17 PEs each have been implemented on the Kintex7 to fully utilize the device. The clock frequency of the design is 250 MHz resulting in a runtime of only 1 h 08 m for a subset of the WTCCC dataset with 10,000 SNPs and 5,000 samples. The resource utilization of the device is stated in Tab. 5.8. Furthermore, the Power Report indicates the energy consumption of the device with only 10.5 W.

For comparison purposes a multi-threaded CPU implementation (using *pthreads*) performing an exhaustive third-order analysis with the mutual information measure has been implemented and executed on an Intel Core i7 Sandy Bridge hexa-core CPU with 3.2 GHz. The runtime for an even smaller dataset with only 5,000 SNPs and 5,000 samples was measured with 1 d 01 h 48 m. Thus, the runtime for the 10,000 SNPs dataset has been extrapolated to 8 d 14 h 26 m, leading to a speedup of about 182 for the

FPGA. This result shows a clear advantage of the FPGA implementation over the CPU implementation.

The runtime has also been measured and estimated for a dataset with 20,000 SNPs and 5,000 samples. For a complete WTCCC dataset the runtime has been estimated as well, resulting in more than 16 years for the analysis. This demonstrates the cubic scaling of the problem and shows that it is still unfeasible to analyze a complete WTCCC dataset with this method on a single FPGA device. All runtime results can be found in Tab. 5.9. The table also contains the runtime estimations of the same design synthesized for a Virtex7-690T device with 12 PE chains each containing 17 PEs, which results in exactly two times the speed of the presented implementation.

Chapter 6

Genotype Imputation

This chapter contains previously unpublished work.

6.1 Motivation

In order to explain the role of DNA sequence variants in the expression of diseases, clinical studies determine the genotype of a set of individuals commonly via microarrays, such as Affymetrix GeneChips [Aff09], although the determined genotypes ignore the person's *diplotype*, the unique content of the two homologous chromosomes inherited by the individual. As a short example, let G be an individual's genotype vector

$$G = \texttt{01012}.$$

With this information alone, the *phase* of the haplotypes is unknown, i.e. it can not directly be followed if the two haplotype vectors for the sample were

$$\begin{aligned} H_1 &= \texttt{00001} \\ \text{and} \quad H_2 &= \texttt{01011} \end{aligned}$$

or

$$\begin{aligned} H_1 &= \texttt{01001} \\ \text{and} \quad H_2 &= \texttt{00011}. \end{aligned}$$

However, knowledge of personal haplotypes is an important prerequisite

to support personalized medicine. Many studies have linked specific haplotypes to drug response, clinical outcomes in transplantations [PMG+07] and to susceptibility or resistance to disease, e.g. the association of human leukocyte antigen (HLA) haplotypes with autoimmune conditions such as multiple sclerosis [JSF72].

The lack of phased genomes is primarily due the lack of experimental approaches for obtaining phase information. Thus, at present, the preferred method is to estimate the personal haplotypes as part of a computational process called *genotype imputation*, also referred to as *SNP imputation*. It describes an approach of *haplotype phasing* and predicting untyped alleles that are not directly assayed in a genetic sample. The key idea is to phase the genotypes determined from samples in a study and then use the estimated haplotypes in combination with reference sets of known haplotypes to infer unobserved genotypes and their phase in the study individuals.

Popular tools handling these steps in one application are BEAGLE [BB07], PHASE [SSD01; SD03; SS05], fastPHASE [SS06] and MaCH [LWD+10]. In order to separate the computationally intensive *phasing* process from the data-intensive *imputation* process to introduce flexibility and to reduce the total computational runtime, a split-up into separate tools was proposed recently [HFS+12]. Delaneau et al. presented SHAPEIT [DMZ12] to do the phasing step alone. This chapter concentrates on the successor SHAPEIT2 [DZM13] which allows phasing across whole chromosomes with high accuracy and scales linearly with the number of samples to be phased. For the subsequent imputation process Minimac2 [FAH15], Minimac3 [Abe] and IMPUTEv2 [HDM09] are the most commonly applied software tools able to use the phased haplotypes from the first step. Though IMPUTEv2 is able to use the phasing information of SHAPEIT2, the primary intention of this tool is to use a *Markov chain Monte Carlo (MCMC)* scheme that switches between phasing and imputation in each iteration and can therefore be used to perform phasing and imputation together. The original IMPUTE [MHM+07] performs genotype imputation solely based on the integration over the unknown phase of the genotypes in a study, i.e. phasing is not required here for the cost of imputation quality.

Imputation quality heavily relies on suitable whole-genome reference panels of previously accurately phased individuals [HMS11]. In order to im-

prove imputation accuracy, the *Haplotype Reference Consortium (HRC)* started creating reference panels of human haplotypes by combining together sequencing data from multiple ongoing whole genome sequencing projects. So far, the HRC collected information on more than 38,000 sequenced whole genomes, aggregated over 20 studies of predominantly European ancestry, to create reference panels of 64,976 haplotypes at 39,235,157 SNPs. In the future, the HRC envisages the reference panel further increasing in size and consisting of samples from a more diverse set of world-wide populations [HRC].

However, phasing as well as imputation are computationally very expensive processes, and with the on-going growth of the reference panel the computational demands increase likewise. Current large-scale genome-wide input data sets with larger haplotype reference panels lead to runtimes of several days, even on well-equipped CPU clusters. This chapter addresses this issue by applying FPGA hardware for the haplotype phasing process according to SHAPEIT2 to reduce the runtime of this process to only a few hours. Sect. 6.2 describes the mathematical methods of a Hidden Markov model and the forward-backward procedure necessary to understand the algorithm description of the phasing process in SHAPEIT2 described in Sect. 6.3.1. It follows the FPGA implementation including evaluation in Sect. 6.3.2.

The imputation process commonly follows similar rules as in the phasing process, i.e. it uses a Hidden Markov model and the forward-backward procedure to estimate unknown genotype information. It is exemplarily presented in Sect. 6.4 based on the IMPUTE tool. Although IMPUTE does not require haplotype phasing for imputation, its underlying mathematical methods form the basis of all before mentioned imputation tools. Thus, IMPUTE has been chosen to serve as an example to explain the imputation process.

6.2 Mathematical Methods

The phasing algorithm of SHAPEIT2 is based on the mathematical definitions of a *Hidden Markov Model (HMM)* as well as the use of the *forward-*

backward procedure [Rab89] for sampling the haplotype estimations in several iterations based upon the HMM. These mathematical expressions will be described in the following.

6.2.1 Hidden Markov Model

A *Hidden Markov model (HMM)* is a stochastic model where a system is described as a *Markov chain* with unobserved, i.e. hidden states. Modeling as a Markov chain means that the system randomly turns from one state into another in one time step whereby the probability of this transition is only dependent on the current state and not on the states already visited before. This property is called the *Markov property* and can be formally defined as

$$P(Z_m = z_m | Z_{m-1} = z_{m-1}) = P(Z_m = z_m | Z_{m-1} = z_{m-1}, Z_{m-2} = z_{m-2}, \ldots, Z_0 = z_0) \quad (6.2.1)$$

whereby $\{Z_m\}_{m \in \mathbb{N}}$ is a random process and z_m the state at time step m. If this property is satisfied, $\{Z_m\}_{m \in \mathbb{N}}$ is also called a *Markov process*. The probability $P(Z_m = z_m | Z_{m-1} = z_{m-1})$ is called *transition probability* and indicates the probability to change to state z_m when the process is in state z_{m-1}.

For an HMM the direct observation of $\{Z_m\}_{m \in \mathbb{N}}$ is generally not possible. Observations can only be made of a second random process $\{X_m\}_{m \in \mathbb{N}}$ whereby it is assumed that it is only dependent on the unobserved state, i.e.

$$P(X_m = x_m | Z_m = z_m) = P(X_m = x_m | Z_m = z_m, \ldots, Z_0 = z_0, X_{m-1} = x_{m-1}, \ldots, X_0 = x_0) \quad (6.2.2)$$

The observations are also called *emissions* and the probability $P(X_m = x_m | Z_m = z_m)$ is referred to as the *emission probability* to emit x_m when the process is in state z_m.

An HMM is called *stationary* if the transition and emission probabilities are constant for each time step.

Formally, an HMM can then be defined as a 5-tuple $\lambda = (S, O, T, E, \pi)$:

$\triangleright$ $S = \{s_i\}$: set of possible (hidden) states, i.e. $z_m \in S$

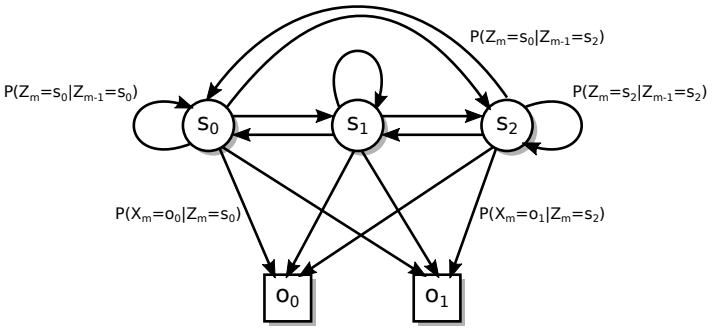


Figure 6.1. Example HMM with three states and two possible observations. The arrows indicate the transition and emission probabilities. Not all probabilities are labeled.

- $O = \{o_i\}$: set of possible observations, i.e. $x_m \in O$
- $T \in \mathbb{R}^{|S|\times|S|}$: transition probabilities, i.e. $P(Z_m = s_i | Z_{m-1} = s_j) \in T$
- $E \in \mathbb{R}^{|O|\times|S|}$: emission probabilities, i.e. $P(X_m = o_i | Z_m = s_j) \in E$
- π: start distribution, i.e. $\pi_i = P(Z_0 = s_i)$ is the probability that s_i is the first state.

Fig. 6.1 illustrates an example HMM with three states and two possible observations with arrows indicating the probabilities. Furthermore, Fig. 6.2 shows the timely development of an HMM where the arrows indicate the conditional dependencies.

6.2.2 Forward-Backward Procedure

Let $\lambda = (S, O, T, E, \pi)$ be an HMM according to the definitions above. In a general HMM it is difficult to calculate the marginal probability to be in a state $Z_m = s_i \in S$ at time m if a certain emission sequence $x_{0:L-1} = x_0 x_1 x_2 \dots x_{L-1}$ with $x_i \in O$ is observed [Rab89]. This *posterior* probability is denoted as $P(Z_m = s_i | x_{0:L-1}, \lambda)$.

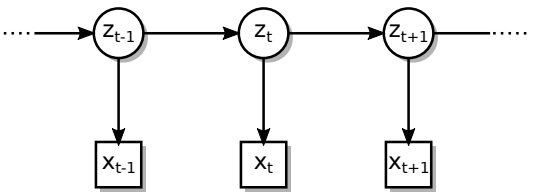


Figure 6.2. Timely development of an HMM. Arrows indicate the conditional dependencies of the observations to the current state and the conditional dependencies of the current state to the previous state.

In order to calculate these posterior probabilities, two types of variables are defined:

- The *forward variables* define the probability of being in state $s_i \in S$ at time $m \in \{0, \ldots, L-1\}$ and observing the first part of the emission sequence $x_{0:m}$, i.e.
$$a_m(i) = P(Z_m = s_i, x_{0:m}|\lambda) \tag{6.2.3}$$
- The *backward variables* define the probability of observing the remaining sequence given the state $s_i \in S$ at time $m \in \{0, \ldots, L-1\}$ as starting point, i.e.
$$b_m(i) = P(x_{m+1:L-1}|Z_m = s_i, \lambda) \tag{6.2.4}$$

Now, the calculation of the posterior probabilities evaluates to [Rab89]:

$$P(Z_m = s_i|x_{0:L-1}, \lambda) = \frac{a_m(i)b_m(i)}{P(x_{0:L-1}|\lambda)} \tag{6.2.5}$$
$$= \frac{a_m(i)b_m(i)}{\sum_j a_m(j)b_m(j)} \tag{6.2.6}$$

Since $P(x_{0:L-1}|\lambda)$ is a constant for a given HMM and observation, it is sufficient for many problems to highlight the proportionality only:

$$P(Z_m = s_i|x_{0:L-1}, \lambda) \propto a_m(i)b_m(i) \tag{6.2.7}$$

The forward and backward variables can be calculated recursively. Firstly, the initialization of the forward variables directly takes the start probability distribution of λ into account and denotes the probability of s_i being the first state while emitting the first symbol $x_0 = o_j$.

$$a_0(i) = P(X_0 = o_j | Z_0 = s_i) P(Z_0 = s_i) \tag{6.2.8}$$
$$= E_{ji} \pi_i \tag{6.2.9}$$

Let $x_{m+1} = o_j$ denote the observation of timestamp $m+1$. Then, all other forward variables are recursively defined as follows.

$$a_{m+1}(i) = P(X_{m+1} = o_j | Z_{m+1} = s_i) \sum_k a_m(k) P(Z_{m+1} = s_i | Z_m = s_k) \tag{6.2.10}$$
$$= E_{ji} \sum_k a_m(k) T_{ki} \tag{6.2.11}$$

Clearly, the evaluation of the forward variables elapses in a forward direction. Secondly, the backwards variables are evaluated similarly in a backwards direction and therefore, the initialization begins with the last time step $L-1$. It can be interpreted as the probability to further emit the empty sequence when the last time step has been reached, which is effectively constant one.

$$b_{L-1}(i) = 1 \tag{6.2.12}$$

All other backward variables are defined as:

$$b_m(i) = \sum_k b_{m+1}(k) P(X_{m+1} = o_j | Z_{m+1} = s_k) P(Z_{m+1} = s_k | Z_m = s_i) \tag{6.2.13}$$
$$= \sum_k b_{m+1}(k) E_{jk} T_{ik} \tag{6.2.14}$$

Since the calculation of the posterior marginal probabilities according to Eq. 6.2.7 requires the complete evaluation of both, forward and backward variables, and the evaluation has to be done in both directions, the process is called the *forward-backward procedure*. Let L denote the number of total time steps and therefore the length of the observation sequence, and N denote the number of states. Then, the runtime complexity of one forward-

backward computation is clearly in $\mathcal{O}(LN)$. This implies the storage of either the complete forward or backward variables in memory. Thus, the memory complexity is in $\mathcal{O}(LN)$ as well.

Analog to the computation of the state probabilities it is also possible to calculate the posterior marginal and transitional probabilities of the emissions using the forward-backward procedure as it is done in SHAPEIT2 (see next section).

6.3 Phasing with SHAPEIT2

The phasing process in genotype imputation determines the phase of each genotype in a study. Generally, the two haplotype vectors which explain the biological origin of a genotype vector, i.e. the diplotype information of the alleles on the two homologous chromosomes of the diploid individual, are unknown. In order to resolve this missing information, the SHAPEIT2 tool takes a set of input genotype vectors, usually a complete study, and estimates a pair of haplotype vectors for each genotype vector. The estimations are based on all other input genotype vectors alone.

SHAPEIT2 iteratively generates a Hidden Markov Model for each genotype vector, based on a subset of all current haplotype estimations from the previous iteration, presenting the states. The possible outcomes of haplotype pairs present the emissions. With the help of the forward backward procedure the posterior transitional probabilities for the emissions can be calculated, which form the basis to sample the haplotypes for the new estimation in this iteration. SHAPEIT2 uses several iterations including burn-in iterations to find a better starting point where the results are discarded, and main iteration from which the results are averaged to form the final estimation. Since each haplotype is sampled with a random number weighted by the transitional probabilities between sites, and each probability only depends on the probability of the previous site, the probabilities form a classical *Markov chain* and the whole sampling process can be found as a *Gibb's sampling* approach, a special case of the more general *Markov chain Monte Carlo* method. This method converges to the joint distribution of all haplotypes and is therefore suitable to find the most probable haplotype

pairs (diplotypes) for a genotype. The algorithm will be explained in detail in the following section.

SHAPEIT2 only determines the phase of input genotypes and may fill single unknown markers of a few samples in a study based on all other genotypes of the study alone. In order to infer larger gaps of unobserved genotypes and to improve result quality, reference sets of known haplotypes have to be taken into account. The phase information from this process is then used to impute the missing genotype information in the imputation process, which is shortly introduced in Sect. 6.4.

6.3.1 SHAPEIT2 Algorithm

Overview

The phasing process in SHAPEIT2 consists of several steps which are presented in a short overview here. A detailed explanation follows in the later paragraphs.

1. At first, SHAPEIT2 divides the input data into *windows* which can be processed independently instead of phasing whole chromosomes at once. The default window size is 2Mb, but can be configured by the user. The size is related to the bases in the genome rather than the number of markers. According to the distribution of markers in the genome, two windows of the same size may contain a significantly different number of markers. Furthermore, windows may contain overlaps, but can be phased independently in general. The next steps therefore refer to a single window rather than the complete dataset.

2. The first step of processing a window is to generate a *compatible* pair of haplotype vectors for each genotype vector of the input dataset. These pairs present the first haplotype estimations and are randomly generated.

3. Any genotype vector is divided into *segments* each containing at most $B = 3$ heterozygous genotypes. Thus, the number of *consistent* haplotype vectors for each segment is $2^B = 8$.

4. For each genotype vector a set of K haplotype vectors is chosen according to similarity. A Hidden Markov model is constructed with K

states presented by the chosen haplotype vectors and 8 possible emissions presented by the indices of a consistent haplotype vector for each segment.

5. Since the direct observation of the HMM is not the segment indices but the underlying genotype vector, the goal is to calculate the posterior transitional probabilities for each site which index might be taken to explain the current genotype given the previous index and the chosen set of K haplotype vectors. These probabilities are calculated using the forward-backward procedure explained before.

6. From the calculated probabilities a new compatible pair of haplotype estimates is sampled by the Gibb's sampler for each genotype which replaces the current estimates. A restriction is that the indices in a segment are not subject to change, i.e. an index and thus, the corresponding haplotypes for this index are sampled segment-wise. A change of the index is therefore only allowed at a segment border.

7. Steps 4 to 6 are repeated in a number of *burn-in iterations*. As in classical Gibb's sampling, all results but those from the last iteration are discarded. These iterations are performed only to find a better starting point for the main iterations than to use a set of haplotypes completely chosen at random for the first estimations. The default number of burn-in iterations is 7.

8. A number of *state pruning and segment merging iterations* follow. Basically, steps 4 to 6 are repeated again but after each iteration the underlying model is simplified. Unlikely states are pruned away and neighboring segments where the index transitions are almost unambiguous are merged together. The default number of iterations is 8. However, these iterations will not be discussed any further, since, according to the authors, they were introduced only to accelerate the computation process on a PC and do not improve the result quality, such that they can be replaced by extra main iterations. They are not included in the FPGA implementation anyway.

9. Steps 4 to 6 are again repeated in a number of *main iterations*. The default

number is 20. Here, the distribution of the estimates is expected to converge to the joint distribution of all haplotypes (following a Gibb's sampling approach). Thus, the results of each iteration are kept to average the final haplotype estimate across all main iterations.

In the following, all necessary steps to sample compatible haplotype estimates from the input data are explained in detail.

Notations

Following notations will be used throughout this section.

N: no. of unrelated individuals in the input dataset.

L: no. of markers/SNPs (biallelic sites) in the current window.

K: no. of used haplotype vectors for reference, i.e. the number of states in the HMM.

$G = \{g_0, \ldots, g_{L-1}\}$: current genotype vector of the input data set, $g_m \in \{0, 1, 2, 3\}$.

$G_{n:m} = \{g_n, \ldots, g_m\}$: subvector of the current genotype vector from index n to m.

$H = \{H_0, \ldots, H_{K-1}\}$: current used haplotype vectors, a subset of all $2N$ haplotype vectors.

$H_k = \{h_{k0}, \ldots, h_{k(L-1)}\}$: kth haplotype vector, $h_{km} \in \{0, 1\}$.

(G_1, G_2): pair of two haplotype vectors compatible with G.

B: no. of heterozygous genotypes in each segment, usually $B = 3$ constant.

C: no. of segments resulting from division of G in segments containing B heterozygous genotypes.

$S_m \in \{0, \ldots, C-1\}$: index of the segment containing site m with $m \in \{0, \ldots, L-1\}$.

$\{s\} = \{m | S_m = s\}$: all sites in segment s.

A_{mi}: the allele of the consistent haplotype vector with index i in segment S_m at site m, $A_{mi} \in \{0, 1\}$.

$X = \{X_0, \ldots, X_{L-1}\}$: X_m is the index of the with G consistent haplotype in segment S_m, also referred to as *label*. It presents the emission of the HMM at site m. $X_m \in \{0, \ldots, 2^B - 1\}$.

$X_{n:m} = \{X_n, \ldots, X_m\}$: subvector of X from indices n to m.

$Z = \{Z_0, \ldots, Z_{L-1}\}$: unobserved (hidden) states in the HMM for each site, $Z_m \in \{0, \ldots, K-1\}$ denotes the "copied" haplotype at site m.

$Z_{n:m} = \{Z_n, \ldots, Z_m\}$: subvector of Z from indices n to m.

$\rho_m = 4N_e r_m$: r_m is the *per generation genetic distance* between sites m and $m-1$. The distance is determined from a *genetic map* the user has to provide along with the input data. The genetic map contains the positions of each marker in the genome. N_e is the effective population size and set per default to $N_e = 15,000$.

$\lambda = \frac{\theta}{2(\theta+K)}$: a constant indicating the mutation rate, $\theta = \left(\sum_{i=1}^{K-1} \frac{1}{i}\right)^{-1}$.

Consistent and Compatible Haplotype Vectors

Let $G = g_0 \ldots g_{L-1}$ be a genotype vector of length L with $g_m \in \{0, 1, 2, 3\}$ ("0" denotes the homozygous wild type, "1" denotes the heterozygous type, "2" denotes the homozygous variant type, "3" is unknown).

Definition: A haplotype vector $H_k = h_{k0} \ldots h_{k(L-1)}$ with $h_{km} \in \{0, 1\}$ ("0" is wild type, "1" is variant type) is called *consistent* with G if:

$$\forall 0 \leqslant m < L : (g_m = 0 \Rightarrow h_{km} = 0) \wedge (g_m = 2 \Rightarrow h_{km} = 1) \tag{6.3.1}$$

In other words, the haplotype has to show the wild type on sites where the genotype is homozygous wild, or it has to show the variant type if

the genotype is homozygous variant. The haplotype is free to chose if the corresponding genotype is heterozygous or unknown.

Definition: A pair of haplotype vectors (G_1, G_2) with $G_i = h_{i0} \dots h_{i(L-1)}$ and $h_{im} \in \{0,1\}$, $i \in \{1,2\}$ is *compatible* to G if G_1 and G_2 are consistent with G and:

$$\forall 0 \leqslant m < L : g_m = 1 \Rightarrow ((h_{1m} = 0 \wedge h_{2m} = 1) \vee (h_{1m} = 1 \wedge h_{2m} = 0)) \quad (6.3.2)$$

In other words, each heterozygous site implies one wild and one variant haplotype on the same site in both haplotype vectors, i.e. when both haplotypes may present a biological explanation of the genotype.

SHAPEIT2 randomly generates a pair of compatible haplotype vectors for each input genotype vector as the first estimation in the first step of processing a window. All further estimations are sampled according to the calculated probabilities from the HMM, but are ensured to be compatible as well.

Segments

Each genotype vector is divided into segments such that each segment contains $B = 3$ heterozygous genotypes. Thus, there exist exactly $2^B = 8$ consistent haplotype vectors or $2^B/2 = 4$ compatible haplotype pairs in any segment. The division is straight forward and does not need further explanation. Figure 6.3 illustrates the segmentation of an example genotype vector.

Choosing a Subset of K Haplotype Vectors

In order to choose a subset of K haplotypes to serve as the states of the HMM, i.e. the set H, all haplotype vectors are ranked by their relation to the current genotype. K is user-definable and is set to 100 per default.

