A Hybrid-parallel Architecture for Applications in Bioinformatics

M.Sc. Jan Christian Kässens

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2017

Kiel Computer Science Series (KCSS) 2017/4 dated 2017-11-08

URN:NBN urn:nbn:de:gbv:8:1-zs-00000335-a3
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 post@jan-kaessens.de

Published by the Department of Computer Science, Kiel University

Computer Engineering Group

Please cite as:

▻ Jan Christian Kässens. *A Hybrid-parallel Architecture for Applications in Bioinformatics* Number 2017/4 in Kiel Computer Science Series. Department of Computer Science, 2017. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Kaessens17,
  author    = {Jan Christian K\"assens},
  title     = {A Hybrid-parallel Architecture for Applications in Bioinformatics},
  publisher = {Department of Computer Science, CAU Kiel},
  year      = {2017},
  number    = {2017/4},
  doi       = {10.21941/kcss/2017/4},
  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. Manfred Schimmler Christian-Albrechts-Universität Kiel
2. Gutachter:	Prof. Dr. Bertil Schmidt Johannes-Gutenberg-Universität Mainz

Datum der mündlichen Prüfung: 10. Oktober 2017

Zusammenfassung

Seit der Einführung der Next Generation Sequencing (NGS)-Technologie steigen die Datenmengen aus der Sequenzierung von Genomen besonders schnell. Die Verfügbarkeit dieser Daten führt wiederum zu der Erschließung neuer Felder in der Molekular- und Zellbiologie, sowie der Genetik, die wiederum neue Daten erzeugen. Andererseits steigt die verfügbare Rechenleistung in Rechenzentren nur linear. In den letzten Jahren zeigte sich jedoch, dass neue Spezialhardware in Rechenzentren Einzug nimmt, insbesondere Grafikprozessoren (GPUs) und, weniger verbreitet, FPGAs (*field-programmable gate arrays*). Durch den Leistungsbedarf angetrieben begannen Entwickler Standardsoftware auf diese neuen Systeme zu portieren, um die speziellen Fähigkeiten ausnutzen zu können. Systeme, die GPUs und FPGAs gemeinsam nutzen, sind jedoch selten zu finden. Besondere Herausforderungen stellt dabei einerseits der Bedarf an tiefgehendem Know-How in zwei diametral verschiedenen Programmierparadigmen dar, sowie das nötige Ingeneurswissen um das für heterogene Systeme typische Nadelöhr in der Kommunikation zwischen den beteiligten Geräten.

Für diese Arbeit wurden zwei Algorithmen aus der Bioinformatik für die Implementierung auf einer neuen, hybrid-parallelen Rechnerarchitektur und Softwareplattform ausgewählt, die die Vorzüge von GPUs, FPGAs und CPUs auf besonders effiziente Weise nutzt. Es wird gezeigt, dass eine solche Entwicklung nicht nur möglich ist, sondern die Rechenleistung homogener FPGA- oder GPU-Systeme vergleichbarer Größe übertrifft und dennoch weniger Energie benötigt. Beide Methoden werden genutzt, um Fall-Kontroll-Daten aus Assoziationsstudien auszuwerten und auf Interaktionen zwischen zwei oder drei Genen zu analysieren. Insbesondere bei letzterem zeigt sich, dass die neu gewonnene Rechenleistung es erstmals ermöglicht, größere Datenmengen zu untersuchen, ohne ganze Rechenzentren über Wochen auszulasten. Der Erfolg der Architektur führt schließlich zu der Entwicklung eines Hochleistungsrechners auf Basis des vorliegenden Konzeptes.

Abstract

Since the advent of Next Generation Sequencing (NGS) technology, the amount of data from whole genome sequencing has been rising fast. In turn, the availability of these resources led to the tapping of whole new research fields in molecular and cellular biology, producing even more data. On the other hand, the available computational power is only increasing linearly. In recent years though, special-purpose high-performance devices started to become prevalent in today's scientific data centers, namely graphics processing units (GPUs) and, to a lesser extent, field-programmable gate arrays (FPGAs).

Driven by the need for performance, developers started porting regular applications to GPU frameworks and FPGA configurations to exploit the special operations only these devices may perform in a timely manner. However, applications using both accelerator technologies are still rare. Major challenges in joint GPU/FPGA application development include the required deep knowledge of associated programming paradigms and the efficient communication both types of devices.

In this work, two algorithms from bioinformatics are implemented on a custom hybrid-parallel hardware architecture and a highly concurrent software platform. It is shown that such a solution is not only possible to develop but also its ability to outperform implementations on similar-sized GPU or FPGA clusters in terms of both performance and energy consumption. Both algorithms analyze case/control data from genome-wide association studies to find interactions between two or three genes with different methods. Especially in the latter case, the newly available calculation power and method enables analyses of large data sets for the first time without occupying whole data centers for weeks. The success of the hybrid-parallel architecture proposal led to the development of a high-end array of FPGA/GPU accelerator pairs to provide even better runtimes and more possibilities.

Acknowledgments

I would not have been able to complete this work without the guidance and support of a lot of people and I would like to take this opportunity to express my gratitude.

Probably the most important person during my academic career was my supervisor, mentor and leader of the Computer Engineering Group, Manfred Schimmler. From my second semester on, I regularly visited his tutoring lessons and got to know an enthusiastic lecturer that never failed to motivate me, and he obviously also didn't fail this time. Besides his deep knowledge and his readiness to provide funds for prototype systems with uncertain value, this is probably the quality with the highest importance for me, thank you. On the administrative side, I wish to thank Brigitte Scheidemann, who did what can best be summarized as "always knowing what to do, even if or especially when nobody else did." I further wish to thank Bertil Schmidt for his continued guidance in publishing articles and fruitful discussions about new technology, Andre Franke and David Ellinghaus for supporting my research even though I should probably have done other things in my working time.

Every scientist in academia is only as good as his team and this is especially the case with me. Thank you for being my colleagues, Vasco Grossmann, Sven Koschnicke and Christoph Starke, for listening to my sometimes confused ideas and providing me with chocolate and sweets of all kinds. Special thanks go to my tutor, mentor, room mate and true friend (in temporal order) Lars Wienbrandt. A rather outstanding property of having him as a colleague and friend is the ability to discuss problems, both personal and professional, in a unique, in-depth, sometimes emotional and sometimes rough way, with him always being open and forgiving. Besides our relationship, he was the one that pulled me into working with FPGAs and eventually hybrid computers. Thanks for what you have done so far.

On the non-academic side, I would like to express my gratitude to my family who always covered my back, especially to my wife Patricia and my daughter Freya who had to do an awful lot of time without me, even when I was technically present, but never complained and always beared with me when things didn't go well. I will never forget that.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Regular desktop computers have been heterogeneous systems since the advent of integrated processors. Every computer chip within such a system has a clearly defined responsibility. For example, a microcontroller processes readings from fan speeds and temperature sensors and moves them to a dedicated system management bus controller (SMBus). Then, they are moved to the central processor for evaluation. If an application decides that a temperature might be too high, a command is issued to the SMBus controller, which in turn instructs the sensor processing controller to raise the fan speed. Additionally, this change in temperature is recorded to the hard disk by first instructing the hard disk bus controllers to move data to the hard disk-integrated firmware controller which writes data on the physical platters though a multitude of intermediate devices.

Most parts of this ecosystem are somehow integrated and encapsulated in a way that the user rarely even knows about their existence. Microcontrollers, bus controllers and other small-scale processors are usually dedicated to certain purposes and have just enough computational power to handle their workloads on time.

Later that time, *accelerators* were added to perform certain computations that could have been done on the CPU but that the accelerator is specifically designed to perform. The purpose of a CPU is to do all kinds of work but specializations usually outperform generalizations with their respective operations. However, they do not automatically accelerate applications. Instead, these applications have to be specifically designed to exploit the available features.

Certainly the most prominent accelerator is the graphics processing unit (GPU). Originally, they have been implemented to help computer-aided

design software (CAD) and video games in calculating transformations in affine and euclidean 2D and 3D spaces and draw a projection of the vector space to a rasterized display, such as computer screens. GPU functions were soon enriched with texture filtering, lighting through radiosity, ray tracing and other complex processes in fixed pipelines. After adding more and more functions, the fixed function pipeline became too static and vendors chose to expose the GPU instruction set to the system, and the concept of general-purpose graphics processing units was born (GPGPU). Scientists started to move climate prediction models, protein folding methods and genome analyses to GPUs to exploit the vectored and parallelized instruction sets and achieved speed-ups that can be reasonably measured in orders of magnitude.

1.1 Motivation

Although GPUs open the world of high performance computing to regular computers, their specialization still lies on graphics. The above mentioned functions typically require low-precision floating-point calculations in dot products, matrix multiplication, and generally multiply/add-heavy algorithms. Field programmable gate arrays (FPGAs), although less popular and more complex, can be used as accelerators to enhance a vastly different type of computation. Their strength lies in the flexibility of configuration. No instruction sets, no scheduler and (almost) no fixed-function entities restrain the kind of work FPGAs can perform. The developer directly imbues combinational and/or sequential logic into the device to connect the different pins and peripherals with each other. This great flexibility comes at a cost. Development time and debugging difficulty tend to be very high while the achievable clock frequency is rather low when compared to GPUs or CPUs. However, this also allows developing applications that could not be run on CPUs and GPUs efficiently, such as systolic structures where large numbers of primitive nodes concurrently work on data streams. One popular example is the development of polynomial evaluators. Using Horner's Rule, building an FPGA configuration to evaluate a several-thousand-degree polynomial every clock cycle is trivial (see Section 2.3.2 for details), while it

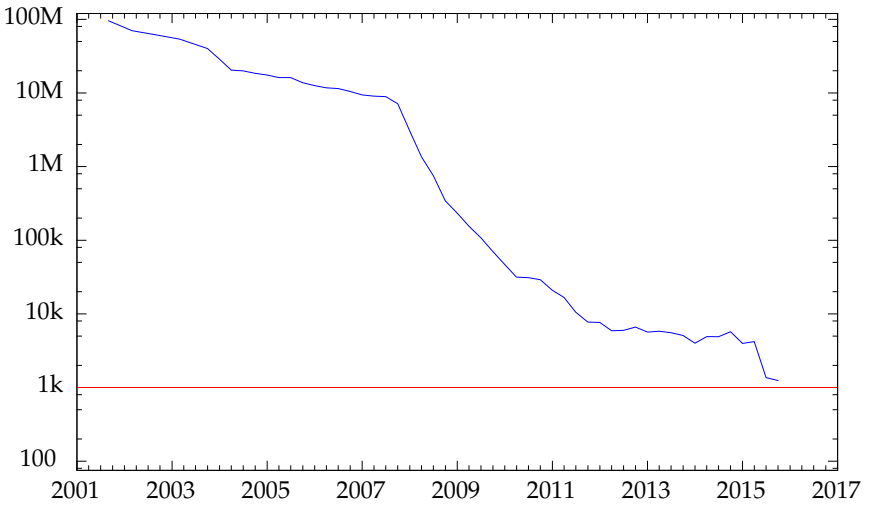

Figure 1.1. Human genome sequencing costs in USD [Wet16]

is hardly possible to do on GPUs and certainly impossible on CPUs in the same time frame.

After all, the relatively low frequencies achievable on FPGAs also reduce energy consumption but the flexibility and concurrency potential can easily make up for it as this work shows. This, however, requires the developer to extensively exploit this potential. Plain sequential processes, complex arithmetical issues or dynamic programming solutions will not perform well but if they can be translated into more suitable structures, FPGA designs become able to generate results faster than any other platform while still keeping the energy footprint low.

Problems that can be partitioned into suitable workloads for GPUs, FPGAs and CPUs will therefore be extremely efficient when compared to implementations on any of these alone. Bioinformatics, for example, is a field where large amounts of data, typically DNA, are analyzed in pipelines with clearly defined stages and therefore provides a workable field for acceleration through the hybrid-parallel architecture proposed in this work.

Next-Generation Sequencing (NGS) revolutionized biotechnology by making fast and economical profiling of genomes possible. Since its introduction, the trend continues as can bee seen in Figure 1.1 and the acquired amounts of data rise faster than the available computing power. Empowered by these new methods, scientists in medicine, genetics and molecular biology discovered genetic features that might be responsible for certain diseases — and created the field of personal medicine and, of course, completely new workloads for already busy data centers.

The rich field of bioinformatics is therefore a grateful target for the development of novel high-efficiency architectures. Although many methods and solutions exist that focus on data centers and GPU clusters, no single implementation is known to exploit the best of all three acceleration models. Graphics cards set high standards for computational power, their commercial and academic communities are vibrant and the programming APIs exceptionally well-designed. FPGA cards, however, do not provide standardized interfaces, simple programming mechanisms and communities with pre-built libraries for all kinds of tasks. The major challenge in developing a hybrid system to cherry-pick from all architectures are the programmability and interconnection and this is exactly what the work at hand focuses on. Moving data between threads or programs on a single device alone often becomes a bottleneck in an application's performance. Moving data between *devices* requires sophisticated threading models, highly efficient buffer management, extensive knowledge of communication protocols and deep integration with all involved operating system drivers. However, this work aims to prove that it is not only possible implement. Instead, it is shown that the hybrid realization of two methods for analysis of genome-wide association studies outperforms all known methods on performance-wise comparable, non-hybrid devices.

A genome-wise association study (GWAS) is often designed by selecting a particular disease, e.g. ulcerative colitis, and collecting human samples that are known to be healthy with respect to this disorder, the *control group*, and samples that do possess colitis, the *case group*. Each individual has its genome sampled at specific places, *loci*, where each locus has one of two or more types. The methods discussed and used in this work try to discover combinations of loci where certain values within the combination

lead to a shift in the case/control distribution. These analyses are performed on contingency tables, i.e. multi-dimensional data points where one table shows the distribution of sampled values of a single locus with respect to the group membership. Therefore, a simple, first-order analysis requires P contingency tables to be created and evaluated. With the number of loci P quickly raising towards millions, even simple statistical tests can become a computational burden for conservative architectures. The superior power of the proposed hybrid system becomes obvious when not only analyzing single contingency tables but all two-combinations. The demand in computational resources rises exponentially in the order of interaction, i.e. P evaluations for single order become almost P^2 evaluations for second order, and so on. For third and higher order interactions, it is almost impossible to find established methods as only very few scientists have access to the computational power required to do these kinds of analyses. For comparison, a sufficiently modern CPU-based system requires a little more than 115 hours for our first method on a reference data set. The implementation of the same method on the hybrid system is faster by a factor of 595, requiring only 12 minutes. For third order analysis, a novel method to model gene-gene-gene interactions based in approaches from information theory is presented and implemented for comparison. Here, due to technical reasons, the speed-up is limited to 72 with respect to a highly optimized CPU-based implementation but should easily achieve higher numbers when these hardware limitations are lifted.

1.2 Related Work

Although hybrid architectures of this type have not been observed in these studies, they have already been proposed in other scientific fields. In 2014, Kocz et al. presented a similar system consisting of CPUs, FPGAs and GPUs to analyze cross-correlation between signals from numerous radio antennas used in astronomics [KGB+14]. The FPGA part in their system samples analog signals from antennas through external analog-digital-converter and applies bandwidth filters through stream-oriented real-time Fourier transforms before sending data to a GPU for the creation of cross-correlation

matrices. Architecture-wise, the design is similar to the proposed bioinformatics platform. Regarding the data flow, though, the throughput requirements are different. Most importantly, the FPGA nodes are solely used for data acquisition, yielding a real-time data rate requirement of less than 50 kiB/s, whereas the data rates between CPU and GPU reaches 51 GiB/s (approx. 6 GiB/s). Due to this asymmetric design, both architectures are solely comparable on a conceptual level.

In [IBS12], Inta et. al. propose an interesting "Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform." They show how algorithms might be implemented on hybrid platforms, e.g. Monte-Carlo classification for approximations of π or Fast Fourier Transform, and describe their hardware components in detail. Also, the interconnect between the individual components is presented to be the central bottleneck when it comes to performance. Unfortunately, they decided to explicitly ignore this limitation as future communication solutions would solve this kind of problem. Therefore, algorithm performance is predicted while assuming the interconnect keeping up, although precisely the communication link has always been known to be the bottleneck in all scales of distributed systems. Additionally, the data and sampling rates are extrapolated from measurement time frames in the order of microseconds and nanoseconds without analysing its uncertainty, making comparisons to actual implementations hard. The authors state that a high-performance backplane is being developed to directly connect FPGAs and GPUs, in addition to a suite of custom kernel drivers to provide an easy-to-use interface. However, by the time of this writing, no follow-up article has been published.

In the field of cryptanalysis, [KL10], clusters of FPGA/GPU computing nodes are used to crack a range of hashes, i.e. finding a plain text that leads to a specific cryptographic hash value. The cluster nodes are connected via Microsoft's Message Passing Interface (MPI) and high-bandwidth Infiniband links, allowing for Remote Direct Memory Access (RDMA). Regarding the concept, FPGAs and GPUs are both used to perform the same task while a CPU dispatches units of work to the locally connected FPGA and/or GPU accelerators on demand. In this work specifically as well as in cryptanalysis generally, the typical workload consists of distributing a hash value to one or more computation devices and waiting for at least one node to produce

a result. Hence, the communication between nodes is non-existent while communication demand between host systems and their accelerators is at a low level. As with [IBS12], the authors state that the cluster is not fully operational yet, as only GPUs can be used due to problems with FPGA designs. Therefore, no performance measurements or predictions have been published.

Another field with a high demand on computational power is physics simulation. In [KDH+06], Kelmelis et. al. propose a desktop-type hybrid computer to model signal propagation through large three-dimensional meshes. The authors put much work into problem separation and achieved runtimes of two days where a standard computer "would have required over a month." However, the problem data is transmitted to the devices in a slow transmission process before the actual calculation starts and does not require device-level concurrency or communication while the calculation is running. As parallelism is only used within a single device, the higher-level process can better be described as sequential.

In [BRF14] and [Gil15], the focus is brought to a sub-aspect of hybrid computing, specifically, the communication part that has not been discussed in great detail in the previously mentioned publications. The authors reverse-engineered the DMA modules of proprietary NVIDIA driver software, modified PCI Express drivers and implemented OpenCL-based interfaces to access the newly-implemented functionality. Using these techniques, Gillert established a direct link between an FPGA and a GPU using standard motherboard features. Thanks to his highly specific knowledge of hardware and software drivers, the achieved data rates from FPGA to GPU and GPU to FPGA were 740 MiB/s and 525 MiB/s, respectively. The theoretically achievable net data rates in corresponding technical specifications is approx. 3800 MiB/s, in both directions (full duplex).

Regarding the field targeted by this work, bioinformatics, available computational resources are dominated by traditional CPU clusters, while CPU/GPU and GPU-only solutions are gaining momentum. In an extensive 2016 survey, [NCT+16] lists 12 CPU/GPU implementations of various methods from molecular biology over sequence alignment to systems medicine with speed-ups ranging from 5 to 50 with respect to a contemporary CPU-based computing node. GPU-only solutions in the same fields

with comparable methods yield speed-ups up to 39. With regard to FPGA/CPU hybrids, only few implementations exist, notably [CSM12]. Chen, Schmidt and Maskell proposed a short read mapping method based on the popular banded Needleman-Wunsch algorithm[NW70]. In optimal global alignments like this, the alignment step is a computationally intensive task where the transmission of data (i.e. nucleotide or amino acid sequences) is runtime-wise negligible.

In 2016, Lars Wienbrandt published an extensive evaluation of FPGAs in bioinformatics, providing implementations of sequence alignment, gene-gene interactions and SNP imputation [Wie16]. It is shown that FPGAs and clusters thereof are capable of outperforming CPU clusters both performance and energy-wise. In fact, Wienbrandt covers the groundwork that lead to the development of early prototypes of parts of the hybrid-parallel computer proposed in this work. Specifically, many thoughts and processes were implemented on the Xilinx KC705 Development Board described in Chapter 4.1.3.

1.3 Structure of This Work

The remaining chapters of this thesis are summarized as follows.

The second chapter, *Architecture Essentials*, aims to impart thorough knowledge about architectural components that form the basis of the proposed hybrid computer. Therefore, a deep understanding of the programming of CPUs, GPUs and FPGAs is built up. As the focus in this work does not lie on these fundamentally individual devices, but on the actual *interconnects*, several programming paradigms and hardware interfaces are discussed. This includes levels of parallel programming, from the lowest in CPU microcode to mid-level many-core applications, and extends to GPU warp scheduling and systolic structure descriptions on FPGAs. Hardware-wise, techniques are presented that enable the above components to interchange data in very efficient ways by using embedded controllers and low-overhead communication protocols, most notably Direct Memory Access (DMA) and PCI Express (PCIe), that form the backbone of the proposed system.

Before the actual implementation can be described, Chapter 3 provides an introduction to the biomedical background that is essential to an understanding of the methods involved. As this work covers two almost diametrically opposed fields of science, it begins with an introductory-level primer on genetics, specifically DNA, chromosomes and genes, and continues with allele configurations, association study design and types of interactions between genes. The second part describes an approach to extract information from said studies by employing methods from statistics and information theory, and therefore, the nature of the applications that have been developed for the proposed hybrid computer. Furthermore, the computational challenges and burdens that need to be addressed are emphasized, such as the large amount of statistical model evaluations.

This continues into the first part of the following main chapter, *The Hybrid Architecture Prototype*, where these challenges are projected to the architectural level. The general strategy in hybrid design, divide-and-conquer, and the actual problem partitioning model is presented, along with the problems that need to be faced, such as the high volumes of data that need to be moved just in time. This indeed becomes a major aspect that is present in all levels of the implementation and emphasizes the need for the most sophisticated communication handling, algorithms, and lead to the spawning of novel data structures such as the *fixed-capacity fine binary min max heap* and the *page-locked buffer stack*. The overall structure of this chapter can best be described as bottom-up. First, the detailed implementations of the GPU kernels and FPGA cores are discussed, including the required transformations of mathematical models and technical structures. Second, the driver and API modules are described that provide low-level interfaces between the operating system kernel, GPUs, FPGAs and the application layer which is presented in the final part. Here, all application subsystems are implemented, from the loading of data files to memory management and eventually the post-processing of results where new data structures and algorithms are shown to be correct and efficient.

Chapter 5, *Evaluation*, with a discussion about how performance can actually be measured across platform boundaries. The various established metrics are presented, including the popular FLOPs (floating-point operations per second), and why these are not sufficient for the special case

of hybrid-parallel FPGA/GPU computers. Very specific measurements are introduced to guide the evaluation of performances with respect and in relation to other components, methods and architectures. The proposed implementations are evaluated in parts to identify possible bottlenecks and establish results that allow deep analyses on the general concept of FPGA/GPU hybrid computing and the specific implementation of the second-order and third-order interaction methods. It is shown that although no other hybrid computer is able to maintain the achieved data rates, the usual bottleneck of communication does not need to be a boundary in hybrid computing when transmission handling is done in a thoughtful and sensible way, even when it comes to data rates in the order of 3 GiB/s. Moreover, significant performance increases could be observed over CPU-based reference implementations. However, due to the lack of available architectures that enable third-order interaction analysis, the prototype system is compared against a hypothetical high-performance system.

In the remaining chapter, the previously established results are summarized and the general suitability for hybrid computers for these types of problems assessed. A preview is given on ongoing work with the successor of the prototype design, a server-grade computer with four high-end FPGA and GPU boards, large amounts of memory and two state-of-the-art 8-core SMT CPUs. Additionally, extensions and future work items are discussed that may enhance the driver interfaces and device communication patterns or raise the quality of results produced by the newly designed third-order implementation of *mutual information*.

Chapter 2

Architecture Essentials

In this section, an overview of the various architectures and their underlying technologies is provided. Before describing and analyzing the concepts and implementation of the methods, a certain knowledge about the hardware composition and developed software is necessary. The following sections help providing this knowledge, focusing on the parts that are relevant for later chapters.

Although most parts of the computational workload does not lie on the system's CPU but on GPUs and FPGAs, it is responsible for managing the memory-intensive data flows, a highly sensitive component as the overall performance superiority rises and falls with the stability and throughput of these data channels. Therefore, short historical overviews are given as an introduction to CPU, GPU and FPGA design and the more delicate matters such as *Direct Memory Access* to understand why the selected devices are the devices of choice for this task and how they can be programmed to provide data operations and memory throughput in the range of several gigabytes per second. Additionally, a few methods and paradigms used in parallel programming are introduced to allow a much more thorough knowledge base on the general practices performed during the course of this work.

Finally, more insights into the inner workings of the most important communication interface in the hybrid architecture, PCI Express, are presented.

2.1 Architectures Overview

2.1.1 Central Processing Units (CPUs)

When the first microprocessors for consumer electronics emerged, block diagrams describing all the functional units could be drawn on a single piece of paper. One representative example, Intel's popular 8080 processor released in 1974 and often described as the "first truly useful microprocessor", only contained a register bank, an arithmetic logic unit (ALU), an instruction decoder and a few multiplexers and flip-flops. Two bus systems, a data bus and an address bus, connect this CPU to peripheral hardware. In the following architectures, the operating frequencies rose while more and more functionality was added. The most interesting extensions are shown in Table 2.1. Notable developments are the introduction of a memory management unit (MMU) in the mid-1980s, and the rise of vectoring instruction sets such as SSE and AVX from 1993 onward. Until the year 2000, CPU core frequencies were increased.

By then, processors had approached the 4 GHz line. Due to switching characteristics of transistors, the power dissipation of a CPU is approximately proportional to the clock speed and (electrical) capacitance, and quadratic to its supply voltage [Cor04]. For achieving high clock rates, developers therefore tried to reduce the voltage and capacitance by choosing special low power transistors and increasing the integration density, thus reducing the capacitance by narrowing and shortening wires. With higher densities though, more heat has to be moved away from the chip per unit of area. The total limits of stable operating frequencies are therefore given by the heat dissipation and conduction of the chip and its package as well as the lowest possible voltage for its transistors. As these limits have recently been approached, developers have moved from increasing the clock rates to reducing the number of cycles per instruction, increasing the number of instructions per second by other means than frequency, and increasing the data width per instruction. These techniques are explained in the following sections.

Table 2.1. CPU Technology Timeline

Year	Microarchitecture	New features
1974	Intel 8080	ALU, register bank
1976	Zilog Z80	DRAM refresh controller, interrupt handling
1982	Intel 80286	Microcoded multiplier, protected mode, memory segmentation
1985	Intel 80386	Paged memory
1989	Intel i486	Floating-point unit
1993	Intel P5	Instruction-level parallelism (superscalarity)
1997	Intel P6	SIMD instruction sets (MMX and SSE)
2000	Intel Netburst	SSE2, SSE3, multi-core, symmetric multithreading (SMT), virtualization
2008	Intel Nehalem	PCI Express and DDR3 SDRAM controllers, integrated GPU
2013	Intel Haswell	SSE4, BMI1–3, FMA3, AVX2 SIMD instruction sets, 3D accelerator
2015	Intel Skylake	USB 3.1, DDR4 controllers, integrated voltage regulators
2016	Intel Xeon Phi x200	Many-core architecture (up to 72 per chip), Hybrid Memory Cube (HMC) controller

Instruction Set Architecture (ISA)

General-purpose as well as special-purpose processors are generally classified and defined upon their set of supported instructions. Before the year 2000, instruction set architectures could be divided into two design philosophies, *Complex Instruction Set Computers (CISC)* and *Reduced Instruction Set Computers (RISC)* [PS81]. RISC designs tried to keep instructions as simple as possible but allow the scheduling of more than one instruction per clock cycle by the introduction of a superscalar instruction pipeline. Here, multiple independant operations are carried out in the same pipeline stage in a parallel way. The constraint of simplicity also mandated that memory-

accessing instructions be separated from calculation instructions. Therefore, RISC-based architectures are often called *Load/Store-Architectures* [Fly95]. Complex Instruction Set Computers instead used instructions that combined these functions. For example, the `ADD` instruction in a CISC system typically allows the specification of a memory address as shown in the Listings 2.1 and 2.2.