Let (G_1, G_2) be the current estimation of G. Then $D(G, H_i)$ describes a measure of the relation between the genotype vector G and a haplotype

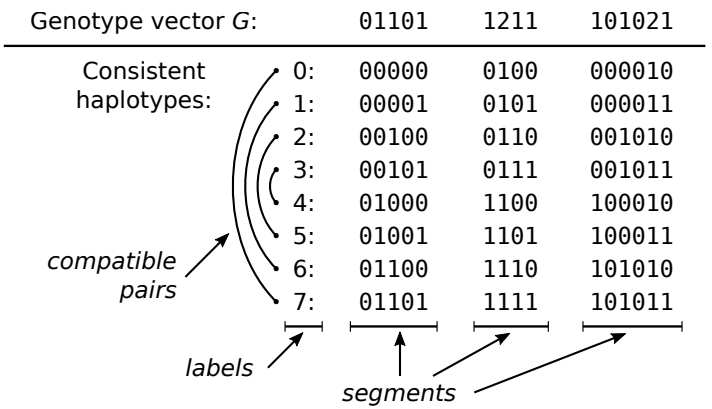


Figure 6.3. Segmentation of a genotype vector in SHAPEIT2. The labels denote the index of a consistent haplotype vector for each segment. Within each segment the compatible pairs of haplotypes are $(0,7)$, $(1,6)$, $(2,5)$, and $(3,4)$.

vector H_i with d_{Hamming} denoting the Hamming distance of two vectors:

$$D(G,H_i) = \min\left\{d_{\text{Hamming}}(G_1,H_i), d_{\text{Hamming}}(G_2,H_i)\right\} \tag{6.3.3}$$

The measurement can be interpreted as a difference between the current genotype and a haplotype. The before mentioned subset H now contains the K haplotype vectors H_i of all current haplotype estimations with the smallest value of $D(G,H_i)$. It is important to mention that H is constructed only of the current estimates of all genotypes other than G, i.e. the current estimates of G are not included in the set.

SHAPEIT2 provides a second way to select H out of the current estimates. According to the authors, the previously described method does not perform well if the input dataset includes closely related relatives. Thus, the second method preserves H from including vectors of too closely related samples. However, this method is not implemented yet in this work and is subject to future work.

Modeling the HMM

For each genotype vector in the current window SHAPEIT2 models a Hidden Markov model to calculate the marginal and transitional probabilities for the corresponding emissions. According to the notations above, the parameters of the HMM are as follows.

Time: A time step in the HMM corresponds to a site m in the genotype vector G (and of course the corresponding sites of the haplotype vectors in H). Neighboring timestamps are the same as neighboring sites. The first site starts with $m = 0$.

States: The states of the HMM are defined as the chosen subset H with K haplotype vectors. Thus the HMM consists of K hidden states.

Emissions: The possible emissions of the HMM are the labels (indices) of the consistent haplotypes in each segment, i.e. $0 \leqslant i < 2^B = 8$.

HMM transition probabilities: The transition probabilities of switching from one state to another in the HMM are defined as:

$$P(Z_m = u | Z_{m-1} = v) = \begin{cases} e^{\frac{-\rho_m}{K}} + \frac{1-e^{\frac{-\rho_m}{K}}}{K} & \text{if} \quad u = v \\ \frac{1-e^{\frac{-\rho_m}{K}}}{K} & \text{if} \quad u \neq v \end{cases} \tag{6.3.4}$$

HMM emission probabilities: The emission probabilities of which label will be emitted in a particular state is defined as:

$$P(X_m = i | Z_m = u, H) = \begin{cases} \lambda & \text{if} \quad H_{um} \neq A_{mi} \\ 1 - \lambda & \text{if} \quad H_{um} = A_{mi} \end{cases} \tag{6.3.5}$$

HMM start distribution: The start distribution to choose the first state follows an equal distribution across all states and can simply be denoted as:

$$P(Z_0 = u) = \frac{1}{K} \tag{6.3.6}$$

Calculation of the Posterior Probabilities

The calculation of $P(X_{m+1} = i_2 | X_m = i_1, H)$, i.e. the posterior transitional probabilities of the emissions, presents the core part of the SHAPEIT2 algorithm. Such a probability indicates how likely it is to obtain the next haplotype for one element of the pair from the consistent haplotype vector of the next segment S_{m+1} with index i_2 if the current index in the current segment S_m is i_1. SHAPEIT2 uses these probabilities to sample a new pair of haplotype estimates for each site of the genotype vector. However, SHAPEIT2 restricts the labels to be identical within each segment, i.e.:

$$\forall m, n : S_m = S_n \Rightarrow X_m = X_n \tag{6.3.7}$$

In other words, all labels within one segment have to be equal. Thus, the probabilities have to be calculated at segment borders only.

An exception is given for the first site $m = 0$. Here, the first label is sampled from the posterior marginal probabilities $P(X_0 = i | H)$.

The computation of both, the marginal and the transitional probabilities, is processed using the forward-backward procedure (see Sect. 6.2). The forward and backward variables as well as the before mentioned probabilities are defined as follows.

Forward variables: The forward variables are defined as the probability to emit index i from state u and to observe the genotype sequence $G_{0:m}$ at timestamp m given the haplotype set H.

$$a_m(i, u) = P(X_m = i, Z_m = u, G_{0:m} | H) \tag{6.3.8}$$

The calculation is defined recursively:

$$a_0(i, u) = P(X_0 = i | Z_0 = u, H) P(Z_0 = u) \tag{6.3.9}$$

$$= \frac{1}{K} \begin{cases} \lambda & \text{if} \quad H_{u0} \neq A_{0i} \\ 1 - \lambda & \text{if} \quad H_{u0} = A_{0i} \end{cases} \tag{6.3.10}$$

$$a_{m+1}(i,u) = P(X_{m+1}=i|Z_{m+1}=u,H) \cdot \begin{cases} \sum_{v=0}^{K-1} a_m(i,v)P(Z_{m+1}=u|Z_m=v) & \text{if} \quad S_m = S_{m+1} \\ \sum_{j=0}^{2^B-1}\sum_{v=0}^{K-1} a_m(j,v)P(Z_{m+1}=u|Z_m=v) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.11}$$

$$= \begin{cases} \lambda\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}a_m(i,\bullet) + e^{\frac{-\rho_{m+1}}{K}}a_m(i,u)\right) & \\ \qquad \text{if} \quad H_{u(m+1)} \neq A_{(m+1)i} \wedge S_m = S_{m+1} & \\ (1-\lambda)\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}a_m(i,\bullet) + e^{\frac{-\rho_{m+1}}{K}}a_m(i,u)\right) & \\ \qquad \text{if} \quad H_{u(m+1)} = A_{(m+1)i} \wedge S_m = S_{m+1} & \\ \lambda\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}a_m(\bullet,\bullet) + e^{\frac{-\rho_{m+1}}{K}}a_m(\bullet,u)\right) & \\ \qquad \text{if} \quad H_{u(m+1)} \neq A_{(m+1)i} \wedge S_m \neq S_{m+1} & \\ (1-\lambda)\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}a_m(\bullet,\bullet) + e^{\frac{-\rho_{m+1}}{K}}a_m(\bullet,u)\right) & \\ \qquad \text{if} \quad H_{u(m+1)} = A_{(m+1)i} \wedge S_m \neq S_{m+1} & \end{cases} \tag{6.3.12}$$

The bullet notation is used according to following definitions:

$$a_m(i,\bullet) = \sum_{u=0}^{K-1} a_m(i,u) \tag{6.3.13}$$

$$a_m(\bullet,u) = \sum_{i=0}^{2^B-1} a_m(i,u) \tag{6.3.14}$$

$$a_m(\bullet,\bullet) = \sum_{i=0}^{2^B-1}\sum_{u=0}^{K-1} a_m(i,u) \tag{6.3.15}$$

Backward variables: The backward variables are defined as the probability to observe the remaining sequence $G_{m+1:L-1}$ at timestamp m given the current state u with emission i and the haplotype set H.

$$b_m(i,u) = P(G_{m+1:L}|X_m = i, Z_m = u, H) \tag{6.3.16}$$

As the forward variables the backward variables are calculated recursively, but starting with the last site $L-1$:

$$b_{L-1}(i,u) = 1 \tag{6.3.17}$$

$$b_m(i,u) = \begin{cases} \sum_{v=0}^{K-1} b_{m+1}(i,v) P(X_{m+1} = i|Z_{m+1} = v, H) P(Z_{m+1} = v|Z_m = u) \\ \qquad\qquad \text{if} \quad S_m = S_{m+1} \\ \sum_{j=0}^{2^B-1} \sum_{v=0}^{K-1} b_{m+1}(j,v) P(X_{m+1} = j|Z_{m+1} = v, H) \\ \qquad P(Z_{m+1} = v|Z_m = u) \quad \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.18}$$

$$= \begin{cases} \frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} B_{m+1}(i,\bullet) + e^{\frac{-\rho_{m+1}}{K}} B_{m+1}(i,u) & \text{if} \quad S_m = S_{m+1} \\ \frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} B_{m+1}(\bullet,\bullet) + e^{\frac{-\rho_{m+1}}{K}} B_{m+1}(\bullet,u) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.19}$$

Here, the modified backward variable $B_m(i,u)$ has been introduced and used with the bullet notation analogue to the forward variables:

$$B_m(i,u) = b_m(i,u) P(X_m = i|Z_m = u, H) \tag{6.3.20}$$

$$= b_m(i,u) \begin{cases} \lambda & \text{if} \quad H_{um} \neq A_{mi} \\ 1-\lambda & \text{if} \quad H_{um} = A_{mi} \end{cases} \tag{6.3.21}$$

$$B_m(i,\bullet) = \sum_{u=0}^{K-1} B_m(i,u) \tag{6.3.22}$$

$$B_m(\bullet, u) = \sum_{i=0}^{2^B-1} B_m(i, u) \tag{6.3.23}$$

$$B_m(\bullet, \bullet) = \sum_{i=0}^{2^B-1} \sum_{u=0}^{K-1} B_m(i, u) \tag{6.3.24}$$

Marginal and transitional probabilities: With the definitions of the forward and backward variables the marginal and transitional probabilities can be calculated as follows:

$$P(X_m = i|H) \propto \sum_{u=0}^{K-1} a_m(i, u) b_m(i, u) \tag{6.3.25}$$

$$P(X_{m+1} = i_2|X_m = i_1, H) = \frac{\sum_{u_0=0}^{K-1} \sum_{u_2=0}^{K-1} P(X_m = i_1, Z_m = u_1, X_{m+1} = i_2, Z_{m+1} = u_2|H)}{\sum_{j=0}^{2^B-1} \sum_{u_1=0}^{K-1} \sum_{u_2=0}^{K-1} P(X_m = i_1, Z_m = u_1, X_{m+1} = j, Z_{m+1} = u_2|H)} \tag{6.3.26}$$

The joint probabilities follow this proportionality relation:

$$P(X_m = i_1, Z_m = u_1, X_{m+1} = i_2, Z_{m+1} = u_2|H) \propto a_m(i_1, u_1) P(Z_{m+1} = u_2|Z_m = u_1) P(X_{m+1} = i_2|Z_{m+1} = u_2, H) b_{m+1}(i_2, u_2) \tag{6.3.27}$$

Note that the probabilities are defined only via proportionality relations. However, this is not a problem as the sampling process in the next step has to normalize from a selection anyway (see next paragraph).

Sampling Haplotype Estimates

In order to sample the haplotype estimates for each genotype a pair of labels (i, j) is randomly selected for each segment according to the prob-

ability distribution gained from the calculated probabilities. The selected pair of haplotype vectors has to be assured to be compatible with the genotype vector in that segment. Therefore, the probabilities are combined by multiplication of only those labels which form a compatible pair.

Let $X_m^{(1)}$ and $X_m^{(2)}$ denote the compatible pair of labels to be sampled. The marginal probabilities are only required for the first site to sample the pair of labels for the first segment. Thus, the first pair is sampled proportional to product probability of the marginals at the first site taking only compatible pairs of labels into account. Let (i, j) be a compatible pair of labels:

$$P(X_0^{(1)} = i, X_0^{(2)} = j|H) \propto P(X_0 = i|H)P(X_0 = j|H) \tag{6.3.28}$$

A restriction for the sampling is, that the labels within one segment are not subject to change. Hence, the pair of labels $X_m^{(1)}$ and $X_m^{(2)}$ has to be identical within each segment, i.e.:

$$\forall m, n : S_m = S_n \Rightarrow X_m^{(i)} = X_n^{(i)} \tag{6.3.29}$$

Thus, the sampling is performed segment-wise. A new pair of labels is only selected at segment borders, i.e. at sites m with $S_m \neq S_{m+1}$. Therefore, the transitional probabilities only need to be calculated for those sites. The next pair (i_2, j_2) is sampled proportional to the product probability of the transitional probabilities given the labels of the previous site (i_1, j_1), again only restricted to compatible pairs. Let (i_2, j_2) be a compatible pair of labels:

$$\begin{aligned} P(X_{m+1}^{(1)} = i_2, X_{m+1}^{(2)} = j_2|X_m^{(1)} = i_1, X_m^{(2)} = j_1, H) \propto \\ P(X_{m+1} = i_2|X_m = i_1, H)P(X_{m+1} = j_2|X_m = j_1, H) \end{aligned} \tag{6.3.30}$$

The sequence of generated label pairs exactly describes the sequences of the haplotype vectors in the compatible pair. Figure 6.4 illustrates an example of the sampling process. These newly generated haplotype sequences replace the estimates from the previous step and the process continues with the next genotype or the next iteration. Over the iterations the estimates are likely to converge to a certain form, with a high probability of

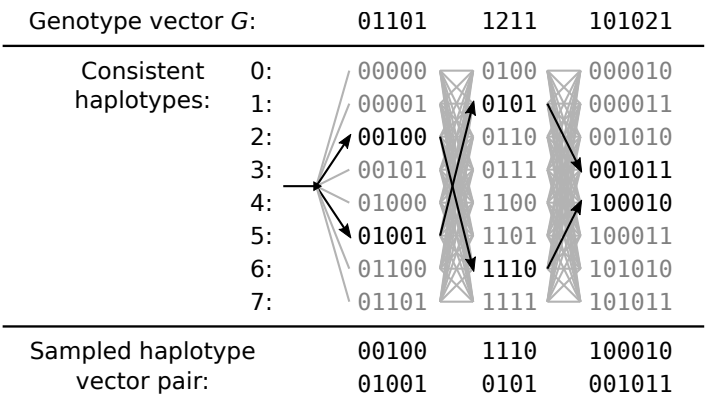


Figure 6.4. SHAPEIT2 sampling example from a genotype vector divided into three segments. Arrows indicate the paths through the consistent haplotypes between the segments for the sampled haplotype vector pair. The sampled label pairs are $(2,5)$, $(6,1)$ and $(4,3)$.

quality improvement after each iteration, which is the goal of this iterative process [DZM13].

Computational Costs

Concluded, SHAPEIT2 performs several iterations for each individual in the input data set. In each iteration it calculates the forward and backward variables for each label at each marker position in each sample. Furthermore, the variables are dependent on each state of the HMM. Thus, with I denoting the total number of iterations, the computational runtime costs are in

$$\mathcal{O}(I \cdot N \cdot L \cdot K \cdot 2^B).$$

During the computation process, the genotypes and the current haplotype estimates of the last iteration need to be accessed. Since all estimates from the main iterations are used to average the final result, they need to be kept in memory after each iteration. Furthermore, estimating a haplotype pair for an individual requires all forward or all backward variables for

the corresponding sample to be calculated and kept in memory in a naive implementation. Thus, the memory complexity is clearly in

$$\mathcal{O}(I \cdot N \cdot L + L \cdot K \cdot 2^B).$$

In the following description of an FPGA implementation it is shown how the runtime and memory complexity can be reduced by spending hardware resources for parallel computation and introducing supporting points for the forward-backward procedure.

6.3.2 FPGA-based SHAPEIT2

Overview

This paragraph gives a short introduction in the FPGA implementation of the SHAPEIT2 core. Details follow in the later paragraphs.

The core of the FPGA implementation of SHAPEIT2 concentrates on the mapping of the sample process of compatible haplotype estimations together with the forward-backward procedure and the underlying Hidden Markov model. The computationally most intensive part of this process is clearly the calculation of the forward and backward variables. Hence, most resources have been spent to solve that part.

The implementation targets a Spartan6-LX150 FPGA of the RIVYERA S6-LX150 architecture and is divided into two parts, one for the host and one for the FPGA. The host part precalculates constants required for the computation, prepares the input data, assigns the windows, generates the first haplotype estimates, and continuously updates the haplotype subset H for each genotype during the iterations. Furthermore, it generates the final haplotype estimates from averaging the estimates of each main iteration and combines the results from the separate windows to single haplotype vector pairs for each input genotype.

An FPGA core performs the forward-backward procedure for the assigned window with the help of the precalculated constants from the host and calculates the marginal and transitional posterior emission probabilities where necessary. The segmentation and sampling of a compatible haplotype pair as well as the generation of the corresponding haplotype

sequences for each segment is also done on the FPGA. The iterations have to be synchronized with the host since the FPGA core reports the newly generated haplotype estimations for the window and relies on the update of the haplotype set H by the host in each iteration.

The FPGA core design basically consists of 16 parallel pipelines for probability calculations. In each iteration eight pipelines concurrently calculate the forward variables and the other eight calculate the backward variables as well as the emission probabilities for each genotype. Eight is the number of possible emissions for each segment ($2^B = 8$), thus each pipeline for the forward or the backward variables does the calculations for a fixed label $X_m = i$.

The pipelines feed a sampling unit which generates the required product probabilities of the emission probabilities and samples a new pair of compatible labels. Subsequently, it generates the corresponding haplotype sequences for the current segment. These directly replace the current estimates of the current genotype.

The forward and backward variables are only of temporary use as they are required only for the calculation of the posterior emission probabilities. However, since they are defined recursively with each type following an opposed direction, the backward variables need to have been calculated already when calculating the first forward variable and the corresponding emission probability. In order to prevent an extensive memory usage, a memory saving strategy taken from [LWD+10] has been implemented. This strategy involves the calculation of all backward variables first, but storing only some supporting points while discarding the rest. This way the required values can be recalculated in time while calculating the forward variables. For more details see the paragraph on *Supporting Points* below.

For any Spartan6 FPGA of the RIVYERA there are two memory modules available with 256 MB each. The first module contains the genotype vectors and all current haplotype estimates of the current window as well as the precalculated constants required for the computation of the forward and backward variables. The second module is reserved for the supporting points for the backward variable calculation and contains a set of pointer arrays. Each array corresponds to one genotype vector and contains the addresses for the haplotype vectors chosen to be in its haplotype subset H.

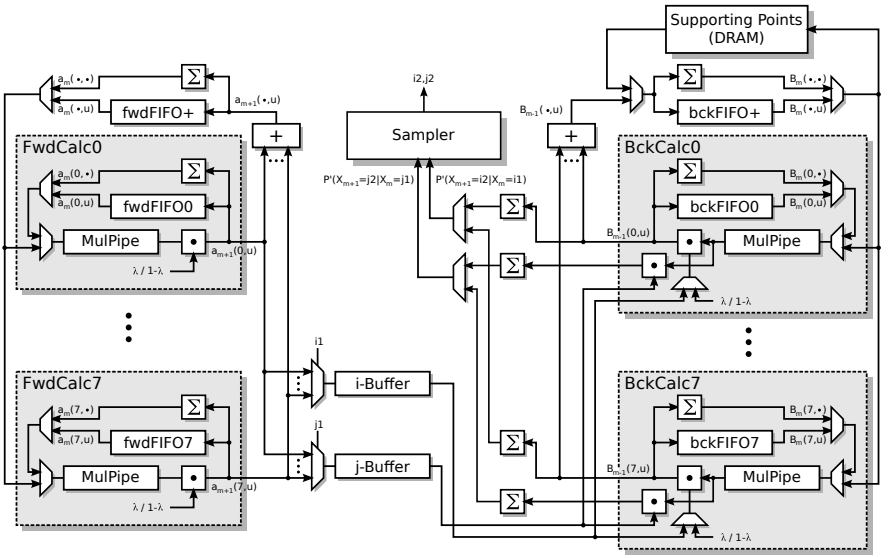


Figure 6.5. Overview of the SHAPEIT2 hardware design.

This array is continuously updated by the host for each iteration.

An overview of the FPGA design is illustrated in Fig. 6.5.

Modified Backward Variables

In order to calculate the required marginal and transitional posterior emission probabilities for the sampling process some transformations of the previously described equations in Sect. 6.3.1 are introduced to enhance efficiency. The transformations are based on the modified definition of the backward variables already introduced in Eq. 6.3.20. With this definition the relation in Eq. 6.3.25 to calculate the marginal emission probabilities evaluates to:

$$P(X_m = i|H) \propto \sum_{u=0}^{K-1} a_m(i,u) \frac{B_m(i,u)}{P(X_m = i|Z_m = u, H)} \tag{6.3.31}$$

This equation is only required once to calculate the marginal probability for the first site, i.e. $m = 0$. Hence, with the definition in Eq. 6.3.9 this evaluates to:

$$P(X_0 = i|H) \propto \sum_{u=0}^{K-1} a_0(i,u) \frac{B_0(i,u)}{P(X_0 = i|Z_0 = u, H)} \tag{6.3.32}$$

$$= \sum_{u=0}^{K-1} P(Z_0 = u) B_0(i,u) \tag{6.3.33}$$

$$= \frac{1}{K} \sum_{u=0}^{K-1} B_0(i,u) \tag{6.3.34}$$

$$= \frac{1}{K} B_0(i,\bullet) \tag{6.3.35}$$

In other words, the modified backward variables of the first site can directly be used to sample the labels for the first segment:

$$P(X_0 = i|H) \propto B_0(i,\bullet) \tag{6.3.36}$$

For all other segments the probabilities are calculated according to the transitional probabilities defined in Eq. 6.3.26. It is based on Eq. 6.3.27 which is easily replaced as follows:

$$P(X_m = i_1, Z_m = u_1, X_{m+1} = i_2, Z_{m+1} = u_2|H) \propto$$
$$a_m(i_1,u_1) P(Z_{m+1} = u_2|Z_m = u_1) B_{m+1}(i_2,u_2) \tag{6.3.37}$$

With the definition of the HMM transition probabilities in Eq. 6.3.4 this evaluates to (c constant):

$$P(X_m = i_1, Z_m = u_1, X_{m+1} = i_2, Z_{m+1} = u_2|H) \propto$$
$$a_m(i_1,u_1) B_{m+1}(i_2,u_2) \begin{cases} e^{\frac{-\rho_{m+1}}{K}} + \frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} & \text{if} \quad u_1 = u_2 \\ \frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} & \text{if} \quad u_1 \neq u_2 \end{cases} \tag{6.3.38}$$

$$\Leftrightarrow \qquad P(X_m = i_1, Z_m = u_1, X_{m+1} = i_2, Z_{m+1} = u_2|H) =$$

$$c\ a_m(i_1,u_1)B_{m+1}(i_2,u_2) \begin{cases} e^{\frac{-\rho_{m+1}}{K}} + \frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} & \text{if} \quad u_1 = u_2 \\ \frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} & \text{if} \quad u_1 \neq u_2 \end{cases} \tag{6.3.39}$$