Listing 2.1. In-memory addition in RISC systems

```
MOV r1, [y]      ; load from memory locations y and z
MOV r2, [z]      ; into registers r1 and r2
ADD r3, r1, r2   ; r3 := r1 + r2
MOV [x], r3      ; store r3 into location x
```

Listing 2.2. In-memory addition in CISC systems

```
MOV r1, [y]      ; load from memory locations y to r1
ADD r3, [z], r2  ; load first operand and add r2
MOV [x], r3      ; store r3 into location x
```

In this case, the CISC `ADD` instruction allows one operand to be a memory location (or a register containing a memory location), while this is generally not implemented in RISC systems. Although the CISC program is one instruction shorter, the added complexity in an instruction with memory access may or may not be able to be processed in one clock cycle. Hence, the runtime or efficiency cannot simple be deduced from the number of instructions in a program.

Modern processors through, can usually not be clearly distinguished into RISC and CISC anymore. Many RISC processors implement more complex instructions and allow memory operations within arithmetic operations while CISC processors support more simple but fast instructions and use RISC-like superscalarity concepts:

- Instructions are conceptually processed from a sequential instruction stream

▻ Processors check for data dependencies between eligible instructions during runtime (as opposed to compile-time checking)

Although these terms are still used in many recent articles, the lines between CISC and RISC become increasingly blurred. This lead to complex, fast and efficient processors that have built-in concepts for automatic parallelization through superscalarity, simple instructions fused together (i.e. fused multiply/add) and wide data instructions.

In many applications, it is necessary to count the number of set bits in a vector, including cryptography/cryptanalysis and hash table operations. This number is also called the *Hamming weight* of a bit string, or *population count*. Although many more sophisticated algorithms exist to determine the Hamming weight, it might be implemented in the naïve way as shown in Listing 2.3.

Listing 2.3. Naïve implementation of Hamming Weight in C

```
unsigned NaiveHamming(unsigned value) {
  unsigned count;
  for(count = 0; value != 0; value >>= 1)
    count += value & 1;
  return count;
}
```

Listing 2.4. Naïve Hamming Weight (compiled)

```
  XOR %hamming, %hamming ; zero hamming weight counter
loop:
  AND $1, %value         ; stores lowest bit in %result
  ADD %hamming, %result  ; add %result to %count
  SHR %value             ; shift %value by one to the right
  JNZ loop               ; jump to beginning if %value is not zero
```

The x86 assembly code shown in Listing 2.4 has been obtained by compiling the C code block in Listing 2.3 with GNU GCC 5.4 and all

general and processor-specific optimizations enabled. For easier reading, the instructions have been annotated. It can be seen that even if we (quite unrealistically) assume that every instruction takes exactly one clock cycle, evaluating a 32-bit value would take 128 clock cycles. Better runtimes can be achieved by using binary counter trees, large look-up tables and other skillful implementations [TAOCP]. However, none of these methods reach the performance of the `POPCNT` instruction that has been introduced to current processors in the SSE 4.2 instruction set (see Sect. 2.1.1) and taking only one clock cycle with a latency of two clock cycles for any 64-bit integer word on an Intel Skylake architecture [Fog16].

While `POPCNT` is one of the simpler instructions, SSE 4.2 (and any other version, for that matter) introduced much more complex ones, such as `DPPS`. To speed up matrix multiplication that is often used in graphics processing, `DPPS` allows performing four fused multiply-and-add operations in a single instruction [IA32, page 293]. Many of these combined and/or vectored instructions make use of special vectoring registers. With the various versions of SSE and AVX, additional registers have been introduced to support 128-bit, 256-bit and 512-bit registers (`xmm`, `ymm` and `zmm`, respectively).

Unfortunately, many compilers, including GCC 5.4, are not able to detect situations where the usage of these extensions would be appropriate, as can be seen in the above Listings 2.3 and 2.4. For these cases, most C/C++ compilers allow the emission of inline assembly code so the software developer may force the respective instruction's usage. One large disadvantage is that the software developer is required to know the exact semantics and instruction sets of the underlying processor where the compiler generally tries to keep the language abstract from these hardware details, for the sake of platform independency. The resulting code is not only less readible and thus less maintainable, the program compiled from Listing 2.5 will not run on processors pre-dating SSE 4.2 introduction without changing the source code and recompiling.

Listing 2.5. The `POPCNT` instruction in inline assembly

```
unsigned InlinePopcount(unsigned value) {
  unsigned count = 0;
```

```
  __asm__ (
    "POPCNT %buffer, %count;"
    : [count] "=&r" (count)
    : "value" (value));
  return count;
}
```

As many others, the GCC authors introduced so-called *intrinsics*. These special functions are not kept pre-compiled in any standard libraries as regular functions but the compiler itself has an intimate knowledge about what these functions do and may create them on-the-fly. This allows compilers for much better integration and is one of the few ways to place a hint on the semantics of the source code.

Listing 2.6. Intrinsic population count function

```
unsigned IntrinsicPopcount(unsigned value) {
  return __builtin_popcount(value);
}
```

When Listing 2.6 is compiled by GCC 5.4, a `POPCNT` instruction is emitted if the target processor supports it, a library implementation otherwise. Although this resolves the problem of processor dependency, it is still compiler-dependent.

GCC supports a large number of built-in functions to aid its detection of semantics. Some of them are of the "instruction type" as `POPCNT` is, and others are functions that are not covered by library functions and/or language standards:

- `__builtin_cpu_supports`: allows querying for specific instruction set support, i.e. `__builtin_cpu_supports("sse4.2")`.
- `__builtin_ia32_crc32si`: uses SSE 4.2's instruction to compute the 32-bit Cyclic Redudancy Check (CRC32) checksum
- `__builtin_mul_overflow`: multiplies two integers and reports whether the multiplication lead to an integer overflow

- `__builtin_expect`: places a hint on a conditional statement to let the CPU's branch prediction unit know what outcome a boolean expression is likely to have
- `memcpy`: copies memory from one region to another, exploiting vector copying instructions, if possible. This is an example of a function shadowed by an intrinsic. It is non-standard behavior and the compiler will fall back to the library function if it is run in *strict standards compliance mode*.

Althrough population count is a rather recent addition to modern instruction sets, it is also rather simple when compared to other recent additions. `VRSQRT28PD` from the "AVX-512" instruction set, for example, computes reciprocals of square roots of 16 single-precision floating-point numbers in a single instruction. There also exist instructions for fused multiply/add operation, dot product computation, comparison and copying for 512-bit vectored numbers. For simplicity, this section only showed the usage of these instructions for `POPCNT` but the same principles basically apply for any other (more complex) instruction.

Simultaneous Multi-Threading and Many-core Architectures

In the previous section, the term "superscalarity" has been used to describe a type of instruction-level parallelism (ILP) where several instructions taken from a sequential stream are executed in parallel in the same pipeline stage in a processor. Due to the high complexity of implementing a superscalar instruction pipeline, efforts have been made to find alternatives for a hardware implementation.

Very Long Instruction Word (VLIW) processors support custom, variable-length instructions. These can be used by compiler software to analyze the instruction stream that would be executed and pack multiple instructions into a single instruction (*instruction group*) where the parts can be processed in parallel. This technique moves the burden of dependency checking and scheduling from the processor hardware to the compiler software. As simpler hardware is less expensive and may consume less energy, this type of architectures is becoming popular in embedded systems [Fis83]. 12 years

after the introduction of VLIW, a partnership of HP and Intel introduced the Explicitly Parallel Instruction Computing (EPIC) architecture [SR00]. Directly evolved from VLIW, EPIC overcomes some of its short-comings and adds new features from other architectures. Today, as with CISC and RISC, VLIW and EPIC architectures have a large set of features in common and it is becoming increasingly hard to differentiate one from the other.

In general-purpose server and consumer hardware, VLIW or EPIC is not widely used. Instead, simultaneous multithreading (SMT) is popular among wide ranges of current processors. In this technique, a CPU core is presented to the operating system as two or more (to some extent) independent CPU cores. This happens in a way that it is generally not trivial for a computer program to detect whether the underlying hardware uses several physical CPUs or several *virtual* cores from a single physical CPU. Although all virtual cores do provide the same functions and same instruction sets, putting them under maximum load, measurements show that the overall performance gain over a single core is approx. 30 % with two-way SMT. This number highly depends in the workload, though [MBH+02]. The explanation is simple. Virtual cores are emulated by single superscalar instruction pipeline. To support the high level of parallelism on the instruction-level and thread-level, many hardware modules are available several times. On the other hand, being a single physical module, all simultaneous threads of execution share caches, main memory, system buses and external peripherals through a shared communication endpoint. Therefore, the highest performance outcome can achieved by running programs on these cores that are as different as possible regarding the instruction characteristics. For example, one thread may be a memory-heavy program while the other focuses in arithmetic problems. Hence, a part of the speed-up through SMT usage may be accounted for I/O and memory latency hiding.

Thread-level parallelism is also brought to a higher level by introducing additional physical processors which, in turn, provide superscalar instruction pipelines by themselves, and therefore more virtual cores. Most processor architectures that have been developed in the last few years have implemented more pipelines by housing several fully independent physical CPUs within the same integrated circuit. Typically, a processor package also houses several controllers, instruction caches, data caches, and other

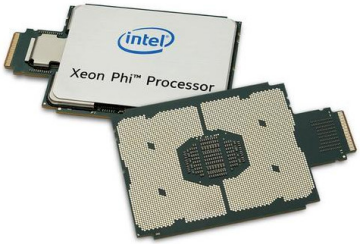

Figure 2.1. Intel Xeon Phi standalone host processor [X200]

peripherals. While cores in a *multi-core* processor are indeed independent, they usually share higher-level caching memories and bus interfaces. This separates multi-core processors from actual separate CPU chips where each package contains a full set of cache hierarchies, frequency and voltage regulators and memory controllers. The amount of cores per package ranges up to several hundreds of cores where multi-core processors become *many-core* processors [Vaj11].

One notable example is the Intel Xeon Phi many-core series. Starting as a PCI Express-connected add-on co-processor cards, on June 2016 Intel launched the "Xeon Phi x200" product line of standalone host processors [X200]. Each device, depending on the model, supports up to 72 cores with four-way SMT and a clock frequency of 1.5 GHz, resulting in 288 AVX-512-capable virtual cores. Additionally, it is typically possible to use more than one of these processors in a single system.

Memory

Most of the vectoring extensions and other forms of parallelism introduced earlier are useless if the choke point of the system is memory access. Hence, it is crucial to understand how and when memory can be accessed efficiently. This knowledge is discussed in the following section.

Most modern general-purpose computers support multi-tasking. Several processes share the same CPU and the same main memory, usually with a high degree of transparency to the application. Each process, or more

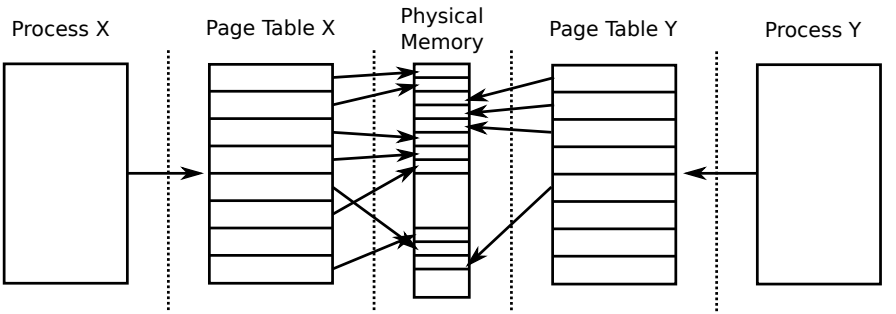

Figure 2.2. Virtual memory translation

specifically, each *thread of execution*, possesses its own set of virtual CPU registers that can almost arbitrarily be saved and restored by the operating system's task scheduler. These context switching operations are often directly supported by specialized CPU instructions. Sharing main memory is different and conceptually more complex. However, the following sections require a thorough understanding of memory organization as the proposed components make heavy use of the various memory abstraction layers involved.

Operating system kernels and CPUs that implement *virtual memory* allow the same level of transparency for memory as well. Each process is therefore presented its own, private main memory and may assume that neither other processes can read or manipulate these ranges, nor the process itself can manipulate foreign memory. Notable exceptions to the principle of *process separation* are threads, i.e. separate threads of execution that live within the same process context and thus share the memory range, and dynamic library injections, where shared objects ("DLLs" in Microsoft Windows) are injected into a process context through debugging interfaces.

Now that a process is allowed to assume that the whole memory is exclusively reserved, the operating system is required to employ a mechanism to translate virtual memory addresses to actual physical addresses. Information on actual translations and memory mappings are stored into the respective process context block that resides in the scheduler. Different

operating systems implement virtual memory handling in different ways but the abstract concept stays the same. Figure 2.2 shows how two processes may share the system's main memory.

A central part of virtual memory translation is the per-process *page table*. Although memory is generally addressable with byte granularity, memory controllers and operating systems typically fetch larger regions at once for better efficiency and caching behavior. In many Linux-based systems, the *page size* is set to 4 096 bytes, or 2^{12} bytes. If a process issues a read request of a single byte, a larger section surrounding that byte is actually read and made ready to subsequent read requests without additional memory controller utilization. Translation tables therefore do not need to store single addresses but may hold whole pages of memory.

Accordingly, access to physical memory might be implemented as follows:

1. An application requests a block of memory from the operating system but does not actually access it yet. It is allocated a block defined by the virtual addresses `4096` to `8191` (4 096 bytes).

2. Address `5000` is written to.

3. Now that the memory is actually accessed, the processor shifts the address `5000` to the right by 12 bits as the system's page size is 4 096 bytes. This yields `1` as the *page table index* which is then used to retrieve the corresponding physical address.

4. The page table entry for index `1` is still unpopulated, i.e. no corresponding physical memory location has been assigned yet. This is now done.

5. The page table look-up yields a physical memory address with respect to the whole page. Hence, the offset into the page ($5000 - 4096$) is added to the physical address.

6. The newly calculated physical memory address is now issued to the memory controller.

In this example, it is assumed that the CPU contains a memory management unit (MMU) that the current process' page table is loaded into upon

the scheduler's context switch. Otherwise, the translation would have to be done in software, incurring a large computational overhead. It also shows other relevant properties of virtual memory. Even if an application requests a large amount of memory, there does not need to be a corresponding physical location until it is actually accessed. Hence, there might be large differences between the allocated *virtual memory size (VMS)* and the actual *resident set size (RSS)*. This allows for *over-commitment* among the processes in the system and can lead to out-of-memory situations even if the memory allocations themselves might have been successful.

Another use case of virtual memory not depicted in the above figure is *paging*. Operating systems keep lists about most frequently used pages, last recently used pages and a few more. These can be used to automatically optimize memory access behavior by re-ordering physical pages to match virtual pages if a sequential access pattern has been detected. If pages have been found that have been allocated and used but some time has passed since then, they can be evicted from physical memory and instead stored to a secondary storage, such as hard disks. This technique called *paging* is used to reduce memory fragmentation and delaying out-of-memory conditions on overcommitted memory systems. Moving a page from memory to hard disk is called *paging out* or *swapping*, while moving a page back into memory is called *paging in*. A virtual memory location may therefore be in one of the following statuses:

- Not allocated. On access, the application is terminated for illegally accessing un-allocated memory.
- Allocated but not backed by actual storage (physical memory or secondary storage).
- Allocated and backed by physical memory. These pages are said to be *resident*.
- Allocated and backed by secondary storage. These pages are also resident but will have to be transparently paged in to be accessed.

Additionally, certain situations require memory to be allocated, resident and prevented from being moved or paged. These pages are called *page-locked* or *pinned* and, once locked, are bound to constant physical memory

locations until unlocked. This is often a central requirement in tightly coupled software/hardware constellations such as the applications that are presented in this work. Obviously, locked memory cannot benefit from memory reordering and overcommitment but will never suffer from the latencies generated by slow hard disk accesses during page-in and first-time physical memory allocations.

Energy Consumption

Energy consumption in CPUs is usually expressed through *power dissipation*. It is generally composed of *dynamic power consumption*, which originates from logic gate activity, *short-circuit* consumption during transistor switching and *transistor leakage* currents. The dynamic power and short-circuit power consumed is approximately proportional to the clock frequency, and to the square of the CPU operating voltage, while the leakage current is only dependent on the voltage [04].

An obvious solution to reduce power consumption is to lower the operating voltage. The main requirement for a certain voltage is the size of a transistor. Therefore, lowering the *structure size* also lowers the minimum voltage. As a result, several microprocessor vendors, such as Samsung Electronics and Intel, started commercial production of integrated circuits with transistor sizes of 10 nm with a target voltage of less than 0.5 Volts.

In data sheets of desktop processors, a measure often found is *Thermal Design Power (TDP)*. It expresses the maximum supported thermal discharge which, however, does not necessarily correspond to the maximum power consumption but gives a good indication. While the TDP is certainly a good specification when designing cooling systems and energy supplies, actual consumption rates in averaged work load conditions are rarely specified, possibly due to the lack of standardized measurement procedures. Table 2.2 gives an overview of selected processors showing the structure size, clock rate, core count and thermal design power from their respective data sheets. Evaluating the efficiency by these figures is almost impossible though, as different architectures may require a different number of clock cycles for a single instruction rendering common ratios such as “Watts per GHz” virtually useless.

Table 2.2. Power dissipation of selected processors

Model	Lithography	Clock Speed	TDP
Intel Pentium III 1400	130 nm	1.40 GHz	31 W
Intel Pentium M 780	90 nm	2.26 GHz	27 W
Intel Core 2 Quad Q6700	65 nm	4×2.66 GHz	95 W
Intel Core 2 Extreme X9100	45 nm	2×3.06 GHz	44 W
Intel Core i7-6950X	14 nm	10×3.00 GHz	140 W
Intel Atom D525	45 nm	2×1.83 GHz	13 W

Figure 2.3. Atari 2600 home entertainment system [Ken01]

2.1.2 Graphics Processing Units (GPUs)

Although the term *Graphics Processing Unit* was popularized when NVIDIA introduced its GeForce 256 game graphics accelerator card, the first dedicated video hardware dates back to the 1970s. Usually dated from 1978 to 1983, the "Golden Age of Arcade Video Games" took place where North America, Europe and Asia saw a rapid spread of video arcade games, both in video game arcade halls and home entertainment systems [Ken01], such as the popular Atari 2600 (Figure 2.3).

Although conceptually the same, a multitude of video hardware had emerged under various names, for example "Video Shifter", "Television Interface Adaptor" or "Video Display Controller." The basic idea has been to

move the burden of memory manipulation, such as drawing simple shapes and textures into the video memory, off the CPU to conserve resources. Over time, video hardware became more and more complex and began including special function units for *blitting*, i. e. the copying of regions from a texture to the screen, color space transformation, and even 3D polygons. Eventually, video controllers moved from simple display units to actual co-processors by exposing own primitive instruction sets.

In the 1990s decade, when home computers were on the rise and graphical work environments such as Microsoft Windows became popular, graphics devices were also used to enhance and accelerate the drawing of 2D surfaces in day-to-day work environments in addition to specialized computer-aided design (CAD) or gaming tasks. By this time, GPUs introduced so-called *transform & lighting (T&L)* support, the foundation for 3D hardware and later developments. The T&L units were able to extract a viewport from a vectored 3D scene and transformed it into a rasterized 2D image to be shown on screen. Operations, from simple object rotations to complex virtual camera calibrations and ray casting in computer vision, evolve around matrix multiplications in the affine 3D space. Thus, the idea for general purpose arithmetic co-processing arose to exploit the computational resources on graphics hardware for non-graphics purposes.

Eventually, frameworks were created to specifically facilitate the utilization of graphics cards for general purpose computation. This mode of computation is often named *general-purpose computing on graphics processing units (GPGPU)*. The most notable frameworks are the Khronos Group's OpenCL programming language, Microsoft's DirectCompute and NVIDIA's CUDA [DWL+12]. While OpenCL and DirectCompute may run programs on almost all recent graphics cards, NVIDIA's proprietary, closed-source CUDA system is restricted to NVIDIA hardware. One notable advantage is the missing abstraction layer. This way, specialized features can be exposed by the CUDA interface without requiring specific support from other vendors. In the evolution of OpenCL and its predecessors, this has constantly been a major hindrance and results in CUDA to be usually faster when compared on supported hardware [DWL+12]. As the general graphics hardware architectures of various vendors have mostly converged, the following sections describe the functions and units of NVIDIA devices as well as the

CUDA programming model, without loss of generality.

NVIDIA traditionally maintains three major product lines. Although the actual processing units are the same in all lines, the peripheral equipment, such as on-board memories, as well as the host system's drivers may differ significantly.

GeForce The GeForce series is built for the low-cost consumer market and is mainly targeted at gaming computer setups. When compared to other series, its products are optimized towards high-performance low-precision computing as needed for gaming.

Quadro NVIDIA Quadro boards are intended to be used for workstations running professional computer-aided design (CAD) and digital content creation (DCC) software. The drivers and API modules are designed for tight interaction with design software suites and the cards themselves typically contain large amounts of memory.

Tesla The Tesla product brand features devices that are specifically built for general purpose computing and target the high performance computing market. Their physical format specifications allow operation in data centers. Compared to the GeForce series, Tesla devices often feature a four-fold increase in double precision floating-point arithmetics performance and is therefore often used in scientific computing, for example in large scale weather simulations, machine leaning applications and bioinformatics. With the exception of the Tesla C-Series, these devices do not even provide display ports.

The CUDA Programming Model

The NVIDIA CUDA framework aims at opening the GPU's instruction sets to established programming languages such as C, C++ and Fortran and can be seen as an extension to their compilers. Since the introduction of 3D space transformations, GPUs have evolved into capable multi-processing systems.

In the NVIDIA Pascal microarchitecture, specifically the GP100 microprocessor used in the most recent Tesla series additions, 54 independent

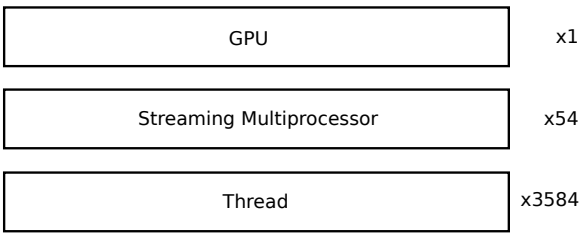

Figure 2.4. NVIDIA GP100 processing hierarchy

so-called streaming multiprocessors (SMs) are implemented. Each SM further contains 64 cores, totalling to a GPU-wide multiprocessing capability of 3 584 parallel threads of execution called *CUDA cores*. The architecture hierarchy with respect to computation units is shown in Figure 2.4.

The CUDA cores in a streaming multiprocessor are organized in *warps* of fixed sizes between 32 and 64 threads, depending on the device model. Every warp receives a single instruction stream, which leads to a lock-step SIMD-style of computation where each thread in the same warp has to execute the same instructions. Sometimes, this is not possible, especially in data-dependent branch instructions. Modern devices permit splitting up the instruction stream to allow executing a small number of branches to be executed in parallel but generally, the different branches are taken sequentially. Then, only the diverged threads are running while others not taking this branch are stalled until the streams are joined again. This can lead to performance impacts when larger parts of a warp are idling. Therefore, when programming with massive parallelism on CUDA-enabled devices, keeping the *warp divergence* on a low level is essential.

Despite executing the same instructions, cores usually operate on different data and therefore require access to thread-local storage. To overcome this issue, each core is allocated a slice of the SM-local register file. This comes with the downside that the number of processes running in parallel is not only limited by the availability of physical cores but also by the size of the register file. If the number of required registers exceeds the amount of available registers, cores are not scheduled instructions for until the workload can be executed. A streaming multiprocessor also keeps a

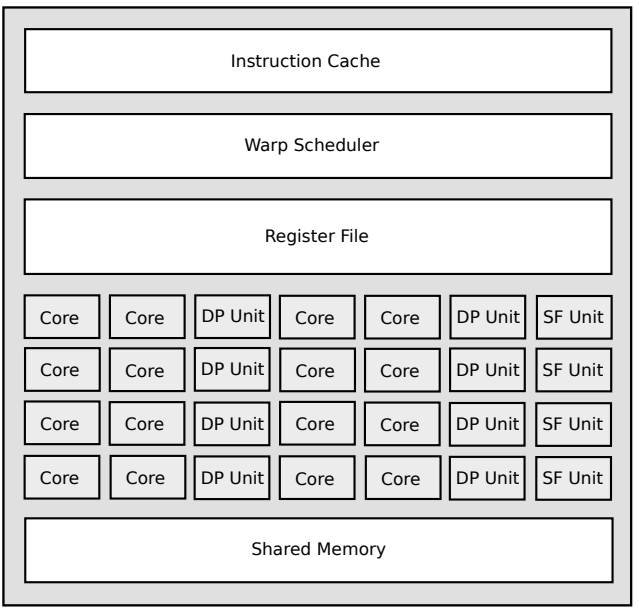

Figure 2.5. Streaming multiprocessor overview

certain amount of SM-local memory that is shared among threads and thus is called *shared memory*. Additionally, cores share special external units for double-precision floating-point arithmetics ("DP Units"), special function units ("SFUs") for transcendental instructions such as sine, cosine and others, and load/store units to access the GPU-global memory. Figure 2.5 gives a simplified overview of a single SM with only a few cores and peripherals.

A larger-scale overview is given in Figure 2.6 and shows how the streaming multiprocessors are embedded in the whole system. Before a program, the *kernel*, is launched to the GPU, a *grid* has be defined. It contains information about how the programmer intends to run the program among the streaming multiprocessors. In particular, it defines how many threads are to be executed by each SM (the *block size*) and how many of such a set of threads should be executed (the *grid size*). In recent developments, if resources allow, several blocks may be scheduled on the same SM concur-

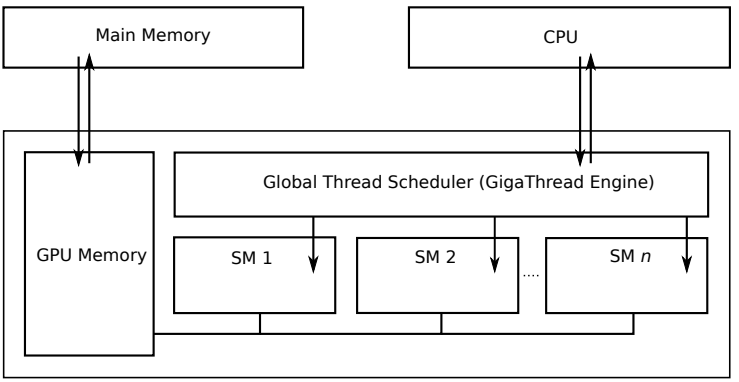

Figure 2.6. GPU overview

rently, much like in the CPU's simultaneous multi-threading engine (see Section 2.1.1).

The execution of a GPU program ("kernel") therefore works as follows:

1. A CUDA program is compiled on the CPU into device code. The programmer defines a *grid* to describe the ordering of threads for the global scheduler.

2. The kernel is launched to the GigaThread Engine, which schedules one or more blocks to the available and idle SMs. Often, more blocks are defined than SMs are available. In this case, they are scheduled on a first-in first-out (FIFO) style of execution and a streaming multiprocessor receives a new block as soon the old block is processed. Each block is accompanied by the same kernel.

3. Each SM then multiplies the instruction stream according to the number of threads to execute as defined by the grid and allocates register bank slices to the individual cores.

4. The SM-local warp schedulers dispatch instructions to the threads which are then executed in a SIMD-style.

Apart from registers and shared memories that are local to threads or streaming multiprocessors, threads may also access a global memory. Although many implementations provide one or more layers of data caching for global memory, only a very limited number of threads may access the memory concurrently. To prevent global memory accesses to be a bottleneck, several techniques have been developed. For example, the load/store units within a thread warp try to reduce the total number of memory transactions by coalescing requests. This is especially efficient if threads in a warp access consecutive addresses in the global memory and no single thread exceeds the memory transaction's maximum data width. In a similar approach, if memory addresses can be easily predicted, the programmer may copy a portion of global memory to the SM-local shared memory and hence reduces memory bus collisions when other SMs issue requests to memory.

Energy Consumption

Kasichayanula et al. show that while energy consumption on GPUs generally follows the same principles as in CPUs, current NVIDIA processors allow precise measurement and prediction of consumption rates with respect to the utilization of double precision units, memory accesses and general-purpose cores [KTL+12]. They further show that under optimal utilization, average power consumption is typically well within 20 % of the specified thermal design power while due to efficient power management, idle power consumption quickly drops to 15 % in total.

2.1.3 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays are, on the architecture side, hardly comparable with CPUs or GPUs as there is no central unit that executes programs based on a defined instruction set. A program for an FPGA is called *configuration* and does not contain sequential or parallel statements that are somehow executed in a certain order, but precisely describes how the circuitry shall connect external pins and internal devices. Hence, a more appropriate name for an FPGA program is *hardware description*. Being a much lower design level paradigm than high-level languages such as C++

or Java, whole processors may certainly be implemented through hardware descriptions that *do* execute regular compiled program code. In fact, this is one of the reasons FPGAs exist.

When highly integrated circuits are designed, transistors and their interconnections are edged into silicon wafers in a highly complex and expensive process. In higher level functions, circuitry tends to be recurring. For example, a 64 bit adder circuit might be built from 64 1 bit adders, each of them in turn build using two half-adders. A half-adder can be built out of an XOR and an AND gate and these consist of a number of transistors. The sum of a 64 bit adder could be stored in a register, which consists of a number of flip-flops and the pattern repeats down to the transistor again. Adders and registers are fairly trivial and their developers can often identify semantic errors by taking a close look. Larger systems, such as microprocessors built of millions of transistors or logic gates, become virtually impossible to debug by the naked eye. Although sophisticated circuit design tools may provide design checks and electrical check to rule out wiring errors, semantic and logic errors may only be found by full-coverage simulations.

Instead of building circuits from part libraries, they might instead be described by a formal language, such as Verilog or VHDL, and then simulated by a suitable tool. The simulation scripts might be annotated with input values and expected output values to automatically detect errors. Waveform diagrams can be used to evaluate timing requirements on wires. When designs become complex enough, providing full code coverage through simulations also becomes increasingly difficult. Some errors even show up for the first time in the finished products but edged silicon cannot be changed anymore. Indeed, early microprocessors have a long history of faulty hardware where operating system designers and compiler builders had to deploy workarounds, such as the infamous “Pentium FDIV Bug,” where an error in the floating-point unit of the Intel Pentium CPU series yielded wrong or inaccurate results [Vet15]. However, in-the-wild tests with real hardware may easily become very expensive due to the complex manufacturing process. At this point, reconfigurable logic emerged to address the issue. There are FPGAs, for example, that can be reconfigured any time and (virtually) arbitrarily often. How this intricate paradigm can be

Table 2.3. One bit full adder truth table

x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

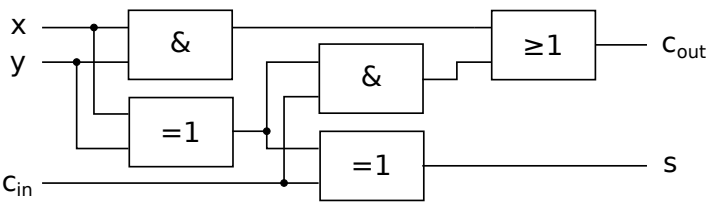

Figure 2.7. Full adder in elementary logic

implemented is shown in the following.

Reconfigurable Logic

When designing formulas in Boolean algebra, the first step is to express the expected results as a function of the input values through a truth table, that can easily implemented as a *look-up table* (LUT), as shown in Table 2.3.

A full adder consists of three inputs, with one bit for each summand and one bit for an incoming carry value that might come from a previous adder or a negative number. The result is a one bit sum and an outgoing carry bit. Further analysis and synthesis yields a elementary-level schematic as shown in Figure 2.7.

Instead of actually implementing a full adder though gate-level logic, the truth table can also be directly stored as-is using memory primitives.

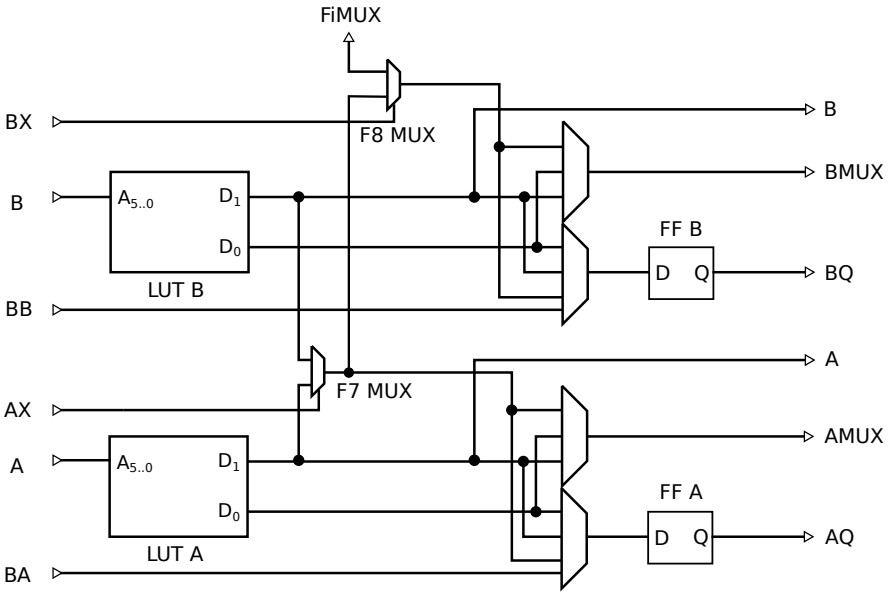

Figure 2.8. Simplified Configurable Logic Block with flip-flops and wide function generators

Here, an 8×2 bit memory may store the two bits of result while the three input wires may be used as address lines to that memory. Reading a value from the address `100` will yield the result `01`, which is exactly the binary value of $1 + 0 + 0$. Given enough memory, arbitrarily large functions can be expressed using storage as look-up tables. Being memory, its contents can also be replaced anytime, implementing completely new or corrected functions if necessary. This method is the fundamental approach used in FPGAs.

Configurable Logic Blocks

On Xilinx FPGAs such as the XC6V690T used in this work, look-up tables are organized in so-called *configurable logic blocks (CLBs)*. A coarse overview

is shown in Figure 2.8. A CLB consists of two 64×2 bit look-up memories tagged as LUT A and B on the left hand side of the schematic. Their output can be selected by asserting the 6 bit address line inputs A and B. The additional AX input for the lower side controls a multiplexer that is part of the *wide function generator*. Inputs to the F7 MUX are the upper bits of both LUTs, so when A and B are set to the same values, AX can be used as an additional address line, merging both LUTs to a 128×2 look-up table. This scheme can be further followed by using the F8 MUX to merge the local LUTs with the LUTs in the next CLB, each adding an additional address line.

On the output side, the lines A and B directly forward the contents of the higher order bit of their respective look-up tables without intermediate logic. The AMUX and BMUX output can be configured through the upstream multiplexers that select the higher order bit, the lower order bit or the output of the wide function generator for output. AQ and BQ offer an additional flip-flop to break long signal paths and ease the meeting of timing requirements. This flip-flop can also be configured as a latch instead of a flip-flop. The last function shown are the BA and BB inputs that allow the flip-flops to be primed with a default value or used as simple registers for signals from outside this CLB. The most trivial function to implement using a CLB is its look-up table to be used as is, i.e. as a plain 64×2 bit memory called *distributed RAM*.

The CLB shown above, though, is simplified in that it only shows logic that is connected to the realization of arbitrarily wide functions by using look-up tables. The actual CLBs contain much more logic that, however, is not required to understand the principles of FPGAs. By using additional resources not shown, LUTs may be connected to build regular shift registers and barrel shifters. Also, additional logic is included to allow efficient development of adders and multipliers.

On-board Peripherals and Integrated Components

Not all functionality can be realized by using plain look-up tables. Some components may not be expressed by simple zero-latency idealized Boolean algebra, such as clock management. Although it could indeed be possible to

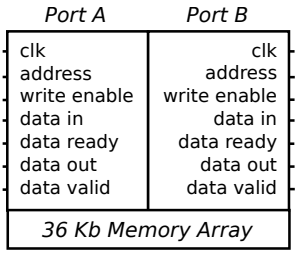

Figure 2.9. Block RAM device

design a configuration without using clock signal generators at all, hardly any communication protocol can be used without synchronization to some low-jitter recurring signal. Besides CLBs, FPGAs contain a number of digital clock managers that can be configured to clean clock signals from an outside source and generate scaled and phase-shifted frequencies by setting phase angles, clock multipliers and dividers. These clock signals are then fed into specialized clock buffers that serve all on-chip components as clock sources.

Generating larger amounts of memory by using the CLB's distributed RAM is possible. However, being such a common use case, dedicated memory blocks have been introduced to lift this burden off the logic resources, the *Block RAM*. A simplified Block RAM cell is shown in Figure 2.9. Each primitive exposes two individual ports to access the underlying 36 kilobit memory array, two accessors do not even have to agree on a common clock signal as the Block RAM interface contains synchronizer logic. It can therefore be also used as two separate 18 kilobit memories by masking the respective bits on the address lines. Additionally, masking out the `write enable` lines turns the RAM block into a ROM block which also allows further optimization by synthesis tools. Block RAM also features arbitrary coalescing into larger memories while keeping a single dual-port interface [UG473].

Another important component is the digital signal processor (DSP) tile. Integer arithmetics also being of frequent usage, Xilinx introduced a number of dedicated units that implement typical operations that are also found in regular CPUs' and GPUs' arithmetic-logical units (ALUs), such as integer

additions, subtractions and bit-wise logical operations (AND, OR, XOR, NOT, etc.). Xilinx DSPs, however, extend this set [UG479]:

- 25×18 bit two's complement multipliers
- 48-bit accumulator with internal accumulator registers
- pre-adders for efficient finite impulse response (FIR) filtering applications
- SIMD operations for dual half-width (24 bit) or quad quarter-width (12 bit) operand addition, subtraction and accumulation

To ease integration, many FPGA boards feature additional off-chip hardware. These often include PCI Express connectors and the respective high frequency transceivers, memory controllers for DRAM, NAND flash modules for external configuration storage, any many others. In modern FPGAs, such as the recent Xilinx UltraScale series, some of these are already integrated into the FPGA package, such as transceivers for PCI Express usage, or DRAM memory controllers [DS890].

Energy Consumption

By reducing the structure size to the ranges of 28 nm and operating voltages to 1.0–0.85 V, FPGA vendors successfully keep their thermal design power typically below 30 W while allowing theoretical peak clock frequencies of 750 MHz. The actual power consumption heavily depends on the program configuration and therefore on the switching activity. Hence, a high utilization and often-changing look-up tables lead to high power consumption. Some peripherals, such as high-frequency transceivers for off-chip communication require higher currents, as well as clock distribution networks and DSP slices [15]. Fortunately, modern design and synthesis tools such as the Xilinx Power Estimator can evaluate the energy consumption and heat dissipation based on the resource utilization within an error margin of less than 100 mW.

2.1.4 Comparison and Suitability Assessment

This work focuses on the partitioning of computational challenges into sub-components suitable for the introduced platforms and especially their interaction. In this section therefore the platform suitability for certain tasks is assessed.

CPUs contain a low number of independent execution units. They are clocked at comparatively high frequencies, have very potent branching and branch prediction units and are capable of executing higher arithmetics, transcendental functions and SIMD operations on register sizes up to 512 bit. This makes CPUs ideal for heavily control-flow oriented applications and iterative processes with data-dependent loop conditions.

GPUs on the other hand contain a rather large number of cores clocked with medium frequencies of approx. 1 GHz. Capabilities of these cores are limited to rather simple ALU operations and low-precision floating-point representations. Co-processing units for cores are available in low numbers, typically in ratios between 1:4 to 1:16, such as special function units (SFUs) for transcendental functions or raised-precision floating-point operation units (DB units). High memory access speeds, sophisticated caching hierarchies and efficient hardware-implemented thread scheduling make GPUs very efficient in highly data-parallel applications without divergence in branching decisions or data dependencies, and simple arithmetics.

The nature of FPGAs not having a fixed execution pipeline or even sequential execution units makes comparing these to more conservative processors difficult. However, based on the available resources and the internal representation of functions through configured CLBs allows to assume that very fine-grained parallelism can be implemented. In fact, parallelism can be implemented on a multitude of application layers, from the lowest of gate-levels to complex processors without any external scheduling supervisor units and the associated overhead of an active scheduler. Typically, FPGAs can use several clock networks where each buffer might operate at frequencies of up to 750 MHz [DS182]. Despite the comparatively low frequencies, with the help of DSPs, Block RAM and the implementation of intermediate clock domains, deep pipelines and systolic fields can be constructed easily. Data word sizes can be specifically tailored towards the

Table 2.4. Platform suitability overview

Task	CPU	GPU	FPGA
Operate on data with many data inter-dependencies or data-controlled loop conditions	very good	bad	bad
Perform simple arithmetics on standard data types	good	very good	very good
Working with custom data types	bad	bad	very good
Simple floating-point operations	good	very good	good
Higher arithmetics	good	medium	bad
SIMD programming style	bad	very good	very good
Pipeline operation	bad	bad	very good
Operation on data streams	very bad	good	very good
Memory access	medium	very good	good
Parallelism potential	low	high	very high

application's requirements as no native register sizes restrict the uses. With careful design, several concurrently operating pipelines can generate large amounts of data as long as no data dependencies arise and the pipeline output can be moved off-chip on time.

These findings are summarized in Table 2.4.

2.2 Data Transfer and Communication

This section gives an overview over the communication technologies used in the implementation of the prototype system. Short historical and motivational statements are given while the technical details required for a thorough understanding of the decisions relating to communication between the involved devices.

2.2.1 Direct Memory Access

In high-throughput hardware and software applications such as the proposed one, managing the data flow and processing protocol handshakes becomes a major burden that even modern processors struggle to handle. This section introduces state-of-the-art technology to move large amounts of data from one subsystem to using dedicated hardware instead of general-purpose processors.

Until the introduction of DMA, around 1981, the only way of moving data between peripherals was Programmed I/O (PIO). In this transmission mode, the CPU is instructed to write to or read from a device and waiting for its completion. Typical peer devices in regular computers were PS/2 mice and keyboards, RS232 serial interfaces, sound cards and IDE/ATA hard disk drives. Moving the head assembly on the actuator arm inside a hard disk to a different track (seeking) easily takes several milliseconds, making the PIO instruction block the CPU in the ranges of millions of clock cycles where no other instruction can be executed. Therefore, no transmission to or from other devices may take place and the current transmission's throughput is directly affected by the CPU's instruction processing time. Even in modern systems with multiple CPU cores and frequencies of several gigahertz, this can become a problem. High-end server-grade network cards often require bandwidths of 10 GBit per second and more. 2 GHz processors would certainly be put under heavy load. Even graphics cards on modern consumer-grade gaming workstations expect the system to deliver 10 GB/s and more.

In 1981, the IBM PC was launched containing an Intel 8088 CPU, one of the first consumer computer systems making use of Direct Memory Access (DMA). This technology allowed a Microprocessor to offload the transmission handling to a separate DMA controller, in these early systems, the Intel 8237. Now, the CPU set up the DMA controller with an offset and a length via PIO and started the transaction. While the DMA transaction was executing, the CPU could continue processing until the DMA controller notified the CPU that the work has been done. Indeed, the 8237 even supported four almost independent channels with transfer throughputs of 1.6 MiB/s per channel, a respectable speed for the early 1980s [Int93].

Since then, the use of DMA technology has evolved heavily. Today, most of the more data-intensive peripherals are connected via high-performance and DMA-capable links, such as SCSI, Serial ATA or PCI Express. Several new transport functions and modes have been added, such as scatter/gather operations, which plays a major role in the transport mechanisms used in our hybrid system.

Scatter-Gather Operation

Modern operating systems organize physical memory in pages. When a process writes to virtual memory, the memory block is separated to (usually) fixed-sized memory *pages* whose virtual addresses are translated to physical addresses. Due to various reasons, such as fragmentation, pages from a virtual memory block are not necessarily contiguous in physical memory. The translation between virtual addresses and physical addresses is usually done with the help of page tables as shown in Fig. 2.2.

DMA controllers though, programmed with only address, offset and length, typically expect a contiguous memory region. Naturally, the maximum buffer size for a DMA transfer is therefore capped by the ability of the operating system to provide such a region. Tests with a custom DMA driver in recent Linux versions have shown that it is hardly possible to acquire contiguous memory blocks larger than 4 MB, regardless of the available physical memory. This has several reasons. Firstly and most importantly, a user-space application does have very little control over the virtual memory allocation modes. In particular, it cannot request memory that is contiguous as the Linux system calls responsible for memory management (`brk`, `sbrk` and `mmap`) [TOG13] purposely do not provide means for that. Therefore, the transmission buffer has to be allocated in the memory space reserved for the kernel, where memory can be allocated directly from the virtual memory subsystem instead of using system calls. Secondly, the Linux kernel's view of memory is separated into *zones*. One of the zones, `ZONE_DMA`, is the memory region generally suitable for DMA. Unfortunately, many devices and buses in a computer do not support full-width addresses. ISA buses, for example, only allow DMA communication with 24 bit addresses [Int93], but the DMA zone claims that memory taken from it *generally works for*

DMA. Therefore, the DMA zone on a x86 sytem is 2^{24} bytes, or 16 MiB in size. However, devices on the PCI bus (or one of its direct successors, such as PCI Express), do not have this limit and are able to work with memory located throughout the (physical) address space, the memory still has to be contiguous, though [LDD3, pp. 214–216].

Generally, when contiguous memory is required, the memory region has to reside in the kernel memory space. As user applications do not have access to kernel memory for security reasons, processes give a userspace-allocated buffer to the kernel which then has to copy the data forth and back. This conceptually unnecessary copy operation reduces the whole DMA operation to absurdity as the CPU has to copy one word at a time from one buffer to the other. Although the intra-memory copying is certainly faster than moving data between devices, the CPU is utilized what DMA was actually designed to reduce.

The solution is DMA operation in scatter/gather mode. This DMA function requires more complex setup in the kernel driver as well as in the DMA controller. In this mode, the userspace process may allocate an arbitrary block of memory by the usual means. The userspace buffer address is then handed over to the responsible driver. Contrary to the previously described DMA approach with kernel buffers (*Streaming DMA*), the driver translates the userspace buffer into a list of pages. Then, entries in the page list that correspond to physical memory regions next to each other, are coalesced into single entries, yielding a list of contiguous memory regions called the *scatterlist*. Modern DMA controllers do also allow programming by whole scatterlists instead of single (address, length, offset)-tuples. For that purpose, the memory occupied by the scatterlist itself has to obey the same restrictions as if it would be transferred with streaming DMA. Therefore, in the extreme case, the scatterlist has to fit into a single memory page. In current Linux systems, a (regular) page is 4,096 bytes. Assuming that addresses are 64 bit, length and offset 32 bit, a scatterlist entry is 16 bytes. Hence, a full hypothetical scatterlist with 256 entries may address only 1 MiB of memory when no coalescing is possible (i.e. due to busy or fragmented memory), so for every megabyte of data, a new DMA transaction has to be started.

Although this is already a huge improvement over the streaming DMA mode, there is still room to further reduce the CPU utilization. One obvious

solution is to use larger pages. Modern CPUs and operating systems already support larger tables (“HugeTLBs” in Linux, “Large Pages” in Windows, “Super Pages” in BSD), but at least in Linux, there is no standardized way to make use if this feature. Another solution is scatterlist chaining. In this method, the last entry of a page-fit scatterlist is marked as “more are following”. When the DMA controller has started processing the scatterlist entries one-by-one, it will eventually reach the last one. The flagged entry tells the controller that there are more entries, despite of the header word which stated the list length, and the next word in the list is a header of a new list. This way, several scatterlists can be chained together without interrupting the transaction. Some DMA controllers organize the scatterlist buffer in a ring buffer. Here, a new scatterlist can be loaded into the controller buffer as soon as the controller has started the second list, ultimately allowing to transmit arbitrary userspace buffers to and from peripheral devices with minimal computing overhead.

Latest developments show that the scale of performance in DMA-enabled communication links is not only required within a computing system, but also in between computing systems. Several implementations of a technology called Remote DMA (RDMA) have evolved to move data between different computer system’s main memories without involving their operating systems and/or CPUs. This could be done, for example, by enabling both system’s interface cards to perform DMA transactions with their local main memory and connecting both cards with a direct link so that no routing would be required. Common implementations of RDMA include InfiniBand, iWARP and RoCE (RDMA over Converged Ethernet).

2.2.2 PCI Express

PCI Express (PCIe) is a standardized communication system in modern computer architectures that is used to interconnect peripherals, such as hard disk drives, network interface cards, and graphics accelerator cards. It is held being the successor to ISA, PCI and AGP, although they do not have much in common. At least the differences between AGP, PCI and PCI Express are usually abstracted in operating systems in a way that generic drivers, i.e. for sound cards, generally do not need to know the actual

communication architecture.

Although often called a "bus", PCIe is actually a system of serial point-to-point connections, where every device is directly connected to a switching fabric, the PCI Express switch, at this level functioning much like its Ethernet equivalent. On the physical layer, also called "PCIe PHY", PCI Express devices are connected through an edge connector. While a standard PCI interface contains 124 connections, 56 of them solely for power supply and signal ground, a PCI Express connector requires only 36 connections, including 18 power supply and ground connections. This is mostly because PCI Express does not need bus arbitration and handshake pins (as it is not a bus), and data is transmitted serially instead of in parallel.

Line Code and Data Rates

On the physical layer, data communication is implemented by using *lanes*, full-duplex pairs of LVDS (low voltage differential signaling) wires, accompanied by the same number of ground connections, totaling to eight additional wires per lane. All lanes together form a full-duplex logical link. For Generation 1, the data rates are specified as 2.5 GT/s, which is an informal description of *the number of transfers per second* and incompatible with the Système international d'unités (SI) due to its symbol name clash (where T is reserved for the Tesla unit) [TT08]. A more technically sound measure would be the symbol rate as it is regularly used in information theory [Bel53]. It is defined as the modulation rate of a signal, i.e. the number of distinct symbol changes per second. Its SI-conforming unit is the baud (Bd). Among many other communication protocols, PCI Express Gen 1 uses the 8b10b line code, where 8 data bits are encoded in 10 line bits. These line bits are generally called *transfers*, as in *2.5 Gigatransfers per second*, and can therefore be written more precisely as 2.5 GBd/s. Now it is clear, that a lane in PCI Express Gen 1 cannot transfer 2.5 Gb/s net payload data but 2.0 Gb/s due to the 8-in-10-bit encoding scheme. 20 % data rate overhead being relatively high, the line code for PCI Express Generations 3 and 4 has been changed to 128b/130b reducing the line-code overhead to approx. 1.5 %.

The data rates shown in Table 2.5 include line-code overhead but do not

Table 2.5. Data rates per lane by PCI Express generation

Generation	Line Code	Line Rate	Net Speed
1.0/1.1	8b/10b	2.5 GBd/s	238 MiB/s
2.0/2.1	8b/10b	5.0 GBd/s	477 MiB/s
3.0/3.1	128b/130b	8.0 GBd/s	939 MiB/s
4.0	128b/130b	16.0 GBd/s	1.869 MiB/s

take PCI Express protocol overhead into account. The actual application-layer net throughput heavily depends on the application profile. Regarding net data rates after protocol overhead, [BK08] states:

> Like other high data rate serial interconnect systems, PCIe has a protocol and processing overhead due to the additional transfer robustness (CRC and acknowledgements). Long continuous unidirectional transfers (such as those typical in high-performance storage controllers) can approach >95 % of PCIe's raw (lane) data rate. These transfers also benefit the most from increased number of lanes (×2, ×4, etc.) But in more typical applications (such as a USB or Ethernet controller), the traffic profile is characterized as short data packets with frequent enforced acknowledgements. This type of traffic reduces the efficiency of the link, due to overhead from packet parsing and forced interrupts (either in the device's host interface or the PC's CPU). Being a protocol for devices connected to the same printed circuit board, it does not require the same tolerance for transmission errors as a protocol for communication over longer distances, and thus, this loss of efficiency is not particular to PCIe.

DMA Usage

As stated earlier in Sect. 2.2.1, PCI Express devices support the use of DMA. Unlike older buses such as ISA, PCI and PCIe do not have a central DMA controller. Instead, any device may become "bus master" to directly initiate the transactions, effectively *becoming* a DMA controller for the

duration of the transaction. Especially in buses or architectures where several devices may become master at the same time, arbitration algorithms have to be employed. Several Quality of Service (QoS) parameters, such as traffic classes and virtual channels are required for standards-conformant operation [BAS03]. This way, connections with real-time priorities, such as audio streams that are naturally transmitted via synchronous means, may be embedded in PCI Express' isochronous transactions.

Link Negotiation

PCI Express link management protocols are designed for highest compatibility in a way that most devices are fully backwards compatible. Therefore, many features that come with updated revisions are optional and negotiable. Newer cards may not only negotiate software peculiarities such as power management features, but even link widths and low-level encodings. For example, consumer motherboards often provide PCI Express female edge connectors with a *physical* length of 16-lane slots but their *electrical* wiring only supports four lanes. On the other hand, (physical) four-lane cards may be inserted in any slot where it *physically* fits, i.e. fully wired 16-lane slots. To help coping with these lane width compatibility features, several loop-back pins exist on these cards: after the first lane pair block, the fourth, eighth and sixteenth lane pair block, two otherwise unused pins are short-circuited. Hence, the PCI Express controller can easily set up initial lane allocations without using the not-yet established communication links.

Unfortunately, many chip sets and processors do not provide enough lanes to satisfy a full allocation request, i.e. the motherboard provides more lanes on its sockets than the controllers support and all sockets contain devices with maximum lane count. In these situations, the computer's system software (BIOS, UEFI, etc.) and operating system may re-negotiate the lane allocations. Some systems even allow the end user to configure allocation limits for certain slots. During the initial negotiation and further re-negotiations, not only the lane allocations but also protocols and features can be changed and agreed or declined upon. Arguably, the most important feature for initial connections might be the PCI Express generation. Generally, the generation used defines the line code and transfer rates. The

current PCI Express 4.0 standard defines the generations listed in Table 2.5. Other features include advanced power management and error reporting and recovery mechanisms.