Taking this into account Eq. 6.3.26 can be evaluated to:

$$P(X_{m+1} = i_2|X_m = i_1, H) =$$

$$= \frac{\sum_{u_1=0}^{K-1}\sum_{u_2=0}^{K-1} c\ a_m(i_1,u_1)P(Z_{m+1} = u_2|Z_m = u_1)B_{m+1}(i_2,u_2)}{\sum_{j=0}^{2^B-1}\sum_{u_1=0}^{K-1}\sum_{u_2=0}^{K-1} c\ a_m(i_1,u_1)P(Z_{m+1} = u_2|Z_m = u_1)B_{m+1}(j,u_2)} \tag{6.3.40}$$

$$= \frac{c\sum_{u_1=0}^{K-1} a_m(i_1,u_1)\sum_{u_2=0}^{K-1} P(Z_{m+1} = u_2|Z_m = u_1)B_{m+1}(i_2,u_2)}{c\sum_{u_1=0}^{K-1} a_m(i_1,u_1)\sum_{u_2=0}^{K-1} P(Z_{m+1} = u_2|Z_m = u_1)\sum_{j=0}^{2^B-1} B_{m+1}(j,u_2)} \tag{6.3.41}$$

$$= \frac{\sum_{u_1=0}^{K-1} a_m(i_1,u_1)\sum_{u_2=0}^{K-1} P(Z_{m+1} = u_2|Z_m = u_1)B_{m+1}(i_2,u_2)}{\sum_{u_1=0}^{K-1} a_m(i_1,u_1)\sum_{u_2=0}^{K-1} P(Z_{m+1} = u_2|Z_m = u_1)B_{m+1}(\bullet,u_2)} \tag{6.3.42}$$

$$= \frac{\sum_{u_1=0}^{K-1} a_m(i_1,u_1)\left(\sum_{u_2=0}^{K-1}\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}B_{m+1}(i_2,u_2) + e^{\frac{-\rho_{m+1}}{K}}B_{m+1}(i_2,u_1)\right)}{\sum_{u_1=0}^{K-1} a_m(i_1,u_1)\left(\sum_{u_2=0}^{K-1}\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}B_{m+1}(\bullet,u_2) + e^{\frac{-\rho_{m+1}}{K}}B_{m+1}(\bullet,u_1)\right)} \tag{6.3.43}$$

$$= \frac{\sum_{u_1=0}^{K-1} a_m(i_1,u_1)\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}B_{m+1}(i_2,\bullet) + e^{\frac{-\rho_{m+1}}{K}}B_{m+1}(i_2,u_1)\right)}{\sum_{u_1=0}^{K-1} a_m(i_1,u_1)\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}B_{m+1}(\bullet,\bullet) + e^{\frac{-\rho_{m+1}}{K}}B_{m+1}(\bullet,u_1)\right)} \tag{6.3.44}$$

Now, the recursion strategy for the backward variables has to be adapted to the modified backward variables $B_m(i,u)$. This new recursive definition replaces Eqs. 6.3.17 and 6.3.19.

$$B_{L-1}(i,u) = b_{L-1}(i,u)P(X_{L-1}=i|Z_{L-1}=u,H) \tag{6.3.45}$$
$$= P(X_{L-1}=i|Z_{L-1}=u,H) \tag{6.3.46}$$
$$= \begin{cases} \lambda & \text{if} \quad H_{u(L-1)} \neq A_{(L-1)i} \\ 1-\lambda & \text{if} \quad H_{u(L-1)} = A_{(L-1)i} \end{cases} \tag{6.3.47}$$

$$B_m(i,u) = b_m(i,u)P(X_m=i|Z_m=u,H) \tag{6.3.48}$$
$$= P(X_m=i|Z_m=u,H) \begin{cases} \sum_{v=0}^{K-1} B_{m+1}(i,v)P(Z_{m+1}=v|Z_m=u) \\ \qquad\qquad \text{if} \quad S_m = S_{m+1} \\ \sum_{j=0}^{2^B-1}\sum_{v=0}^{K-1} B_{m+1}(j,v) \\ \qquad P(Z_{m+1}=v|Z_m=u) \\ \qquad\qquad \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.49}$$

$$= P(X_m=i|Z_m=u,H) \begin{cases} \frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} B_{m+1}(i,\bullet) \\ \qquad +e^{\frac{-\rho_{m+1}}{K}} B_{m+1}(i,u) \\ \qquad\qquad \text{if} \quad S_m = S_{m+1} \\ \frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} B_{m+1}(\bullet,\bullet) \\ \qquad +e^{\frac{-\rho_{m+1}}{K}} B_{m+1}(\bullet,u) \\ \qquad\qquad \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.50}$$

$$= \begin{cases} \lambda \left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} B_{m+1}(i,\bullet) + e^{\frac{-\rho_{m+1}}{K}} B_{m+1}(i,u) \right) \\ \qquad \text{if} \quad H_{um} \neq A_{mi} \wedge S_m = S_{m+1} \\ (1-\lambda) \left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} B_{m+1}(i,\bullet) + e^{\frac{-\rho_{m+1}}{K}} B_{m+1}(i,u) \right) \\ \qquad \text{if} \quad H_{um} = A_{mi} \wedge S_m = S_{m+1} \\ \lambda \left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} B_{m+1}(\bullet,\bullet) + e^{\frac{-\rho_{m+1}}{K}} B_{m+1}(\bullet,u) \right) \\ \qquad \text{if} \quad H_{um} \neq A_{mi} \wedge S_m \neq S_{m+1} \\ (1-\lambda) \left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} B_{m+1}(\bullet,\bullet) + e^{\frac{-\rho_{m+1}}{K}} B_{m+1}(\bullet,u) \right) \\ \qquad \text{if} \quad H_{um} = A_{mi} \wedge S_m \neq S_{m+1} \end{cases} \tag{6.3.51}$$

With this observation and comparison of Eq. 6.3.51 with Eq. 6.3.12 it is clear to see that the calculation of the forward variables $a_m(i,u)$ shares exactly the same rule as the calculation of the modified backward variables $B_m(i,u)$. Thus, the same hardware description can be used for both calculations. Furthermore, a closer look on Eq. 6.3.44 shows that the same rule can be applied again for the separate calculation of the numerator and the denominator for the transition probability. The only required modification is that λ or respectively $1-\lambda$ has to be replaced by the forward variable $a_m(i,u)$. (It can be shown that the calculation of the denominator and the division is even not necessary. See paragraph on *Sampling* for more details.) This leads to a uniform pipeline structure for all three computations and will be described in the paragraph on *The Computation Pipeline* in detail.

Scaling

All presented equations are based on probabilities, i.e. on numbers between zero and one. In fact, most values act in an area very close to zero and get closer during the extensive multiplications presented in the equations. Hence, it is crucial to provide a scaling mechanism not to lose too much

precision or even lose a very small number to zero. For this purpose scaling means to calculate the desired probabilities with scaled forward and backward variables, namely $\hat{a}_m(i,u)$ and $\hat{B}_m(i,u)$. The scaling factor has to be constant for each recursion step and is denoted c_m.

Scaling method: The following redefinitions for the calculation of the forward and backward variables are based on the scaling mechanism described in [Rab89].

The calculation of the forward variables according to Eqs. 6.3.9 and 6.3.11 is newly defined as:

$$\hat{a}_0(i,u) = c_0 a_0(i,u) \tag{6.3.52}$$

$$\hat{a}_{m+1}(i,u) = c_{m+1} P(X_{m+1}=i|Z_{m+1}=u,H) \cdot \begin{cases} \sum_{v=0}^{K-1} \hat{a}_m(i,v) P(Z_{m+1}=u|Z_m=v) & \text{if} \quad S_m = S_{m+1} \\ \sum_{j=0}^{2^B-1} \sum_{v=0}^{K-1} \hat{a}_m(j,v) P(Z_{m+1}=u|Z_m=v) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.53}$$

The recursion for the modified backward variables according to Eqs. 6.3.46 and 6.3.49 is redefined as:

$$\hat{B}_{L-1}(i,u) = c_{L-1} B_{L-1}(i,u) \tag{6.3.54}$$

$$\hat{B}_m(i,u) = c_m P(X_m=i|Z_m=u,H) \begin{cases} \sum_{v=0}^{K-1} \hat{B}_{m+1}(i,v) P(Z_{m+1}=v|Z_m=u) & \text{if} \quad S_m = S_{m+1} \\ \sum_{j=0}^{2^B-1} \sum_{v=0}^{K-1} \hat{B}_{m+1}(j,v) P(Z_{m+1}=v|Z_m=u) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.55}$$

According to these definitions Eqs. 6.3.12 and 6.3.51 can be used with simply replacing $a_m(i,u)$ by $\hat{a}_m(i,u)$ and $B_m(i,u)$ by $\hat{B}_m(i,u)$ and applying

the multiplicative scaling factor. In the following it is shown that the scaling has no consequence on the calculation of the required marginal and transitional emission probabilities.

Let C_m and D_m be products of the so far used scaling factors in the forward and the backward process respectively, i.e.:

$$C_m = \prod_{\mu=0}^{m} c_\mu \tag{6.3.56}$$

$$D_m = \prod_{\mu=m}^{L-1} c_\mu \tag{6.3.57}$$

Regarding the original forward and backward variables, the following conditions hold (the proofs can be found below):

$$\hat{a}_m(i,u) = C_m a_m(i,u) \tag{6.3.58}$$

$$\hat{B}_m(i,u) = D_m B_m(i,u) \tag{6.3.59}$$

The marginal probability for the first site is calculated in the following way (according to Eq. 6.3.36):

$$P(X_0 = i|H) \propto \frac{\hat{B}_0(i,\bullet)}{D_0} \tag{6.3.60}$$

$$\Leftrightarrow \qquad P(X_0 = i|H) \propto \hat{B}_0(i,\bullet) \tag{6.3.61}$$

According to Eq. 6.3.44 the calculation of the transitional probabilities evaluates to:

$$\begin{aligned} &P(X_{m+1} = i_2 | X_m = i_1, H) = \\ &= \frac{\sum_{u_1=0}^{K-1} \frac{\hat{a}_m(i_1,u_1)}{C_m} \left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} \frac{\hat{B}_{m+1}(i_2,\bullet)}{D_{m+1}} + e^{\frac{-\rho_{m+1}}{K}} \frac{\hat{B}_{m+1}(i_2,u_1)}{D_{m+1}} \right)}{\sum_{u_1=0}^{K-1} \frac{\hat{a}_m(i_1,u_1)}{C_m} \left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K} \frac{\hat{B}_{m+1}(\bullet,\bullet)}{D_{m+1}} + e^{\frac{-\rho_{m+1}}{K}} \frac{\hat{B}_{m+1}(\bullet,u_1)}{D_{m+1}} \right)} \end{aligned} \tag{6.3.62}$$

$$= \frac{\frac{1}{D_0}\sum_{u_1=0}^{K-1}\hat{a}_m(i_1,u_1)\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}\hat{B}_{m+1}(i_2,\bullet)+e^{\frac{-\rho_{m+1}}{K}}\hat{B}_{m+1}(i_2,u_1)\right)}{\frac{1}{D_0}\sum_{u_1=0}^{K-1}\hat{a}_m(i_1,u_1)\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}\hat{B}_{m+1}(\bullet,\bullet)+e^{\frac{-\rho_{m+1}}{K}}\hat{B}_{m+1}(\bullet,u_1)\right)} \tag{6.3.63}$$

$$= \frac{\sum_{u_1=0}^{K-1}\hat{a}_m(i_1,u_1)\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}\hat{B}_{m+1}(i_2,\bullet)+e^{\frac{-\rho_{m+1}}{K}}\hat{B}_{m+1}(i_2,u_1)\right)}{\sum_{u_1=0}^{K-1}\hat{a}_m(i_1,u_1)\left(\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}\hat{B}_{m+1}(\bullet,\bullet)+e^{\frac{-\rho_{m+1}}{K}}\hat{B}_{m+1}(\bullet,u_1)\right)} \tag{6.3.64}$$

Hence, Eq. 6.3.44 can still be used to calculate the transitional probabilities with simply replacing $a_m(i,u)$ by $\hat{a}_m(i,u)$ and $B_m(i,u)$ by $\hat{B}_m(i,u)$ as well.

Proof for scaled forward variable (Eq. 6.3.58): The proof is by induction.

Base case:

$$\hat{a}_0(i,u) = c_0 a_0(i,u) \tag{6.3.65}$$

$$= C_0 a_0(i,u) \tag{6.3.66}$$

If $\hat{a}_m(i,u) = C_m a_m(i,u)$, then:

$$\hat{a}_{m+1}(i,u) = c_{m+1}\begin{cases} P(\ldots)\sum_{v=0}^{K-1}\hat{a}_m(i,v)P(\ldots) & \text{if} \quad S_m = S_{m+1} \\ P(\ldots)\sum_{j=0}^{2^B-1}\sum_{v=0}^{K-1}\hat{a}_m(j,v)P(\ldots) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.67}$$

$$= c_{m+1}\begin{cases} P(\ldots)\sum_{v=0}^{K-1}C_m a_m(i,v)P(\ldots) & \text{if} \quad S_m = S_{m+1} \\ P(\ldots)\sum_{j=0}^{2^B-1}\sum_{v=0}^{K-1}C_m a_m(j,v)P(\ldots) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.68}$$

$$= c_{m+1}C_m \begin{cases} P(\ldots) \sum_{v=0}^{K-1} a_m(i,v)P(\ldots) & \text{if} \quad S_m = S_{m+1} \\ P(\ldots) \sum_{j=0}^{2^B-1} \sum_{v=0}^{K-1} a_m(j,v)P(\ldots) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.69}$$

$$= C_{m+1}a_{m+1}(i,u) \qquad \square \tag{6.3.70}$$

Proof for scaled backward variable (Eq. 6.3.59): The proof is by induction.

Base case:

$$\hat{B}_{L-1}(i,u) = c_{L-1}B_{L-1}(i,u) \tag{6.3.71}$$

$$= D_{L-1}B_{L-1}(i,u) \tag{6.3.72}$$

If $\hat{B}_{m+1}(i,u) = D_{m+1}B_{m+1}(i,u)$, then:

$$\hat{B}_m(i,u) = c_m P(\ldots) \begin{cases} \sum_{v=0}^{K-1} \hat{B}_{m+1}(i,v)P(\ldots) & \text{if} \quad S_m = S_{m+1} \\ \sum_{j=0}^{2^B-1} \sum_{v=0}^{K-1} \hat{B}_{m+1}(j,v)P(\ldots) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.73}$$

$$= c_m P(\ldots) \begin{cases} \sum_{v=0}^{K-1} D_{m+1}B_{m+1}(i,v)P(\ldots) & \text{if} \quad S_m = S_{m+1} \\ \sum_{j=0}^{2^B-1} \sum_{v=0}^{K-1} D_{m+1}B_{m+1}(j,v)P(\ldots) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.74}$$

$$= c_m D_{m+1} P(\ldots) \begin{cases} \sum_{v=0}^{K-1} B_{m+1}(i,v)P(\ldots) & \text{if} \quad S_m = S_{m+1} \\ \sum_{j=0}^{2^B-1} \sum_{v=0}^{K-1} B_{m+1}(j,v)P(\ldots) & \text{if} \quad S_m \neq S_{m+1} \end{cases} \tag{6.3.75}$$

$$= D_m B_m(i,u) \qquad \square \tag{6.3.76}$$

Applied scaling mechanism: The scaling mechanism applied to the FPGA implementation is based on the backward variables and is chosen to be effectively applicable on-the-fly with floating point operations.

The scaling factor for the last site is 1, i.e. no scaling:

$$c_{L-1} = 1 \tag{6.3.77}$$

For all other sites with $m < L - 1$:

$$c_m = 2^{-t_m} \tag{6.3.78}$$

with

$$t_m = \left\lceil \log_2 \left(\hat{B}_{m+1}(\bullet, \bullet) \right) \right\rceil \tag{6.3.79}$$

In other words, the scaling factor is taken from the largest value of the backwards computation $\hat{B}_{m+1}(\bullet, \bullet)$ in the previous step. Thus, it can directly be applied on-the-fly to the current computations since it is already known at the beginning of this step, which is contrary to many other proposed scaling mechanisms (such as described in [Rab89]) that determine the scaling factor for each step at the end and recall all already computed variables for scaling.

Furthermore, the scaling factor can directly be taken from the exponent in the floating point representation of $\hat{B}_{m+1}(\bullet, \bullet)$, and scaling itself is nothing else than subtracting this value from the exponent of the currently computed value, which makes it very efficient to be implemented in hardware.

Supporting Points

The forward and backward variables are calculated recursively, but with each type following an opposed direction. The calculation of the transitional emission probability of any site requires the presence of the corresponding forward and backward variable (see Eq. 6.3.64) and the calculation of the first marginal probability requires the presence of the backward variable of the first site. Thus, for the sampling process, which follows a forward direction, all backward variables must have already been computed and

stored while the computation of the forward variables could be processed on-the-fly.

However, the storage of the backward variables would require the capacity for at least $2^B KL$ values, which gets impossible very quickly for the small storage capabilities on the FPGA. Thus, a memory saving strategy similar to the one used in [LWD+10] has been implemented. The proposed strategy requires only a small fraction of memory from the naive strategy at the cost of calculating all backward variables twice. The first run stores *supporting points* every n sites and discards the rest. Then, only those backward variables beginning from the last supporting point up to the current site are kept in memory while calculating the forward variables. When the process passes the last supporting point, the current backward variables are discarded and the ones beginning from the next supporting point up to the current location are calculated again and kept in memory.

Regarding the application here, the emission probabilities are only calculated at the segment borders. Thus, the supporting points could be set to exactly those sites. In detail, the site for a supporting point is always chosen to be the first of a segment. This ensures the condition $S_m \neq S_{m+1}$ to be valid for the calculation of the next backward variables. According to Eq. 6.3.51 only the accumulated backward variables $\hat{B}_{m+1}(\bullet, u)$ and $\hat{B}_{m+1}(\bullet, \bullet)$ are required for this computation. These values are chosen to present a supporting point now. Therefore, the required memory for all supporting points is only $(K+1)C$ values, whereby the number of segments C is clearly considerably smaller than the number of sites L.

However, these values can not directly be used to calculate the transitional probabilities according to Eq. 6.3.64 since this requires $\hat{B}_{m+1}(i, u)$ and $\hat{B}_{m+1}(i, \bullet)$ again. So, discarding these values in the first run saves memory resources but requires the recalculation of these values in the second run. Nevertheless, the computation of the transitional probabilities is again only necessary at the segment border. Thus, there is no need to store all backward variables of one segment until they are used. Only the variables of the first site of a segment are required. Beyond that, let a segment border be between sites m and $m+1$. The recalculation of the backward variables for a segment S_{m+1} happens concurrently to the calculation of the forward variables of the previous segment S_m. Hence, when both computations have

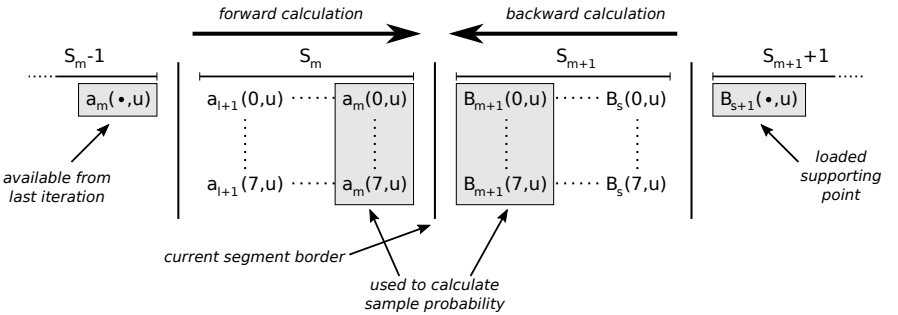


Figure 6.6. Using supporting points to calculate necessary forward and backward variables concurrently. In the example the current segment border is located between sites m and $m+1$. Thus, in order to calculate the sample probabilities $P_m^{\text{sample}}(i,j)$ to sample a new index pair (i,j) for segment S_{m+1}, it is necessary to calculate the forward variables $\hat{a}_m(x,u)$ and the backward variables $\hat{B}_{m+1}(x,u)$. These are calculated concurrently based on the forward variables $\hat{a}_l(\bullet,u)$ at the last segment border available from the last iteration, and the backward variables $\hat{B}_{s+1}(\bullet,u)$ available from the loaded supporting point at the next segment border. Arrows indicate the calculation direction.

reached the segment border, the corresponding transitional probability can be calculated and the process continues with the calculation of the forward variables for segment S_{m+1} while the backward variables are calculated for segment $(S_{m+1}+1)$. Figure 6.6 illustrates this procedure.

Besides the storage of the variables $\hat{B}_{m+1}(\bullet,u)$ and $\hat{B}_{m+1}(\bullet,\bullet)$ it is also necessary to store the site information m together with the supporting point because the lengths of each segment may differ and the process computing the backward variables needs to know at which site to start. With this information the total memory requirements for the supporting points evaluates to $(K+2)C$ values.

The Computation Pipelines

The computation of either forward or backward variables share almost the same pipeline structure and implement Eqs. 6.3.12 and 6.3.51 with scaled variables. The following explanations focus on the implementation of the

backward variables but can be applied analogue to the forward variables.

There is a pipeline implemented for each index $0 \leqslant i < 2^B$ in a segment, i.e. eight pipelines for each type of variables. Each pipeline consists of only two floating point multipliers and one floating point adder as well as one integer adder for scaling the exponent of the resulting floating point and a floating point accumulator. In the first computation cycle of the pipeline the first multiplier computes either $\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}\hat{B}_{m+1}(i,\bullet)$ or $\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}\hat{B}_{m+1}(\bullet,\bullet)$ depending on whether m is at a segment border or not. The result is stored in a register and used for the subsequent computation cycles. Here, the same multiplier computes either $e^{\frac{-\rho_{m+1}}{K}}\hat{B}_{m+1}(i,u)$ or $e^{\frac{-\rho_{m+1}}{K}}\hat{B}_{m+1}(\bullet,u)$. The adder is used to add the result from the first cycle to this value. The second multiplier then multiplies either λ or $1-\lambda$ to the result, depending whether the current haplotype of this site and state H_{um} matches the corresponding allele A_{mi} for this index at this site or not. At last, a scaling factor is applied via the integer adder to the exponent of the floating point representation of the result.

All operations are completely unrolled such that an operation can be started within each clock cycle. The process iterates over all K states in H for each site m. Thus, $K+1$ cycles plus latency are required to calculate all variables for one site. In each cycle a new variable $B_m(i,u)$ is generated and presented at the output port. Furthermore, all variables are accumulated on-the-fly to generate $B_m(i,\bullet)$ using the accumulator (see Sect. 6.3.2 below for details on the pipelined floating point accumulator).

Each pipeline is connected to a local buffer implemented as a FIFO in BRAM. The buffer contains all variables computed for the previous site, i.e. for the corresponding index i the values $\hat{B}_{m+1}(i,u)$ and $\hat{B}_{m+1}(i,\bullet)$ whereby the latter is stored in a register besides the FIFO since it is generated at last in the previous iteration but required first for the current iteration. During the computation, the FIFO is updated with the newly generated variables $\hat{B}_m(i,u)$ and $\hat{B}_m(i,\bullet)$ such that they will be available for the next iteration.

The values for $e^{\frac{-\rho_{m+1}}{K}}$ and $\frac{1-e^{\frac{-\rho_{m+1}}{K}}}{K}$ are constant for each site and taken from a *coefficient buffer* which automatically prefetches this information from external memory (see paragraph on *Memory Buffers* for more details). λ and

$1 - \lambda$ are constant again and stored in registers on the FPGA.

One additional FIFO buffer is required for the storage of the values $\hat{B}_{m+1}(\bullet, u)$ and $\hat{B}_{m+1}(\bullet, \bullet)$. This buffer is implemented exactly as the local buffers for each pipeline and is directly attached to the block of eight pipelines for either forward or backward variables. The buffer provides the before mentioned variables to all eight pipelines in the case where the current site m is located at a segment border, i.e. $S_m \neq S_{m+1}$. The values in the buffer are updated during the process via an adder tree which accumulates all outputs of all eight pipelines to generate $\hat{B}_m(i, \bullet)$ and $\hat{B}_m(\bullet, \bullet)$.

For the backward variables this buffer can be externally loaded with a supporting point to allow a site change to the supporting point. This functionality is not required for the forward variables.

Furthermore, the pipelines for the calculation of the backward variables are also used to calculate the numerator of transitional probability described in Eq. 6.3.64. It is easy to see that this computation equals the computation of $\hat{B}_m(i, \bullet)$ without scaling if m were not at a segment border and λ and $1 - \lambda$ were replaced by the required forward variables $\hat{a}_m(i_1, u)$ whereby i_1 presents the first element of the last sampled index pair (i_1, j_1). One additional multiplier is implemented to concurrently calculate the numerator of the transitional probability for the index j_1 from the index pair. The one input is shared with the before mentioned multiplier, but the second input is connected to the forward variables $\hat{a}_m(j_1, u)$. Furthermore, the computation of the denominator for this probabilities is not necessary as it is shown in the paragraph of *Sampling*. Thus, only this additional multiplier is required for calculating the two probabilities.

The pipeline structure is depicted in Fig. 6.7.

Implementation of a Pipelined Floating-Point Accumulator

All computational cores for basic floating point operations, such as multiplication or addition, are provided by the Xilinx Coregen. An exception is presented by the floating point accumulator which is only supported for 7-series FPGAs or newer. Since the latency of the adders used in this application is four clock cycles, an accumulator which is able to accept one

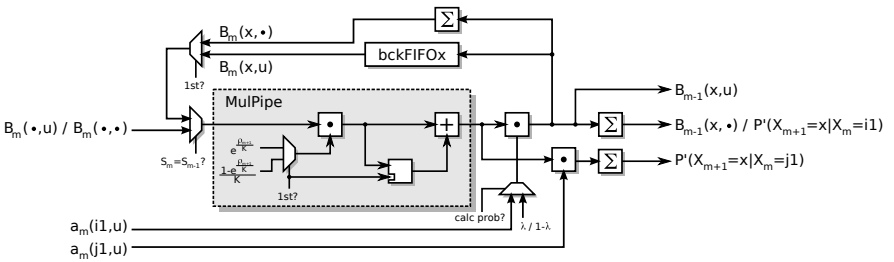


Figure 6.7. The calculation pipeline used in the SHAPEIT2 core. Exemplarily shown is the pipeline used to calculate the backward variables for an index x and the numerator of the corresponding transition probability indicated with P'.

value in each cycle is not a trivial task. The solution provided here uses two pipelined floating point adders with four cycles latency.

The first adder features a simple backward path that provides the current sum as one input. The other input is served by the values which are to be accumulated. This way, the first adder produces four partial sums of the desired accumulated value in four consecutive time slots.

The second adder is used to accumulate the four partial sums from the first adder. The accumulation is divided into four time slots corresponding to the latency of the second adder. First of all, the output of the first adder is registered, such that the value of the previous clock cycle is available to the second adder at the same time with the current value. In the second time slot the second adder now adds the two partial sums from the first adder of the first and the second time slot. The same applies to the fourth time slot where the partial sums from the third and the fourth time slot are added. Thus, after four clock cycles, two partial sums in time slots two and four are available to form the total sum. Hence, the output of the second adder is registered for one and for three cycles, such that the first time slot of the second adder can be used to accumulate these two parts. After four more cycles the final accumulation value is available in time slot one.

The total latency of the accumulator depends on the time slot where the last value that is to be added arrives at the first adder. If it is available in the first time slot, it passes through all register stages supplementary to the

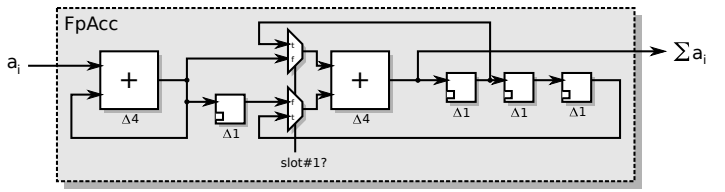


Figure 6.8. Principle of a floating point accumulator using two adders with four clock cycles latency each. Flags to indicate a valid datum or the end of the accumulation are omitted. The result is only valid at time slot #1 between 13 and 16 clock cycles after the last datum. The corresponding indicator is omitted as well.

three additions, resulting in a total latency of 16 clock cycles. The minimum is 13 clock cycles with only one register stage extra to the three additions. Figure 6.8 illustrates the functionality of the accumulator. The last value for the accumulation is flagged to signalize the end of the accumulation and hence, a valid output. This flag is not shown in the illustration.

Memory Buffers

The computation of the forward and backward variables are iterated over H for each site m, i.e. during the process the value u running from 0 to $K-1$ changes in each clock cycle. u addresses a haplotype vector from the current estimates in the external RAM while m addresses the current site in each vector. Thus, a constant stream of haplotypes h_{um} from all vectors in H is required for the current site m. Since all vectors in H are addressed via a pointer located in the external RAM as well and furthermore only one haplotype at the current site is required from each vector, extracting the data directly from the RAM to generate the stream during the process is not practicable.

For this reason, a *haplotype buffer* prefetches and stores the required haplotypes for a couple of sites in the local BRAM. The buffer consists of two separately controlled BRAMs and provides space for the haplotypes of 2×128 sites. Hence, all K subvectors of H of the length 128 from two certain sections are contained in the buffer. Independent of the addressing, the read request on a BRAM takes only one clock cycle. Hence, to access

the before mentioned sequence of single haplotypes for a particular site, each cycle first addresses the 32 bit words (corresponding to the port size) in each haplotype subvector containing the requested haplotype and then picks the haplotype out of that word with a multiplexer.

The BRAMs are true dual-ported such that the forward and the backward process can concurrently access the buffers with different addresses. A control process keeps track of the currently accessed sites and triggers a buffer reload if necessary. Since the directions of both processes are known, the next sites which need to be accessed are easily predictable. Thus, if the sites contained in one of the two BRAMs will not be required anymore, the contents are immediately replaced by the sites next to the last site contained in the other buffer. With the introduction of a maximum segment size of 64 it is ensured that the reload happens concurrently to the computations and thus, generally no delay can be observed (see paragraph on *Memory Utilization and Core Control* for more information on the maximum segment size). The reload process accesses the required addresses for the haplotype vectors in H from another local buffer which holds all K pointers for the current genotype. Figure 6.9 illustrates the functionality of the haplotype buffer.

Analogue to the haplotype buffer a *genotype buffer* and *coefficient buffer* is implemented. In contrast to the haplotype stream required from the haplotype buffer, each process requires only one genotype g_m and two coefficients (namely $e^{\frac{-\rho m}{K}}$ and $\frac{1-e^{\frac{-\rho m}{K}}}{K}$) for each site m. Thus, the required local RAMs are implemented as simple dual-ported (i.e. one port provides read access and the other only write access for reloading). Furthermore, much less space is required for each site when compared to the haplotype buffer. Therefore, two separate BRAMs are not required to implement the same reload strategy as for the haplotype buffer. This is simply realized by dividing the address space of a BRAM into two separate spaces. The genotype buffer provides space for 2×4096 genotypes while the coefficient buffer stores 2×512 coefficients.

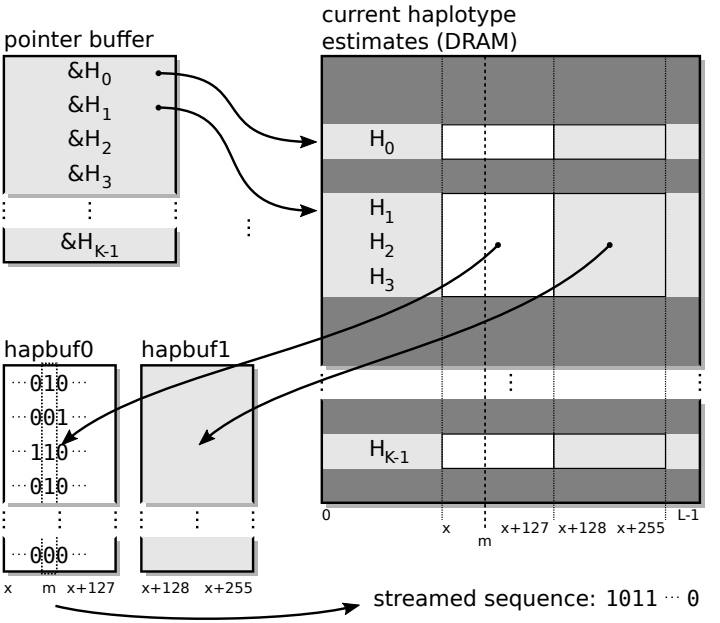


Figure 6.9. Buffering of haplotypes and generation of a haplotype stream. The example shows that only required haplotypes according to the pointer buffer are loaded into the haplotype buffers. The buffers contain windows of 128 sites which are required for the current or next iterations. If one buffer contains only sites which will not be required by neither the forward process nor the backward process anymore, the buffer is reloaded with required sites for upcoming iterations.

Sampling

Sampling denotes the determination of an index pair corresponding to a pair of haplotype vectors compatible to the current genotype vector in the current segment. Only one pair is sampled for each segment. There are always eight possible indices $0 \leqslant i < 2^B = 8$ available, resulting from the maximum of $B = 3$ heterozygous haplotypes in each segment.

The first index pair (i, j) is sampled proportional to the product proba-

bility of the marginal probabilities of the first site:

$$P_0^{\text{sample}}(i,j) \propto P(X_0 = i|H)P(X_0 = j|H) \tag{6.3.80}$$

All other pairs (i_2, j_2) are sampled proportional to the product probability of the transitional probabilities conditional on the pair of the previous segment (i_1, j_1):

$$P_m^{\text{sample}}(i_2,j_2) \propto P(X_{m+1} = i_2|X_m = i_1, H)P(X_{m+1} = j_2|X_m = j_1, H) \tag{6.3.81}$$

Considering that not every possible pair presents a compatible pair, the sample probability has to be normalized taking only compatible pairs into account. Let Q denote the set that contains all compatible index pairs (i,j). Then, the sample probabilities are calculated as follows:

$$P_0^{\text{sample}}(i,j) = \frac{P(X_0 = i|H)P(X_0 = j|H)}{\sum\limits_{(i,j)\in Q} P(X_0 = i|H)P(X_0 = j|H)} \tag{6.3.82}$$

$$P_m^{\text{sample}}(i_2,j_2) = \frac{P(X_{m+1} = i_2|X_m = i_1, H)P(X_{m+1} = j_2|X_m = j_1, H)}{\sum\limits_{(i_2,j_2)\in Q} P(X_{m+1} = i_2|X_m = i_1, H)P(X_{m+1} = j_2|X_m = j_1, H)} \tag{6.3.83}$$

Now, let $0 \leqslant r < 1$ be a random number. The sampling process accumulates the sampling probabilities for each compatible index pair one after the other until the sum exceeds r. The sampled index pair (i,j) is that pair whose corresponding probability leads to the sum being larger than r. In particular, the sample process is divided into two steps. Firstly, all compatible pairs are accumulated in advance. Instead of implementing a divider, this value is then multiplied with the random value r. Secondly, all pairs are accumulated again, but this time the current sum is compared to the multiplied random value after each pair according to the before mentioned procedure. Figure 6.10 demonstrates the sampling process.

Rather than computing the complete transitional probability for each index, it is possible to apply the procedure described above to only the numerators of Eq. 6.3.64. The denominators do not need to be calculated

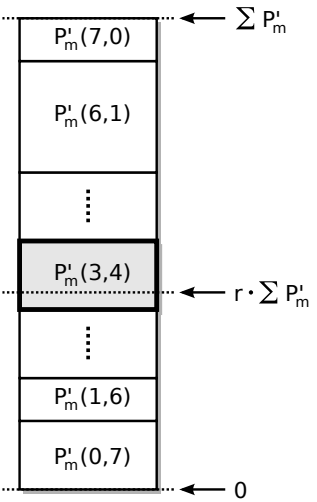


Figure 6.10. Sampling a compatible index pair. The numerators of the transitional probabilities $P_m^{\text{sample}}(i,j)$ for all compatible index pairs (i,j), indicated as $P'_m(i,j)$, are accumulated first. The sum is then multiplied with a random value $0 \leqslant r < 1$. The sampling result is the index pair whose accumulated value exceeds the calculated random value. In the example, index pair $(3,4)$ is sampled.

since they are not dependent on the indices of the pair to be sampled (i.e. i_2 or j_2 respectively) and therefore, the normalization step would cancel them out anyway.

The determination of which index pairs are compatible with the genotype in the current segment follow a simple fixed rule. The indices are chosen to always set the haplotypes corresponding to the heterozygous sites in each segment in the same way, e.g. index $i = 0$ sets all heterozygous genotypes in the segment to haplotype 0 and index $i = 7$ sets all heterozygous genotypes to haplotype 1. Thus, the pair $(0,7)$ always forms a compatible pair. The complete association list can be found in Tab. 6.1. According to that, the pairs $(0,7)$, $(1,6)$, $(2,5)$, $(3,4)$, $(4,3)$, $(5,2)$, $(6,1)$, and $(7,0)$ are always compatible.

It is also possible to handle *unknown genotypes* in a genotype vector.

Table 6.1. Association of a segment index to a haplotype sequence regarding only heterozygous or unknown genotype sites.

Index	Sequence
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

Throughout the whole computations unknown genotypes can be considered as heterozygous sites since they allow haplotypes 0 or 1 in their haplotype estimations as well. However, an issue arises when considering compatible index pairs because the restriction that both haplotype vectors must contain opposite haplotypes at heterozygous site does not apply for unknown sites, i.e. for an unknown site both haplotypes may be the same. Hence, depending on which sites of the segment are unknown, the list of compatible index pairs grows, e.g. if there were three unknown sites instead of three heterozygous sites in a segment, the list of compatible index pairs contains all 64 possible pairs $(0,0)$, $(0,1)$, ..., $(0,7)$, $(1,0)$, ..., $(7,7)$. Fortunately, for all eight possible occasions of appearances of unknown genotypes in a segment the lists of compatible index pairs is fixed. Thus, each sequence for the sampling process is implemented in a large LUT and can be selected according to the unknown genotypes of a segment.

Memory Utilization and Core Control

The complete computation process follows three main steps which will be explained below.

1. *Initialization.* The host prepares the external FPGA memory including genotype vectors, coefficients, first haplotype estimates and pointers for

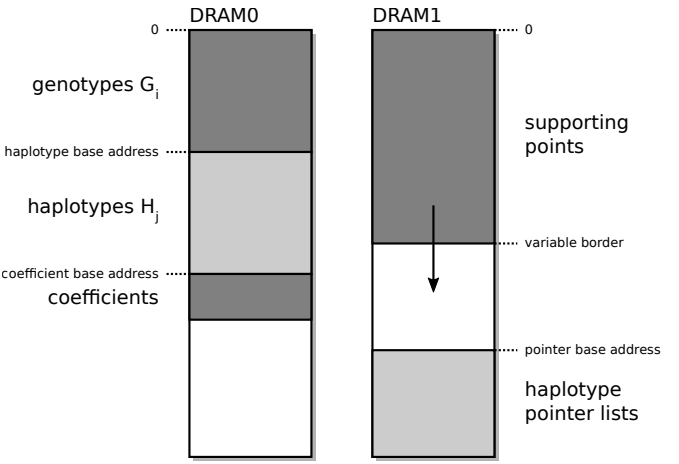


Figure 6.11. Memory organization of the SHAPEIT2 hardware design.

the haplotype subset H for each genotype vector, and provides other necessary constants.

2. *Iteration.* The FPGA iterates over all provided genotype vectors and generates new haplotype estimates based on the current estimates. This process underlies the forward-backward procedure and the sampling of segment indices from which the new haplotype estimates are generated. It is controlled by the core state machine described below.

3. *Results.* After each iteration the host requests the newly generated haplotype estimates and generates a new subset H for each genotype vector for the next iteration. Thus, the FPGA does not need to distinguish between burn-in and main iterations since the host decides whether to discard the current results or not.

The initialization step is only required once for each data set to be phased. Steps 2 and 3 are repeated for the burn-in and main iterations.

Initialization. In the initialization step the host provides the necessary information to the FPGA core and prepares the external memory. As de-

Table 6.2. Encoding of genotypes used in the SHAPEIT2 FPGA core.

Genotype	Encoding	Description
0	0 0	homozygous wild
1	0 1	heterozygous
2	1 0	homozygous variant
3	1 1	unknown

Table 6.3. Encoding of haplotypes used in the SHAPEIT2 FPGA core.

Haplotype	Encoding	Description
0	0	wild
1	1	variant

picted in Fig. 6.11 the two external RAM modules are divided into five parts. The first part contains all genotype vectors. The genotypes are each encoded in 2 bit according to Tab. 6.2 and the vectors are aligned to 64 bit words. The haplotype estimates directly follow the genotypes such that the order of the vectors follows the order of the genotype vectors and each pair of haplotype vectors corresponds to one genotype vector. The haplotype vectors are therefore aligned to 32 bit words. After the initialization the host is allowed to read the current haplotype estimates at any time. However, it will do so only at the end of each iteration. Haplotypes are trivially encoded in 1 bit which is stated in Tab. 6.3. At the end of the first module, the precalculated coefficients $e^{\frac{-\rho_m}{K}}$ and $\frac{1-e^{\frac{-\rho_m}{K}}}{K}$ follow for each site m. The coefficients are encoded in a 32 bit floating point format.

The second module is reserved for the supporting points of the forward-backward procedure. However, aligned to the end of the memory, the pointer lists for the haplotype subsets H are stored for each genotype. Each pointer presents a 32 bit address pointing to the beginning of a haplotype vector in the first module. These lists can continuously be updated by the host.

The constants which have to be provided by the host include the size of

each vector L in the number of sites, the number of genotype vectors N, the number of states K, the total number of iterations to be performed (burn-in and main iterations), and the start addresses for the haplotypes, coefficients and the pointer lists.

Iteration. Each iteration is controlled via the core state machine. A simplified diagram of this state machine can be found in Fig. 6.12. The states can be divided into two main parts. The first part controls the backwards process while the second part controls the forward process and the sampling.

Backward process. The computation starts with the backward process at the last site $L-1$. First of all, the memory buffers load the contents of the last sites for genotypes, haplotypes and coefficients. As already mentioned before, the buffers keep their content up-to-date in the background such that generally no delay due to a buffer update is caused during the computation.

Once the buffers are ready, the computation of the backward variables for the last site starts. The genotype buffer and the coefficient buffer hold the genotype and the coefficients for this site while the haplotype buffer provides a continuous haplotype stream of all haplotypes $h_{u(L-1)}$ of the set H at the last site. The procedure uses eight of the sixteen pipelines since the forward process is yet idle. Each of the pipelines automatically stores its calculated backward variables $\hat{B}_m(i,u)$ and $\hat{B}_m(i,\bullet)$ in their FIFO, while $\hat{B}_m(\bullet,u)$ and $\hat{B}_m(\bullet,\bullet)$ are stored in an extra FIFO. The process iterates over each site m and calculates all backward variables according to Eq. 6.3.51 until the first site $m=0$ is reached.

The computation of the backwards variable of the last site presents a special condition since it does not depend on a previous step. According to Eq. 6.3.46 this condition can be satisfied if the coefficient $e^{\frac{-\rho_L}{K}}=0$ and the other coefficient $\frac{1-e^{\frac{-\rho_L}{K}}}{K}=1$ together with $\hat{B}_L(\bullet,\bullet)=1$ in Eq. 6.3.51. This is manually achieved by providing the required coefficients for this imaginary site L by the host (which stores them together with the other coefficients), and by initializing the register that stores $\hat{B}_m(\bullet,\bullet)$ with 1.

At each change of site the state machine watches the current genotype. If the third heterozygous or unknown genotype is observed, a segment border

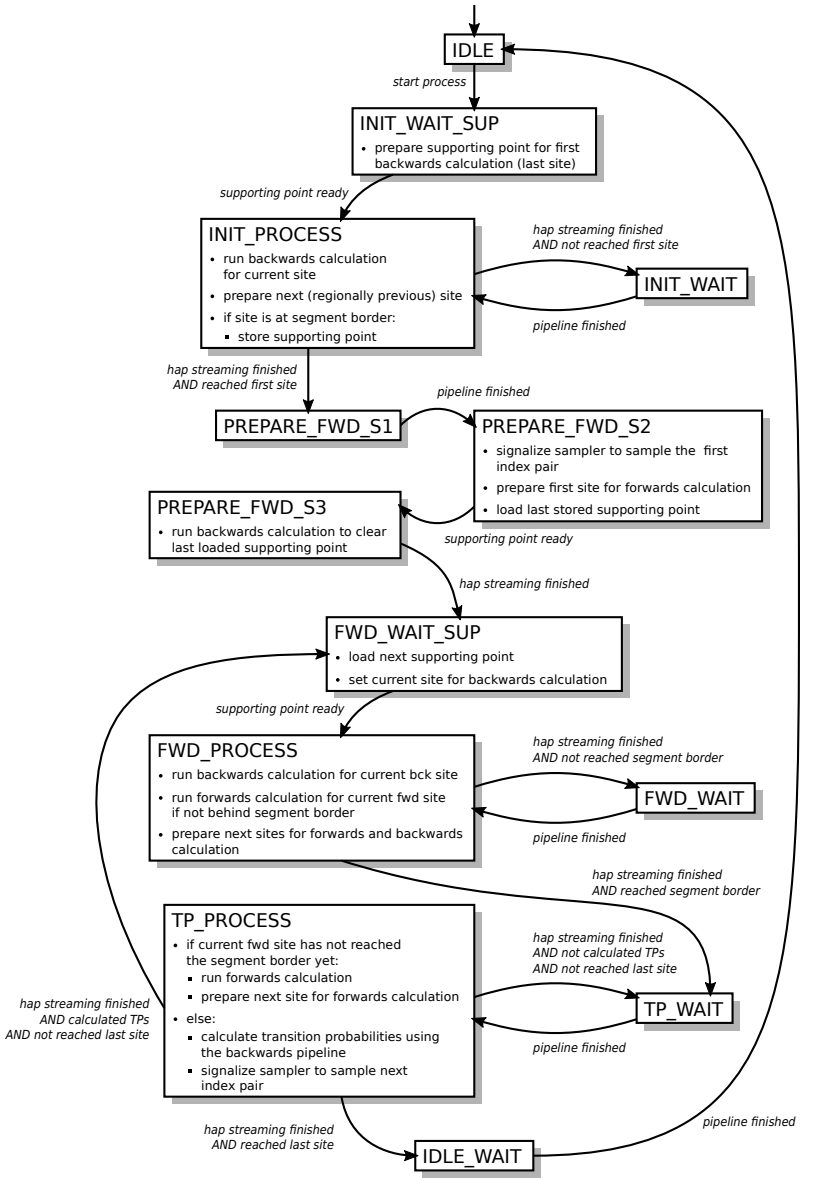


Figure 6.12. Simplified diagram of the core state machine.

is reached and the process prepares the storage of a supporting point, i.e. the variables $\hat{B}_m(\bullet, u)$ and $\hat{B}_m(\bullet, \bullet)$ together with the site information are buffered and written to the supporting point memory afterwards.

In order to satisfy the limited capacity of the haplotype buffer the segment size is bounded by at most 64 sites. If the 64th site of a segment is reached, the process manually inserts a segment border and stores a supporting point as if three heterozygous sites have occurred. This influences the results only in that way that for this segment several indices may define the same haplotype sequence.

Forward process and sampling. Once the backward process has finished the first site $m = 0$ the backward pipelines provide the variables $\hat{B}_0(i, \bullet)$. These values are used to sample the first index pair according to Eqs. 6.3.61 and 6.3.82. In the meantime, the last stored supporting point is loaded and discarded since it supports only the recalculation of the first segment which is not required anymore.

Then, the next supporting point is loaded and the backward process starts to recalculate the second segment. Concurrently, the forward process starts to compute the forward variables $\hat{a}_m(i, u)$ for the first segment according to Eq. 6.3.12 using the eight forward pipelines. Now, all 16 pipelines are utilized in parallel.

As for the last site in the backward process, the first site $m = 0$ states a special condition for the forward process according to Eq. 6.3.9. This condition can be satisfied if the coefficients $e^{\frac{-\rho_0}{K}} = 0$ and $\frac{1-e^{\frac{-\rho_0}{K}}}{K} = \frac{1}{K}$ together with $\hat{a}_{-1}(\bullet, \bullet) = 1$ in Eq. 6.3.12. Again, these coefficients for the site $m = 0$ are provided by the host and stored together with the other coefficients, and the register that stores $\hat{a}_m(\bullet, \bullet)$ is initialized with 1.