2.3 Parallel Computing

The implementations presented in this work are heavily built upon parallel programming building blocks. To truly penetrate this matter, it is essential to develop a thorough mental model of the parallelization aspects in this architecture. This section aims at laying the foundation and introducing these building blocks.

2.3.1 Parallelism and Machine Models

Parallelism and parallelization may occur at different abstraction levels throughout an application. Many of these mechanisms happen automatically or implicitly, in a way that does not need any interaction with or even knowledge by users, programmers or software architects. Other methods have to be explicitly enabled and carefully used. The most important aspects of automatic parallelization and vectorization have been discussed in Sect. 2.1.1, such as pipeline superscalarity. Assisted parallelization and concurrency can be easily used through pre-built software libraries, such as OpenMP and MPI. More abstract and higher-level techniques can only be employed with deep knowledge of a specific problem, such as the problem partitioning used in the proposed work. In some cases, programmers and hardware designers even create new parallelization techniques and functions from scratch to be used for a certain class of applications. The applications and architectures developed here are carefully designed to include almost every layer of parallelism, from single instructions to a rough hardware component schematic and are required to be understood to properly evaluate the nature of the hybrid-parallel prototype architecture.

Flynn's Taxonomy

When parallelization and parallel operations became popular in computer science, Michael J. Flynn proposed a classification scheme for parallel execution models later known as "Flynn's Taxonomy" [Fly72].

The original classes are defined as follows.

Single Instruction stream Single Data stream (SISD) describes the simplest execution mode. Each scheduled instruction is executed exactly one after another and processes exactly one datum before the next instruction is scheduled. This is a typical mode for consumer-line CPUs before the streaming instruction set extensions were introduced.

Single Instruction stream Multiple Data streams (SIMD) is a mode where a single instruction operates on multiple data streams as supported in modern processor since the Multimedia Extensions (MMX). Current instruction set extensions, such as Advanced Vector Extensions 512-bit (AVX-512) and Fused Multiply/Add (FMA), usually focus on SIMD instructions.

Multiple Instruction stream Single Data stream (MISD) is a rather exotic mode with low coverage in regular CPUs. Applications requiring high fault tolerance may schedule a number of instructions on the same data set where the different execution engines then have to agree on the correct result. Another example is a systolic array where a single data streams is processed by pipelined execution nodes. To some degree, the classification of systolic arrays in Flynn's Taxonomy is contested. This will be discussed in detail in Sect. 2.3.2.

Multiple Instruction stream Multiple Data stream (MIMD) became more popular with multiprogramming processors and superscalar systems. Operating systems schedule multiple instructions to be executed at the same time on different nodes, each one associated with its own data set. MIMD systems range from simple multi-core CPUs to distributed platforms. As opposed to SIMD, instructions need not to be executed in lock-step but fully asynchronously.

When parallel processing and parallel architectures became a more popular field of research, some authors felt the need to extend or further subdivide Flynn's taxonomy to account for modern developments. An interesting example that is hard to classify with the mentioned terms is the Cell processor found in Sony's home entertainment system "PlayStation 3". It is being developed by Sony, Toshiba and IBM since 2001 and features a number of general-purpose processors, each one accompanied by eight service processors that are specialized towards data-intensive processing and floating-point calculations [GHF+06]. Service processors and main processors operate in a master/slave environment and run different programs and the same time but asynchronously. Althrough this seems to qualify for a MIMD classification, the slave processors' work units are directly scheduled by their master processor, not by some higher-level operating system, as opposed to the master processor itself. This kind of work-sharing construct would not be properly reflected without a subdivision of MIMD that is called *Multiple Programs, Multiple Data Streams (MPMD)* [GFB+04].

A more commonly used subcategory proposal for MIMD is *Single Program, Multiple Data Streams (SPMD)* [DGN+88; Dar01]. Analogous to MPMD, multiple instructions are executed in parallel with multiple data streams but within a single program. This is a typical class for all kinds multithreaded programs and therefore easily the most common style of parallel programming.

2.3.2 Systolic Arrays

Most processors such as CPUs and GPUs are either based on Von Neumann or Harvard sequential architectures and as such depend on complex centrally orchestrated memory accesses, instruction execution pipelines, program counters, register files and others. Although vendors have recently added certain small-scale parallelism functionality, the underlying architecture is still tied to sequential processes. As a large part of the problem is implemented using a systolic array-based solution, a short introduction to systolic structures using the simple example of polynomial solving is given.

A systolic array is a tightly coupled network of identical, primitive nodes. These nodes can be interconnected arbitrarily, but most often, linear

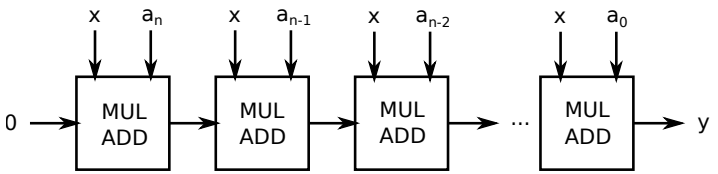

Figure 2.10. Polynomial evaluation using systolic arrays

arrays, two-dimensional meshes, hypercubes, and other popular network topologies are used. No central unit coordinates data or control flows and the overall operation is similar to pipelining.

For example, polynomials can be computed efficiently using linear systolic arrays. A polynomial in its base form

$$y = a_0 + a_1 x + a_2 x^2 + \ldots + a_k x^k$$

can be transformed into a suitable expression using Horner's Rule:

$$y = ((((a_n x + a_{n-1}) \times x + a_{n-2}) \times x + a_{n-3}) \times x \ldots a_1) \times x + a_0$$

The factorization now consists of a recursive sequence of multiply-and-add operations and can be implemented using a linear systolic array with inputs from two directions. The result is shown in Figure 2.10 using fused multiply/add units. The first node multiplies its input (0) by x and adds a_n, and forwards the result to the next node, which multiplies the value again by x and adds a_{n-1}. The first two nodes therefore calculate the subterm $a_n x + a_{n-1}$. Every successive node adds another pair. Once the first output has been generated after n cycles (with n being the array length), one polynomial can be evaluated every cycle with a latency of n cycles. As seen from a CPU-like standpoint, the network processes n `MUL/ADD` instructions per cycle.

As processing nodes only perform primitive operations, they also con-

sume much less resources in terms of logic gates and transistors than fully pipelined CPUs. Furthermore, due to their simplicity, they can also be cascaded into large networks and with rising numbers of nodes, easily outperform even the most sophisticated vector extensions (i.e. AVX-512).

While systolic arrays are very efficient by exploiting low level parallelism, they can only be used with problems that can be expressed in a suitable and simple form. Additionally, common architectures such as CPUs and GPUs are not flexible enough for systolic chain implementations as inter-core communication can easily become a bottleneck. Naturally, FPGAs do provide the required flexibility and are a regular target for systolic array-based signal processing algorithms (convolution, Fourier transform, lattice filters), higher dimensional arithmetics (matrix multiplication, inversion, decomposition) and artificial neural networks [ZP02].

Chapter 3

Applications

3.1 A Primer on Bioinformatics

With the hybrid-parallel architecture focusing on applications in bioinformatics, the following sections provide an overview on the biological and genetic backgrounds. This knowledge is required to fully understand how and why these applications and algorithms have been selected as well as the importance of the results and consequences.

3.1.1 DNA and Chromosomes

All known lifeforms on earth, from complex mammals such as humans to the simplest viruses and mycoplasma, store their hereditary information in special macromolecules called deoxyribonucleic acid (DNA) [AJL+15, Chap. 1]. DNA consists of a potentially long array of nucleotide pairs. Nucleotides contain desoxyribose (a sugar) as a fixture and one of four nucleobases, adenine, thymine, guanine or cytosine as the actual genetic code, usually abbreviated by their first letter. These sugar fixtures are fused together with phosphates, creating a helix structure. The DNA though, contains a second strand, anti-parallel to the first. By their molecular structure, adenine may only oppose thymine while guanine may only oppose cytosine. Both nucleobases are tied together by hydrogen bridges, forming what is called a *base pair*. Besides several biological implications, the double helix is more fault-tolerant than a single strand because of its redundant nature. A schematic overview of the DNA topology is shown in Fig. 3.1.

From an information scientist's point of view, the DNA may act as a universal linear data storage. The information itself is encoded with an

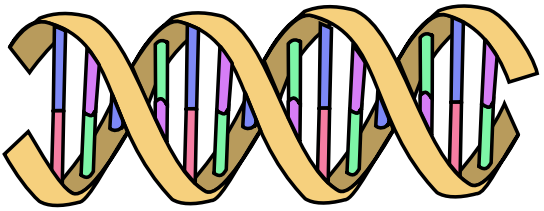

Figure 3.1. A DNA schematic with nucleotides and sugar/phosphate fixture

alphabet of four different symbols where every symbol is stored along with its inverse, introducing a certain level of redundancy.

Eukaryotes, organisms with cell membranes and distinctive cell organelles, contain their genetic material within a nucleus. During (regular) cell division, the DNA is to be replicated to provide a copy for the newly created cell. This process is called *mitosis* for regular cells and *meiosis* for gametes. In the very first step during mitosis, the *interphase*, the DNA is duplicated. During the next phase, the *prophase*, the single long double-helix is tightly condensed and cast into chromosomes, stabilized by specialized proteins called *nucleosomes*. A human cell typically contains 46 chromosomes of varying lengths, divided into two gonosomes and 44 autosomes. As most mammals, human cells are diploid. They contain 22 autosomes and one gonosome from their direct maternal ancestor and the same set from their parental ancestor. Genetic information that will be used for purposes in bioinformatics is usually extracted during the interphase or the following metaphase, yielding chromosomes instead of a single double-helix, as shown in Fig. 3.2. In literature and biobanks, chromosomes are sorted and indexed by length, 1 being the longest chromosome, 22 the shortest. Due to their special roles, gonosomes are not indexed by number but X or Y, depending on their type.

The data stored in DNA serves a multitude of purposes. Many regions encode the physical structure of proteins. When a protein is to be synthesized, a certain region from the DNA is copied and processed into messenger ribonucleic acid (mRNA), a single nucleotide strand similar to these known from DNA. The mRNA encodes the protein structure by the

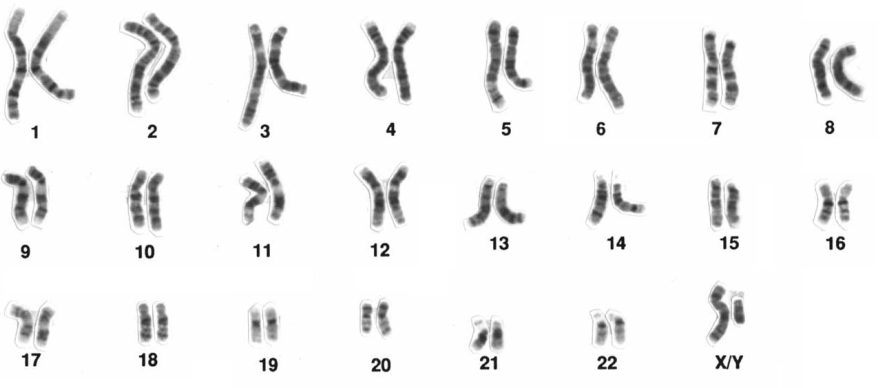

Figure 3.2. Set of 23 chromosomes from a human male

use of *codons*, triplets of nucleotides. With four possible nucleotides, 64 possible codons exist, where 61 are used to encode the 20 canonical amino acids proteins are created from. The remaining three codons mark the end of an encoding re Dieselbe Frage stellt sich auf Seite 25 und einigen weiteren Seiten.gion and terminate the translation process. The synthesis product is a chain of amino acids where the exact types and overall ordering determine the physical structure and therefore, function.

3.1.2 Genes

A *gene* is a region in the DNA with a certain function. In an evolutionary point of view, a gene is a molecular *unit of heredity* [AJL+15], where its transmission to offspring often means the transmission of phenotypic traits. Although only parts of a DNA strand are currently considered to be genes, the whole DNA including its associated mechanisms (ribonucleic acid, genomes of mitochondia, etc.) and its non-coding regions is called *genome* or *genetic material*. Currently, approx. 20 000 genes are estimated to be present in the human genome [PS10].

Mutations are alterations to the genetic material of an organism. They may result from a multitude of causes. Popular examples are sunburns

as a form of direct DNA damage, where UV rays with relatively high energy impact the thymine's ability to bind with adenosine, effectively breaking the base pair at this location [Goo01]. The DNA not being a hard crystalline formation but an organic mobile part, the strand may overlap itself. This may also happen with chromosomes overlapping each other or themselves. At these locations, one strand may just cut the other, permanently reducing the information. This process is known as *deletion* and can affect any number of nucleotide pairs [Lew04]. Another source of mutation is the DNA replication and synthesis mechanism itself, where a number of base pairs may be inserted into the strand, deleted or replaced as results of molecular decay or error-prone repair processes [AJL+15, Chap. 5].

3.1.3 Single Nucleotide Polymorphisms (SNPs)

For this work however, only single nucleotide polymorphisms (SNPs) are interesting. As the name suggests, a single nucleotide pair is replaced with another consistent pair. Prior research suggests that many traits and (genetic) diseases can be explained by the characteristics of a single SNP [TKF+11], such as sickle-cell anemia [Ing56]. SNPs can be found throughout the genome, within genes and also within non-coding regions. A SNP is defined by its base position in the DNA and its possible base pair variations are said to be *alleles* for this position.

As explained in Sect. 3.1.1, the human genome consists of a maternal and a paternal set of chromosomes. Therefore, every SNP at a specific location in the genome has two related base pairs, of which only the first pair part is usually mentioned as the second part is redundant. For example, if an individual has the guanine/cytosine pair at both the maternal and paternal SNP's location, it may be written as `GG`. If the variation *G* at the SNP's location is the predominant configuration (*allele*) in the individuals respective population, it generally called *wild type* as opposed to the *variant type*. Therefore, a SNP may have the following configurations:

- *Homozygote wild*, where both the maternal and paternal part contain the predominant allele in the individual's population.
- *Homozygote variant*, where both parts possess a variant type allele.

▷ *Heterozygote,* where one part has a variant allele and the other part a wild type allele.

3.1.4 Genome-wide Association Studies (GWAS)

Association studies are examinations of potentially large numbers of genetic variations in different individuals to find correlations between genetic variants and certain traits like diseases. In a typical set-up, individuals are grouped into *cases* where a certain trait is present and in *controls* where it is not. Results from genome-wide association studies (GWAS) may therefore identify SNPs that are *somehow associated* with and *influential* to a disease, but cannot specify which genes actually cause the trait to manifest [Man10].

One of the more popular GWA studies is extensively used in this work and provided by the Wellcome Trust Case Control Consortium (WTCCC). The study is designed around 2,000 sampled individuals from each of seven diseases, type 1 diabetes, type 2 diabetes, coronary heart disease, hypertension, bipolar disorder, rheumatoid arthritis and Crohn's disease. For the control group, 3,000 individuals have been sampled from the same population based in Great Britain [BCC+07].

3.1.5 Epistasis

In genetics, the term *epistasis* refers to the phenomenon where a gene is influenced by its *genetic background*. These influences may be dependencies on the presence or absence of one or more other genes called *modifier genes*. Other examples are influences between alleles in a single heterozygote or between a gene and the environment. Often, the combined expression cannot be explained solely by the sum of the effects of the single genes. Therefore, epistatic interaction in classified as follows:

▷ Additive Epistasis. Purely additive effects of the single genes. As most genes exhibit at least some level epistatic interaction, this type is relatively rare [Kau93].

▷ Magnitudal Epistasis. The combined effect is larger than the sums of the genes' primary effects. It is suggested that magnitudal epistasis be

further divided into positive and negative [Phi08; BCP+04] or synergistic and antagonistic effects [CC10].

- Sign Epistasis. A genetic mutation has the oppisite effect when in presence of another certain genetic mutation [WWC05].

It is therefore not only appropriate to search for single genes but also combination of genes.

3.1.6 Acquiring Data

When planning and organizing a genome-wide gene-gene association study, a large number of SNPs is analyzed, whose positions in the genome are already known. For example, in the WTCCC1 study, approx. 500 000 SNPs have been analyzed, within approx. three billion base pairs. This subset of the genome is called *genotype*. Before software tools are able to analyze SNPs and their alleles by investigating encodings of wild types and variant types they first have to be translated from a *biological* representation, i.e. the actual DNA, to a *digital* representation that computers can process.

One approach to digitize the SNPs' characteristics works by chemically marking the desired SNPs in the genetic material. Every marker individually identifies a certain SNP. Furthermore, the alleles in question are replaced with fluorescent equivalent base pairs. Then, the material flows over a special chip called *microarray*. This device contains an array of probes where each probe allows one of the previously installed markers to bind. After removing the leftovers, determining the allele configurations becomes a matter of taking a photograph of the chip to capture the fluorescent emissions. This photograph can then be easily processed into digital formats. A fragment of a DNA microarray shot is shown in Figure 3.3.

While the first commercially available microarrays only allowed very few SNPs for processing, current technologies typically support any number between hundreds, thousands, and up to 4.5 million SNPs on a single array in the Illumina Inc. Omni series [Ill16]. With arrays becoming larger over time and epistasis analysis targeting higher orders instead of singular SNP effects, analysis in bioinformatics is becoming a major challenge when it comes to available computational resources.

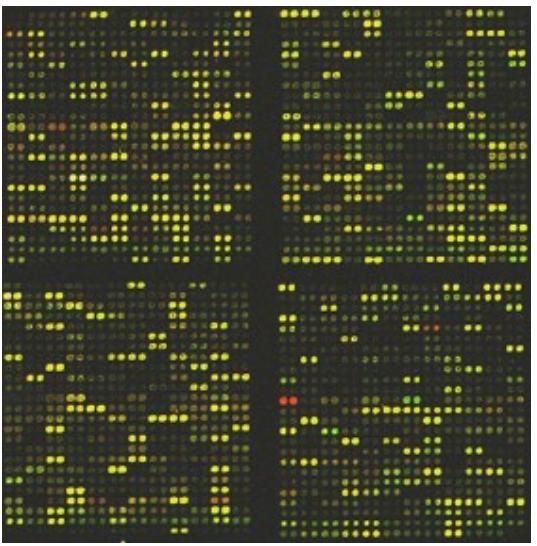

Figure 3.3. Fluorescent emissions from a microarray

3.2 Exhaustive Interaction Search

Data from genome-wide association studies are well suited for analyses of traits where the phenotypic outcome can be clearly distinguished into "affected" and "not affected". These traits are termed *binary traits*. Even though single-SNP effects are certainly accountable for a large number of traits, analyses of even millions of SNPs are trivial. However, research suggests that epistatic gene-gene interactions might play a major role in trait expression [Mah08; MAW10]. Single-SNP analysis methods may combine SNPs with strong effects but hence, the SNP combination may only express *additive epistasis*. This method of interaction analysis is obviously not powerful enough to model interactions between SNPs that do not have primary (single-SNP) effects.

Evaluating all actual combinations of two SNPs raises the computational by an order of magnitude – from linear in the number of SNPs to quadratic. The exact number of tests can be calculated with the help of the elemen-

tary combinatorics, specifically the binomial coefficient, by counting all 2-combinations without repetitions. Let n be the number of SNPs and $k = 2$ the combination order.

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

More formally, a $k-$combination of a set is a subset of k distinct elements of it. In this case, it yields all sets of SNP combinations of length 2 where the order does not matter, i.e. $\{i,j\} = \{j,i\}$, and a combination may not contain the same SNP two times, $\{i,i\}$ does not exist. In this work, the terms "2-sets" and "pairs" are interchangeably used although in the strictly mathematical sense, a pair is a 2-tuple and tuples are defined to be ordered sequences instead of unordered sets where the ordering of items is irrelevant.

When evaluating the already introduced WTCCC1 data set, more than 500 000 SNPs have to be analyzed [BCC+07]. Although this is already a fairly large study, even by today's standards, the evaluation time is certainly a matter of seconds, depending on the actual method used. With the power raised to second order interactions, 500 000 evaluations become $C(500000,2) \approx 125 \times 10^9$ evaluations. Whole data set analyses in these orders of magnitude are clearly a much more interesting problem when it comes to computational complexity and motivates the invention of specially designed hardware and software architectures.

Recent research also suggests even more complex traits, such as the expression of psioriasis, that seems to be following a larger network of genes and might even influence other serious diseases [ZSX+16; ALI+00; TE14]. These include inflammatory bowel diseases, hypertension, diabetes and others [KMS+07]. These findings show that the search for interactions of orders higher than two may also reveal interactions. As a proof of concept, this work additionally presents an implementation of third order epistatic interaction analysis. Considering the above data set, the number of evaluations grows to $C(500000,3) \approx 2.1 \times 10^{16}$, exponentially in the order of interaction.

Even if such an evaluation consumed as few as ten CPU clock cycles at the highest available frequencies, an exhaustive analysis would almost take just under two years of runtime. Hence, only a suitably sized subset of

		SNP A				SNP A		
	Cases	w	h	v	*Controls*	w	h	v
SNP B	w	n_{000}	n_{010}	n_{020}	w	n_{001}	n_{011}	n_{021}
	h	n_{100}	n_{110}	n_{120}	h	n_{101}	n_{111}	n_{121}
	v	n_{200}	n_{210}	n_{220}	v	n_{201}	n_{211}	n_{221}

Figure 3.4. An unnormalized contingency table

WTCCC1 can be analyzed using the methods presented here.

3.2.1 Contingency Tables

To model other forms of interaction as plain additive combinations, such as magnitudal or sign epistasis, the interactions themselves are to be investigated. A common tool in survey research is the contingency table. It is a type of matrix that generally displays the possibly multivariate frequency distribution of the involved variables and can be used to assess the relations between them. Figure 3.4 shows a suitable contingency table format for two-SNP interaction. Each half of the displayed table describes allele distribution of a SNP combination, with the case group on the left hand side and the control group on the right hand side. The "w/w" cell on the right side, for example, contains the number of control group individuals where the alleles of SNP A and SNP B are both of the *wild type*. Note that this type of table where the cells contain actual *counters* does not show a frequency distribution. When programmatically creating contingency tables, this is often an easier format but can trivially be processed into actual frequencies by normalizing the cell values to the respective group size.

This unnormalized type of display has a few mathematical implications that will be important in the implemented methods in later chapters. For example, the column sum of SNP A's wild types in the case group equals the total number of wild types for SNP A's cases and is constant for every two-set of SNPs where SNP A is part of. This observation holds true for any column and row in the table and will later be used to reduce the required contingency table storage by removing this redundancy.

In principle, these kinds of contingency tables can be generated for any order. For humans, these become more difficult to evaluate or even write down as Figure 3.5 suggests by showing a table for third-order interaction search. Unfortunately, raising the order does not only increase the number of required tables exponentially but also the size of each table. Specifically, each SNP in the combination adds three additional alleles to each group, therefore, the number of counters in a table can be expressed as 2×3^k, with k being the order of interaction. Thus, second-order interaction requires 18 values, third order 54 values and fourth order 162 values per table. Operating on these large data sets poses a major challenge in all involved areas of computation, especially communication paths, their associated stream handling, and the arithmetical units to calculate the respective measures.

3.2.2 Second Order Interaction Measures

To evaluate the grade of interaction between two SNPs, a large number of algorithms have been implemented in the last years. Most of them employ complex statistical models that measure the differences of the allele frequency distribution between the case group and the control group.

One of the simpler measures is iLOCi [PNI+12]. In this method, the case group distribution is formulated as a probability mass function and compared to a ground truth constructed from the control group. A different approach is taken by GWIS [GRW+13] where classification models are proposed, based on receiver operating characteristics (ROC curves), which are often used in machine learning algorithms. When going further in this direction, many methods have evolved around random forest-based variable prediction, such as Random Forest Fishing [YC14] or Permuted Random Forests [LMA+16]. Generating ground truth assumptions through random permutation is also well-established and used in multi-factor dimensionality reduction-based methods, such as MB-MDR [JLS11] and its various descendants.

Another complex algorithm takes a classical regression-based approach. In BOOST [WYY+10], two logistic regression models M_H and M_S are obtained from a contingency table, where M_S is called *saturated* and is

cases ($l = 0$)			SNP A		
			w	h	v
SNP B w	SNP C	w	n_{0000}	n_{0100}	n_{0200}
		h	n_{0010}	n_{0110}	n_{0210}
		v	n_{0020}	n_{0120}	n_{0220}
SNP B h	SNP C	w	n_{1000}	n_{1100}	n_{1200}
		h	n_{1010}	n_{1110}	n_{1210}
		v	n_{1020}	n_{1120}	n_{1220}
SNP B v	SNP C	w	n_{2000}	n_{2100}	n_{2200}
		h	n_{2010}	n_{2110}	n_{2210}
		v	n_{2020}	n_{2120}	n_{2220}

controls ($l = 1$)			SNP A		
			w	h	v
SNP B w	SNP C	w	n_{0001}	n_{0101}	n_{0201}
		h	n_{0011}	n_{0111}	n_{0211}
		v	n_{0021}	n_{0121}	n_{0221}
SNP B h	SNP C	w	n_{1001}	n_{1101}	n_{1201}
		h	n_{1011}	n_{1111}	n_{1211}
		v	n_{1021}	n_{1121}	n_{1221}
SNP B v	SNP C	w	n_{2001}	n_{2101}	n_{2201}
		h	n_{2011}	n_{2111}	n_{2211}
		v	n_{2021}	n_{2121}	n_{2221}

Figure 3.5. Contingency tables for cases and controls. n_{ijkl} reflect the number of occurrences for the corresponding genotype combination in a given SNP triple.

based on the joint probability mass function between SNP variables and the case/control variable. The so-called *homogeneous* model M_H is used for regression analysis regarding the case/control group affiliation as a *dependent variable* while the SNP variables are treated as *predictor variables*. The difference between both models' estimates describes the probability shifts in the case/control affiliation prediction as a function of the case and control allele frequency distributions. Due to its widespread use and high quality results, this method has been chosen for implementation on the hybrid-parallel architecture.

3.2.3 Third Order Interaction Measures

Due to the required computational power involved in exhaustive third-order interaction search, only very few methods have been published and even less have been implemented to solve this issue [WYY+10; WLo11]. Guo et al. use dynamic clustering methods to reduce the measure complexity to clusters of SNPs [GMY+14]. Lacking established methods that are performed exhaustively on every single combination, a new measure is proposed.

In the field of information theory, the mutual information I describes the mutual dependence between two or more random variables. In the previous section about second order interaction methods, the model of BOOST has been introduced where regression models are used to describe the difference of outcomes of a SNP combination with and without prior knowledge of the case/control group affiliation. Wan et al. show that it can be implemented by calculating the Kullback-Leibler divergence D_{KL}, a measure of the difference of two probability distributions. Mutual Information describes a very similar measure but is defined through the entropy of its random variables [CT06, Chap. 2.1–2.3]:

$$I(X_1, X_2, X_3; Y) = H(X_1, X_2, X_3) + H(Y) - H(X_1, X_2, X_3, Y) \qquad (3.2.1)$$

In discrete distributions such as contingency tables, the Mutual Information test is equal to the G-Test, a well-known dependency test in statistics and the successor to Pearson's Chi Squared test [Pea00; Hoe12]. Addition-

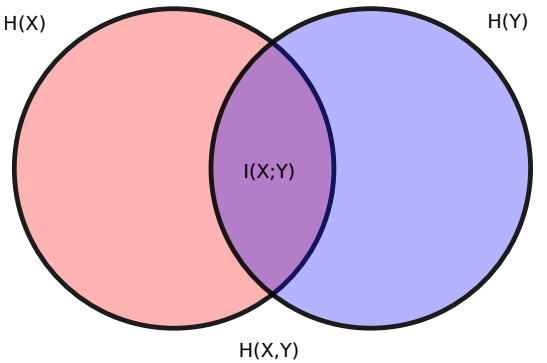

Figure 3.6. Venn diagram for mutual information and entropy

ally, in the discrete case it can then be defined through the Kullback-Leibler divergence as used in BOOST and therefore allows assumptions about its validity and expressiveness [CT06, Chap. 2.3].

Figure 3.6 depicts an interpretation of entropies and the mutual information. The left circle describes the entropy (uncertainty) of the random variable X and the right circle of Y. The union of both is the *joint entropy* of X and Y, while the intersection of both is the mutual information. Therefore, it can be expressed through the difference between the joint entropy $H(X,Y)$ and the sum of their individual entropies $H(X)$ and $H(Y)$, as shown in Equation 3.2.1. Regarding the SNP combinations, the joint entropy of all SNPs $H(X) \leftarrow H(X_1, X_2, X_3)$ is intersected with the entropy of the case/control variable Y.

Chapter 4

The Hybrid Architecture Prototype

Parts of this chapter have been previously published in [GWK+15a; GKW+15; KGW+14; WKG+14; KWG+15; GWK+15b; GSK+14; WKH+17].

This chapter consists of a thorough description of the prototype used to implement the architecture. In particular, all components that were developed and engineered on all levels are presented, from top-level application threading protocols down to electrical interfaces and wiring.

4.1 Overview

4.1.1 The Problem Statement

The overall problem that this hybrid architecture is designed to solve is to create a larger platform to exploit the very different strengths of the very different architectures with respect to the individual architectures' power and potential of computation as explained in Chap. 2. The combined platform targets a high efficiency in terms of energy and computational power, achieved by synergistic effects and therefore performing better than the sum of its components.

The system presented here is generally capable of solving all kinds of problems efficiently if they are reasonably divisible in several mostly self-contained parts. For this work though, two interesting methods from the area of genetics and bioinformatics have been selected as a proof of concept. These methods are described in Sections 3.2.2 and 3.2.3. From an algorithmical point of view, these problems are quite similar but their implementations require different foci on different aspects and therefore lead to different sophisticated approaches as the following sections show.

When accelerating applications or algorithms through parallelization and concurrency, a common bottleneck is communication between the participating components. Hence, it is declared the central challenge to solve and great effort has been made to provide high-performance interfaces with as low as possible set-up overhead and latency.

4.1.2 Problem Partitioning

Both problems described in 3.2.2 and 3.2.3 can be separated into following steps:

1. The input data, a database of genetic material, is read from storage and converted into a format suitable for the remaining parts.
2. SNP data is processed into contingency tables.
3. A statistical test is executed on each table.
4. Tables are filtered depending on their score.
5. Remaining SNP pair information and scores are written to storage.

The conversion step will naturally be executed on the CPU as it is directly attached to the storage and heavily data-dependent. The formats typically used, as well as our application's native format, encode the SNP data, i.e. whether a certain SNP's locus in a certain individual is a heterozygous, homozygous wild or homozygous variant site, in a compressed stream for reasons of efficiency. Hence, its interpretation involves operations on single bits. As contingency tables are created by counting exactly these bit

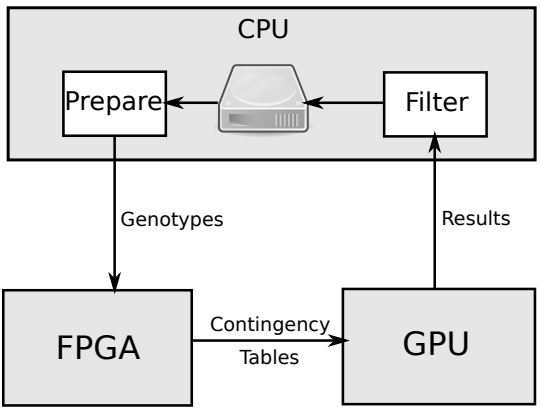

Figure 4.1. General system overview

patters, a highly efficient FPGA implementation could be developed. The mutual information and Kullback-Leibler measures both especially involve arithmetics on real numbers without data inter-dependencies and, hence, are an appropriate choice for GPU platforms. The filtering and reporting steps are implemented directly on the host system for the same reasons as the input data conversions: its "warp divergence potential" and its locality to the storage subsystem. An overview of the various data paths involved is depicted in Figure 4.1.

4.1.3 System Set-up

The hardware platform can be clearly divided into three parts: the GPU module, FPGA module and the CPU, among other tasks, serving as a host. Following the design principles described in Sect. 4.1.1 regarding communication bandwidth, PCI Express (see Sect. 2.2.2) has been chosen as the system's communication network for its wide availability and scalable performance among its simplicity. A system summary is shown in Tables 4.1 and 4.2. Besides the high operating frequency of up to 4.4 GHz, the Intel Core i7–4790K provides a PCI Express controller with 16 Generation-3 lanes with a total gross throughput of 128 Gb/s. Furthermore, the processor

Table 4.1. Prototype configuration: CPU and GPU

	CPU	GPU
Platform	Intel Core i7–4790K	Nvidia GTX 780 Ti
Architecture	4 physical cores, SMT	2,880 CUDA cores
Frequency	800–4,400 MHz	875–928 MHz
Main Memory	32 GB DDR3-RAM	3 GB GDDR5-RAM
PCIe Connectivity	16 lanes Gen 3, 8 lanes Gen 2	16 lanes Gen 3
Compiler	G++ 5.3.1	NVCC 7.5.17 / G++ 4.9.3
System Software	Linux 4.3.0	CUDA 7.5

features four physical cores supporting symmetric multi-threading for eight concurrently running threads [4790]. In addition to the CPU's PCI Express controller, the MSI Z97 XPOWER AC motherboard's platform controller hub, an Intel DH82Z97 chipset, provides eight more PCI Express lanes, Generation 2 [Z97]. The 16 Gen-3 lanes are occupied by an NVIDIA GeForce 780 Ti graphics adapter, featuring a GK110B GPU with 2,880 CUDA cores distributed over 15 streaming multiprocessors. 3,072 MiB of GDDR5 memory are attached to the 384 bit-wide memory bus. The floating-point performance is stated as 5,040 GFLOPs but this only holds with 32-bit IEEE 754 single-precision floating-point operations. Due to the low number of double precision units in the NVIDIA Kepler microarchitecture, only 210 GFLOPs can be achieved with double precision calculations [12].

The prototype architecture has been designed with two different FPGA types, a Xilinx Kintex 7K325T and a Xilinx Virtex 7V690T, two series that differ significantly in price/performance ratios. The Kintex FPGA is hosted on a Xilinx KC705 evaluation board. The FPGA is backed with 1 GiB DDR3 memory, an 8-lane PCI Express edge connector and more peripherals that are not relevant to this work. The FPGA chip itself provides approximately 325,000 logic cells, 840 DSP slices and 16 Mb Block-RAM distributed over 445 units with 36 Kb each. As shown in Table 4.2, the Virtex FPGA offers considerately more resources, 690,000 logic cells, 1,470 Block-RAM units totaling to 52.9 Mb and 3,600 DSP slices, providing resources for much more sophisticated designs.

Table 4.2. Prototype configuration: FPGAs

	Xilinx Kintex 7	**Xilinx Virtex 7**
Platform	Xilinx KC705 Eval Board	Alpha Data ADM-PCIE-7V3
Logic Cells	326,080	693,120
DSP Slices	840	1,470
Block RAM (36 kb)	445	1,470
Frequency	200–250 MHz	200–250 MHz
Memory	1 GiB DDR3-SDRAM	16 GiB DDR3-SDRAM
PCIe Connectivity	8 lanes Gen 2	8 lanes Gen 2

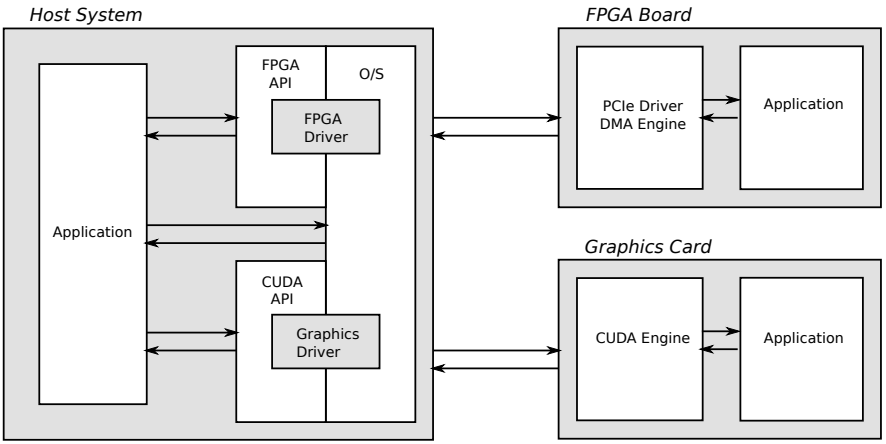

Figure 4.2. Prototype overview

4.1.4 Software

Obviously, the coarse software architecture follows the hardware architecture in terms of work division as described in Sect. 4.1.2. To give a better overview, Figure 4.2 summarizes the prototype system setup graphically.

Between the three application parts, a lot of resource management takes place. On the host side, an operating system manages access to the devices on the PCI Express network. For the specific functions that the devices are to perform, for example, moving data to and from the host application

or PCI Express parameter negotiation, drivers are implemented on both ends of the link. On the host side, the driver is a piece of software being executed within the Linux operating system kernel. It can only be accessed by certain system calls and through the devices nodes anchored in the file system hierarchy, as described in more detail in Sect. 4.4. These accessor functions are combined into shared libraries that make up the Application Programming Interfaces (API). Both FPGA boards in use require different drivers. The Xilinx Evaluation board is bundled with an open-source driver module whose sole purpose was to conduct throughput tests. Although some work and principles could be re-used, large parts of the Xilinx driver had to be rewritten to support the intended use cases in an efficient way. The API encapsulating the Xilinx driver interface was completely written from scratch. The Alpha Data Virtex 7 board, though, came with a full-featured open-source kernel driver and APIs and required little work for adaption to our needs. Both boards' suppliers delivered ready-to-use PCI Express and DMA modules for the FPGA implementation.

As CUDA applications are written in a relatively high-language, there is no direct hardware or driver access, neither on the graphics hardware, nor on the host system. NVIDIA provides well-documented but proprietary and closed-source APIs and drivers for the CUDA development environment. This brings several limitations as the following sections will show.

4.2 FPGA Configuration

The FPGA's main task is creating the contingency tables that are delivered to the GPU part for further processing. Being one of the more complex parts, this section describes the FPGA configuration in detail.

4.2.1 Data Flow and Structural Overview

In a rather abstract overview, the FPGA design is composed of several larger structural components as shown in Figure 4.3. After an initial configuration, the host system sends a certain set of genotypes to the FPGA, ordered by SNPs where for each SNP, the alleles of all samples at that specific

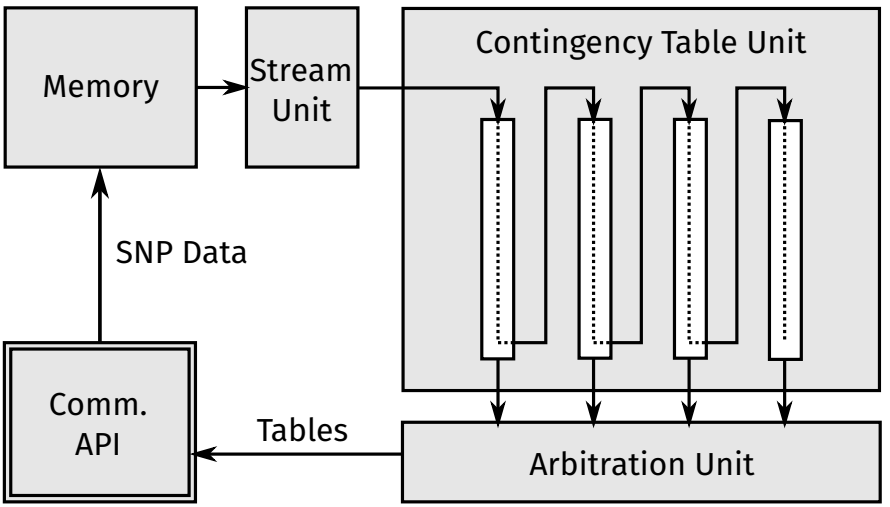

Figure 4.3. Data flow schematic

positions are taken consecutively. The double-bounded symbol "Comm. API" describes the PCI Express/DMA endpoint where these genotypes arrive. From there, they are moved directly into the attached SD-RAM memory module.

Once done, a streaming unit starts moving data to the contingency table unit where tables are created. The table creation is a rather complex process that will be explained more detailed in Section 4.2.3. For this overview though, it is sufficient to know that the creation pipeline is constructed as a large systolic array of small and primitive processing nodes (see Sect. 2.3.2 for an introduction to systolic arrays). To ease the result collection from the nodes, several paths have been inserted where contingency tables are moved off the array.

In the last step, the "Arbitration Unit" fetches the generated contingency tables from several result paths and distributes them over the available DMA channels to send them back to the host for further processing. Naturally, the structure of the arbitration unit significantly changes with the ratio between return paths and available DMA channels and is therefore different in 2-way

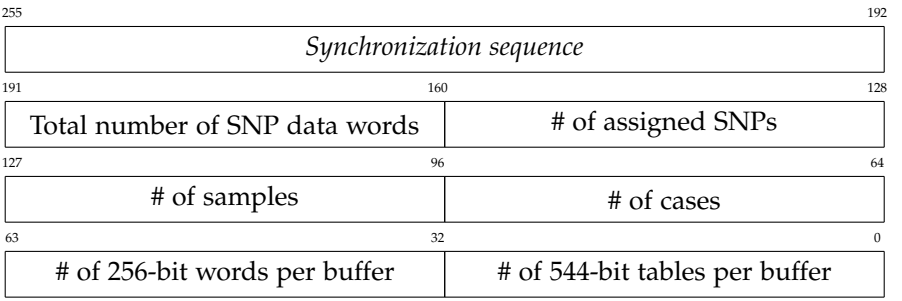

Figure 4.4. Constant initialization

and 3-way interaction analyses.

4.2.2 Initialization

Before the FPGA can set to work, a two-phase initialization sequence has to be performed. In the first step, a set of integer constants is received.

Constants

When designing hardware, only a certain amount of resources is available. For every calculation that is done in a certain instant, hardware has to be provided. These allocations are done in a highly complex synthesis during development and cannot be changed afterwards without re-running this process. To achieve a high efficiency it is therefore desirable to use the allocated hardware primitives as often as possible.

In some cases though, hardware has to be created that is only used once during the initialization without positive effects on the performance. One example would be the number of 544-bit words that will fit into a single transmission buffer. Although this could directly be calculated from the number of 256-bit words that fit into a buffer, hardware would have to be spent to calculate this one value that would not change throughout the whole program run. As seen in Figure 4.4, instead both values are transmitted. Here, the host system calculates this value on the CPU. Due to the nature of using instruction-based processors instead of hard-wired

application-specific processors, the resources used for that computation are not allocated permanently. Therefore, the one-shot evaluation of small constant values is best done on CPUs and beneficial to the FPGA design.

These constants are not necessarily constants in the traditional sense where "constants never change". Instead, they are data-specific values that remain constant with respect (only) to the FPGA. They are computed and derived on the host system based on the nature of the input data among other factors.

The very first value in the constant block is a synchronization sequence. When an application does not properly clean up or is forcefully terminated by the operating system, data may still reside in intermediate buffers between the FPGA's PCI Express endpoint and the host application. Hence, this synchronization sequence can be used to synchronize to the actual start of the application. Any other data word is discarded silently until the defined state is reached by detecting this sequence.

As genotype data is a largely unstructured data stream, further processing entities require certain parameters for correct operation. One example would be the data streamer which moves data SNP-wise and is therefore required to know how large a SNP actually is. Furthermore, the process of creating contingency tables needs to separate control and case genotypes. The final two values indicate how many 256-bit words and how many (544-bit) contingency tables will fit into a single host-supplied transfer buffer. Although this information is not actually required for the FPGA as the API endpoint operates in streaming mode, an additional constraint has to be followed where a transmission buffer must not contain incomplete tables.

Genotypes

In the second part of the initialization process, genotypes are copied from the host application to the FPGA-attached memory where the streaming unit will later fetch its SNP data. These genotype are expected in SNP-major mode, i.e. all genotypes with respect to a single SNP will be collected from all individuals. Similar to the constraint put on transmission buffers, the size of every SNP is required to be a multiple of the memory's word size, 512 bit. Every genotype is expected to take two bits of data that

Table 4.3. Genotype encoding

Genotype	Encoding
Homozygous Wild	`00`
Heterozygous	`01`
Homozygous Variant	`10`
Invalid/Padding	`11`

is encoded as depicted in Table 4.3. If the number of genotypes does not completely fill 512-bit words, the remaining space is filled with genotypes of the "Invalid/Padding"-type. These will be ignored later when contingency tables are created from this data. Immediately after the last SNP data word has arrived, processing begins.

4.2.3 Processing Element Arrays

Figure 4.5 shows the table construction unit as a data flow diagram. This is valid for both 2-way and 3-way methods with the notable difference that a processing node for three-dimensional tables makes use of a second SNP buffer (dotted box). The principles stay the same.

At the beginning of the data path, a data streaming unit "SNP Streamer" reads a SNP from memory and injects 8 genotypes into the array, every clock cycle, until the SNP has been completely transmitted. Then, the second SNP is sent. This continues until all SNPs have been sent.

The very first node consumes all genotypes of the first SNP and stores these into its "SNP Buffer", a Block RAM module of 16 kbit capable of storing 8 000 genotypes. While the second SNP is streamed, the second node does the same and stores it into its buffer without forwarding it to the next node. Concurrently, the first node now uses these genotypes on the fly and reads its own SNP buffer at the same pace to generate combined allele counters resulting in a contingency table. Naturally, this only holds for two-dimensional tables. In third-order interaction, a node will also have to fill its second SNP buffer before a valid table can be created. When a SNP first arrives at the third node, the first node will also pick up the "current"

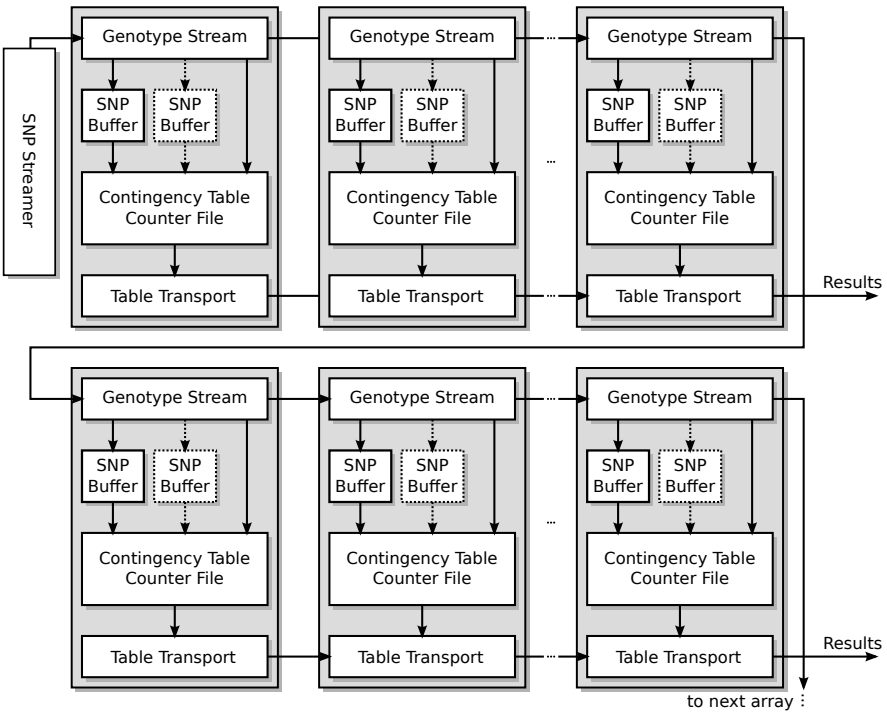

Figure 4.5. Systolic processing element array

SNP and continues to create contingency tables. The first node will create tables for the pairings 1/2 and 1/3, the second node creates 2/3 and the third node still waits for the fourth SNP. For the first few SNPs, the emission will be as follows:

1. 1/2
2. 1/3, 2/3
3. 1/4, 2/4, 3/4
4. 1/5, 2/5, 3/5, 4/5

For third-order interactions, the table generation is delayed by one additional SNP and is analogous to the previous example:

1. 1/2/3
2. 1/2/4, 2/3/4
3. 1/2/5, 2/3/5, 3/4/5
4. 1/2/6, 2/3/6, 3/4/6, 4/5/6

When the last SNP has been injected, all nodes invalidate their SNP buffers and the SNP stream restarts at the Pth SNP with P being the number of nodes in the array. This results in the first newly created table being $(P+1)/(P+2)$ and continues in the same triangular pattern as before. This process is analogous for three-dimensional tables but requires much more streaming iterations to generate all possible triplets of SNPs.

Table Transport

A typical, for benchmarking widely used data set stems from the Wellcome Trust Case Control Consortium (WTCCC). Their first version consists of approx. 500 000 SNPs, 2 000 individuals in the case group and 3 000 individuals in the control group [BCC+07]. For this to work in the presented architecture, a cell in a contingency table requires a minimum width of $\lceil \log_2 \max(2000; 3000) \rceil = 12$ bits. A whole contingency table therefore takes up $2 \times 9 \times 12 = 216$ or $2 \times 27 \times 12 = 648$ bits of space, respectively. Due to this low space requirement, it is stored in distributed memory instead of Block RAM.

The maximum size of the node array depends on the amount of data to be processed. Generated tables are moved off the array in a transport bus, one half table (i.e. cases *or* controls) per clock cycle. For efficiency reasons, the table generation is done separately for case samples and control samples. With 2 000 case genotypes per SNP and 8 genotypes per clock cycle, a node has to be capable of creating a (half) contingency table within 250 clock cycles. This holds for every node in the array and therefore results in 250 contingency tables that are generated in total under full bus load. Therefore,

the array length is capped as shown in Eq. 4.2.1, apart from available system resources. More nodes than this number would lead to bus congestion and eventually lower efficiency.

$$\begin{aligned} P &= \min\{N_{\text{Cases}}; N_{\text{Controls}}\} \div \text{Genotypes per cycle} \\ &= \min\{2000; 3000\} \div 8 \\ &= 250. \end{aligned} \tag{4.2.1}$$

To overcome this limitation a major structural change has been introduced. As shown in Figures 4.3 and 4.5, additional transport buses have been introduced that separate the contingency table streams into equally sized sub-arrays. With $k = 4$ partitions, each transport bus can now take a half table every cycle and hence effectively quadruples the total transport bus capacity and will allow array sizes of up to $P \cdot k = 1000$ nodes.

Due to routing channel congestion on the FPGA, the design achieves a core frequency of 200 MHz. A whole system of 1 000 two-way interaction nodes therefore generates a stream of 320×10^6 contingency tables per second, or 10.7 GiB/s. In Section 2.2.2, the transmission speeds in PCI Express communication systems have been described in some detail. Generation 2 interfaces with 8 lanes, such as in the Alpha Data FPGA board, support a maximum theoretical net throughput of 3 816 MiB/s. Even if all intermediate devices were able to meet this rate, the table rate would require a threefold of this throughput capability. This motivates further analyses on the nature of contingency tables with a reduction of volume in mind.

Counter Compression

As shown in Sections 3.2.2 and 3.2.3, the sum of all counters in a contingency table equals the total number of samples. More specifically, this also holds true for each half part of the table. The sum of all counters in the case part equals the total number of cases. These numbers though, are constant with respect to the data set, and are already known by all participating peripherals and subsystems. An implicit transmission is therefore redundant and can be left out and can simply be done by removing and indeed not

Table 4.4. Table generation rates at 200 MHz and 5 000 samples

	2-way		3-way
Max. group size	16 384	65 536	16 384
Array length	1 000	288	180
Partitions	4	2	3
Tables/s (per node)	320×10^3	320×10^3	320×10^3
Tables/s (per partition)	60×10^6	30×10^6	45×10^6
Tables/s (total)	320×10^6	92.2×10^6	57.6×10^6
Data rate (full tables)	10.7 GiB/s	3.1 GiB/s	5.8 GiB/s
Data rate (reduced tables)	4.9 GiB/s	1.4 GiB/s	4.0 GiB/s

counting a single field per half table. This will not only losslessly reduce the required data rate but also allows to reduce the size of processing nodes in the array and eventually the transport bus widths, freeing resources to implement even more nodes.

With the help of the host system's parser (see Sect. 4.5.2 for details), even more resources can be saved. In a two-dimensional contingency table such as in Figure 3.4, the sum of the leftmost column equals the total number of homozygote wild types in SNP A. If this number is known in advance, one additional value will be saved per column. On the downside, the parser is required to extract and store these counters. Fortunately, as the parsing process processes input data genotype-by-genotype anyway, this results in very little computational burden. Furthermore, these few counters are stored for each SNP, not for each combination of SNPs, proving its efficiency. This scheme of counter saving continues and results in size-reduced contingency tables that only contain 8 values. The same principles apply to three-dimensional tables where the number of counters can be reduced from 54 to 40. Table 4.4 gives an overview of the different configurations and shows that the table data rate could be reduced to 4.9 GiB/s (46 %) and 4.0 GiB/s (71 %) for two-dimensional and three-dimensional tables, respectively.

4.2.4 Transmission Unit

The largest difference between FPGA-implemented two-way and three-way interaction methods is in the Arbitration and Transmission Unit. For the third order Mutual Information approach, the logic is designed for 180 nodes in three partitions, i.e. 60 nodes are serviced by one result channel. The PCI Express API endpoint is configured to use three DMA channels, so the channel mapping is trivial here.

An FPGA configuration supporting up to 4 096 samples per group requires a field with of $\log_2 4096 = 12$ bit per counter. Therefore, a 40-counter table occupies 480 data bits of storage. Each table is accompanied by an ID triplet that uniquely identifies the table so subsequent processing entities know which SNP combination this table comes from. For example, to allow up to two million SNPs, an ID field takes 21 bit, totaling to 543 bit. The DMA channel's port width is 256 bit, requiring a 543-to-256 bit data width changer. This can be implemented by using a FIFO with asymmetric ports. Converting between widths requires a shift register with a size of the lowest common multiple of both port widths. In the case of 256 and 543, the lcm is 139 008. Resource-wise, a 543×256 bit shift register is rather inefficient. Increasing the table size by 1 bit to 544 by adding a bit reduces the distinct prime product from 543 to 17. A 17×256 bit shift register is much more efficient at the low cost of a slightly reduced transmission speed of a single bit per table. The resulting protocol is shown in Figure 4.6.

4.3 GPU Kernels

The implementation for the GPU is much simpler than a hardware description on FPGAs due to its similarity to regular CPU-based programming. Furthermore, the input data as created by the FPGA implementations contain contingency tables that can be operated on without data interdependencies. This allows a programming model known as *embarrassingly parallel*. Nevertheless, a number of challenges arises in the face of the large amounts of data where every snippet of code has to be analyzed down to the instruction to provide the most efficient implementation.