In order to sample the index pair for the next segment, the forward and the backward process have to meet at the segment border. Therefore, the process that reaches the border first has to wait for the other process before continuing. At the segment border the values $\hat{a}_m(i, u)$ as well as $\hat{B}_{m+1}(i, u)$ and $\hat{B}_{m+1}(i, \bullet)$ have been calculated. Now, the backward pipelines are utilized to calculate the numerator of the transitional probability as stated in Eq. 6.3.64. Note that according to Eq. 6.3.83 the denominator does not

need to be computed (see paragraph on *Sampling* for details). The process selects the appropriate forward variables $\hat{a}_m(i_1, u)$ in accordance with the previously sampled index i_1 from the pair (i_1, j_1) for the current segment and replaces λ or $1 - \lambda$ with these variables during the calculation. In parallel, the variables $\hat{a}_m(j_1, u)$ are selected to be used with the other parallel multiplier in the backwards pipelines. Finally, a new index pair (i_2, j_2) is sampled according to Eq. 6.3.83. Please refer back to the paragraph on *The Computation Pipelines* for details on the calculations.

Results. Concurrently to the computation process an entity generates two streams of haplotypes according to the current genotype and sampled index pairs. The streams are buffered in a separate FIFO each and whenever the memory is ready, the current haplotype estimates for the genotype are updated with the contents of the FIFOs. The host is able to directly read this part of the memory with the current estimates and does so after each iteration when all estimates where updated by processes described above. It updates H for each genotype by finding the K best fitting estimates according to the paragraph on *Choosing a Subset of K Haplotype Vectors* in Sect. 6.3.1. The corresponding pointers to these vectors are directly written into the reserved space for H in the second RAM module and the next iteration starts.

Depending on whether the FPGA processes a burn-in or main iteration, the host discards the estimates after determining H or it keeps them for each iteration to average them all for a final result.

Computational Costs

Due to the introduction of several computation pipelines working in parallel in each FPGA core, the runtime complexity gets independent of the number of labels in each segment. Furthermore, the runtime costs are reduced by processing as many windows concurrently as possible, i.e. preferably the number of windows should not exceed the number of available FPGA cores.

In this case, the total runtime complexity can be estimated to be in

$$\mathcal{O}\left(\frac{I \cdot N \cdot L \cdot K}{W}\right)$$

with W denoting the number of windows and I denoting the number of iterations.

Furthermore, the memory complexity has been reduced by introducing supporting points. The input data set as well as the haplotype estimates for each iteration still have to be kept in memory. However, all calculated backward variables but the supporting points can be discarded. Since the supporting points are located at each segment border, the calculation of the next backward variable depends only on the accumulated sum of backward probabilities from the previous site (see Eq. 6.3.51). Thus, the memory complexity for the supporting points only depends on the number of segments C and the number of states K. Therefore, the total memory complexity can be estimated to be in

$$\mathcal{O}(I \cdot N \cdot L + C \cdot K)$$

whereby the host system only needs to store the result of each iteration, i.e.

$$\mathcal{O}(I \cdot N \cdot L).$$

An FPGA only needs to store its current window and the supporting points, such that under the assumption of an equal distribution the memory complexity of a single FPGA can be approximated with

$$\mathcal{O}\left(\frac{N \cdot L + C \cdot K}{W}\right).$$

Evaluation

FPGA-based SHAPEIT2 has been evaluated against SHAPEIT2 release 837 on a PC system equipped with an Intel Core i7 4790K quad-core CPU clocked at 4 GHz and 32 GB RAM runnning 64bit Ubuntu Linux 15.04. The targeted FPGA system was the RIVYERA S6-LX150 with 128 Xilinx

Table 6.4. Device utilization of the SHAPEIT2 implementation on a Spartan6-LX150.

	Occupied Slices	Slice Registers	Slice LUTs	Block RAM (18k)	DSP48A1
Used	19,113	42,893	58,282	92	168
Available	23,038	184,304	92,152	268	180
Utilization	82%	23%	63%	34%	93%

Table 6.5. Test scenarios used for evaluating FPGA-based SHAPEIT2.

Scenario	1	2	3	4	5
Chromosomes	all	all	1, 9, 22	9, 13, 22	1, 9, 22
# states (K)	100	500	100, 200, 500, 1,000	100	100
window size (W in Mb)	2	2	2	1, 2, 4, 8	2
# burn-in	7	7	7	7	7, 12
# prune	8/0	8/0	8/0	8/0	8/0
# main	20/28	20/28	20/28	20/28	20/28, 25/33
population size (N_e)	11,418				
# threads (PC only)	4				

Spartan6-LX150 FPGAs. The resources of an FPGA were almost consumed completely, i.e. 82% of the slices were occupied and 93% of the device's DSPs were used (see Table 6.4 for more details). The core frequency of the design was 100 MHz.

The test input data was a set of genotypes from 1,108 samples genotyped at a total of 741,560 markers distributed over the chromosomes 1 to 22. The set was divided into 22 subsets, one for each chromosome. The number of markers for each set range from 9,496 on chromosome 22 to 63,192 on chromosome 2.

Test scenarios. Five test scenarios were processed equally on the PC and on the RIVYERA system. The first scenario involves phasing all 22 test sets with SHAPEIT2 default parameters, i.e. $K = 100$ states, a window size of $W = 2$Mb, 7 burn-in iterations, 8 pruning iterations and 20 main iterations. Since the FPGA version does not support pruning, the number of pruning iterations has been added to the main iterations on the FPGAs. Thus, both systems perform a number of 35 iterations.

The second scenario phases all 22 test sets again with the same parameters, but with an increased number of states, i.e. $K = 500$.

The third and the fourth scenario observe the impact on changing the number of states or the window size on selected chromosomes while keeping the other parameters at default. The runtimes were measured for chromosome 1, 9 and 22 with $K = 100$, 200, 500 and 1,000 states and other parameters at default (Scenario 3), and for chromosomes 9, 13 and 22 for window sizes $W =$ 1Mb, 2Mb, 4Mb and 8Mb with other parameters at default as well (Scenario 4).

Scenario 5 observes the runtime change when changing the number of iterations, i.e. for chromosomes 1, 9 and 22 the runtimes were measured firstly for 7 burn-in and 20 main iterations, secondly for 12 burn-in and 20 main iterations, thirdly for 7 burn-in and 25 main iterations, and finally for 12 burn-in and 25 main iterations. The number of pruning iterations were kept at 8 for the PC system, or respectively added 8 main iterations for the FPGA system.

For all scenarios the effective population size was set to $N_e = 11,418$ to match the European ancestry of the samples, and the number of threads for the PC system was set to 4 to match all four available CPU cores. Table 6.5 gives an overview over the parameters in the different scenarios.

Results. For Scenarios 1 and 2 the runtimes are listed in Tables 6.6 and 6.7 respectively. The results show that for the larger chromosomes 1 to 13 the speedup for the FPGA system in Scenario 1 is relative constant at 17 to 19. It declines down to 6.5 for the smaller chromosomes. Similar results shows Scenario 2 with the highest speedup of 46.4 for chromosome 2. This indicates that the more FPGAs are involved in solving the problem and the more states are used for the HMM implying longer calculations in each

Table 6.6. Runtimes (in seconds) and Speedup for FPGA-based SHAPEIT2 vs. SHAPEIT2 in Scenario 1 ($K = 100$).

Set	# markers	SHAPEIT2 software	FPGA-based SHAPEIT2	# FPGAs used	FPGA-Speedup
chr1	60,507	7,514	420	105	17.89
chr2	63,192	7,710	458	112	16.83
chr3	52,028	6,033	317	92	19.03
chr4	47,945	5,523	373	89	14.81
chr5	48,399	5,619	311	83	18.07
chr6	48,352	5,486	308	79	17.81
chr7	40,248	4,552	239	72	19.05
chr8	41,776	4,675	276	67	16.94
chr9	35,279	3,943	215	53	18.34
chr10	41,235	4,529	254	62	17.83
chr11	37,960	4,210	239	62	17.62
chr12	36,152	3,980	230	61	17.30
chr13	29,435	3,102	179	45	17.33
chr14	23,893	2,504	180	40	13.91
chr15	22,141	2,359	160	39	14.74
chr16	23,488	2,553	221	37	11.55
chr17	17,232	1,937	155	37	12.50
chr18	22,592	2,491	168	36	14.83
chr19	9,887	1,139	123	25	9.26
chr20	19,608	2,150	176	29	12.22
chr21	10,715	1,165	159	17	7.33
chr22	9,496	1,026	157	16	6.54

iteration, the higher is the speedup of the FPGA system. However, Fig. 6.13 shows the speedups over the number of used FPGAs. It reveals, that the speedup stagnates if more than 40 FPGAs are used if the number of states is $K = 100$. This effect is underlined by the results from Scenario 3.

The results of Scenario 3 are listed in Table 6.8. Compared to Scenarios 1 and 2, it shows additional speedups for $K = 200$ and $K = 1,000$ on chromosomes 1, 9 and 22. Figure 6.14 illustrates these speedups indicating an increase with an ascending number of states, but stagnation for a larger number of states. The smaller speedups for less states are caused by the time

Table 6.7. Runtimes (in seconds) and Speedup for FPGA-based SHAPEIT2 vs. SHAPEIT2 in Scenario 2 ($K = 500$).

Set	# markers	SHAPEIT2 software	FPGA-based SHAPEIT2	# FPGAs used	FPGA-Speedup
chr1	60,507	28,237	634	105	44.54
chr2	63,192	29,401	633	112	46.45
chr3	52,028	25,036	602	92	41.59
chr4	47,945	23,379	542	89	43.13
chr5	48,399	22,338	544	83	41.06
chr6	48,352	21,992	578	79	38.05
chr7	40,248	19,125	523	72	36.57
chr8	41,776	19,334	541	67	35.74
chr9	35,279	16,488	572	53	28.83
chr10	41,235	18,626	589	62	31.62
chr11	37,960	17,500	579	62	30.22
chr12	36,152	16,673	550	61	30.31
chr13	29,435	13,404	530	45	25.29
chr14	23,893	11,146	504	40	22.12
chr15	22,141	10,438	505	39	20.67
chr16	23,488	11,174	508	37	22.00
chr17	17,232	8,398	471	37	17.83
chr18	22,592	10,501	522	36	20.12
chr19	9,887	4,684	359	25	13.05
chr20	19,608	9,277	554	29	16.75
chr21	10,715	5,109	467	17	10.94
chr22	9,496	4,550	415	16	10.96

required by the host software to fetch the results and to update the current estimates in H for each FPGA. If this time is higher than the execution time on the FPGAs it causes the FPGAs to remain in an idle state for longer. The almost equal runtimes for $K = 100$ and $K = 200$ on the FPGA system for chromosome 1 highlight this behavior and reveal that the host computation and communication requires more time than the FPGA computation in these cases.

The results from Scenario 4 (see Table 6.9) show that the SHAPEIT2 software is relative unaffected by changing the window size. However, since

Table 6.8. Runtimes (in seconds) and Speedup for FPGA-based SHAPEIT2 vs. SHAPEIT2 in Scenario 3 (different number of states (K) for chromosome 1, 9 and 22 with 60,507, 35,279 and 9,496 markers respectively).

Set	# states (K)	SHAPEIT2 software	FPGA-based SHAPEIT2	# FPGAs used	FPGA-Speedup
chr1	100	7,514	420	105	17.89
	200	12,725	423	105	30.08
	500	28,237	634	105	44.54
	1,000	49,408	1,089	105	45.37
chr9	100	3,943	215	53	18.34
	200	6,790	324	53	20.96
	500	16,488	572	53	28.83
	1,000	28,851	1,084	53	26.62
chr22	100	1,026	157	16	6.54
	200	1,901	218	16	8.72
	500	4,550	415	16	10.96
	1,000	8,523	901	16	9.46

the window size directly determines the total number of windows for the data set to be processed, and the windows are equally distributed over the available FPGAs in the FPGA-based solution, a large window size implies less FPGAs to be used resulting in longer runtimes for the FPGA system. Furthermore, smaller windows reduce the load for each FPGA such that communication and calculation overhead on the host gains more influence on the runtime. Thus, runtimes get worse with larger windows, but show a slight increase for very small windows as well. Chromosomes 9 and 13 were exemplarily chosen since they nearly use all available FPGAs for a window size of 1Mb. Additionally, chromosome 22 was chosen for being the smallest. The runtimes are depicted as a graph in Fig. 6.15.

Scenario 5 reveals, as expected, that increasing the number of iterations increases the runtime in the same manner. This is likewise for the software and the FPGA-based solution. However, a closer look at the results indicates that increasing the number of burn-in iterations affects the runtime in software-based SHAPEIT2 stronger than increasing the main iterations,

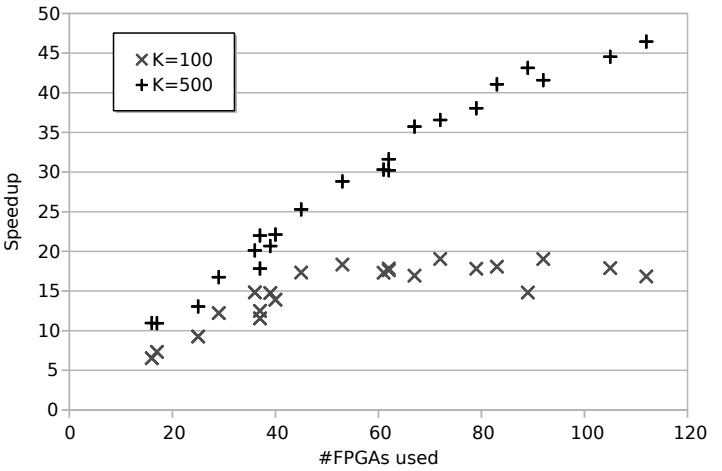


Figure 6.13. Speedups of FPGA-based SHAPEIT2 vs. SHAPEIT2 in Scenarios 1 ($K = 100$) and 2 ($K = 500$) over the number of used FPGAs.

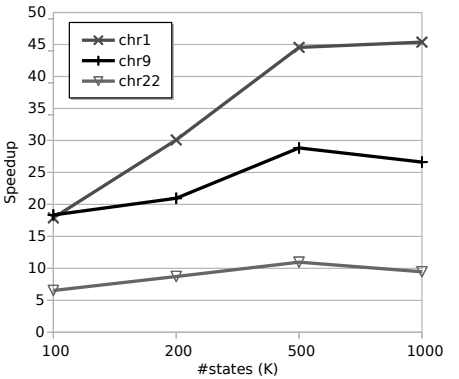


Figure 6.14. Speedup of FPGA-based SHAPEIT2 vs. SHAPEIT2 in Scenario 3 for chromosomes 1, 9 and 22.

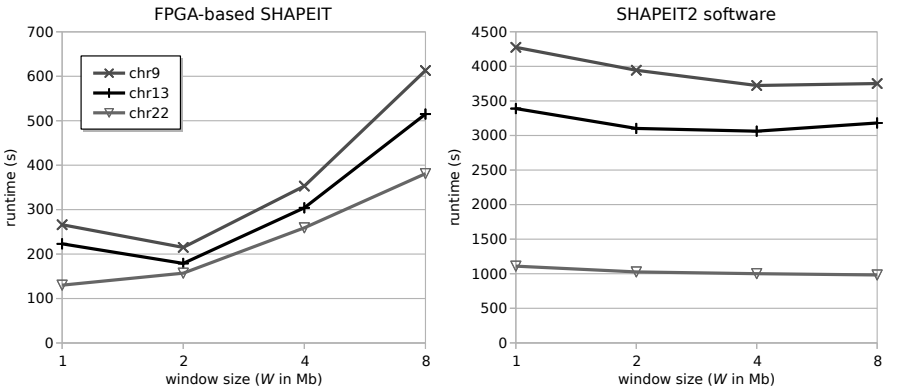


Figure 6.15. Runtimes of FPGA-based SHAPEIT2 and SHAPEIT2 in Scenario 4 for chromosomes 9, 13 and 22.

Table 6.9. Runtimes (in seconds) and Speedup for FPGA-based SHAPEIT2 vs. SHAPEIT2 in Scenario 4 (different window sizes (W) for chromosomes 9, 13 and 22 with 35,279, 29,435 and 9,496 markers respectively).

Set	window size (W)	SHAPEIT2 software	FPGA-based SHAPEIT2	# FPGAs used	FPGA-Speedup
chr9	1Mb	4,275	266	99	16.07
	2Mb	3,943	215	53	18.34
	4Mb	3,723	353	28	10.55
	8Mb	3,751	613	15	6.12
chr13	1Mb	3,389	223	87	15.20
	2Mb	3,102	179	45	17.33
	4Mb	3,062	304	24	10.07
	8Mb	3,181	515	9	6.18
chr22	1Mb	1,110	130	29	8.54
	2Mb	1,026	157	16	6.54
	4Mb	1,001	259	9	3.86
	8Mb	982	381	5	2.58

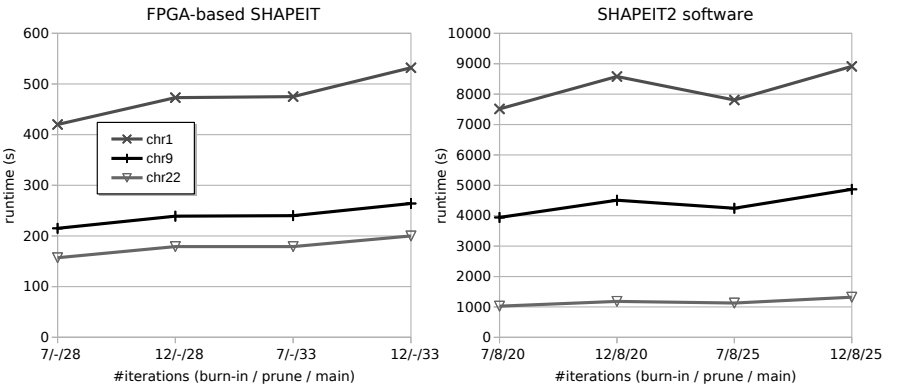


Figure 6.16. Runtimes of FPGA-based SHAPEIT2 and SHAPEIT2 in Scenario 5 for chromosomes 1, 9 and 22.

while this effect is not visible for the FPGA-based solution. The reason is the pruning step between burn-in and main iterations for the software solution. It simplifies the internal Hidden Markov model such that main iterations can be processed faster. This step is omitted in the FPGA-based solution. Table 6.10 shows the runtimes and speedups for this test scenario. In Fig. 6.16 the runtimes are depicted as a graph underlining the previously mentioned effect.

Conclusion. Concluded, the total runtime for all datasets for the SHAPEIT2 software was almost one day at default parameters and more than four days for $K = 500$ states. The FPGA-based solution was able to analyze all datasets subsequently in about 1 hour and 28 minutes, and 3 hours and 15 minutes respectively, resulting in a total speedup of 15.83 and 29.59. The highest speedup in this evaluation was 46.44. It was gained with the largest data set (chromosome 2 with 63,192 markers) at $K = 500$ states, windows size $W = 2$Mb and 112 utilized FPGAs. In numbers, this setup reduces the runtime from 8 hours and 10 minutes on a standard PC to only 10 minutes and 33 seconds on the RIVYERA system. However, the different test scenarios revealed, that the speedup of the FPGA-based implementation

Table 6.10. Runtimes (in seconds) and Speedup for FPGA-based SHAPEIT2 vs. SHAPEIT2 in Scenario 5 (different number of burn-in and main iterations for chromosomes 1, 9 and 22 with 60,507, 35,279 and 9,496 markers respectively).

Set	# iterations (b / p / m) software	FPGA	SHAPEIT2 software	FPGA-based SHAPEIT2	# FPGAs used	FPGA-Speedup
chr1	7/8/20	7/-/28	7,514	420	105	17.89
	12/8/20	12/-/28	8,580	473	105	18.14
	7/8/25	7/-/33	7,807	475	105	16.44
	12/8/25	12/-/33	8,913	532	105	16.75
chr9	7/8/20	7/-/28	3,943	215	53	18.34
	12/8/20	12/-/28	4,510	239	53	18.87
	7/8/25	7/-/33	4,246	240	53	17.69
	12/8/25	12/-/33	4,870	264	53	18.45
chr22	7/8/20	7/-/28	1,026	157	16	6.54
	12/8/20	12/-/28	1,182	179	16	6.60
	7/8/25	7/-/33	1,128	179	16	6.30
	12/8/25	12/-/33	1,320	200	16	6.60

is highly sensitive to the runtime parameters and the nature of the test set. Thus, running FPGA-based SHAPEIT2 on the chromosome 22 data set with 9,496 markers, $K = 100$ states and window size $W = 8$Mb utilizes only 5 FPGAs, and therefore results in a speedup of only 2.58.

Consequently, the FPGA-based solution is most suitable for data sets that include a large number of markers such that as many FPGAs as possible are utilized. This includes the selection of the window size as well since the larger the window size is the less FPGAs can be used. Furthermore, the workload of the FPGAs has to be high enough for each iteration such that communication overhead and processor workload of the host system do not become the dominant factors. This can be achieved by selecting a higher number of states such as $K = 500$.

Of course, the workload of an FPGA cannot be increased infinitely. However, due to the low memory requirements for the FPGA-based solution, a single FPGA would require only slightly more than 100MB of DRAM

if it had to process the largest dataset in this evaluation (chromosome 2) with $K = 1,000$ states alone. Since each FPGA features 512MB of DRAM, the problem size of the dataset could still be increased approximately by a factor of 5.

A major disadvantage of the FPGA-based solution is the low speedup at small datasets. This results from the incomplete utilization of all available FPGAs. One suggestion would be to modify the host software such that it is able to analyze more than one dataset at once for a better utilization of the FPGAs. For larger datasets the speedup could be improved by equally distributing the number of markers to all FPGAs as well. For now, the number of markers is bound to each window since each window has to be analyzed independently, and the window size is specified as a genomic size, i.e. megabases (Mb), and not in the number of markers such that the number of markers in each window will be different. However, calculating with different window sizes such that the number of markers will be equally distributed will be a major aberration to the original software and may result in quality reductions. Thus, analysis and improvement of the workload situation is subject to future work.

6.4 Imputation of Phased Haplotypes

Minimac2 [FAH15], Minimac3 [Abe] and IMPUTEv2 [HDM09] present recent tools which are generally used to impute unobserved genotypes in a study using the phase information determined by SHAPEIT2 in the phasing process. Minimac2 significantly improves the original tool Minimac [HFS+12] by introducing vectorization in its implementation, but uses the same mathematical methods. IMPUTEv2 [HDM09] as successor of the IMPUTE tool [MHM+07] introduces a new technique that switches between phasing and imputation in each iteration, but can be used to impute the missing genotypes from the phased haplotypes in SHAPEIT2 as well.

Fig. 6.17 illustrates the problem of untyped genotypes in a study. The sampled individuals may miss genotype information at a number of markers which might be necessary for further investigation e.g. on the relation of genetics to certain illnesses. This information might be available for samples

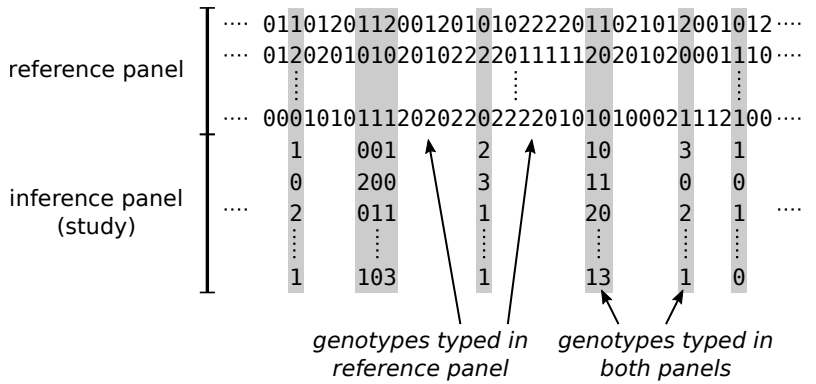


Figure 6.17. Typed and untyped genotype information in a study and in a reference panel.

in a reference panel, such as panels by the *Haplotype Reference Consortium (HRC)* [HRC]. In order to project the information of the reference panel to information on untyped markers in the study, most imputation methods look for perfect or near perfect matches between the phased haplotypes in the study and the corresponding haplotypes at the same markers in the reference panel. It is then assumed that for those candidates the unknown haplotypes would also match the corresponding haplotypes in the reference at the untyped markers in the study. This states the fundamental basis of genotype imputation [HDM09].