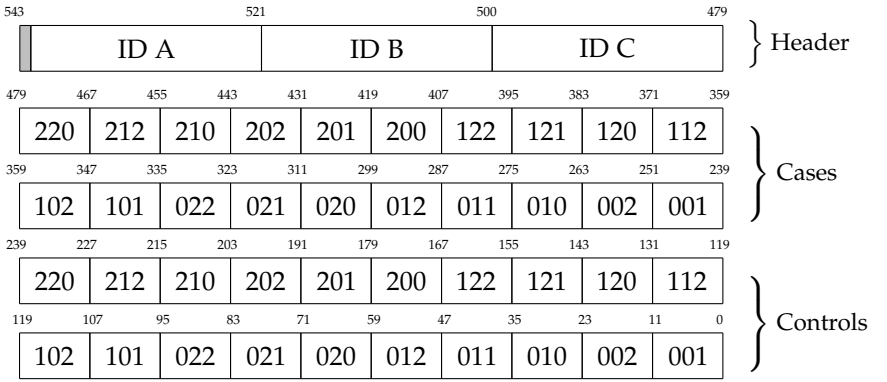

Figure 4.6. Table transmission format

4.3.1 Programming Model and Data Distribution

Moving data from FPGA or similar non-NVIDIA peripherals to the CUDA cores requires the contingency table data to be converted from a data stream to discrete data packets that are handled one-by-one. Typical sizes used in development are 256 MiB, 512 MiB and 1 024 MiB. In second-order analysis, a reduced contingency table as described in the FPGA chapters consists of eight values 16 bit each, 16 bytes in total, which precisely fits into power-of-two-sized buffers. Three-dimensional contingency tables require 40 values of 12 bit each and additional 3×21 bit for their identifier, totaling to 68 bytes. Unfortunately, powers of two are not divisible by 68 and therefore, the buffers are padded to full size with invalid data.

As explained in more detail in Section 2.1.2, before a kernel is launched to the GPU, the execution scheme has to be defined beforehand. Configurable parameters are the number of threads per block and the total number of blocks known as *grid size*. In the case of an NVIDIA GeForce 780 Ti, the graphics processing unit used in the given prototype, every streaming multiprocessor (SMX) contains a register file with 65 536 entries that are dynamically shared among all concurrently executed threads [12]. Therefore, the maximum size of a block is a function of the kernel's space complexity with respect to the number of registers allocated per thread. Within an

SMX, a block is divided in *warps* of 32 threads in a SIMD-style lock-step architecture. Every thread performs the same instruction at the same time. If one thread performs a branch while others in the same warp do not, the branch bodies are executed sequentially by their respective thread. This may lead to a single warp taking longer than other concurrently running warps if their threads did not branch. An SMX can run 192 threads in parallel, or 6 warps, and only a whole set of warps can be replaced, i.e. an execution takes as long as the longest-running warp. To counter the effects of a high warp divergence, the block size can be reduced enough to allow running multiple *blocks* concurrently, removing the barrier functionality at the end of a warp at the cost of increased scheduling overhead. As these parameters are highly dependent on the actual workload and both application's kernels significantly differ regarding warp divergence and occupancy, they will be described and discussed separately in the following sections.

The distribution itself is implemented in the most efficient way possible for a GPU. If every thread on the GPU processes a single table from the buffer, there will be no data or control flow dependencies and therefore no time-consuming synchronization necessary. Furthermore, accesses to the memory are strictly ordered. All 192 threads in an SMX access 192 consecutive memory locations. With this workload, 192 individual memory accesses can automatically be merged to the largest read size the respective memory controller supports. This technique is called Memory Coalescing and can reduce the memory bus load and increase cache hit ratios significantly.

4.3.2 Counter Reconstruction

To minimize the required bandwidth between FPGAs, the host system and GPUs, the FPGA configuration removes some counters off contingency tables that are redundant. As an example, Figure 4.7 shows a $2 \times 3 \times 3$ contingency table in its reduced form. The reduction scheme is explained in more detail in Sect. 4.2.3.

Before a statistical test can be performed, the removed counters have to be reconstructed. Knowing the allele counts SNP-wise beforehand is required as the following reconstruction equations show.Let $L[s, a, k]$ be the number of alleles $a \in \{w, h, v\}$ present in SNP s in the case or control group

		SNP A				SNP A		
	Cases	w	h	v	*Controls*	w	h	v
SNP B	w	n_{000}	~~n_{010}~~	n_{020}	w	n_{001}	~~n_{011}~~	n_{021}
	h	~~n_{100}~~	~~n_{110}~~	~~n_{120}~~	h	~~n_{101}~~	~~n_{111}~~	~~n_{121}~~
	v	n_{200}	~~n_{210}~~	n_{220}	v	n_{201}	~~n_{211}~~	n_{221}

Figure 4.7. A reduced $2 \times 3 \times 3$ contingency table

($k \in \{\text{cases}, \text{controls}\}$).

$$
\begin{aligned}
n_{010k} &= L[B, w, k] - n_{000k} - n_{020k} \\
n_{210k} &= L[B, v, k] - n_{200k} - n_{220k} \\
n_{100k} &= L[A, w, k] - n_{000k} - n_{200k} \\
n_{110k} &= L[A, h, k] - n_{010k} - n_{210k} \\
n_{120k} &= L[A, v, k] - n_{020k} - n_{220k}
\end{aligned}
$$

A three-dimensional contingency table as in Figure 4.8 that naturally contains 2×27 values is reduced to 2×20 values and can be reconstructed analogous to the two-dimensional kind.

$$
\begin{aligned}
n_{000l} &= L[A, w, l] - \sum_{jk \neq 00} n_{0jkl} \\
n_{100l} &= L[B, w, l] - \sum_{ij \neq 10} n_{ij0l} \\
n_{101l} &= L[C, w, l] - \sum_{ik \neq 11} n_{i0kl} \\
n_{111l} &= L[A, h, l] - \sum_{jk \neq 11} n_{1jkl} \\
n_{211l} &= L[B, h, l] - \sum_{ij \neq 21} n_{ij1l} \\
n_{212l} &= L[C, h, l] - \sum_{ik \neq 22} n_{i1kl}
\end{aligned}
$$

cases ($l=0$)				SNP A w	SNP A h	SNP A v
SNP B	w	SNP C	w	~~n_{0000}~~	n_{0100}	n_{0200}
			h	n_{0010}	n_{0110}	n_{0210}
			v	n_{0020}	n_{0120}	n_{0220}
	h	SNP C	w	~~n_{1000}~~	n_{1100}	n_{1200}
			h	~~n_{1010}~~	~~n_{1110}~~	n_{1210}
			v	n_{1020}	n_{1120}	n_{1220}
	v	SNP C	w	n_{2000}	n_{2100}	n_{2200}
			h	n_{2010}	~~n_{2110}~~	n_{2210}
			v	n_{2020}	~~n_{2120}~~	~~n_{2220}~~

controls ($l=1$)				SNP A w	SNP A h	SNP A v
SNP B	w	SNP C	w	~~n_{0001}~~	n_{0101}	n_{0201}
			h	n_{0011}	n_{0111}	n_{0211}
			v	n_{0021}	n_{0121}	n_{0221}
	h	SNP C	w	~~n_{1001}~~	n_{1101}	n_{1201}
			h	~~n_{1011}~~	~~n_{1111}~~	n_{1211}
			v	n_{1021}	n_{1121}	n_{1221}
	v	SNP C	w	n_{2001}	n_{2101}	n_{2201}
			h	n_{2011}	~~n_{2111}~~	n_{2211}
			v	n_{2021}	~~n_{2121}~~	~~n_{2221}~~

Figure 4.8. Three-dimensional contingency tables for cases and controls.

$$n_{222l} = L[A, v, l] - \sum_{jk \neq 22} n_{2jkl}$$

Notably, these equations only show counters that may be recalculated with prior knowledge about allele counters of single SNPs. These are acquired while parsing the initial data as a side product, therefore having a negligible impact on runtime. More counters could be left out with knowledge about allele occurrences in *combinations of SNPs* but is much harder to obtain as it basically solves the second-order interaction problem.

4.3.3 BOOST Measure Calculation

In BOOST [WYY+10], the strength of interaction between two SNPs is measured by the difference of log-likelihood estimates $\hat{L}_H$ and $\hat{L}_S$ between the logistic regression models M_H and M_S as described previously in Chapter 3.2.2. This difference is proportional to the difference between the joint distribution obtained from the saturated model and its factorization and can be expressed through the Kullback-Leibler divergence

$$\begin{aligned}\hat{L}_s - \hat{L}_H &= \sum_{i,j,k} \left(n_{ijk} \log \frac{n_{ijk}}{\mu_{ijk}} \right) \\ &= n \sum_{ijk} \left(\hat{\pi}_{ijk} \log \frac{\hat{\pi}_{ijk}}{\hat{p}_{ijk}} \right) \\ &= n \cdot D_{\mathrm{KL}} \left(\hat{\pi}_{ijk} || \hat{p}_{ijk} \right)\end{aligned}$$

where $\hat{\pi}_{ijk}$ and $\hat{p}_{ijk}$ are the normalized versions of the joint distribution n_{ijk} and the distribution constructed from its factorization $\hat{\mu}_{ijk}$, and D_{KL} the Kullback-Leibler divergence as defined in [KL51]. Unfortunately, M_H does not have a closed-form and has to be calculated iteratively, i.e. using Newton's method. Iterative data dependent methods, though, are hardly computable efficiently. Therefore, Wan et al. et al. propose to approximate M_H with the Kirkwood Superposition Approximation (KSA):

$$\begin{aligned}\hat{p}_{ijk}^{\mathrm{KSA}} &= \frac{1}{\eta} \frac{\pi_{ij.} \pi_{i.k} \pi_{.jk}}{\pi_{i..} \pi_{.j.} \pi_{..k}} \\ \eta &= \sum_{ijk} \frac{\pi_{ij.} \pi_{i.k} \pi_{.jk}}{\pi_{i..} \pi_{.j.} \pi_{..k}}\end{aligned}$$

where η is a normalization factor. They further prove that this approximation using the KSA is an upper bound for M_H. The normalization factor $1/\eta$ is still a large computational burden.

Seeking further computational simplification, a large number of tests have shown that the unnormalized KSA has always been an upper bound of the normalized KSA [GWK+15a]. For this test, all contingency tables of 500 000 artificial, randomized SNPs and 5 000 samples similar to the

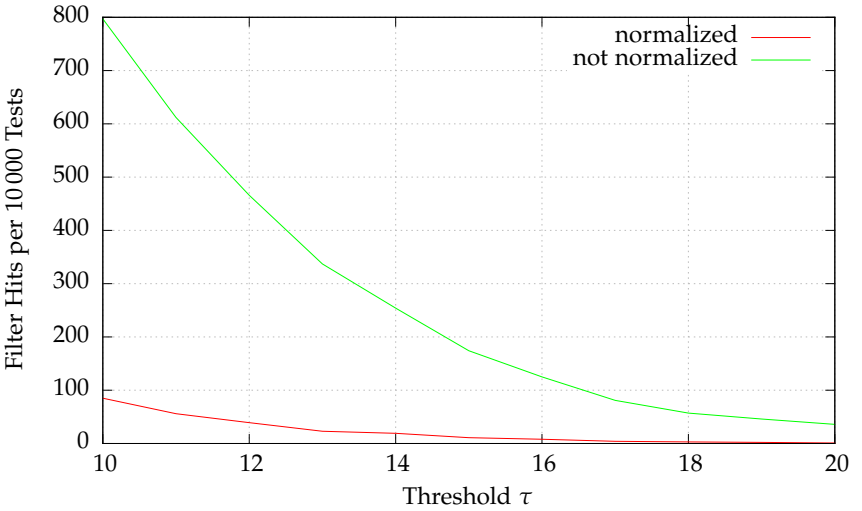

Figure 4.9. Approximation of the Kirkwood Superposition Approximation

first WTCCC data set have been evaluated. Figure 4.9 shows both measure methods with their respective filter passes and a threshold. Though not mathematically proven yet, the unnormalized value is at least a good estimation of an upper bound of the normalized value. The test results have further shown that the unnormalized filter has a rather large type I error (false positive) with respect to the KSA measure while the evaluation did not find a single type II error (false negative). Therefore, the additional approximation is a good candidate for a prefiltering system and is subsequently called the Kirkwood Superposition Approximation Superposition Approximation (KSASA). Assuming that the KSASA is indeed an upper bound of the KSA, the following filter chain can be constructed:

$$\hat{L}_{\mathrm{KSA}} \leftarrow \frac{1}{\eta} L_{\mathrm{KSASA}}$$

$$\hat{L}_S - \hat{L}_H \leqslant \hat{L}_S - \hat{L}_{\mathrm{KSA}} \leqslant \hat{L}_S - \hat{L}_{\mathrm{KSASA}}$$

The overall execution on the GPU follows the scheme given in Al-

gorithm 1. Once a contingency table passes the KSASA pre-filter, the computationally more expensive KSA measure is calculated and compared to its respective, user-supplied threshold. The original BOOST and GBOOST [YYW+11] implementations feature a downstream "log-linear test", where the actual model M_H is evaluated iteratively. As the application's goal is to provide a drop-in replacement for the original software, the implementation presented here also features an M_H evaluation as the last filter in the pipeline being the most expensive test method. For further processing, a positive score is only delivered to the system if it passes the final test, otherwise it is set to zero.

Algorithm 1: GPU Filter Chain

Input: Thresholds τ_1, τ_2 and τ_3 for log-linear testing, KSA and KSASA; a contingency table T
Result: $L_S - L_H$ as an interaction method

```
1 Score ← 0.0;
2 if KSASA(T) ⩾ τ3 then
3 |  if KSA(T) ⩾ τ2 then
4 |  |  Score ← LogLinear(T);
5 |  |  if Score < τ1 then
6 |  |  |  Score ← 0.0;
7 return Score;
```

4.3.4 Mutual Information Calculation

The Mutual Information is a commonly used measure of the mutual dependence of two or more random variables. As described in Section 3.2.2, this implementation uses Mutual Information to describe the interaction between three SNPs. It operates on $2 \times 3 \times 3 \times 3$ contingency tables generated by the FPGA and yields an "amount of information" in a random variable that can be obtained through another random variable. The result is also called *relative entropy* [CT06]. Regarding the actual application, one

random variable is the allele distribution in a SNP combination and the second variable is the actual outcome, i.e. whether the distribution relates to the case group or the control group. Without additional knowledge, the case/control variable is uniformly distributed with maximum *uncertainty*. Calculating the Mutual Information between these two variables reduces this uncertainty and might introduce a distribution bias and therefore give an indication whether a SNP combination influences the occurrence of the trait that has been used to divide individuals into cases and controls.

The Mutual Information for this case is defined as in Section 3.2.2, but can be trivially transformed into equations that are easier on the resources. Here,

- n_l is the number of samples per group (with $l = 0$ for cases and $l = 1$ for controls)
- π_{ijkl} and n_{ijkl} are the relative allele frequencies and absolute allele occurrences d with respect to a contingency table. The field index is defined as i, j, k and $l \in \{0, 1\}$ as previously used.
- $H(X_1, X_2, X_3)$ is the *joint entropy* of the allele frequency tables of three SNPs
- $H(X_1, X_2, X_3, Y)$ is the *joint entropy* of the allele frequencies of a combination of SNPs including the *trait status* (case or control)

$$
\begin{aligned}
n_{ijkl} &= \pi_{ijkl} / n_l \\
I(X_1, X_2, X_3; Y) &= H(X_1, X_2, X_3) + H(Y) - H(X_1, X_2, X_3, Y) \\
H(X_1, X_2, X_3) &= -\sum_{ijk} \frac{n_{ijk0} + n_{ijk1}}{n} \log \frac{n_{ijk0} + n_{ijk1}}{n} \\
H(X_1, X_2, X_3, Y) &= -\sum_{ijkl} \frac{n_{ijk}}{n} \log \frac{n_{ijk}}{n} \\
H(Y) &= -\frac{n_0}{n} \log \frac{n_0}{n} - \frac{n_1}{n} \log \frac{n_1}{n}
\end{aligned}
$$

The actual implementation only calculates the joint entropies and their differences, i.e. $H(X_1, X_2, X_3) - H(X_1, X_2, X_3, Y)$. The calculation of the case/control entropy $H(Y)$ on the other hand, consists only of values that are constant with respect to the data set. It is therefore a data-independent offset that will not influence the natural order of a sorted result list. Hence, the offset is only added to those few results that will eventually be reported back to the user, removing some computational burden from the GPU. It is worth noting that opposed to the factors in BOOST, no larger products over floating-point numbers are required, so higher result precision can be expected as the GPU uses a floating-point implementation based on IEEE 754.

Additionally, calculating logarithms is usually implemented as iterative Taylor-series approximations [Ori07] and is therefore a rather complex operation when compared to others. In the special case of contingency tables though, when factoring out the quotient n in the above equations, the logarithm is only calculated for integers in the range $[0..n]$ which makes using a look-up table feasible, instead of occupying the limited amount of special function units (SFUs) and waiting for complex arithmetics to finish. Hence, a look-up table with $n + 1$ entries and double precision values is pre-computed on the host system and then moved to a cache-efficient read-only memory section on the GPU that the CUDA cores may then take advantage of. Furthermore, NVIDIA's logarithm implementation is known to produce results with rather low precision. Hence, the logarithm look-up can be expected to additionally increase the quality of the results.

4.3.5 Result Reporting

Both implemented methods store their results in the same way. This allows the host system to operate on these data without distinguishing between the various methods and/or kernels that could have been used to create them. Both involved methods can be invoked with either IEEE 754 single precision floating-point numbers or double precision. The possible result frames that are delivered back to the host are shown in Figure 4.10. They always contain three SNP IDs that uniquely identify the underlying contingency table. This is required to enable the host to associate the score with an actual SNP

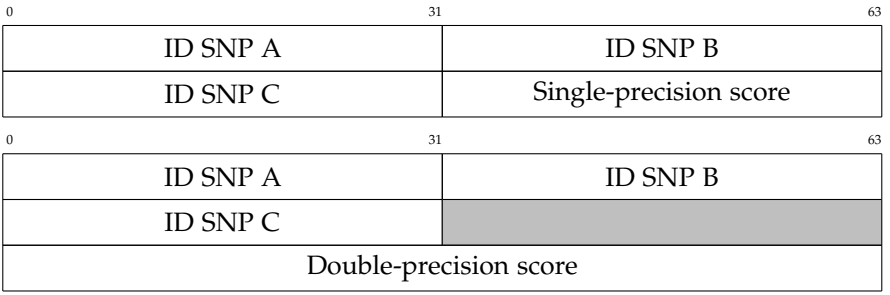

Figure 4.10. GPU result formats

triplet. The fourth field contains the actual score, using 32 bit or 64 bit. A large number of tests has shown that when ordered by score, the ordering of results does not change in dependence of the result score width. As BOOST and the Mutual Information measure are primarily designed as upstream screening applications, higher precision may be of little use. However, if results are required to have a precision higher than approx. 10^{-6}, runtime may be traded, although data center GPU models (i.e. the NVIDIA Tesla series) with a large number of dedicated double precision floating-point units could make the difference in runtime negligible.

On a CPU, data can be most efficiently accessed if their address is a multiple of the word size. For single-precision, this is fairly trivial as the ID fields also come in widths of 32 bit, so no shifting could happen that would move the score field to an unaligned position, assuming the whole buffer starts at such a boundary. For double precision, the value should be aligned to 64 bit boundaries. For the cost of 32 wasted padding bits, the double precision score can be properly aligned.

Once all blocks from the grid have been executed, the on-chip caches are flushed and the results are ready for being copied from the device memory into the host memory for final filtering and processing.

4.4 Host Drivers

On the host computer system, more specifically in its operating system, a layer of driver software connects hardware with the kernel's abstraction layer. As each hardware module is unique and often does not implement standardized interfaces, the hybrid system makes use of a plethora of drivers of which the most important ones will be described.

The kernel's abstraction layer helps developers to write software for generic hardware, such as "sound cards". Of course, the "sound cards" themselves are not generic but the kernel tries to provide a common interface that makes specific sound cards work without the software requiring to know the exact brand and type. The functions specific to sound cards are kept together in a common sound card interface. While this layer is certainly useful, it only makes sense for hardware devices that share a common set of functions. In many occasions, hardware exposes certain special functions that do not belong to this set, such as selecting equalizer presets in sound cards. Drivers for special-purpose hardware may choose to not even use the abstraction layer at all. To handle these non-standard functions, the respective drivers are usually delivered with an Application Programming Interface (API) that seek to hide the complexity involved in driver relations.

The NVIDIA graphics card in this setup uses a closed-source hardware driver and an extensive closed-source API to be used by applications. Apart from the API interface specification, programming guides and rough hardware descriptions, no documentation about the inner workings is available to the public. For the Virtex 7-series FPGA board, Alpha Data provided an open-source kernel driver and an API while the Kintex 7 board did neither come with a driver nor an API. Although the Alpha Data driver exposes much more functionality, the custom driver and software interfaces follow the same principles. Therefore, only these will be described in the following sections.

In Linux-based operating systems, programs can roughly be divided into two realms, those running in user-space and those running in kernel-space. User-space programs are run by users, terminated by the operating system in case of malfunctioning, and closed when done or requested by the user. The operating system provides various services to application

during set-up, during runtime and when the program is terminating (in any case), such as freeing claimed memory resources and closing left-open file descriptors. In general, it is cared for applications to not cause harm to other applications or the system itself as well as reducing resource exhaustion due to misbehaving or badly programmed software. In kernel-space, driver modules are just like regular programs as they may be started and and removed from the kernel during runtime. The major difference is that these modules are directly linked into the running kernel and hence, are not stand-alone programs anymore. Instead of being associated with it, they become a part of it. Therefore, the usual *address space separation* techniques along with its memory protection mechanisms do not work anymore. There is no higher-layer entity that cleans up after a module or terminates it in case of memory corruption. When a module writes to a memory location that it is not supposed to write to, the whole system may lock up or, in case the memory location belongs to a file system or hard disk driver, silent data corruption may occur, usually seen as a far worse consequence than a full system lock-up. Due to this very low grade of protective measures, kernel driver development requires great care and responsibility while only very few assertions on the environment can be made as opposed to writing user-space programs. To support driver developers, Linux provides a large library of commonly used data structures and well-established algorithms and puts a strong focus on separation and encapsulation of problems. This leads to software designs that heavily divert from regular user-space application design principles.

Regular permissions that can be given to users and superusers do not apply for the kernel. Instead, it is running in a special environment with almost uncontrolled access to other drivers, subsystems or bare hardware. For reasons of stability and responsibility, as stated above, drivers should only contain code that is absolutely required to run within kernel-space. This leads to the development of user-space interfaces, so software parts that do not require kernel privileges can be kept in well-defined and operating system-protected APIs in user-space. Of course, this applies to the Kintex FPGA's driver as well.

4.4.1 Interfacing

To protect the running kernel from user-space applications, the only means of communication between both realms is via the *System Call Interface*. System calls are functions exported by the kernel and callable by applications and provide access to the file system (for example by `open` or `read`), the networking subsystem (`bind`, `accept`, `send`), scheduling, process management and many other areas. In Linux, these calls follow a very special call semantic. For every exported function, a unique number is allocated. When an application wishes to issue a system call, it places this number and the call's parameters in certain processor registers and schedules a special instruction `TRAP`. This instruction transfers control of execution to the kernel and jumps to a fixed location where a system call dispatcher handles the request. After completion, control is returned to the application and execution continues at the instruction following `TRAP` [TB14, Chap. 1.6].

It is common among drivers to intercept system calls to the file system for direct user-space communication. Linux itself provides functionality to do so by supporting *special device files* that typically reside within the `/dev/` directory in the file system tree. A driver module may therefore establish a special device file and associate a handler with it. When an application issues system calls that relate to that file, the dispatcher redirects the request to the driver module instead of the regular file system layer. The KC705 driver makes heavy use of this interface. Besides these file-system endpoints, the kernel expects modules to provide certain entry points for management. The complete set of defined entry points is summarized in Table 4.5. The `init`, `exit` and `remove` functions will not be further discussed as they only perform kernel-related housekeeping.

When the driver is loaded into the kernel, it registers itself to be a service operator for a given class of devices. Whenever such a device is plugged in, the kernel iterates over all modules with a matching class until the first module *claims* the yet unclaimed and therefore uninitialized device. This process also takes place after a module has been loaded and unclaimed devices with the specified class exist, and is called *probing*.

The `probe` function of the Kintex FPGA module claims a device if its reported vendor/device identifier is `10EE/7082`. These numbers are default

Table 4.5. KC705 driver entry points

Entry point	Called by	Description
`init`	kernel	Performs module initialization and registration tasks, called right after module insertion
`exit`	kernel	Clean-up and de-registration, called right before module removal
`probe`	kernel	Called when a device has been inserted/-found, is unclaimed and matches the module's class. Performs device-specific initialization such as buffer set-up and installing new file system entry points.
`remove`	kernel	Called when a device has been removed from the system
`open`	user	Registers an application for exclusive access to the FPGA device and establishes a handle
`close`	user	Releases an application's access handle
`write`	user	Schedules a user-supplied buffer for DMA transmission to the FPGA
`read`	user	Queues a user-supplied target buffer for DMA transmission from the FPGA
`ioctl`	user	Requests status information on read or write queues, can set or get metadata for a transmission
`inthandler`	device	The driver is notified that a transmission has completed

values for generic, DMA-capable devices manufactured by Xilinx Corporation and are managed by a group of volunteers [PMV]. In the probing stage, the device is set-up following a specific protocol:

1. Formally enable the device. Basic communication structures are allocated and initialized by the PCI Express subsystem. The device is notified that it should start its initialization routines. The chip set and the device agree on initial PCI Express parameters, such as link width, frequencies and protocol generation as explained in Sect. 2.2.2. Although dependent on the actual hardware and kernel version, these are often set to the lowest permissible values, i.e. generation 1.0 and one single lane.

2. Enable bus-mastering. The device may now provide its DMA controller services to the system bus.

3. Initialize regions. The PCI Express controller in the FPGA design contains a memory region that is now exposed to the host system. It is used for configuration purposes, status reporting and DMA control. Furthermore, it provides a low-bandwidth side channel for data when a DMA channel is not yet available.

4. Allocate file system entry points for communication between a user-space application and the driver module. These entry points already shown in Table 4.5 encapsulate a large portion of the driver complexity and are discussed in more detail in the next section.

5. Install interrupt handler and activate interrupt sources. The device may now actively send notifications to the driver. This is used to signalize the end of a DMA transfer.

6. Re-negotiate link speed and lane count. The engine parameters are set to the highest values supported by all involved components of the hybrid prototype.

When these steps have been performed successfully, the device should be fully operational. In some situations, one or more of these steps may fail, such as hardware errors or resource exhaustion of the host system. Then,

all operations that have already been performed have to be rolled back and resources released to ensure a defined system state. The `probe` function will indicate an error and the device will be left unclaimed.

4.4.2 Data Transfer

The most demanding function of the driver interface is high-speed data transmission between a user-space application and the FPGA device. For the application endpoint, a special device file is installed in the system. The KC705 device driver module intercepts all accesses to this file and allows communication by the operating system's standard I/O facilities. This approach has already been summarized in Sect. 4.4.1.

Before an application may use the modules and therefore the device's capabilities, the driver requires it to first issue an `open` system call on the device file. For our hybrid prototype, concurrent accesses of multiple processes are not desired. Therefore, a binary semaphore is established immediately as file system entry points are generally re-entrant, i.e. many processes may open a file at the same time and the execution threads of the `open` method are interleaved or in parallel. As a binary semaphore only allows a single thread to continue, any other thread is declined access until the *ownership* of the device file is explicitly released. As exclusive access is now guaranteed, a device handle is issued to the process so that it can be uniquely identified in subsequent system calls. No other process can acquire a handle until the `release` call is invoked. Furthermore, no process may successfully call other functions on the file without providing a valid handle. This is particularly true for `release`.