Although the IMPUTE tool [MHM+07] does not require haplotype phasing for imputation, the underlying mathematical models form the basis of all before mentioned imputation tools. Thus, IMPUTE serves as an example here to explain the mathematical background of the imputation process based on a Hidden Markov model.

However, the FPGA implementation of an imputation tool remains subject to future work.

6.4.1 The IMPUTE Algorithm

In order to determine the missing information at untyped markers, imputation works similar to the phasing process but uses a reference set of haplotypes instead of current haplotype estimates from several iterations. The IMPUTE tool creates a similar HMM as in SHAPEIT2 with the genotypes as emissions and the haplotypes in the reference set as states. Other tools may choose a subset of haplotypes from the reference according to the phased haplotypes or include the haplotypes in the reference set, but not using them as states. The emissions may also be converted to diplotypes, i.e. a pair of haplotypes, as it is done e.g. in IMPUTEv2.

Based on the HMM IMPUTE calculates the marginal probability of each possible genotype `0`, `1`, or `2`, for each untyped marker from which the actual genotype estimation will be sampled. The tool processes windows of 10Mb per default. For a better understanding, the following notations are chosen to be similar to those adapted from the SHAPEIT2 phasing process in Sect. 6.3.1.

Notations

Following notations will be used throughout this section to explain the IMPUTE application.

N: no. of unrelated individuals in the input dataset (study).

L: no. of markers in the reference panel in the current window.

K: no. of haplotype vectors in the reference panel, i.e. the number of states in the HMM.

$G = \{G_0, \ldots, G_{N-1}\}$*:* all genotype vectors for all individuals in the study.

$G_i = \{g_{i0}, \ldots, g_{i(L-1)}\}$*:* ith genotype vector of the input data set, $g_{im} \in \{0, 1, 2, 3\}$. States the emissions of the HMM.

$H = \{H_0, \ldots, H_{K-1}\}$*:* the haplotypes from the reference panel.

$H_k = \{h_{k0}, \ldots, h_{k(L-1)}\}$*:* kth haplotype vector, $h_{km} \in \{0, 1\}$.

$Z_i = \{Z_{i0}, \ldots, Z_{i(L-1)}\}$: unobserved (hidden) states in the HMM for each site in the ith genotype vector, $Z_{im} \in \{0, \ldots, K-1\}$ denotes the "copied" haplotype at site m.

$\rho_m = 4N_e r_m$: r_m is the *per generation genetic distance* between sites m and $m-1$. The distance is determined from a *genetic map* the user has to provide along with the input data. The genetic map contains the positions of each marker in the genome. N_e is the effective population size and set per default to $N_e = 11,418$.

$\lambda = \frac{\theta}{2(\theta+K)}$: a constant indicating the mutation rate, $\theta = \left(\sum_{i=1}^{K-1} \frac{1}{i}\right)^{-1}$.

Modeling the HMM

The goal in the IMPUTE application is to calculate the joint distribution of observed and missing genotype data in the input data set. Furthermore, it is assumed that each genotype vector can be estimated independently of others. This can be expressed mathematically as follows. Let G be partitioned into an observed component G_O and a missing component G_M such that $G = \{G_O, G_M\}$. Then,

$$P(G_M|G_O, H) \propto P(G_M, G_O|H) = P(G|H) = \prod_{i=0}^{N-1} P(G_i|H). \tag{6.4.1}$$

Now, each individuals genotype vector, i.e. $P(G_i|H)$, is modeled with a HMM where the states are a sequence of pairs of haplotypes from the reference set that are "copied" for each location to form the genotype which is considered as the emission of the HMM. Let $Z_i^{(1)} = \{Z_{i0}^{(1)}, \ldots, Z_{i(L-1)}^{(1)}\}$ and $Z_i^{(2)} = \{Z_{i0}^{(2)}, \ldots, Z_{i(L-1)}^{(2)}\}$ be sequences of haplotypes. The marginal probability can then be calculated as

$$P(G_i|H) = \sum_{Z_i^{(1)}, Z_i^{(2)}} P\left(G_i \left| Z_i^{(1)}, Z_i^{(2)}, H\right.\right) P\left(Z_i^{(1)}, Z_i^{(2)} \left| H\right.\right) \tag{6.4.2}$$

$P(Z_i^{(1)}, Z_i^{(2)}|H)$ models how the sequence changes along the sequence and $P(G_i|Z_i^{(1)}, Z_i^{(2)}, H)$ models how the haplotypes are copied to form the genotype vector G_i. The genotypes will not result directly from the copied haplotypes since this process mimics the effects of mutation and therefore takes a mutation probability into account (see below).

For the sequence changes a Markov chain is modeled as follows. The initial state is uniform on all possible K^2 states:

$$P\left(Z_{i0}^{(1)}, Z_{i0}^{(2)} \middle| H\right) = \frac{1}{K^2} \tag{6.4.3}$$

The transition probabilities from site $m-1$ to site m are given by

$$P\left(Z_{im}^{(1)}, Z_{im}^{(2)} \middle| Z_{i(m-1)}^{(1)}, Z_{i(m-1)}^{(2)}, H\right) = \begin{cases} \left(e^{\frac{-\rho_m}{K}} + \frac{1-e^{\frac{-\rho_m}{K}}}{K}\right)^2 \\ \qquad \text{if} \quad Z_{im}^{(1)} = Z_{i(m-1)}^{(1)}, Z_{im}^{(2)} = Z_{i(m-1)}^{(2)} \\ \left(e^{\frac{-\rho_m}{K}} + \frac{1-e^{\frac{-\rho_m}{K}}}{K}\right)\left(\frac{1-e^{\frac{-\rho_m}{K}}}{K}\right) \\ \qquad \text{if} \quad Z_{im}^{(1)} = Z_{i(m-1)}^{(1)}, Z_{im}^{(2)} \neq Z_{i(m-1)}^{(2)} \\ \qquad \text{or} \quad Z_{im}^{(1)} \neq Z_{i(m-1)}^{(1)}, Z_{im}^{(2)} = Z_{i(m-1)}^{(2)} \\ \left(\frac{1-e^{\frac{-\rho_m}{K}}}{K}\right)^2 \\ \qquad \text{if} \quad Z_{im}^{(1)} \neq Z_{i(m-1)}^{(1)}, Z_{im}^{(2)} \neq Z_{i(m-1)}^{(2)} \end{cases} \tag{6.4.4}$$

Overall, the prior distribution on the hidden states follows as

$$P\left(Z_i^{(1)}, Z_i^{(2)} \middle| H\right) = P\left(Z_{i0}^{(1)}, Z_{i0}^{(2)} \middle| H\right) \prod_{m=1}^{L-1} P\left(Z_{im}^{(1)}, Z_{im}^{(2)} \middle| Z_{i(m-1)}^{(1)}, Z_{i(m-1)}^{(2)}, H\right) \tag{6.4.5}$$

Table 6.11. IMPUTE emission probabilities $P(G_{im}|Z_{im}^{(1)}, Z_{im}^{(2)}, H)$ of mutating the derived genotype by copying the haplotypes from the two states $Z_{im}^{(1)}$ and $Z_{im}^{(2)}$ to the observed genotype G_{im}.

		G_{im}		
		0	1	2
$H_{Z_{im}^{(1)}m} + H_{Z_{im}^{(2)}m}$	0	$(1-\lambda)^2$	$2\lambda(1-\lambda)$	λ^2
	1	$\lambda(1-\lambda)$	$\lambda^2 + (1-\lambda)^2$	$\lambda(1-\lambda)$
	2	λ^2	$2\lambda(1-\lambda)$	$(1-\lambda)^2$

The genotype distribution is given by

$$P\left(G_i \left| Z_i^{(1)}, Z_i^{(2)}, H\right.\right) = \prod_{m=0}^{L-1} P\left(G_{im} \left| Z_{im}^{(1)}, Z_{im}^{(2)}, H\right.\right) \tag{6.4.6}$$

with the emission probabilities $P(G_{im}|Z_{im}^{(1)}, Z_{im}^{(2)}, H)$ defined according to Table 6.11. These emission probabilities take the mutation probability λ into account that an allele mutates to its complementary allele. The underlying assumption is that the mutations are independent on both haplotypes and across sites.

Given these equations it is possible to determine the overall distribution $P(G|H)$. With these probabilities genotypes can be sampled for the untyped sites in the input data set. If the emissions are sampled with an iterative Gibb's sampling approach or if simply the genotype is taken with the highest probability depends on the user. The authors of IMPUTE even suggest to use the probabilities directly for further tests, such as association tests [MHM+07].

Chapter 7

Conclusion

Summary

This thesis concentrates on the hardware acceleration of common compute intensive problems related to the area of bioinformatics. The implementation on FPGA hardware has remarkable advantages over the software implementation targeting CPUs. Though CPUs feature higher clock frequencies in general, they are bound to a fixed instruction set, and in order to process each instruction the CPU has to hold and support the required resources. This may lead to an uneffective resource usage and results in longer runtimes and higher energy consumption. By exploiting FPGA hardware, this thesis has shown that the hardware design can be perfectly optimized and adapted to the computational problem effectively using the available resources. It was possible to implement the compute intensive part of a problem in small processing elements requiring only a fraction of the FPGAs available resources. Thus, it was able to utilize many processing elements in a single FPGA design. Since the resources are completely configurable, pipelines could be successfully implemented with an almost arbitrary depth. Together with the implementation of several processing elements, a high-degree of fine-grained parallelism was achieved even on single FPGAs, resulting in low execution times and low energy consumption. By harnessing the special FPGA-based architecture RIVYERA S6-LX150 with 128 Spartan6 FPGAs an additional speedup was accomplished through implicit coarse-grained parallelization over several FPGAs in all targeted applications.

Three major topics in bioinformatics were addressed in this thesis: sequence alignment, SNP interaction detection and genotype imputation. In sequence alignment the popular protein sequence alignment tool

BLASTp [AGM+90; AMS+97] was implemented on the RIVYERA architecture to speedup protein database lookups. With two pipelines on one FPGA each containing three necessary application steps *hit finding*, *two-hit finding* and *ungapped extension*, together with a *gapped extension* filter for both pipelines, the RIVYERA setup reached a speedup of up to 23.5 when compared to the original NCBI BLASTp software [NCBa] running on sixteen threads on two Intel Xeon E5520 quad-core CPUs.

The SNP interaction detection problem has been addressed in this thesis by the implementation of several tools. The popular software tools BOOST [WYY+10a] and iLOCi [PNI+12] detect pair-wise interactions in *Genome-wide Association Studies (GWAS)*. A statistical test, i.e. measuring the *Kullback-Leibler divergence* in BOOST and a self created ρ-distance in iLOCi, on each contingency table created for all possible marker pairs of the underlying case-control study reveals possible interaction candidates. Both tools require about 19 hours for interaction detection on an Intel Core i7 hexa-core CPU (BOOST) respectively two Intel Xeon quad-core CPUs (iLOCi) for a *WTCCC* dataset [WTCCC07] containing 500,000 markers and 5,000 samples. In contrast, the presented FPGA designs on the RIVYERA system finished the same analysis in below 6 minutes for BOOST and below 3 minutes for iLOCi, resulting in speedups of 214 and 492 respectively. The BOOST design was implemented with 80 processing elements in one chain on each Spartan6-FPGA for the creation of contingency tables, while the length of the process element chain for iLOCi was 123 PEs.

Furthermore, the BOOST design has been adapted for the Kintex7-325T FPGA on the KC705 development board. Here, 108 process elements were distributed over 4 chains. The clock frequency and the genotype throughput were also increased by factors 2.5 and 4 respectively, resulting in a speedup of 13.5 when compared to a single Spartan6-LX150 FPGA of the RIVYERA architecture.

Furthermore, it is possible to apply the *mutual information* measure to any order of SNP combinations. This thesis demonstrated how to apply this measure to all possible third-order SNP combinations in a case-control study. The implementation targeted the Kintex7 FPGA of the KC705 board as well. 102 processing elements distributed over 6 chains gained a performance speedup of 182 compared to an Intel Core i7 hexa-core CPU. A dataset with

10,000 SNPs and 5,000 samples could be analyzed in only 1 hour and 8 minutes.

The third topic genotype imputation was addressed by the adaption and implementation of the haplotype phasing tool SHAPEIT2 [DZM13]. From an input dataset containing genotypes from a study, the goal of genotype imputation is to impute missing genotypes at untyped positions according to a reference panel. The process is generally divided into two steps. The phasing step performed by SHAPEIT2 is necessary to increase the result quality of the imputation part in genotype imputation. It is based on a *Hidden Markov model (HMM)* and the forward-backward procedure iteratively applied on the input genotypes. The implementation of SHAPEIT2 was realized using 8 forward and 8 backward pipelines on one Spartan6 FPGA. Even with a high number of states (e.g. parameter $K = 1,000$), this configuration would be able to phase several hundred thousand genotypes distributed over all samples on a single FPGA. However, in order to reduce runtime, the genotypes are divided into windows and processed separately distributed over the available FPGAs in the RIVYERA system for a coarse-grained parallelization. This resulted in a significant speedup, e.g. for a dataset containing 63,192 markers in 1,108 samples, a window size of 2Mb and $K = 500$ states, the resulting speedup was 46.45 when compared to the original SHAPEIT2 software on an Intel Core i7 quad-core CPU at 4 GHz with four threads.

The imputation step requires a reference panel to determine the genotypes at untyped positions, but again, applies a HMM for the underlying data structure. Imputation was explained on the basis of the simple tool IMPUTE [MHM+07]. However, its implementation remains subject to future work.

Concluded, this thesis has demonstrated how reconfigurable hardware based on FPGAs can help to significantly speedup compute intensive problems in the area of bioinformatics. FPGA hardware is able to support daily work of biomedicals and improves the workflow by either faster runtimes or the ability to solve problems in an order which has not been possible or unfeasible before. With its low requirements in space and energy it presents a remarkable alternative to utilizing large computer clusters.

Future Work

The most obvious target for future work is the missing hardware design of the imputation process besides the already implemented phasing part in genotype imputation. Since both parts require almost the same computational effort, another significant improvement would be achieved if both parts were hardware accelerated, and since both parts apply the same underlying models and methods, the expected speedup is the same as in the phasing part.

Furthermore, the workload distribution can be improved, e.g. analyzing several chromosomes at once prevents a lot of FPGA resources from being idle most of the time. Looking back at the results in Table 6.7 in Chapter 6.3.2 the total number of utilized FPGAs was 1,258. This indicates that the complete dataset over all chromosomes could have been analyzed in only 10 rounds on the RIVYERA platform featuring 128 FPGAs instead of 22 rounds with one round for each chromosome. The runtime of each round would be around 630 seconds such as for chromosome 2 that nearly exploits all FPGA resources completely. Thus, the total runtime would be 6,300 seconds and the expected speedup when compared to the CPU system would result in about 55. This is almost twice the speedup which has been achieved now, and it would grow larger if the number of markers could be more equally distributed around the computing windows as well.

In other areas it would be interesting to see if application acceleration can benefit from a combination of FPGAs with other technologies, such as GPUs. For SNP interaction detection GPUs seem to be perfectly suitable to calculate the statistical test that usually includes a lot of floating point operations. These operations typically require a lot of resources on the FPGA which could be used to create contingency tables instead. This process is typically effectively implementable on an FPGA without spending too many resources since it can benefit from simple data streaming through an optimized self-implemented bus system, as described in Chapter 5. Then, the GPU has to simply calculate the test on a stream of contingency tables using its internal resources such as the available FPUs (floating point units). A problem for this combinatorial solution is the tremendous amount of data traffic between FPGA and GPU, which might be tackled with ultra-fast bus

systems and an efficient data organization.

The area of bioinformatics still hosts a lot of other compute intensive applications, while a great part seems suitable for hardware implementation. Since this thesis has already proven to be successful for certain applications, other tools might benefit from exploiting FPGAs as well. For example, in the area of sequence alignment promising targets can be found. The short-read alignment tool BWA [LD09] or one of its derivates use the Burrows-Wheeler transformation [BW95] and the FM-index [FM00] to quickly align short reads against a transformed reference. For each of the billions of reads resulting from a standard sequencer run, the same task is repeated again and again, which makes it extremely suitable for FPGA implementation. A proof-of-concept can already be found in [Wie13] indicating a great expected speedup for real-world applications.

De-novo assembly is also a suitable target due to the heavy amount of data. Here, it has to be estimated if the computational procedure requires too many memory lookups which might decrease the expectations of a high speedup. Furthermore, graph reduction algorithms may not be trivially parallelized such that a high-effort must be taken into an efficient implementation of such a tool. Yet, de-novo assembly very often requires a preliminary error correction method before building the graph structure, such as SHREC [SSP+09], Qamar [SS12] or Musket [LSS13]. In the case of Qamar simple voting tables indicate erroneous reads and how they should be corrected most likely. This data structure seems to be eligible for a straight-forward implementation in hardware with a high expectation of a remarkable speedup.

Last but not least, analysis of genotypes is also part of the determination of ancestry relations. RAxML [Sta14] is a popular tool in inferring phylogenetic trees of an input set of individuals with the help of the *maximum likelihood* method. This NP-hard problem results in long runtimes of more than an hour for a dataset of 8,000 taxons on a CPU cluster with 36 quad-core CPUs. This problem targeted by FPGA hardware leaves the expectation of the ability to solve larger input data sets in a still feasible runtime.

This last topic gives a direct hint that all biological research may lead back to Darwin, his theory on evolution and his first estimation of the phylogenetic tree of life [Dar59]. In closing this thesis, the reader might have

recognized how much effort is taken in all research throughout this area, but also how much motivation is put into practice to allow humans a little more insight into one of the most fascinating and complex but fundamental topics ever since. Life.

Appendix A

Curriculum Vitae

Dipl.-Inf.
Lars Wienbrandt

Date of birth: November 22nd, 1980
Place of birth: Eckernförde, Germany

Study

02/2009 - 02/2016 PhD thesis at the Department of Technical Computer Science at Christian-Albrechts-University of Kiel
Title: "FPGAs in Bioinformatics - Implementation and Evaluation of Common Bioinformatics Algorithms in Reconfigurable Logic"

10/2001 - 02/2009 Study of Computer Science with minor subject Electrical Engineering at Christian-Albrechts-University of Kiel

Diploma degree: with distinction

Title of thesis: "Massiv parallelisierte DNA-Motivsuche auf COPACOBANA – Hardware-Implementierung in VHDL und Effizienzvergleich mit einem Standard-PC"

Research

08/2013 - 02/2016 Excellence cluster project "A Power-saving Benchtop Machine for Ultra-fast Genetic Data Analysis and Interpretation in the Inflammation Clinic".

07/2013 - 02/2016 Mentoring of AiF project "Development of a hybrid-parallel computing architecture for bioinformatics".

08/2009 - 08/2013 Research in the area of high-performance bioinformatics applications on FPGAs.

02/2009 - 08/2009 HWT project "Bio-Engine".

Projects

from 07/2013 AiF (ZIM-KOOP) project "Development of a hybrid-parallel computing architecture for bioinformatics", co-application under protectorate of Prof. Manfred Schimmler

from 03/2013 Excellence cluster project "A Power-saving Benchtop Machine for Ultra-fast Genetic Data Analysis and Interpretation in the Inflammation Clinic", co-application under protectorate of Prof. Manfred Schimmler

Teaching

03/2009 - 03/2015 Direction of the seminar and advanced seminar "Algorithmen der Bioinformatik / bioinformatics algorithms"

10/2010 - 03/2015 Direction of master project "Algorithmenentwurf für massiv parallele Hardware / development of algorithms for massively parallel hardware"

04/2011 - 09/2014 Direction of exercises for lesson “Implementierung massiv paralleler Systeme / implementation of massively parallel systems” respectively “FPGA-Entwurf mit VHDL / FPGA development in VHDL”.

03/2009 - 09/2013 Direction of the “Hardware-Praktikums / hardware lab” and the computer science part of the “Grundpraktikum für Ingenieure I+II / basic lab for engineers pts. I+II”.

04/2005 - 02/2009 Student assistant for the “Hardware-Praktikum / hardware lab”.

07/2006 - 01/2007 student assistant for the advanced lab “Das eingebettete Orchester / the embedded orchestra”.

10/2005 - 02/2006
10/2004 - 02/2005 student assistant for exercises in “Systemorientierte Informatik I: Digitale Systeme / system oriented computer science I: digital systems”.

Mentoring

07/2015 Sven Gundlach (Master), “Massiv parallele, FPGA-basierte Implementierung von MB-MDR in einer optimierten Version von maxT”

12/2014 Florian Xaver Schmidt (Master), “Efficient JPEG2000 decoding for diagnostic purposes on the RIVYERA-Architecture”

03/2014 Daniel Siebert (Diploma), “BLASTp auf RIVYERA – Implementierung von BLASTp auf der RIVYERA S6-LX150”

02/2014 Simon Timmermann (Diploma), “Short Read Error Correction on the Parallel Platform RIVYERA – An Algorithmic Concept for a Distributed High-Performance Architecture”

09/2013	Nabil Imran (Bachelor), “Mining digitaler Währungen mit Hilfe von RIVYERA”
09/2013	Sven Hüser (Bachelor), “Detektion von Gen-Gen-Interaktionen in genomweiten Fall-Kontroll-Studien mit Hilfe von FPGAs”
03/2013	Martin Kruse (Diploma), “Massively parallel implementation of Navier-Stokes equations for modeling solar winds”
03/2013	Ulf Rüegg (Master), “Parallelized FPGA-based Graph Creation for De Novo Genome Assembly – Leveraging the Reconfigurable High-Performance Platform RIVYERA”
03/2013	Florian Nommensen (Bachelor), “Exaktes lokales Short-Read-Alignment mit möglichem Softclipping”
12/2012	Jan Christian Kässens (Master), “A Distributed Shared Memory Core for FPGA Clusters”
12/2012	Claas Anders Rathje (Master), “Massively Parallel Attack on WPA2 – A Dictionary Attack Using the Reconfigurable Hardware Platform RIVYERA”
09/2012	Rian Voß (Bachelor), “Implementierung einer PBKDF2 unter Verwendung von HMAC-RIPEMD-160 auf der RIVYERA S3-5000”
06/2012	Jost Bissel (Diploma), “Entwicklung einer generischen Alignment-Anwendung auf der massiv-parallelen Hardwareplattform RIVYERA”
03/2012	Björn Stade (Diploma), “Implementierung eines effizienten Verfahrens zur Identifizierung gekoppelter Nukleotid-Polymorphismen aus Exom-Sequenzierungen in das Programmpaket ’snpActs”’

12/2011 Martin Kruse (student research project), "Implementation of the Smith-Waterman algorithm as part of the short read alignment algorithm SMARTI on the FPGA based hardware architecture RIVYERA"

12/2010 Michael Lange (Diploma), "Implementierung des SM-Busses für massiv parallele FPGA-Architekturen"

12/2009 Carol May Yen Yeo (Master), "Migration and Optimization of DNA Motif Search Algorithm on the Reconfigurable Parallel Hardware Platform COPACOBANA 5000"

Publications

2015 Michael Forster, Silke Szymczak, David Ellinghaus, Georg Hemmrich-Stanisak, Malte Rühlemann, Lars Kraemer, Sören Mucha, **Lars Wienbrandt**, Martin Stanulla, UFO Sequencing Consortium within I-BFM Study Group, and Andre Franke, *Vy-PER: eliminating false positive detection of virus integration events in next generation sequencing data*, Scientific Reports 5:11534, 2015. [FSE+15]

Jorge González-Domínguez, Jan Christian Kässens, **Lars Wienbrandt**, and Bertil Schmidt, *Large-Scale Genome-Wide Association Studies on a GPU Cluster Using a CUDA-Accelerated PGAS Programming Model*, International Journal of High Performance Computing Applications (IJHPCA), vol. 29(4), pp. 506–510, 2015. [GKW+15]

Jan Christian Kässens, **Lars Wienbrandt**, Jorge González-Domínguez, Bertil Schmidt, and Manfred Schimmler, *High-Speed Exhaustive 3-locus Interaction Epistasis Analysis on FPGAs*, International Conference on Computer Science (ICCS), Journal of Computational Science, vol. 9, pp. 131–136, 2015. [KWG+15]

Jorge González-Domínguez, **Lars Wienbrandt**, Jan Christian Kässens, David Ellinghaus, Manfred Schimmler, and Bertil Schmidt, *Parallelizing Epistasis Detection in GWAS on FPGA and GPU-accelerated Computing Systems*, IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 12(5), pp. 982–994, 2015. [GWK+15]

2014

Ayman Abbas, Rian Voß, **Lars Wienbrandt**, and Manfred Schimmler, *An Efficient Implementation of PBKDF2 with RIPEMD-160 on Multiple FPGAs*, 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), pp. 454–461, Dec 2014. [AVW+14]

Jan Christian Kässens, Jorge González-Domínguez, **Lars Wienbrandt**, and Bertil Schmidt, *UPC++ for Bioinformatics: A Case Study Using Gnome-Wide Association Studies*, 2014 IEEE International Conference on Cluster Computing, pp. 248–256, Sep 2014. [KGW+14]

Jorge González-Domínguez, Bertil Schmidt, Jan Christian Kässens, and **Lars Wienbrandt**, *Hybrid CPU/GPU Acceleration of Detection of 2-SNP Epistatic Interactions in GWAS*, Euro-Par 2014 Parallel Processing, Lecture Notes on Computer Science, vol. 8632, pp. 680–691, 2014. [GSK+14]

Lars Wienbrandt, *The FPGA-based High-Performance Computer RIVYERA for Applications in Bioinformatics*, International Conference on Computability in Europe (CiE), Language, Life, Limits, Lecture Notes on Computer Science, vol. 8493, pp. 383–392, 2014. [Wie14]

Lars Wienbrandt, Jan Christian Kässens, Jorge González-Domínguez, Bertil Schmidt, David Ellinghaus, and Manfred Schimmler, *FPGA-based acceleration of detecting statistical epistasis in GWAS*, International Conference on

Computer Science (ICCS), Procedia Computer Science, vol. 29, pp. 220–230, Jun 2014. [WKG+14]

2013 **Lars Wienbrandt**, *Bioinformatics Applications on the FPGA-based High-Performance Computer RIVYERA*, in "High Performance Computing Using FPGAs" edited by Wim Vanderbauwhede, Khaled Benkrid, Springer, pp. 81–103, 2013, ISBN 978-1-4614-1790-3. [Wie13]

Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, **Lars Wienbrandt**, and Ralf Zimmermann, *High-Performance Cryptanalysis on RIVYERA and COPA-COBANA Computing Systems*, in "High Performance Computing Using FPGAs" edited by Wim Vanderbauwhede, Khaled Benkrid, Springer, pp. 335–366, 2013, ISBN 978-1-4614-1790-3. [GKN+13]

2012 Ayman Abbas, Claas Anders Rathje, **Lars Wienbrandt**, and Manfred Schimmler, *Dictionary Attack on TrueCrypt with RIVYERA S3-5000*, 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), pp. 93–100, Dec 2012. [ARW+12]

Florian Schatz, **Lars Wienbrandt**, and Manfred Schimmler, *Probability model for boundaries of short-read sequencing*, 2012 International Conference on Advances in Computing and Communications (ICACC), pp. 223–228, Aug 2012. (best paper award). [SWS12]

Christoph Starke, Vasco Grossmann, **Lars Wienbrandt**, and Manfred Schimmler, *An FPGA implementation of an Investment Strategy Processor*, International Conference on Computer Science (ICCS), Procedia Computer Science, vol. 9, pp. 1880–1889, 2012. [SGW+12b]

Lars Wienbrandt, Daniel Siebert, and Manfred Schimmler, *Improvement of BLASTp on the FPGA-Based High-*

Performance Computer RIVYERA, International Symposium on Bioninformatics Research and Applications (ISBRA), Lecture Notes in Computer Science, vol. 7292, pp. 275–286, 2012. [WSS12]

Christoph Starke, Vasco Grossmann, **Lars Wienbrandt**, Sven Koschnicke, John Carstens, and Manfred Schimmler, *Optimizing Investment Strategies with the Recongurable Hardware Platform RIVYERA*, International Journal of Reconfigurable Computing, vol. 2012, 10 pages, 2012. [SGW+12a]

2011 **Lars Wienbrandt**, Stefan Baumgart, Jost Bissel, Florian Schatz, and Manfred Schimmler, *Massively parallel FPGA-based implementation of BLASTp with the two-hit method*, International Conference on Computer Science (ICCS), Procedia Computer Science, vol. 4, pp. 1967–1976, 2011. [WBB+11]

2010 **Lars Wienbrandt**, and Manfred Schimmler, *Collecting Statistical Information in DNA Sequences for the Detection of Special Motifs*, Proceedings of BIOCOMP2010, pp. 274–278, 2010. [WS10]

Manfred Schimmler, **Lars Wienbrandt**, Tim Güneysu, and Jost Bissel, *COPACOBANA: A Massively Parallel FPGA-Based Computer Architecture*, in "Bioinformatics: High Performance Parallel Computer Architectures" edited by Bertil Schmidt, CRC Press, pp. 223–262, 2010, ISBN 978-1-4398-1488-8. [SWG+10]

Lars Wienbrandt, Stefan Baumgart, Jost Bissel, Carol May Yen Yeo, and Manfred Schimmler, *Using the reconfigurable massively parallel architecture COPACOBANA 5000 for applications in bioinformatics*, International Conference on Computer Science (ICCS), Procedia Computer Science, vol. 1 (1), pp. 1027–1034, 2010. [WBB+10]

2008 Jan Schröder, **Lars Wienbrandt**, Gerd Pfeiffer, and Manfred Schimmler, *Massively Parrallelized DNA Motif Search on the Reconfigurable Hardware Platform COPACOBANA*, 3rd IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB), Lecture Notes in Computer Science, vol. 5265, pp. 436–447, 2008. [SWP+08]

Invited Talks

06/2014 Invited Talk in the "Special Session on Bio-inspired Computation" at the "10th Conference in Computability in Europe (CiE 2014)" in Budapest, Hungary: *The FPGA-based High-Performance Computer RIVYERA for Applications in Bioinformatics*.

02/2011 Invited Talk at the "Workshop on Theoretical Biology" at the Max-Planck-Institute for Evolutionary Biology in Plön, Germany: *Hardware implementation and massive parallelization of BLAST*

Reviews

2015 Program committee member of "IEEE International Conference on High Performance Computing (HiPC 2015)".

Program committee member at the workshop on "Parallel Computational Biology (Parallel Bio-Computing (PBC))" in conjunction with the "11th International Conference on Parallel Processing and Applied Mathematics (PPAM 2015)".

2013 Program committee member at the workshop on "Parallel Computational Biology (Parallel Bio-Computing

(PBC))" in conjunction with the "10th International Conference on Parallel Processing and Applied Mathematics (PPAM 2013)".

2012 Program committee member at the 4th workshop on "Emerging Parallel Architectures (WEPA)" in conjunction with the "International Conference on Computational Science (ICCS 2012)".

2011 Reviewer for the Open Access Journal "Sensors" by MDPI.

Program committee member at the 3rd workshop on "Emerging Parallel Architectures (WEPA)" in conjunction with the "International Conference on Computational Science (ICCS 2011)".

Bibliography

[Abe] Abecasis Group. *Minimac3*. accessed 10/2015. URL: http://genome.sph.umich.edu/wiki/Minimac3.

[Aff09] Affymetrix. *Genome-Wide Human SNP Array 6.0*. Data sheet. 2009. URL: http://www.affymetrix.com/support/technical/datasheets/genomewide_snp6_datasheet.pdf.

[AGM+90] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. "Basic Local Alignment Search Tool". In: *Journal of Molecular Biology* 215.3 (Oct. 1990), pp. 403–10.

[Agr12] Alan Agresti. *Categorical Data Analysis*. 3rd ed. Wiley, Dec. 2012. ISBN: 978-0470463635.

[AL06] Peter J. Ashenden and Jim Lewis. *The Designer's Guide to VHDL*. 3rd ed. Morgan Kaufmann, Nov. 2006. ISBN: 978-0120887859.

[Alt] Altera Corporation. URL: https://www.altera.com/.

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[AMS+97] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs". In: *Nucleic Acids Research* 25 (17 1997), pp. 3389–402.

[ARW+12] Ayman Abbas, Claas Anders Rathje, Lars Wienbrandt, and Manfred Schimmler. "Dictionary Attack on TrueCrypt with RIVYERA S3-5000". In: *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2012, pp. 93–100. DOI: 10.1109/ICPADS.2012.23.

[Atm] Atmel Corporation. URL: http://www.atmel.com/.

[AVW+14] Ayman Abbas, Rian Voß, Lars Wienbrandt, and Manfred Schimmler. "An Efficient Implementation of PBKDF2 with RIPEMD-160 on Multiple FPGAs". In: *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2014, pp. 454–461. DOI: 10.1109/PADSW.2014.7097841.

[Bar] Blaise Barney. *POSIX Thread Programming*. URL: https://computing.llnl.gov/tutorials/pthreads/.

[BB07] Sharon R. Browning and Brian L. Browning. "Rapid and Accurate Haplotype Phasing and Missing-Data Inference for Whole-Genome Association Studies". In: *American Journal of Human Genetics* 81 (5 Nov. 2007), pp. 1084–1097. DOI: 10.1086/521987.

[Bos04] P. Bose. "Computer architecture research: Shifting priorities and newer challenges". In: *IEEE Micro* 24 (6 2004), p. 5. DOI: 10.1109/MM.2004.68.

[BW95] M. Burrows and D. J. Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. Tech. rep. Digital Systems Research Center Research Reports, July 1995.

[CCD+11] Tom Cattaert, M. Luz Calle, Scott M. Dudek, Jestinah M. Mahachie John, François van Lishout, Victor Urrea, Marylyn D. Ritchie, and Kristel van Steen. "A detailed view on Model-Based Multifactor Dimensionality Reduction for detecting gene-gene interactions in case-control data in the absence and presence of noise". In: *Annals of Human Genetics* 75.1 (Jan. 2011), pp. 78–89. DOI: 10.1111/j.1469-1809.2010.00604.x.

[CHW+13] Ryan L. Collins, Ting Hu, Christian Wejse, Giorgio Sirugo, Scott M. Williams, and Jason H Moore. "Multifactor dimensionality reduction reveals a three-locus epistatic interaction associated with susceptibility to pulmonary tuberculosis". In: *BioData Mining* 6.1 (2013), p. 4. DOI: 10.1186/1756-0381-6-4.

[CLC] CLC bio, a QIAGEN Company. URL: http://www.clcbio.com.

[Com05] Douglas E. Comer. *Essentials of Computer Architecture*. Upper Saddle River, NJ 07458: Pearson Prentice Hall, 2005. ISBN: 978-0131491793.

[Cor02] Heather J. Cordell. "Epistasis: what it means, what it doesn't mean, and statistical methods to detect it in humans". In: *Human Molecular Genetics* 11.20 (July 2002), pp. 2463–2468.

[Cor09] Heather J. Cordell. "Detecting Gene-Gene Interactions that Underlie Human Diseases". In: *Nature Review Genetics* 10.6 (2009), pp. 392–404. DOI: 10.1038/nrg2579.

[Cul07] R. Culverhouse. "The Use of the Restricted Partition Method with Case-Control Data". In: *Human Heredity* 63.2 (2007), pp. 93–100. DOI: 10.1159/000099181.

[DAR] Beman Dawes, David Abrahams, and Rene Rivera. *BOOST C++ Libraries*. accessed 11/2015. URL: http://www.boost.org/.

[Dar59] Charles Darwin. *On the Origin of Species*. Albemarle Street, London, UK: John Murray, Nov. 1859.

[dbSNP] NCBI. *RELEASE: NCBI dbSNP Build 144*. accessed 10/2015. URL: http://www.ncbi.nlm.nih.gov/projects/SNP/snp_summary.cgi?build_id=144.

[DMZ12] Olivier Delaneau, Jonathan Marchini, and Jean-François Zagury. "A linear complexity phasing method for thousands of genomes". In: *Nature Methods* 9 (2012), pp. 179–181. DOI: 10.1038/nmeth.1785.

[Don13] Jack Dongarra. *Visit to the National University for Defense Technology Changsha, China*. Tech. rep. University of Tennesse, June 2013.

[DZM13] Olivier Delaneau, Jean-François Zagury, and Jonathan Marchini. "Improved whole chromosome phasing for disease and population genetic studies". In: *Nature Methods* 10 (2013), pp. 5–6. DOI: 10.1038/nmeth.2307.

[EMB] EMBL-EBI. *UniProtKB/TrEMBL PROTEIN DATABASE RELEASE 2015_10 STATISTICS*. accessed 10/2015. URL: http://www.ebi.ac.uk/uniprot/TrEMBLstats.

[FAH15] Christian Fuchsberger, Gonçalo R. Abecasis, and David A. Hinds. "minimac2: faster genotype imputation". In: *Bioinformatics* 31 (5 2015), pp. 782–784. DOI: 10.1093/bioinformatics/btu704.

[FCD+76] W. Fiers et al. "Complete nucleotide sequence of bacteriophage MS2 RNA: primary and secondary structure of the replicase gene". In: *Nature* 260 (5551 Apr. 1976), pp. 500–7.

[FHW+12] Gang Fang, Majda Haznadar, Wen Wang, Haoyu Yu, Michael Steinbach, Timothy R. Church, William S. Oetting, Brian Van Ness, and Vipin Kumar. "High-Order SNP Combinations Associated with Complex Diseases: Efficient Discovery, Statistical Power and Functional Interactions". In: *PLoS ONE* 7.4 (2012), e33531. DOI: 10.1371/journal.pone.0033531.

[FM00] Paolo Ferragina and Giovanni Manzini. "Opportunistic Data Structures with Applications". In: *2000 41st Annual Symposium on Foundations of Computer Science*. Nov. 2000, pp. 390–398. DOI: 10.1109/SFCS.2000.892127.

[FSE+15] Michael Forster et al. "Vy-per: eliminating false positive detection of virus integration events in next generation sequencing data". In: *Scientific Reports* 5.11534 (2015). DOI: 10.1038/srep11534.

[GAA+11] Jiang Gui, Angeline S. Andrew, Peter Andrews, Heather M. Nelson, Karl T. Kelsey, Margaret R. Karagas, and Jason H. Moorea. "A robust multifactor dimensionality reduction method for detecting gene-gene interactions with application to the genetic analysis of bladder cancer susceptibility". In:

Annals of Human Genetics 75.1 (Jan. 2011), pp. 20–28. DOI: 10.1111/j.1469-1809.2010.00624.x.

[GBM+06] S. G. Gregory et al. "The DNA sequence and biological annotation of human chromosome 1". In: *Nature* 441 (7091 May 2006), pp. 315–321. DOI: 10.1038/nature04727.

[GKN+13] Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, Lars Wienbrandt, and Ralf Zimmermann. "High-Performance Cryptanalysis on RIVYERA and COPACOBANA Computing Systems". In: *High-Performance Computing Using FPGAs*. Ed. by Wim Vanderbauwhede and Khaled Benkrid. Springer, 2013, pp. 335–366. ISBN: 978-1-4614-1790-3. DOI: 10.1007/978-1-4614-1791-0.

[GKW+15] Jorge González-Domínguez, Jan Christian Kässens, Lars Wienbrandt, and Bertil Schmidt. "Large-Scale Genome-Wide Association Studies on a GPU Cluster Using a CUDA-Accelerated PGAS Programming Model". In: *International Journal of High Performance Computing Applications (IJHPCA)* 29.4 (Nov. 2015), pp. 506–510. DOI: 10.1177/1094342015585846.

[GMY+14] Xuan Guo, Yu Meng, Ning Yu, and Yi Pan. "Cloud computing for detecting high-order genome-wide epistatic interaction via dynamic clustering". In: *BMC Bioinformatics* 15.1 (2014), p. 102. DOI: 10.1186/1471-2105-15-102.

[GN96] Priscilla E. Greenwood and Mikhail S. Nikulin. *A guide to chi-squared testing*. New York: Wiley, 1996. ISBN: 047155779X.

[GRW+13] Benjamin Goudey, David Rawlinson, Qiao Wang, Fan Shi, Herman Ferra, Richard M. Campbell, Linda Stern, Michael T. Inouye, Cheng Soon Ong, and Adam Kowalczyk. *GWIS - model-free, fast and exhaustive search for epistatic interactions in case-control GWAS*. May 2013. DOI: 10.1186/1471-2164-14-S3-S10.

[GSK+14] Jorge González-Domínguez, Bertil Schmidt, Jan Christian Kässens, and Lars Wienbrandt. "Hybrid CPU/GPU Acceleration of Detection of 2-SNP Epistatic Interactions in GWAS". In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva,

Inês Dutra, and Vítor Santos Costa. Vol. 8632. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 680–691. ISBN: 978-3-319-09872-2. DOI: 10.1007/978-3-319-09873-9_57.

[GWD11] Xinyu Guo, Hong Wang, and Vijay Devabhaktuni. "Design of a FPGA-Based Parallel Architecture for BLAST Algorithm with Multi-hits Detection". In: *Proceedings of ITNG'11*. IEEE Computer Society, 2011, pp. 689–694.

[GWK+15] Jorge González-Domínguez, Lars Wienbrandt, Jan Christian Kässens, David Ellinghaus, Manfred Schimmler, and Bertil Schmidt. "Parallelizing Epistasis Detection in GWAS on FPGA and GPU-accelerated Computing Systems". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 12 (5 Jan. 2015), pp. 982–994. DOI: 10.1109/TCBB.2015.2389958.

[HDM09] Bryan N. Howie, Peter Donnelly, and Jonathan Marchini. "A Flexible and Accurate Genotype Imputation Method for the Next Generation of Genome-Wide Association Studies". In: *PLoS Genetics* 5 (6 June 2009). DOI: 10.1371/journal.pgen.1000529.

[HFS+12] Bryan Howie, Christian Fuchsberger, Matthew Stephens, Jonathan Marchini, and Gonçalo R. Abecasis. "Fast and accurate genotype imputation in genome-wide association studies through pre-phasing". In: *Nature Genetics* 44 (8 July 2012), pp. 955–959. DOI: 10.1038/ng.2354.

[HGP] National Human Genome Research Institute. *An Overview of the Human Genome Project*. accessed 10/2015. URL: http://www.genome.gov/12011238.

[HGSC01] International Human Genome Sequencing Consortium. "Initial sequencing and analysis of the human genome". In: *Nature* 409 (6822 2001), pp. 860–921.

[HLZ+10] Xiaohan Hu, Qiang Liu, Zhao Zhang, Zhiqiang Li, Shilin Wang, Lin He, and Yongyong Shi. "SHEsisEpi, a GPU-enhanced genome-wide SNP-SNP interaction scanning algorithm, efficiently reveals the risk genetic epistasis in bipolar

disorder". In: *Cell Research* 20 (7 May 2010), pp. 854–857. DOI: 10.1038/cr.2010.68.

[HMS11] Bryan N. Howie, Jonathan Marchini, and Matthew Stephens. "Genotype Imputation with Thousands of Genomes". In: *G3: Genes, Genomics, Genetics* 1 (6 Nov. 2011), pp. 457–70. DOI: 10.1534/g3.111.001198.

[HRC] The Haplotype Reference Consortium. accessed 10/2015. URL: http://www.haplotype-reference-consortium.org/.

[HTW+11] Gibran Hemani, Athanasios Theocharidis, Wenhua Wei, and Chris Haley. "EpiGPU: exhaustive pairwise epistasis scans parallelized on consumer level graphics cards". In: *Bioinformatics* 27.11 (Apr. 2011), pp. 1462–1465. DOI: 10.1093/bioinformatics/btr172.

[Ill06] Illumina. *Infinium II Assay Workflow*. 2006. URL: http://www.illumina.com/documents/products/workflows/workflow_infinium_ii.pdf.

[Ill15a] Illumina. *HiSeq X Series of Sequencing Systems*. Data sheet. July 2015. URL: http://www.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/datasheet-hiseq-x-ten.pdf.

[Ill15b] Illumina. *Human Commercial and Consortia Infinium Arrays*. Information sheet. Oct. 2015. URL: http://www.illumina.com/content/dam/illumina-marketing/images/applications/genotyping/human-commercial.pdf.

[Int15] Intel. *Accelerate Big Data Insights with the Intel Xeon Processor E7-8800/4800 v3 Product Families*. Product Brief, accessed 11/2015. 2015. URL: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e7-v3-family-brief.pdf.

[ISE] Xilinx Inc. *ISE Design Suite*. accessed 11/2015. URL: http://www.xilinx.com/products/design-tools/ise-design-suite.html.

[JBP+74] Ernest Jay, Robert Bambara, R. Padmanabhan, and Ray Wu. "DNA sequence analysis: a general, simple and rapid method for sequencing large oligodeoxyribonucleotide fragments by mapping". In: *Nucleic Acids Research* 1 (3 Mar. 1974), pp. 331–353.

[JLB+08] Arpith Jacob, Jospeh Lancaster, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. "Mercury BLASTp: Accelerating Protein Sequence Alignment". In: *ACM Transactions on Reconfigurable Technology and Systems* 1 (2 June 2008), p. 9. DOI: 10.1145/1371579.1371581.

[JSF72] C. Jersild, A. Svejgaard, and T. Fog. "HL-A antigens and multiple sclerosis". In: *The Lancet* 1 (7762 June 1972), pp. 1240–1.

[KAB+03] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. "Leakage current: Moore's law meets static power". In: *IEEE Computer* 36 (12 Dec. 2003), pp. 68–75. DOI: 10.1109/MC.2003.1250885.

[KBL08] Server Kasap, Khaled Benkrid, and Ying Liu. "Design and Implementation of an FPGA-based Core for Gapped BLAST Sequence Alignment with the Two-Hit Method". In: *Engineering Letters* 16.3 (2008).

[KBL09] Server Kasap, Khaled Benkrid, and Ying Liu. "A High Performance FPGA-based Implementation of Position Specific Iterated BLAST". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '09. Monterey, California, USA: ACM, 2009, pp. 249–252. ISBN: 978-1-60558-410-2. DOI: 10.1145/1508128.1508169.

[KGW+14] Jan Christian Kässens, Jorge González-Domínguez, Lars Wienbrandt, and Bertil Schmidt. "UPC++ for bioinformatics: A case study using genome-wide association studies". In: *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2014, pp. 248–256. DOI: 10.1109/CLUSTER.2014.6968770.

[Khr] Khronos Group. *OpenGL – The Industry's Foundation for High Performance Graphics*. accessed 10/2015. URL: https://www.opengl.org/.

[KLL+12] Petr Klus, Simon Lam, Dag Lyberg, Ming Cheung, Graham Pullan, Ian McFarlane, Giles Yeo, and Brian Lam. "BarraCUDA – a fast short read sequence aligner using graphics processing units". In: *BMC Research Notes* 5.1 (2012), p. 27. DOI: 10.1186/1756-0500-5-27.

[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, Andy Rupp, and Manfred Schimmler. "How to Break DES for €8,980". In: *SHARCS '06 – Special-purpose Hardware for Attacking Cryptographic Systems*. 2006.

[KWG+15] Jan Christian Kässens, Lars Wienbrandt, Jorge González-Domínguez, Bertil Schmidt, and Manfred Schimmler. "High-Speed Exhaustive 3-locus Interaction Epistasis Analysis on FPGAs". In: *Journal of Computational Science* 9 (July 2015), pp. 131–136. DOI: doi:10.1016/j.jocs.2015.04.030.

[Lat] Lattice Semiconductor. URL: http://www.latticesemi.com/.

[LD09] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows-Wheeler transform". In: *Bioinformatics* 25.14 (July 2009), pp. 1754–1760. DOI: 10.1093/bioinformatics/btp324.