For actual transmission of data, the device file also supports `read` and `write` operations that, as the following paragraphs will explain, are implemented in a very similar way. Before discussing the exact semantics, it is important to recall how the FPGA is supposed to operate the PCI Express endpoint. A somewhat simplified PCI Express/DMA interface on the FPGA is shown in Listing 4.1.

Listing 4.1. Simplified PCI Express/DMA endpoint

```
entity PciExpressEndpoint is port (
  s2c_valid      : in std_logic;
  s2c_ready      : out std_logic;
  s2c_data       : in std_logic_vector(255 downto 0);
  s2c_metadata   : in std_logic_vector(31 downto 0);
  c2s_valid      : out std_logic;
  c2s_ready      : in std_logic;
  c2s_data       : out std_logic_vector(255 downto 0);
  c2s_metadata   : out std_logic_vector(31 downto 0)
);
end entity PciExpressEndpoint;
```

It can be seen that data that is to be sent or received is stored in 256 bit-wide registers. The transfer itself is managed by the handshake signals `ready` and `valid` that will be asserted *high* whenever the terminal is ready to handle a new data word or the currently displayed data word is actually valid, respectively. The overall behavior can therefore be described as streaming operation with handshake.

In programs written in imperative programming languages such as C or C++, it is not only cumbersome but highly inefficient to handle a large number of 256 bit-sized buffers. Instead, a single, large memory buffer should be supplied to the driver where the DMA engine may store 256 bit words in consecutive word addresses. In terms of handshake, `s2c_valid` and/or `c2s_ready` should be asserted *high* whenever the DMA engine has claimed a buffer for a (host-side) write or read operation, respectively. The "conversion" between stream-oriented and packet-oriented transmission has an undesired consequence: the FPGA may not write data to the host whenever it would be ready to. Instead, the host system is required to exactly know when to expect data from the FPGA as it has to actively supply a target data buffer. The other direction is less counter-intuitive. Whenever the `s2c_valid` is *high*, the FPGA application "knows" that data is expected.

The naïve implementation approach would be to supply a buffer to the DMA endpoint directly after a successful `probe` and provide it to the client application as soon as a `read` call is invoked. As previously mentioned, the kernel provides address space separation and transparent process management for all user-space applications, but not for applications running *within* the kernel. Hence, while a driver can certainly allocate memory within the kernel, it cannot be easily moved to the application, or vice-versa. Moving memory between kernel and user-space implicates moving memory from physical memory to virtual memory. The only way to move data here is to use a byte-by-byte copy mechanism provided by the virtual memory subsystem which would introduce a tight performance bottleneck. The reason why we use DMA in the first place is to remove the burden of copying data which is exactly what we would impose on it by kernel/user-space data movement.

In Sections 2.1.1 and 2.2.1, the foundations of virtual memory and DMA transfers have been explained in detail. DMA scatterlists are lists of physical memory pages that ought to be transferred. Fortunately, the virtual memory subsystem also provides functions to inspect memory regions and page tables which contain the mapping between virtual memory and physical memory. Note, though, that this description of page tables is highly simplified but this abstract knowledge will be sufficient for our cause. Therefore, a more sophisticated approach than kernel-allocation and copying has been chosen. The KC705 module makes use of the complex `get_user_pages` kernel API function as it is defined in Listing 4.2.

Listing 4.2. Definition of `get_user_pages` from the Linux API documentation

```
int get_user_pages (struct task_struct * tsk,
                    struct mm_struct * mm,
                    unsigned long start,
                    int nr_pages,
                    int write,
                    int force,
                    struct page ** pages,
                    struct vm_area_struct ** vmas);
```

The Linux Kernel API Documentation for `get_user_pages` reads as follows:

> Returns number of pages pinned. [...] get_user_pages walks a process's page tables and takes a reference to each struct page that each user address corresponds to at a given instant. That is, it takes the page that would be accessed if a user thread accesses the given user virtual address at that instant.

Although the documentation, and certainly the idea of `get_user_pages` too, focus on the aspect of *pinning*, it also returns a list of pages that have been mapped into the user memory block that is given via the `mm` parameter. Fortunately, as the memory is also pinned, it is guaranteed that every address that is accessible in the buffer is actually backed by a physical memory page. Instead, it could have been paged out to disk or not even been allocated in case some memory pages had not been accessed before. The returned structures that store information about the process's pages also contain the physical memory address that is required to fill a DMA scatterlist. The pathway between a specific location in the physical memory and a virtual memory address does not only contain page table translations but also make heavy use of caching. Hence, after directly writing to the respective physical memory (i.e. from an FPGA), user page caches are required to be flushed to prevent lost updates or memory corruption in general. For the same reason, the user-space application may not access the memory block in any way until the memory is un-pinned and released. Not only lost updates occur, but this may also trigger new caching efforts of the operating system, possibly further corrupting data. Even read-only access therefore has to be suppressed completely.

Depending on whether the requested operation is a `read` or a `write`, the list of memory pages is then queued up in the *read page descriptor ring* or the *write page descriptor ring*. Whenever the FPGA's DMA controller signalizes that it is ready to perform a transaction, and the respective `ready` or `valid` signals are asserted, the scatterlist is loaded with entries from one of the ring buffers. As the DMA engines proceeds to process one page after another, its scatterlist is refilled from the ring. By using this technique, only one long-running DMA transaction is required, at least theoretically. When

processing stops, which may happen if the FPGA is not ready to take more data, the start pointer of the ring buffer is moved back to the last page that has not been successfully transmitted and the scatterlist is purged so the system memory bus can be released by the controller. The scatterlist itself is stored into the DMA engine through the PCI Express configuration space (see Sect. 2.2.2). Every time the DMA engine has successfully processed a scatterlist entry or stopped processing, an interrupt is sent to the driver announcing a change of state. The state itself is then retrieved from the configuration space and the driver decides whether the scatterlist should be refilled, the transfer be partly rolled back or completely aborted.

Historically, motherboard chip sets provided dedicated *interrupt lines* for every peripheral, so the CPU could be notified asynchronously. Without interrupt lines, peripherals had to be periodically polled for status changes which is not only inefficient but also makes real-time behavior as required by sound cards hard or impossible. Interrupt lines are then aggregated by an interrupt controller which in turn is connected to a CPU. On the software side, Linux therefore may only implement a single function to communicate with the interrupt controller. When a driver registers a function to handle interrupts, a pointer is placed into the interrupt handler list. On interrupt, each registered handler is called until the first handler indicates that the interrupt has been handled and can be acknowledged on the interrupt controller interface. A new interrupt may hence only arrive if the previous has been acknowledged via controller handshake. Thus, interrupt processing blocks subsequent interrupts. In Linux, it is therefore advised to split interrupt handlers in half if the handler is considered to be a *lengthy task* that could seriously delay other probably timing-critical interrupts [LDD3, pages 275ff.]. In the so-called *top half*, the interrupt is acknowledged and timing-critical operations may be performed. It then declares a *bottom half* that does lengthy operations such as data input/output to/from PCI Express configuration memories or page table processing. Before the top half returns control to the interrupt context, the system scheduler is instructed to execute the bottom half in a less critical execution context. This way, long-running tasks can be processed on interrupt events without delaying subsequent notifications at the expense of much higher programming and development complexity.

When the full scatterlist associated with the user buffer is processed by the DMA engine, the pages are un-pinned and the user-space application may use the released memory again. If an application calls `write`, waits until the data is written and then calls `write` again to send another memory buffer, some time may have passed without transmitting data. To maximize data throughput, `write` and `read` have therefore designed to operate *non-blocking*, i.e. they return control back to the application as soon as the memory buffer is successfully enqueued to the respective transmission ring. Applications may now schedule more buffers until the transmission ring is full, theoretically keeping the DMA engines running for an unlimited amount of time. The downside is that the application is responsible for not corrupting the associated memory by accessing it and activating caching mechanisms.

As `write` and `read` cannot signal the transmission's completion, a special `ioctl` entry point has been implemented. `ioctl` is considered to be a universal file system I/O call with custom functionality. For clarification, its manual reads as follows:

```
int ioctl(int fd, unsigned long request, ...);
```

> The ioctl() function manipulates the underlying device parameters of special files. [...] The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally char *argp (from the days before void * was valid C), and will be so named for this discussion. An ioctl() request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes.[...]
>
> [It is] conforming to no single standard. Arguments, returns, and semantics of ioctl() vary according to the device driver in question (the call is used as a catch-all for operations that don't cleanly fit the UNIX stream I/O model).

In this driver, it has been used to query the status of both transmission rings. Therefore, the module defines the `IOCTL_KC705_TX_STATUS` and `IOCTL_KC705_RX_STATUS` constants as request identifiers. The number of free

transmission buffer entries is returned. As one entry describes a single memory page and on the hybrid prototype's processor, a page is fixed to 4 096 bytes, the caller may calculate the actual amount of memory space that may be queued. As a security precaution, a buffer may not be queued in parts. That is, even if *some* pages may be queued, queueing is rejected if there is not enough space to hold *all* pages associated with it. Due to the sensitive nature of execution in kernel-space, several more measures are taken that relate to the safety and validity of supplied user data. A bad address supplied as a memory pointer might very well bring the whole system down or worse, cause silent data corruption as has been previously mentioned. A great amount of responsibility lies therefore on the programmer specifically when designing user-space interfaces for driver modules.

Furthermore, it is worth noting, that `write` and `read` may be called concurrently. Their processing pathways and data structures in the kernel module are strictly kept seperate to allow unhindered full-duplex transfers. At no time though, two `writes` or `reads` may be called concurrently. One of both calls will return with a "Device or resource busy" error code, as defined in the Linux standard libraries as `EBUSY`.

4.4.3 Application Programming Interface

To ease application devlopment, an application programming interface (API) has been devloped. Its main purpose is to hide the interface complexity of the driver making development more usable. This section describes the noteworthy internals and presents the interface that applications should use for FPGA connectivity. It is written in the C programming language, in its standard version ISO/IEC 9899:1999, also known as C99.

The core functionality of the API is split in two parts: the client functions and the polling thread. The latter is responsible for polling the ioctl interface as described in the previous section. The former are the actual exported functions.

Listing 4.3. KC705 open and close function prototypes

```
int kc705_open(const char *device_filename,
               struct kc705_handle **new_handle,
               FILE *log);

void kc705_close(struct kc705_handle *hdl);
```

Before an application may communicate with the FPGA, a device handle has to be obtained by calling `kc705_open` as defined in Listing 4.3. On success, the FPGA device identified by the device file `device_filename` is opened and a handle is stored in the pointer given in the `new_handle` parameter. Optionally, a file handle may be supplied as a logging target, warnings and error messages are written to the given descriptor. If no logging file is supplied, the respective messages are written to the standard output stream, usually the terminal from where the application has been started. Typical "open functions", such as the Standard C Library's `fopen` return the handle directly and use a separate global variable as an error indicator. Instead of polluting the variable namespace, every KC705 API function returns an error code while actual values that might be of interest to the application are placed in pointed-to structures supplied by the caller. The `struct kc705_handle` is a complex structure containing many internals that are not of interest for the user. Therefore, the structure is realized as an *opaque type*. That is, a type that is formally defined without data members or values. Hence, it is impossible to access any of the structure's members. The actual list of members is declared within the non-public library headers so the API implementation can make use of them anyway.

On the internal side, the call to `kc705_open` creates a thread using the operating system's *POSIX Threading* library. Otherwise, it would be the application's responsibility to regularly poll the `ioctl` interface whenever a buffer is expected to be processed. How exactly the polling thread is implemented will be explained in the following paragraphs.

Listing 4.4. KC705 asynchronous read and write function prototypes

```
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);

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);
```

Listing 4.4 refers to the prototype definitions of the basic read and write functions of the exported KC705 API application interface. The first parameter specifies a device handle that has been previously obtained by successfully calling `kc705_open`. `buffer` and `buffer_length` are used to provide a memory region whose contents should be written to the FPGA or written to by the FPGA. After calling one of these functions, the buffer address is given to the device driver that pins the memory and queues it for DMA transfer. These functions do not wait until the transfer request has been processed. Their return status simply indicates whether the buffer has been successfully queued for transmission. If that is the case, the buffer must not be accessed until it is released by the driver. The `callback` parameter allows a user application to register a function that is called by the KC705 API when the supplied buffer has been processed. That is, the buffer may not be accessed until the callback notifies the completion (or failure) of the transfer. The callback function is given a set of parameters such as the buffer address, the number of bytes that have actually been written or read and a

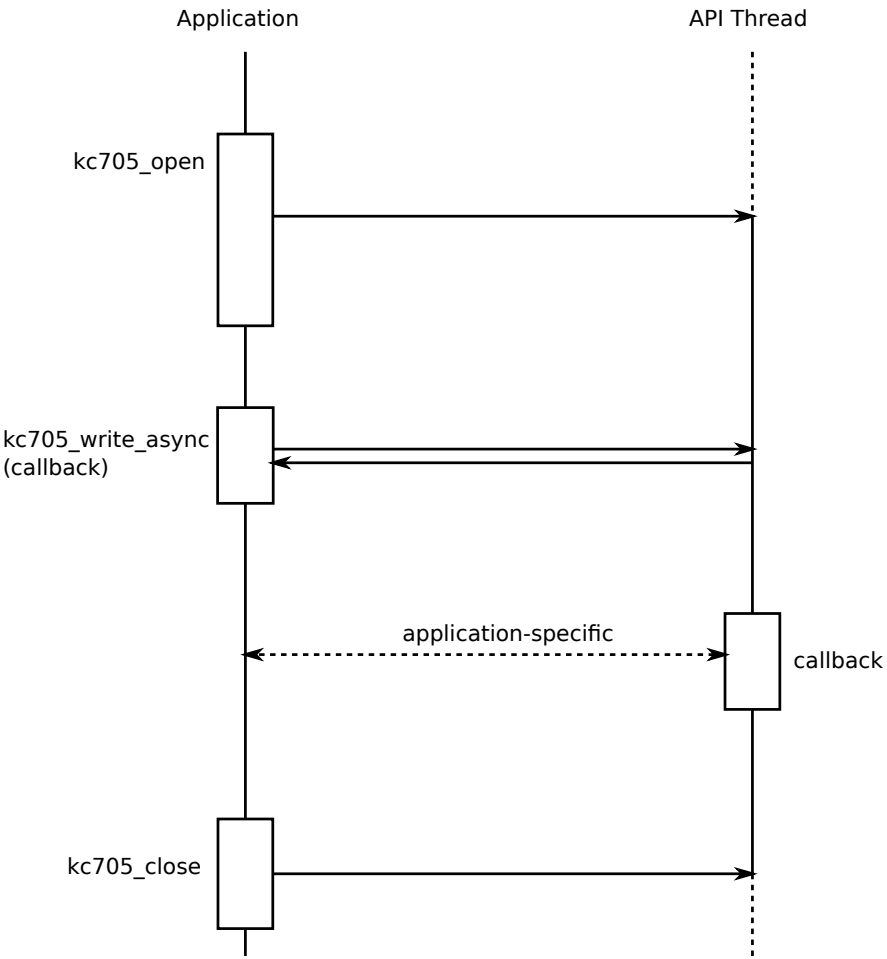

Figure 4.11. KC705 API threading diagram

`user_data` field that contains an arbitrary value that has been supplied in the `user_data` parameter in the actual API call. This allows the application to easier correlate the calling of the function to the respective API call.

In addition to these parameters, an application may write a `metadata` value that is kept constant throughout the transfer and delivered in a side channel to the FPGA where it is available in the first clock beat where `s2c_valid` is asserted high. The FPGA interface also allows transmission of metadata to the host. Obviously, this value is not available until the data has been retrieved from the FPGA and therefore cannot be delivered in the API call. Instead, the registered callback function of `kc705_read_async` receives this value along with the actual data. This metadata functionality can be used to transfer higher-level protocol data or encode other side-channel information.

The callback functions are called by a special thread created in the `kc705_open` function call. Its special purpose is to regularly inquire about the buffers' transmission statuses, relate page lists to buffers and eventually initiate callbacks to the application. It therefore consists of a delayed loop that calls `ioctl` to receive the list of pages in the transmission and reception rings. Then, the lists are compared against the list of user-supplied buffers. As soon as a buffer has been processes, i.e. the page list corresponding to a buffer has been completely removed from either the transmission or reception list, the associated callback function is executed as shown in the threading diagram in Figure 4.11. This function is executed in the KC705 API's thread context. The application is therefore required to establish its own thread communication mechanism. Furthermore, the function call delays the polling function and should be as short as possible, similar to the kernel interrupt routines described earlier. To maximize throughput, several writes and reads can be initiated without waiting for completion of previous transmissions. The respective callbacks are executed in-order which further emphasizes the focus on runtime efficiency of the callback functions.

To reduce the amount of set up code and ease development, two convenience functions have been introduced that encapsulate the read/callback and write/callback mechanisms in synchronous forms. The corresponding prototypes are shown in Listing 4.6.

Listing 4.5. KC705 synchronous convenience wrappers for reading and writing

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

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

These functions can be used much more intuitively and do not require a deep understanding of the C Programming Language regarding function pointers. While the asynchronous versions introduced earlier can lead to an unstable system if the user application accidentally or maliciously accesses a pinned buffer, the synchronous wrapper functions hide the complexity and render this virtually impossible. For a prototype system though, where performance potential is more important than security, access to asynchronous functions is fundamental as it is not possible to queue up multiple buffers when using only synchronous functions.

As shown in Listing 4.6, the KC705 API uses internal callback handlers for reads and writes. To synchronize communication between the callback thread and the application thread, a mutual exclusion primitive (*mutex*) and a condition variable are employed. The former is implemented as a binary semaphore, which can only be locked by one thread. If any other thread tries to lock the mutex, its execution is paused by the scheduler until the mutex is unlocked. Then, a random thread is selected and allowed to lock the mutex again. While a thread is waiting for a lock, it does not consume CPU time.

The condition variable provides the possibility to send a signal to one or more threads that are waiting for a mutex. To wait for a signal, the corresponding mutex has to be locked first. Then, when calling `wait`, the lock is implicitly released without resuming scheduling. This allows more than one thread at a time to use the condition variable. Another thread,

such as our callback thread, can now lock the mutex, and perform the condition variable's `notify` operation and then unlock it again. As soon as the mutex is unlocked, one waiting thread is randomly selected to continue. This would be the application thread in the case of the KC705 API.

Listing 4.6. KC705 convenience wrapper for writing (simplified)

```
struct notify_handle {
  condition_variable cv; mutex mtx; unsigned bytes_written;
} nhandle;

initialize_cv(nhandle.mtx);
lock(nhandle.mtx);
int status
    = kc705_write_async(device,
                        buffer, buffer_length,
                        metadata,
                        internal_write_callback, &nhandle);

if(status == OK) {
  wait(nhandle.cv);
  unlock(nhandle.mtx);
  return nhandle.bytes_written;
} else {
  unlock(nhandle.mtx);
  return FAILURE;
}
```

4.5 Host Application

The host application takes up a special role in the hybrid system. Despite FPGAs and GPUs carry the major computational burden, they are not independent systems. Instead, they are peripherals to the CPU and the applications and drivers running on the CPU. Therefore, it is the host

system's responsibility to set up, instruct and direct almost all aspects and devices in the hybrid system's ecosystem. Some of these duties are carried out by the Linux kernel and drivers as explained in Section 4.4, but a large part remains to be handled by the actual user-space software package.

4.5.1 Overview

The most demanding part in the host application is data flow handling. The size of data sets is usually measured in gigabytes. These data are to be parsed from a set of various formats, text-based, binary and/or compressed, transformed into a special encoding suitable for processing on FPGAs and efficiently stored. These records are distributed to a number of FPGAs. Each FPGA generates several high-throughput data streams that have to be served. Then, data is moved to a number of GPUs, results are fetched, parsed, filtered and converted to the desired output formats.

The highest efficiency can only be achieved if sending and receiving the various data streams is done in parallel. For example, four FPGAs with three streams each require 12 concurrently running threads to be communicating with each other only to move data out of the FPGA. It is not only critical to correctly orchestrate such a large number of threads but also to provide synchronization and data structures that are capable of handling these amounts of data in time.

An overview of data flows is given in Figure 4.12. Before data can be processed into contingency tables, they are first read from hard disk and converted into a suitable format. A scheduling algorithm distributes the resulting SNP data equally to the available FPGAs through the use of multiple threads. Each FPGA then uses a number of DMA channels (see Sect. 2.2.1) to move the contingency tables to the host's main memory. To reduce times where no reception buffer resides at the DMA endpoint, every channel is serviced by a dedicated thread. The work buffers are then stored into a concurrently used queue ("Table Buffer Queue") where each GPU is supplied with. Again, each GPU is serviced by a dedicated thread. A second instance of a work queue is then used to distribute GPU result sets to a number result processors on the CPU. The number of threads in use by result processing is determined by and adapted to the host's threading

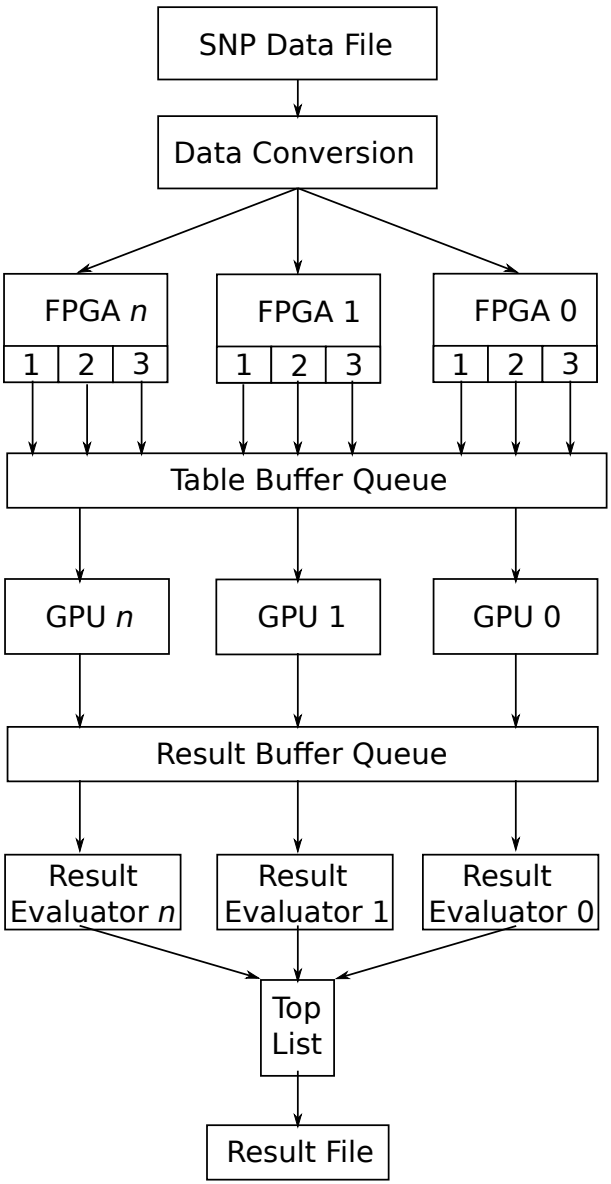

Figure 4.12. Application data flow schematic

capabilities and the current backlog of work. Once the backlog reaches a certain number, additional threads are spawned for result processing. Here, GPU results are filtered either by threshold or by top lists of user-defined length. Finally, the filtered lists are merged and written to disk.

These steps will be explained in detail in the following sections.

4.5.2 Input Data Files and Conversion

In bioinformatics, especially in methods generating or using data from genome-wide association studies, apparently no consensus exists on data formats. Instead, every software requires its own formats, so a large number of format converters have emerged. The host application does support a number of formats, where the two most important ones are described in the following sections. Additionally, an efficient loading mechanism is presented. All formats contain at least the following information that is required to set up a GWAS analysis:

- a list of individuals ("samples"), where each record contains a pseudonym and whether they belong to the case or to the control group
- a list of SNP markers with distinct names
- for every SNP and individual, either two genotypes are given or a type classification (homozygous wild/variant or heterozygote)

As not only interaction searches are performed on GWAS, databases often contain additional information, such as individual pedigrees, chromosome numbers or genome positions. These are not used by the methods presented here.

The PLINK Data Set Format

The authors of PLINK, a large tool set for all kinds of operations on sample data, managed to establish a format that is supported by most current methods and data providers [CCT+15].

The basic format consists of two files, a pedigree file and a map file. Both are text-based formats where each record is stored per line. The pedigree

file contains one line per sample. The first six space-separated columns contain arbitrary family, individual, paternal and maternal identifiers, a sex column and a phenotype, also described as "affection status". From the seventh column onwards, bi-allelic genotypes are specified, i.e. two genotypes per SNP. All lines have to have the same number of fields and the number of SNPs that have been genotypes provided for has to match the number of SNPs specified in the map file. The map file contains definitions for the SNPs used in the pedigree file, one per line. Each line contains the chromosome, an arbitrary SNP identifier, and absolute and relative positions of the SNP in the genome. Examples of these files are shown in Listings 4.7 and 4.8. The example pedigree file defined three individuals `MOMMY`, `DADDY` and `SON` who belong to the same family `FAM001`. All individuals have three bi-allelic SNPs defined. According to the map file, the first SNP originates from chromosome 1 and is called `rs123456`, the second on chromosome 20 while the third SNP's origin is not (yet) known.

Listing 4.7. PLINK pedigree file example

```
FAM001 MOMMY 0 0 2 2 A A G G A C
FAM001 DADDY 0 0 1 2 A A A G 0 0
FAM001 SON   1 2 1 1 A A G G A 0
```

Listing 4.8. PLINK SNP map file example

```
 1  rs123456  0 1234555
20  rs234567 42 1234522
 0  rs443322  0       0
```

Efficiency-wise, storing data in text formats is not optimal. A data set composed of 500 000 SNPs and 5 000 samples may easily require 10 GiB of storage space. The PLINK toolkit therefore supports a binary version of the above. In this more efficient format, the first six columns are left intact and moved to a family file. The remaining genotype columns are then encoded into two bits for every SNP. Homozygote types are stored as `00` or `11` depending on the type, wild or variant, respectively. Heterozygotes are

encoded as `10`, no matter how the alleles are ordered. Unknown or missing genotypes are encoded as `01`. The very first genotype is stored in the lowest significant bits of a byte. In an eight-bit field (or byte), the first individual in the above pedigree file might therefore be encoded as `01101100`. The two highest significant bits are set to *unknown* as a fourth genotype does not exist. After the first individual as been encoded, the next individual is started at the next byte. Hence, if the number of SNPs is not divisible by four, every individual has to be padded to fill up the last byte.

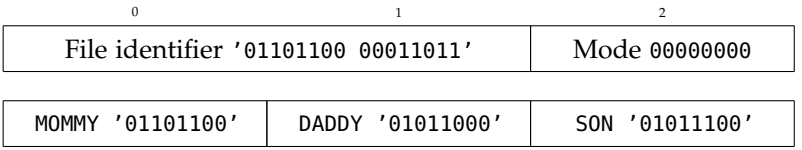

Figure 4.13. PLINK binary pedigree file in individual-major format

The actual implementation as seen in Figure 4.13 also includes three header bytes per file. The first two bytes identify the binary format and the third byte indicates whether the genotypes are stored as individual-major as described above, or SNP-major, where the pedigree file is effectively transposed. In addition to the binary format, a transposed version also exists for the text format.