[LJG+13] François Van Lishout, Jestinah M Mahachie John, Elena S Gusareva1, Victor Urrea, Isabelle Cleynen, Emilie Théâtre, Benoît Charloteaux, Malu Luz Calle, Louis Wehenkel, and Kristel Van Steen. "An efficient algorithm to perform multiple testing in epistasis screening". In: *BMC Bioinformatics* 14.138 (Apr. 2013). DOI: 10.1186/1471-2105-14-138.

[LJL+14] Sangseob Leem, Hyun-hwan Jeong, Jungseob Lee, Kyubum Weea, and Kyung-Ah Sohn. "Fast detection of high-order epistatic interactions in genome-wide association studies using information theoretic measure". In: *Computational Biology and Chemistry* 50 (June 2014), pp. 19–28. DOI: 10.1016/j.compbiolchem.2014.01.005.

[LLL+12] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. "Comparison of next-generation sequencing systems". In: *Journal of Biomedicine and Biotechnology* 2012.251364 (July 2012). DOI: 10.1155/2012/251364.

[LSM10] Yongchao Liu, Bertil Schmidt, and Douglas Maskell. "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions". In: *BMC Research Notes* 3.1 (2010), pp. 93+. DOI: 10.1186/1756-0500-3-93.

[LSM11a] Weiguo Liu, Bertil Schmidt, and Wolfgang Müller-Wittig. "CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 8 (6 Nov. 2011), pp. 1678–1684. DOI: 10.1109/TCBB.2011.33.

[LSM11b] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI". In: *BMC Bioinformatics* 12.85 (Mar. 2011). DOI: 10.1186/1471-2105-12-85.

[LSM12] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform". In: *Bioinformatics* 28 (14 July 2012), pp. 1830–7. DOI: 10.1093/bioinformatics/bts276.

[LSS13] Yongchao Liu, Jan Schröder, and Bertil Schmidt. "Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data". In: *Bioinformatics* 29.3 (2013), pp. 308–315. DOI: 10.1093/bioinformatics/bts690.

[LTP+09] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". In: *Genome Biology* 10.3 (2009), R25. DOI: 10.1186/gb-2009-10-3-r25.

[LWD+10] Yun Li, Cristen J. Willer, Jun Ding, Paul Scheet, and Gonçalo R. Abecasis. "MaCH: Using Sequence and Genotype Data to Estimate Haplotypes and Unobserved Genotypes". In: *Genetic Epidemiology* 34 (8 Dec. 2010), pp. 816–834. DOI: 10.1002/gepi.20533.

[LWS13] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. "CUDASW++3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions". In: *BMC Bioinformatics* 14.117 (Apr. 2013). DOI: 10.1186/1471-2105-14-117.

[LYL+09] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah W. Lam, Siu-Ming M. Yiu, Karsten Kristiansen, and Jun Wang. "SOAP2: an improved ultrafast tool for short read alignment". In: *Bioinformatics* 25.15 (Aug. 2009), pp. 1966–1967. DOI: 10.1093/bioinformatics/btp336.

[Mah08] Brendan Maher. "Personal Genomes: the Case of the Missing Heritability". In: *Nature* 456.7218 (2008), pp. 18–21. DOI: 10.1038/456018a.

[MAW10] Jason H. Moore, Folkert W. Asselbergs, and Scott M. Williams. "Bioinformatics Challenges for Genome-Wide Association Studies". In: *Bioinformatics* 26.4 (2010), pp. 445–455. DOI: 10.1093/bioinformatics/btp713.

[MH10] Atabak Mahram and Martin C. Herbordt. "Fast and Accurate NCBI BLASTp: Acceleration with Multiphase FPGA-based Prefiltering". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS '10. Tsukuba, Ibaraki, Japan: ACM, 2010, pp. 73–82. ISBN: 978-1-4503-0018-6. DOI: 10.1145/1810085.1810099.

[MHM+07] Jonathan Marchini, Bryan N. Howie, Simon Myers, Gil McVean, and Peter Donnelly. "A new multipoint method for genome-wide association studies by imputation of genotypes". In: *Nature Genetics* 39 (7 2007), pp. 906–913. DOI: 10.1038/ng2088.

[Mic] Microsemi Corporation. URL: http://www.microsemi.com/.

[MPI] *The Message Passing Interface (MPI) standard*. URL: http://www.mcs.anl.gov/research/projects/mpi/.

[NCBa] NCBI. *BLAST – Basic Local Alignment Search Tool*. http://blast.ncbi.nlm.nih.gov/Blast.cgi.

[NCBb] NCBI. *BLAST matrices*. accessed 11/2015. URL: ftp://ftp.ncbi.nih.gov/blast/matrices/.

[NCBc] NCBI. *RefSeq: NCBI Reference Sequence Database*. URL: http://www.ncbi.nlm.nih.gov/RefSeq/.

[NCB13] NCBI Insights. *Introducing the New Human Genome Assembly: GRCh38*. Dec. 24, 2013. URL: http://ncbiinsights.ncbi.nlm.nih.gov/2013/12/24/introducing-the-new-human-genome-assembly-grch38/.

[NKF+01] M. R. Nelson, L. R. Kardia, R. E. Ferrell, and C. F. Sing. "A Combinatorial Partitioning Method to Identify Multilocus Genotypic Partitions That Predict Quantitative Trait Variation". In: *Genome Research* 11.3 (Mar. 2001), pp. 458–470. DOI: 10.1101/gr.172901.

[NVIa] NVIDIA. *CUDA Parallel Computing Platform*. accessed 10/2015. URL: http://www.nvidia.com/object/cuda_home_new.html.

[NVIb] NVIDIA. *GeForce GTX TITAN Z Specifications*. accessed 10/2015. URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-z/specifications.

[NW70] Saul B. Needleman and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3 (Mar. 1970), pp. 443–453.

[Owe12] Mark Owen. *Practical Signal Processing*. Cambridge University Press, Dec. 2012. ISBN: 978-1107411821.

[PBS+09] Gerd Pfeiffer, Stefan Baumgart, Jan Schröder, and Manfred Schimmler. "A Massively Parallel Architecture for Bioinformatics". In: *Computational Science – ICCS 2009*. Ed. by Gabrielle Allen, Jarosław Nabrzyski, Edward Seidel, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot.

Vol. 5544. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 994–1003. ISBN: 978-3-642-01969-2. DOI: 10.1007/978-3-642-01970-8_100.

[Pea00] Karl Pearson. "On the criterion that a given system of derivations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50.5 (1900), pp. 157–175.

[PMG+07] Effie W. Petersdorf, Mari Malkki, Ted A. Gooley, Paul J. Martin, and Zhen Guo. "MHC Haplotype Matching for Unrelated Hematopoietic Cell Transplantation". In: *PLoS Medicine* 4 (1 Jan. 2007). DOI: 10.1371/journal.pmed.0040008.

[PNI+12] Jittima Piriyapongsa, Chumpol Ngamphiw, Apichart Intarapanich, Supasak Kulawonganunchai, Anunchai Assawamakin, Chaiwat Bootchai, Philip J. Shaw, and Sissades Tongsima. "iLOCi: a SNP interaction prioritization technique for detecting epistasis in genome-wide association studies". In: *BMC Genomics* 13.Suppl 7 (2012), S2+. DOI: 10.1186/1471-2164-13-s7-s2.

[PNT+07] Shaun Purcell et al. "PLINK: A Tool Set for Whole-Genome Association and Population-Based Linkage Analyses". In: *American Journal of Human Genetics* 81 (3 Sept. 2007), pp. 559–575. DOI: 10.1086/519795.

[PP12] Snehit Prabhu and Itsik Pe'er. "Ultrafast genome-wide scan for SNP–SNP interactions in common complex disease". In: *Genome Research* 22.11 (Nov. 2012), pp. 2230–2240. DOI: 10.1101/gr.137885.112.

[Rab89] Lawrence R. Rabiner. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". In: *Proceedings of the IEEE* 77.2 (Feb. 1989), pp. 257–286.

[RHR+01] Marylyn D. Ritchie, Lance W. Hahn, Nady Roodi, L. Renee Bailey, William D. Dupont, Fritz F. Parl, and Jason H. Moore. "Multifactor-Dimensionality Reduction Reveals High-Order Interactions among Estrogen-Metabolism Genes in Sporadic Breast Cancer". In: *American Journal of Human Genetics* 69.1 (July 2001), pp. 138–147. DOI: 10.1086/321276.

[RL10] Guillaume Rizk and Dominique Lavenier. "GASSST: global alignment short sequence search tool". In: *Bioinformatics* 26 (20 Aug. 2010), pp. 2534–2540. DOI: 10.1093/bioinformatics/btq485.

[Sci13a] SciEngines GmbH. *RIVYERA API – Host-API (C/C++)*. Feb. 2013. URL: http://www.sciengines.com/.

[Sci13b] SciEngines GmbH. *RIVYERA API – Machine-API (VHDL)*. Feb. 2013. URL: http://www.sciengines.com/.

[SD03] Matthew Stephens and Peter Donnelly. "A Comparison of Bayesian Methods for Haplotype Reconstruction from Population Genotype Data". In: *American Journal of Human Genetics* 73 (5 Nov. 2003), pp. 1162–1169. DOI: 10.1086/379378.

[SD07] Euripides Sotiriades and Apostolos Dollas. "A General Reconfigurable Architecture for the BLAST Algorithm". In: *VLSI Signal Processing* 48.3 (2007), pp. 198–208. DOI: 10.1007/s11265-007-0069-2.

[SE] SciEngines GmbH. URL: http://www.sciengines.com.

[SGW+12a] Christoph Starke, Vasco Grossmann, Lars Wienbrandt, Sven Koschnicke, John Carstens, and Manfred Schimmler. "Optimizing Investment Strategies with the Reconfigurable Hardware Platform RIVYERA". In: *Internation Journal of Reconfigurable Computing* 2012 (2012), 10 pages. DOI: 10.1155/2012/646984.

[SGW+12b] Christoph Starke, Vasco Grossmann, Lars Wienbrandt, and Manfred Schimmler. "An FPGA Implementation of an Investment Strategy Processor". In: *Procedia Computer Science* 9 (2012), pp. 1880–1889. DOI: 10.1016/j.procs.2012.04.206.

[SNC77] F. Sanger, S. Nicklen, and A. R. Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences of the United States of America* 74 (12 Dec. 1977), pp. 5463–5467.

[SS05] Matthew Stephens and Paul Scheet. "Accounting for Decay of Linkage Disequilibrium in Haplotype Inference and Missing-Data Imputation". In: *American Journal of Human Genetics* 76 (3 Mar. 2005), pp. 449–462. DOI: 10.1086/428594.

[SS06] Paul Scheet and Matthew Stephens. "A Fast and Flexible Statistical Model for Large-Scale Population Genotype Data: Applications to Inferring Missing Genotypes and Haplotypic Phase". In: *American Journal of Human Genetics* 78 (4 Apr. 2006), pp. 629–644. DOI: 10.1086/502802.

[SS12] Mohammed Sahli and Tetsuo Shibuya. "Qamar – A More Accurate DNA Sequencing Error Correcting Algorithm". In: *2012 International Conference on Bioscience, Biochemistry and Bioinformatics*. Vol. 31. 2012, pp. 53–58.

[SSD01] Matthew Stephens, Nicholas J. Smith, and Peter Donnelly. "A New Statistical Method for Haplotype Reconstruction from Population Data". In: *American Journal of Human Genetics* 68 (4 Apr. 2001), pp. 978–989. DOI: 10.1086/319501.

[SSL+] Haixiang Shi, Bertil Schmidt, Weiguo Liu, and Wolfgang Müller-Wittig. "A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware". In: *Journal of Computational Biology* 17 (4), pp. 603–15. DOI: 10.1089/cmb.2009.0062.

[SSP+09] Jan Schröder, Heiko Schröder, Simon J. Puglisi, Ranjan Sinha, and Bertil Schmidt. "SHREC: a short-read error correction method". In: *Bioinformatics* 25.17 (2009), pp. 2157–2163. DOI: 10.1093/bioinformatics/btp379.

[Sta14] Alexandros Stamatakis. "RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies". In: *Bioinformatics* 30.9 (2014), pp. 1312–1313. DOI: 10.1093/bioinformatics/btu033.

[Ste11] Kristel van Steen. "Travelling the world of gene–gene interactions". In: *Briefings in Bioinformatics* 13.1 (2011), pp. 1–19. DOI: 10.1093/bib/bbr012.

[Sup] Superfamily. *HMM library and genome assignments server*. URL: http://supfam.cs.bris.ac.uk/SUPERFAMILY/.

[SW81] Temple F. Smith and Michael S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147 (1981), pp. 195–197.

[SWG+10] Manfred Schimmler, Lars Wienbrandt, Tim Güneysu, and Jost Bissel. "COPACOBANA: A Massively Parallel FPGA-Based Computer Architecture". In: *Bioinformatics – High Performance Parallel Computer Architectures*. Ed. by Bertil Schmidt. CRC Press, July 2010, pp. 223–262. ISBN: 978-1-4398-1488-8.

[SWJ+09] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J.M. Jones, and İnanç Birol. "ABySS: A parallel assembler for short read sequence data". In: *Genome Research* 19 (June 2009), pp. 1117–1123. DOI: 10.1101/gr.089532.108.

[SWP+08] Jan Schröder, Lars Wienbrandt, Gerd Pfeiffer, and Manfred Schimmler. "Massively Parallelized DNA Motif Search on the Reconfigurable Hardware Platform COPACOBANA". In: *Pattern Recognition in Bioinformatics*. Ed. by Madhu Chetty, Alioune Ngom, and Shandar Ahmad. Vol. 5265. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 436–447. ISBN: 978-3-540-88434-7. DOI: 10.1007/978-3-540-88436-1_37.

[SWS12] Florian Schatz, Lars Wienbrandt, and Manfred Schimmler. "Probability model for boundaries of short-read sequencing". In: *2012 International Conference on Advances in Computing*

and Communications (ICACC). Aug. 2012, pp. 223–228. DOI: 10.1109/ICACC.2012.51.

[SX13] Kaiqi Sun and Yang Xia. "New insights into sickle cell disease: a disease of hypoxia". In: *Current Opinion in Hematology* 20 (3 2013), pp. 215–221. DOI: 10.1097/MOH.0b013e32835f55f9.

[SyC] IEEE.org. *1666-2011 – IEEE Standard for Standard SystemC Language Reference Manual*. accessed 11/2015. URL: http://standards.ieee.org/findstds/standard/1666-2011.html.

[Tan06] Andrew S. Tanenbaum. *Structured Computer Organization*. 5th ed. Upper Saddle River, NJ 07458: Pearson Prentice Hall, 2006. ISBN: 978-0131485211.

[TOP] TOP500.org. *TOP 500 The List*. accessed 11/2015. URL: http://www.top500.org/.

[TWJ+09] Wanwan Tang, Xuebing Wu, Rui Jiang, and Yanda Li. "Epistatic module detection for case-control studies: a bayesian model with a gibbs sampling strategy". In: *PLoS Genetics* 5.5 (May 2009). DOI: 10.1371/journal.pgen.1000464.

[VB13] Wim Vanderbauwhede and Khaled Benkrid. *High-Performance Computing Using FPGAs*. Springer, 2013. ISBN: 978-1-4614-1790-3.

[WBB+10] Lars Wienbrandt, Stefan Baumgart, Jost Bissel, Carol May Yen Yeo, and Manfred Schimmler. "Using the reconfigurable massively parallel architecture COPACOBANA 5000 for applications in bioinformatics". In: *Procedia Computer Science* 1 (1 May 2010), pp. 1027–1034. DOI: 10.1016/j.procs.2010.04.114.

[WBB+11] Lars Wienbrandt, Stefan Baumgart, Jost Bissel, Florian Schatz, and Manfred Schimmler. "Massively parallel FPGA-based implementation of BLASTp with the two-hit method". In: *Procedia Computer Science* 4 (2011), pp. 1967–1976. DOI: 10.1016/j.procs.2011.04.215.

[WC53] J. D. Watson and F. H. C. Crick. "A Structure for Deoxyribose Nucleic Acid". In: *Nature* 171 (4356 1953), pp. 737–738.

[WDR+10] Jing Wu, Bernie Devlin, Steven Ringquist, Massimo Trucco, and Kathryn Roeder. "Screen and Clean: a tool for identifying interactions in genome-wide association studies". In: *Gentic Epidemiology* 34.3 (Apr. 2010), pp. 275–285. DOI: 10.1002/gepi.20459.

[Wie13] Lars Wienbrandt. "Bioinformatics Applications on the FPGA-Based High-Performance Computer RIVYERA". In: *High-Performance Computing Using FPGAs*. Ed. by Wim Vanderbauwhede and Khaled Benkrid. Springer, 2013, pp. 81–103. ISBN: 978-1-4614-1790-3. DOI: 10.1007/978-1-4614-1791-0.

[Wie14] Lars Wienbrandt. "The fpga-based high-performance computer rivyera for applications in bioinformatics". In: *Language, Life, Limits*. Ed. by Arnold Beckmann, Erzsébet Csuhaj-Varjú, and Klaus Meer. Vol. 8493. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 383–392. ISBN: 978-3-319-08018-5. DOI: 10.1007/978-3-319-08019-2_40.

[WKG+14] Lars Wienbrandt, Jan Christian Kässens, Jorge González-Domínguez, Bertil Schmidt, David Ellinghaus, and Manfred Schimmler. "FPGA-based Acceleration of Detecting Statistical Epistasis in GWAS". In: *Procedia Computer Science* 29 (2014), pp. 220–230. DOI: 10.1016/j.procs.2014.05.020.

[WLF+11] Yue Wang, Guimei Liu, Mengling Feng, and Limsoon Wong. "An empirical comparison of several recent epistatic interaction detection methods". In: *Bioinformatics* 27.21 (2011), pp. 2936–2943. DOI: 10.1093/bioinformatics/btr512.

[WS10] Lars Wienbrandt and Manfred Schimmler. "Collecting Statistical Information in DNA Sequences for the Detection of Special Motifs". In: *Proceedings of BIOCOMP2010*. 2010, pp. 274–278.

[WSS12] Lars Wienbrandt, Daniel Siebert, and Manfred Schimmler. "Improvement of BLASTp on the FPGA-Based High-Performance Computer RIVYERA". In: *Bioinformatics Research and Applications*. Ed. by Leonidas Bleris, Ion Măndoiu, Russell Schwartz, and Jianxin Wang. Vol. 7292. Lecture Notes in

Computer Science. Springer Berlin Heidelberg, 2012, pp. 275–286. ISBN: 978-3-642-30190-2. DOI: 10.1007/978-3-642-30191-9_26.

[WTCCC07] The Wellcome Trust Case Control Consortium. "Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls". In: *Nature* 447.7145 (June 2007), pp. 661–78. DOI: 10.1038/nature05911.

[WYY+09] Xiang Wan, Can Yang, Qiang Yang, Hong Xue, Nelson L. S. Tang, and Weichuan Yu. "MegaSNPHunter: a learning approach to detect disease predisposition SNPs and high level interactions in genome wide association study". In: *BMC Bioinformatics* 10 (2009), p. 13. DOI: 10.1186/1471-2105-10-13.

[WYY+10a] Xiang Wan, Can Yang, Qiang Yang, Hong Xue, Xiaodan Fan, Nelson L.S. Tang, and Weichuan Yu. "BOOST: A Fast Approach to Detecting Gene-Gene Interactions in Genome-wide Case-Control Studies". In: *American Journal of Human Genetics* 87.3 (Sept. 2010), pp. 325–340. DOI: 10.1016/j.ajhg.2010.07.021.

[WYY+10b] Xiang Wan, Can Yang, Qiang Yang, Hong Xue, Nelson L.S. Tang, and Weichuan Yu. "Predictive rule inference for epistatic interaction detection in genome-wide association studies". In: *Bioinformatics* 26.1 (2010), pp. 30–37. DOI: 10.1093/bioinformatics/btp622.

[WYY+13] Xiang Wan, Can Yang, Qiang Yang, Hongyu Zhao, and Weichuan Yu. "The complete compositional epistasis detection in genome-wide association studies". In: *BMC Genetics* 14.7 (Feb. 2013). DOI: 10.1186/1471-2156-14-7.

[Xil] Xilinx Inc. URL: http://www.xilinx.com/.

[Xil10] Xilinx Inc. *Spartan-6 FPGA Configurable Logic Block – User Guide*. UG384 (v1.1). Feb. 2010. URL: http://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

[Xil11a] Xilinx Inc. *Spartan-6 Family Overview*. DS160 (v2.0). Oct. 2011. URL: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf.

[Xil11b] Xilinx Inc. *Spartan-6 FPGA Block RAM Resouces – User Guide*. UG383 (v1.5). July 2011. URL: http://www.xilinx.com/support/documentation/user_guides/ug383.pdf.

[Xil13] Xilinx Inc. *Spartan-3 FPGA Family Data Sheet*. DS099 (v3.1). June 2013. URL: http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf.

[Xil14a] Xilinx Inc. *7 Series DSP48E1 Slice – User Guide*. UG479 (v1.8). Nov. 2014. URL: http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

[Xil14b] Xilinx Inc. *7 Series FPGAs Configurable Logic Block – User Guide*. UG474 (v1.7). Nov. 2014. URL: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[Xil14c] Xilinx Inc. *7 Series FPGAs Memory Resources – User Guide*. UG473 (v1.11). Nov. 2014. URL: http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[Xil14d] Xilinx Inc. *Spartan-6 FPGA DSP48A1 Slice – User Guide*. UG389 (v1.2). May 2014. URL: http://www.xilinx.com/support/documentation/user_guides/ug389.pdf.

[Xil15a] Xilinx Inc. *7 Series FPGAs Clocking Resources – User Guide*. UG472 (v1.11.2). June 2015. URL: http://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf.

[Xil15b] Xilinx Inc. *7 Series FPGAs Overview*. DS180 (v1.17). May 2015. URL: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[Xil15c] Xilinx Inc. *Spartan-6 FPGA Clocking Resources – User Guide*. UG382 (v1.10). June 2015. URL: http://www.xilinx.com/support/documentation/user_guides/ug382.pdf.

[XLJ12] Minzhu Xie, Jing Li, and Tao Jiang. "Detecting genome-wide epistases based on the clustering of relatively frequent items". In: *Bioinformatics* 28.1 (Jan. 2012), pp. 5–12. DOI: 10.1093/bioinformatics/btr603.

[YHW+09] Can Yang, Zengyou He, Xiang Wan, Qiang Yang, Hong Xue, and Weichuan Yu. "SNPHarvester: a filtering-based approach for detecting epistatic interactions in genome-wide association studies". In: *Bioinformatics* 25.4 (2009), pp. 504–511. DOI: 10.1093/bioinformatics/btn652.

[YYW+11] Ling Sing Yung, Can Yang, Xiang Wan, and Weichuan Yu. "GBOOST: a GPU-based tool for detecting gene-gene interactions in genome-wide case control studies". In: *Bioinformatics* 27.9 (2011), pp. 1309–1310. DOI: 10.1093/bioinformatics/btr114.

[ZB08] Daniel R. Zerbino and Ewan Birney. "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs". In: *Genome Research* 18 (Mar. 2008), pp. 821–829. DOI: 10.1101/gr.074492.107.

[ZHZ+10] Xiang Zhang, Shunping Huang, Fei Zou, and Wei Wang. "TEAM: efficient two-locus epistasis tests in human genome-wide association study". In: *Bioinformatics* 26.12 (July 2010), pp. i217–i227. DOI: 10.1093/bioinformatics/btq186.

[ZKD+14] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. "UPC++: A PGAS Extension for C++". In: *2014 IEEE 28th International Parallel & Distributed Processing Symposium*. May 2014, pp. 1105–1114. DOI: 10.1109/IPDPS.2014.115.

[ZL07] Yu Zhang and Jun S. Liu. "Bayesian inference of epistatic interactions in case-control studies". In: *Nature Genetics* 39.9 (Sept. 2007), pp. 1167–1173. DOI: 10.1038/ng2110.

[Zwo03] Mark Zwolinski. *Digital System Design with VHDL*. 2nd ed. Prentice Hall, Nov. 2003. ISBN: 978-0130399854.