Reading, Queuing and Parsing

When programs load data from hard disks or other non-memory or even non-local media, a large amount of time is typically spent in `read` system calls. Effectively, the currently running thread is suspended by the system scheduler until the request can be completed. In an efficient system, this time can be used to perform non-I/O work, such as parsing data that has been previously acquired. This requires the use of threads.

The application therefore spawns two threads. The first thread just extracts records (i.e. text lines, individuals or SNPs) and spends most of the time in a waiting state while the second thread converts the record into an FPGA-compatible native format. Communication between threads, though, has been and is still a major research topic in operating system

theory and parallel architectures [BO03, Chap. 13]. Simple synchronization schemes may be implemented by the use of semaphores but their use in high-frequency locking/unlocking is generally considered excessively demanding [ADK+12; OGK+12]. A rather obvious but nonetheless complex solution is to find data structures that do not require locking for shared access. In the previously mentioned work, the authors develop a lock-free single-producer single-consumer queue (in short, SPSC queue) first introduced by Leslie Lamport in 1979 [Lam79]. This queue structure has been implemented to move data from the reader thread to the parser thread for maximum efficiency and CPU saturation.

The expected efficiency is heavily dependent on the input format. Currently, plain-text PLINK transposed ("TPED/TFAM"), SNP-major PLINK binary ("BIM/BED/FAM"), the custom data file format used in the original BOOST software [WYY+10] and the native format described in 4.5.2. These formats put a different computational burden on the parser thread. Additionally, all formats may also be compressed using `gzip` (an implementation of the DEFLATE method [Deu96]) or `bzip2` (a Burrows-Wheeler-based Huffman compressor [Sew]). The decompression is implemented as a reading filter, so the computational burden lies on the I/O thread.

To further allow reconstruction of contingency table values that have been optimized out during generation (see Sect. 4.3.2 for details), the parsing thread also counts the number of wild and variant type homozygotes and heterozygotes, divided in cases and controls. These six counters are stored along by the SNPs.

The parsing result, regardless of the input data format, is designed to be stored efficiently. As described in Section 4.2, the FPGA expects data in SNP-major ordering in a 2-bit encoding as it is described the native format introduction in Section 4.5.2. Before the first SNP is written to the internal database, the number of case and control samples and the number of SNPs have already been determined. Knowing these parameters, the exact size of the full data set can be calculated. Hence, the database does not need to grow with the data but can be allocated a constant size. This reduces memory fragmentation and avoids costly memory allocation re-sizing. The database structure itself consists of a single, continuous byte buffer and a number of accessor methods that allow efficient and concurrent random access to any

ranges of SNPs. This is especially important when distributing SNP data to FPGAs. Additionally, when using certain command-line arguments (see Appendices A.1 and A.2), a native data dump can be generated for faster loading times in subsequent program calls.

The Native Format

The native format can be generated from any other input format to reduce subsequent loading times when working with the same data sets, i.e. if just a threshold parameter is changed. As it directly resembles the internal database format, loading and storing does not require any expensive parsing or format conversion. Often, input files such as the PLINK format include more information than required. These extra data are stripped off. Hence, many formats can be converted into the native format but not generated from it. It can be seen as a lossy, one-way transformation. The exact structure is described in the following paragraphs.

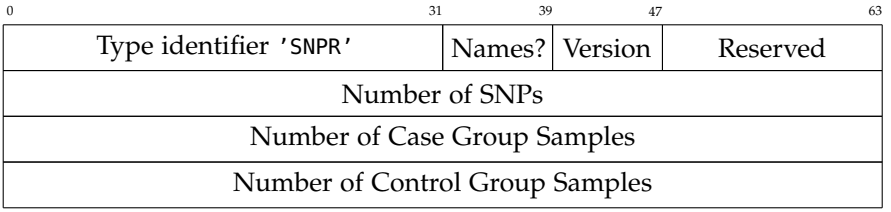

Figure 4.14. SNPDB file format header

The binary layout begins with a header. It is shown in Figure 4.14. The first field is a type identifier that allows the application to identify the given file as a SNP database dump. The second field is a byte flag indicating whether the dump contains a list of SNP names or not. This is especially important as not every source format supports SNP names, such as the BOOST layout. During development, the file layout has undergone several changes. The next field therefore contains the format version. When loading, the format version is checked such that incompatible versions are rejected and compatible but older versions force a slower compatibility loading mode. The following three fields hold the total numbers of SNPs, case

group samples and control group samples.

The actual data follow the header structure. First, allele counts are stored for every SNP. That is, during parsing, wild types, variant types and heterozygotes are counted per case/control group that are now stored within the file. These are required for counter reconstruction done by the GPUs as explained in Section 4.3.2. Then, the full genotype database generated during the parsing process is written to the file, in SNP-major ordering. Depending on the "Names?" field in the header, an optional list of SNP identification names is stored.

4.5.3 Memory Management

From the point where the database has been created and populated, data will be extracted and moved to FPGAs, to GPUs and back to the host system as shown in the overview. Repeatedly allocating, pinning, un-pinning and releasing memory buffers can be costly operation.

To reduce these costs, a buffer management mechanism has been introduced. During compile-time, i.e. by setting compiler macros, the buffer size and maximum number can be set. Then, an application thread may request one of these buffers for transmission.

The buffer handling system allocates a certain amount of buffers from the operating system. Through the use of a custom allocation function, each allocated buffer is automatically aligned at a 4 096 byte address boundary, i.e. it is aligned to the beginning of a memory page as described in Section 2.2.1, so the driver does not need to take care of odd buffer start addresses and partly filled pages in the beginning. After allocation, each buffer is pre-faulted. During this phase, the whole buffer is written to in a 4096-byte-stride, forcing the operating system to actually map every virtual memory page to an actual physical page. Finally, the memory is pinned, locking these pages into memory, preventing that a memory may be paged out to disk. These operations guarantee most efficient access to the managed buffers.

During buffer construction, a custom *deleter* is installed. When the reference count of a buffer reaches zero, i.e. it is not in use anymore, it is automatically taken back into the buffer manager. Internally, references to

these buffers are kept on a stack. Therefore, the last buffer that has been released by a thread will be the first buffer supplied to a requesting thread. It is expected that this will further improve efficiency by better use of memory caching, though these effects might become less visible with larger buffer sizes due to the relatively small and fixed-size CPU cache and sequential access patterns.

4.5.4 FPGA Initialization

After buffer allocation, the API module for FPGA communication is initialized and queried for the number of FPGAs that are present and ready. While the KC705 API for the Xilinx Kintex FPGA board only supports a single full-duplex data channel, the Alpha Data API supports up to four simplex data channels that can be freely configured. The final implementation for Virtex devices uses one channel to send SNP data. In the BOOST implementation, a single result return channel is sufficient while the Mutual Information implementation for 3-way interaction requires three channels to maximize data throughput.

For every FPGA, that is, for every sending channel, one thread is created. These threads extract ranges of SNPs from the databases and move them to their FPGAs. The actual range is determined by a number of factors, most importantly the number of FPGAs and GPUs available to the application. As the help outputs of both application show (see Appendices A.1 and A.2), the number of FPGAs and GPUs to use can be explicitly specified to allow ratio tuning when the performance capabilities of the FPGA and GPU types significantly differ. The devices in the system prototype have been chosen to match in performance figures.

Data Distribution

In the trivial setting that only one FPGA/GPU combination is available, all SNPs are sent to the FPGA for processing. When using more FPGAs, the number of pairs or triplets should be divided among all FPGAs equally and the corresponding SNP ranges determined and sent. For kth order interaction, k ranges of SNPs are selected per FPGA. In the example show

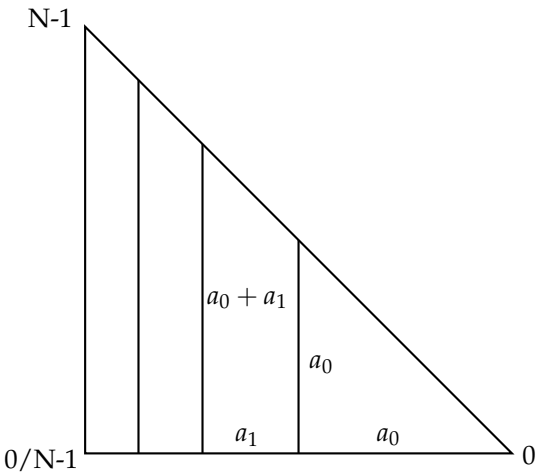

Figure 4.15. Distribution of SNPs in second-order interaction for four FPGAs

in Figure 4.15, it can be seen that for $k = 2$, each partition represents a trapezoid.

For 2-way SNP interaction, the distribution can be achieved as follows. Let

- $N > 0$ be the total number of SNPs to be analyzed for interactions
- $F > 0$ be the number of FPGAs to be used
- $0 \leqslant a_i < N \quad \text{with} \quad 0 < i \leqslant F$ be the number of SNPs for FPGA i.
- A^* be the total number of pairs to be calculated with $A = A^*/F$ pairs for each FPGA.

Then, the following recursive equations for a_i hold:

$$
\begin{aligned}
A &\leftarrow \frac{A^*}{F} && \text{with } A^* = \binom{N}{2} = \frac{N^2 - N}{2} \\
a_0 &= \sqrt{2A} \\
a_1 &= \sqrt{a_0 + 2A} - a_0 && \text{(Area of right trapezoid)}
\end{aligned}
\tag{4.5.1}
$$

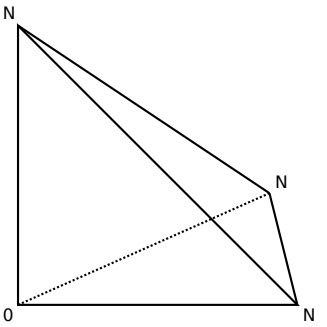

Figure 4.16. A right isosceles prism representing the space of 3-combinations of SNPs

$$= \sqrt{\sqrt{2A} + 2A} - \sqrt{2A}$$
$$a_{i+1} = \sqrt{a_i + 2A} - a_i \tag{4.5.2}$$

In this range calculations, the right triangle that represents the area of SNP pairings has been partitioned into $F = 4$ equally-sized right trapezoids. The rightmost trapezoid has a zero-length right ankle making it an isoscicle right triangle. As this is the only right ankle known beforehand, this represents the starting point for the recursive definition in Equation 4.5.2. In Equation 4.5.1, the general area definition for irregular trapezoids, $A = a_1 \cdot h_2 - (h_2 - h_1) \cdot a$, where a is the basis and h_1, h_2 the parallel ankles at the right angles, is used. In our case, h_1 is the previously obtained value for a, i.e. a_0. This can then be transformed to solve a_1 as A is already known and leads to Equation 4.5.1 and eventually the general form in a recursive definition. With higher orders of interaction, the area calculation of trapezoids can be exchanged for their respective representations in their dimension. A third order calculation would therefore be done analogous over right trapezoidal prisms instead of right planar trapezoids. Figure 4.16 shows such a right isosceles prism as representing the space of all possible 3-combinations of SNPs.

4.5.5 Data Movement

After the FPGAs have been initialized with SNP data, the data path to the FPGA is not required anymore. Instead, the threads now prepare for back-reading the results the FPGAs generate from the SNP streams. If necessary, each FPGA thread now spawns enough additional threads to be able to service all available reader channels in parallel to minimize idle times.

Listing 4.9. FPGA Reader Thread Operation (simplified)

```
while(resultsRemaining(i) > 0) {
  buffer& ctables = buffers.request();
  status = fpga[i].read(ctables);
  if(status == SUCCESS) {
    tablequeue.push(ctables);
  }
}
```

Each service thread will now read data from its respective channel as long as data is expected. The target buffers are requested from the buffer manager described in Section 4.5.3. When the reading call returns, two possible outcomes are expected. Firstly, the transaction may have been aborted. This may only happen if the user terminates the application by sending an interruption signal, i.e. by pressing `Ctrl+C` on the keyboard. Then, it is the applications reponsibility to leave the API in a clean and defined state. This involves tearing down DMA transactions. Additionally, another error condition has been implemented. In cases where the FPGA behaves unexpectedly and stops transmission right in the middle, a configurable time-out actively aborts a transaction as if the user had pressed `Ctrl+C`. The common case is of course the successful transfer of a data buffer. In that case, a reference to the received buffer is submitted to a special Multiple Producer/Multiple Consumer Queue (MPMC Queue). In Figure 4.12, this is depicted as "Table Buffer Queue" and serves as a central storage for distribution of work units to a number of GPUs. A simplified draft is displayed in Listing 4.9.

Whether a channel is expected to deliver data is not trivial. Unfortunately, there is no efficient way for the FPGA to let the host system know

that the transfer has been completed. One method could be to place a flag into the buffer. However, due to the nature of the data stream on the FPGA, the condition can only be detected when the stream actually ends. Therefore, the flag would have to be after the last valid data word in a buffer. This would require the host to scan through the whole buffer to find this flag. Another way of detection could be via side-channel data in the PCI Express configuration interface (see 2.2.2 for details), a very slow way of communication. Fortunately, processing data on FPGA is always deterministic and predictable, if done right. Hence, knowing the number of processing elements, SNPs, processing element chains and processing elements per chain, the exact outcome per transmission channel can reliably be calculated. The application pre-calculates these numbers for every transmission channel so servicing threads can efficiently query for the number of remaining results without performing costly non-DMA I/O operations or memory scanning. As soon as a service thread detects that its work is done, it joins with the main application thread, i.e. it terminates and notifies the application of the status change.

Listing 4.10. GPU servicing thread operation (simplified)

```
while(moreResultsComing == true) {
  buffer& tables = tablequeue.pop(); // wait if necessary
  buffer& results = buffers.request();

  status = gpu[i].runKernel(tables, results);
  if(status == SUCCESS) {
    resultqueue.push(results);
  }
  // table buffer implicitly released
}
```

Right after the creation of FPGA threads, similarly, servicing threads are spawned to process table buffers from the aforementioned MPMC queue on the GPUs. For every GPU available, one thread is responsible to take a table buffer from the table buffer queue, move tables to the GPU. As the GPU

requires another memory block to store the resulting scores, another buffer is requested and provided as is shown in Listing 4.10. It is worth noting that the table buffer that has been moved out of the table queue is now the sole reference to that memory block. This buffer reference leaves the lexical scope of the block defined by the `while` after it has been processed by the GPU. The reference count of the buffer becomes zero and hence triggers the *deleter* function described earlier in Sect. 4.5.3. The unused buffer, instead of being released to the operating system, is being put back on the buffer stack inside the buffer management operation. It becomes available again for requesting by other functions and threads.

As soon as the GPU kernel has finished processing the table buffer, the result buffer is submitted to another MPMC queue, depicted as "Result Buffer Queue" for the final filtering and post-processing.

4.5.6 Result Collection and Processing

The final step is the post-processing of the results produced by the GPUs. Now that the CPU is required to process the results into the final output format, the buffer contents are defined as an array of structures as shown in Figure 4.17. For every table entry in the original table buffer, one score structure is generated. Both single precision (32 bits) and double precision (64 bits) as defined in IEEE 754 are supported.

The structure has been carefully crafted so that each field can be accessed by the CPU in a single memory access. Generally, a field has to be placed at a memory address that is dividable by the size of the field itself. In this case, all 16 bit fields have been aligned to 16 bit addresses and the score field is aligned to a 64 bit address (and thus also to a 32 bit address). Without following this alignment rule, the CPU would have to read two adjacent words, mask the beginning and end of the first and second word, respectively, and shift it into the right position so arithmetic operations can correctly be performed.

Even though the structure is accessible in an efficient way, a single buffer of one gigabyte on a second-order interaction method contains more than 67 million test results and a new buffer arrives every few hundred milliseconds. To avoid this buffer processing being a bottleneck, a configurable

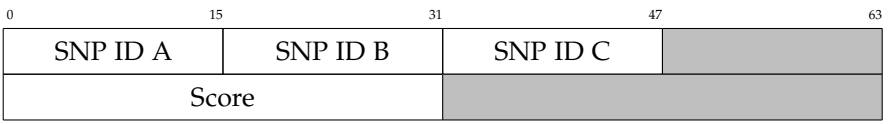

Figure 4.17. Third order test result structure with single precision score

number of threads are created to process buffers in parallel. As opposed to the I/O threads that work towards FPGAs and GPUs, these threads are actually spawned to increase processing throughput on the CPU instead of bridging transmission gaps. Although the number of threads may be specified by the user, it is also possible to let the application decide.

Adaptive Threading

Before actually creating threads in the adaptive method, the system topology is analyzed. The operating system is queried for the number of physical CPUs, physical cores per processor and the order of symmetric multiprocessing units ("hardware threads", see Section 2.1.1 for a deeper explanation). With these numbers known, an efficient threading policy can be set up.

By default, two threads are created to process results. If a backlog develops, i.e. buffers queue up for result processing, another thread is spawned until the maximum number of hardware-supported threads or a user-configurable limit is reached. Let s be the number of physical CPUs, c be the number of physical cores per processor and t the number of supported hardware threads per physical core. Then, the first $s \cdot c$ threads that are created are tied to the available cores, with at most one worker thread per core. This is done by setting the scheduler affinity in the operating system kernel. Although it is not guaranteed that the threads actually run on the desired cores, the scheduler takes greater care than without this specification. By using scheduling affinities, two consequences can be exploited:

- The total computational power of all hardware threads sharing a single physical core is rarely larger than 130 % of a single thread running on that core [ECX+11]. By first distributing threads over cores instead of

hardware threads, the first c threads are able to utilize 100 % each. Every further thread only increases the load by up to 30 %.

▷ Threads with a set affinity have a much lower chance to be re-scheduled for another processor/core/hardware thread. This reduces overhead from context switching and hence, cache invalidation.

Additionally, when worker threads start becoming idle because they process buffers faster than they arrive, they are not terminated. Instead, they are put into sleep mode and therefore skipped from scheduling. This way, they can be reactivated faster than newly creating a thread without consuming too much system resources. This is also done by setting scheduling affinities and priorities.

Result Processing

Every result worker now processes the structures as shown in Figure 4.17 one-by-one. In second-order interaction search, the user supplies a threshold by which the results are filtered. Every structure is then compared against this threshold and those that feature a higher score than the threshold are written to the result output file. Due to the nature of multiple threads writing to the same data structure, the results will be delivered unordered.

In third-order interaction, an additional mode is supported. The user may instead request only the best results to be reported, with a configurable number of results. This is especially useful when evaluating new test methods as a clear threshold can not be given initially. Also, new methods may be implemented where the altitude of scores vary among data sets. Selecting the best results from particularly large data sets can be a computational and algorithmical problem. This is evaluated in the following section.

List of Best Results

For both applications, it is desirable to not only filter values based on a certain, fixed threshold but also acquire a list of the m best results where m is much smaller than the total number of results generated r, i.e. $m \ll r$.

The choice of data structures and algorithms has to satisfy a number of requirements.

First of all, it is clear that r results cannot be kept in memory. Even if every result of a 3-SNP-interaction on 500 000 SNPs would require a single byte, almost 21 PiB of main memory would be required. This restricts us to view the generation of results as streaming where only one-way sequential access is possible. A naïve implementation could be an array of fixed size m which is kept sorted at all times. When a new result is taken from the stream, it has to be compared to the smallest item in the array. Random access in arrays can be done in constant time, so neither reading the smallest element nor replacing it would create a bottleneck. Instead of re-sorting the list which could easily take $\mathcal{O}(m \cdot \log m)$ using QuickSort or MergeSort, a binary search can be done to find the correct index for the new item in $\mathcal{O}(\log m)$. Being an array, a new item cannot be easily moved in between two adjacent items. Therefore, all items between the determined index and the end of the array would have to be moved one place to the right. Therefore, the runtime complexity is $\mathcal{O}(m + \log m)$ for each result. Furthermore, every insertion would also require $\mathcal{O}(m + \log m)$ accesses to the memory. The memory bus is already quite busy by the concurrent DMA transfers, so a lower memory access complexity is desirable.

Aside from a simple array, a priority queue is a natural choice for this kind of work. Implemented as a max-heap in a binary tree, it can be stored in a regular array in consecutive memory locations. The largest element can always be accessed in $\mathcal{O}(1)$ being the root in the tree as the heap property ensures that a node is always larger than its children. Insertion and deletion and extraction of the minimum value can be done in $\mathcal{O}(\log m)$ [CLR+07, Chap. 6]. This also applies to the memory accesses. Now, for every result, we have to extract the minimum and compare it to the result in question to decide whether it makes sense to insert it into the max-heap. For such large numbers even logarithmic runtime w.r.t. m is problematic.

The solution and actual implementation is a *Min-Max Fine Binary Heap*. Here, a min-heap and a max-heap are combined in a single heap [ASS+86]. For this to work, the heap property is extended to define that a max-heap node only has min-heap children and min-heap node only has max-heap children. When viewed in level-order, every second level contains min-heap

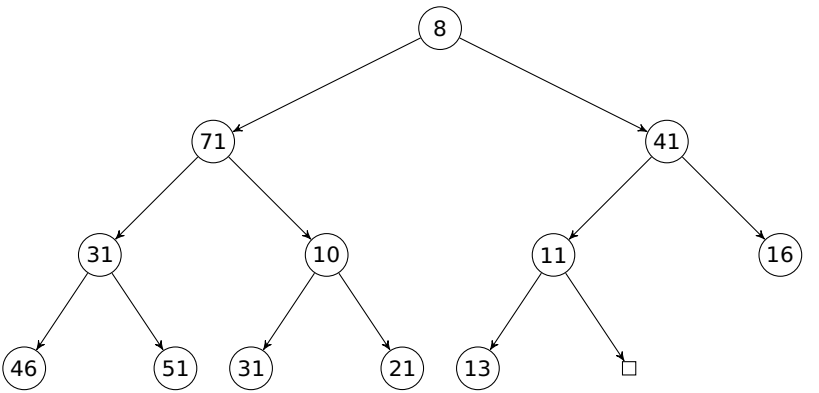

Figure 4.18. Min-max heap

nodes while every other level contains max-heap nodes. An example is shown in Fig. 4.18. When the root node is a min-heap node, it contains the minimum element in the whole heap and one of its two children contains the maximum element. Therefore, accessing the minimum or maximum is in $\mathcal{O}(1)$ while maintaining a "regular" heap's runtime footprint. A further optimization has been proposed by Carlsson et. al. in [CCM94], where for each data item, an additional bit is stored that indicates whether the left or right child is the smaller or larger one, depending on the node type. Although this does not change the complexity class, only one node needs to be loaded from memory instead of two, and finding of the minimum or maximum becomes obsolete as it is already known.

Additionally, the list may only contain up to k elements. This is ensured as shown in Algorithm 2. Before an attempt to actually insert an element into a full list, it is first checked whether the element is larger than the current top list's minimum. If it was not, it would become the $(k+1)$th element in a list of the k largest elements and thus cut away to maintain the correct list size. Therefore, FindMin is called for every candidate whereas DeleteMin and Insert are only called for those elements that will actually become part of the list. As FindMin is constant in time complexity, as opposed to deletion and insertion, a large amount of runtime can be saved as only few elements

would actually be inserted. Unfortunately, the score distribution over all generated contingency tables is unknown so no authoritative prediction can be made about the actual efficiency of this measure, though actual measurements clearly show its usefulness (see Section 5.2.1). However, this initial filtering against the minimum of the heap is implemented using AVX SIMD instructions to compare eight double-precision floating-point scores against the minimum in a single instruction. The `VCMPESTRM` AVX2 instruction used here stores a mask that allows identification of all values in the eight-tuple that are larger than the heap's minimum — and can therefore be inserted. An introduction to CPU SIMD instructions is given in Section 2.1.1.

Algorithm 2: Min-Max Fine Binary Heap Insertion

Data: a heap with i elements and a capacity of m elements
Input: an item e to be inserted
Result: a heap with at most m elements

```
1 if i = m then
2 |   min ← FindMin(H);
3 |   if e > min then
4 |   |   DeleteMin(H);
5 Insert(H, e);
```

In an intuitive (and naïve) implementation would use pointers to store the children node addresses in their parents. While this certainly works, and is the de facto standard for languages in the functional programming domain, it is rather inefficient in procedure-based programming. Instead, the properties of a binary tree have been exploited to create an array-based implementation. Given the heap shown in Figure 4.18, the array representation is as depicted in Figure 4.19. The root node's value is stored in the first cell and its two children in the following two nodes. The left-hand grandchildren are stored in cell four and five while the right-hand grandchildren are stored in six and seven. This construction rule can be generalized:

Let $I_{(i)}$ be the i-th node in the tree, counted from top to bottom and left

8	71	41	31	10	11	16	46

Figure 4.19. Array representation of min-max heap

to right, with n being the total number of nodes and $i \in \mathbb{N}_0$ with $i < n - 1$:

$$\begin{aligned} I_{(0)} &= 0 \\ \textsc{Parent}(i) &= (i-1)/2 \\ \textsc{LeftChild}(i) &= 2i+1 \\ \textsc{RightChild}(i) &= 2i+2 \end{aligned}$$

This rule yields a node storage in level-order and has several advantages over the pointer-based solution:

- ▻ Each node only requires storage for the datum itself. Assuming a pointer size of 8 bytes, the storage overhead per node is reduced from 16 bytes (two child pointers) to zero.
- ▻ With all level nodes being stored in consecutive ordering, it is very cache friendly as nodes in the direct neighborhood are often accessed afterwards (good cache locality).
- ▻ Instead of one allocation per node, all nodes are stored in a single, consecutive block of memory in successive addresses. As the size of the heap is constant and known in advance, the whole storage can be allocated in a single system call, making the insertion of new nodes very efficient. Due to the heap insertion mechanism, the binary tree is always optimally balanced, hence, the array is also compact with no empty cell between nodes.

Result Output

Finally, the results have to be written to a file to be of any use, whether they originate from a top list or from a threshold filter. In the whole course from the database, through the FPGAs and GPUs and eventually the result processors, SNPs are only identified by their relative number in the database.

For the researcher, these numbers do not have any meaningful information. Therefore, instead of SNP IDs, their names are retrieved from a large look-up table. Due to the nature of look-up tables, this access can be done in $\mathcal{O}(1)$ with respect to the runtime.

The actual access to the file system is serialized among the result worker threads with mutual exclusion primitives to prevent data corruption. Furthermore, the results are not written to the file but stored in an intermediate buffer. Otherwise, for every score, a few bytes would be written through the `read` system call which is then diverted to the respective file system driver. In addition to the high overhead of repeatedly issuing a system call for a small amount of data, many file systems store data in blocks of a certain size. For every write modification to a block that is less than the block size, a read-modify-write cycle is performed. On the other hand, if the data is buffered to write multiples of the file system's block size, no read or modify operations would have to be performed. Unfortunately, Linux supports a broad variety of file systems but does not provide a standardized way of querying for the block size. Usually though, they are specified as rather small powers of two, such as 512 Bytes, 4 KiB or up to 512 KiB. Choosing a buffer size of a few megabytes certainly exceeds all common block sizes and is still a multiple thereof. This leaves the `ResultWriter` module with a relatively small system call overhead and efficient file system I/O methods.

The output format is borrowed from the original BOOST publication in [WYY+10]. An example of the output format for third order interaction is shown in Listing 4.11. Two or three SNP names are displayed in whitespace-separated columns, depending on the order of the interaction, and the last column contains the score that has been assigned by the statistical method in use.

Listing 4.11. Third order interaction result output example

```
SNP_A      SNP_B      SNP_C      SCORE
rs123456   rs331234   rs443111   23.441345
rs001337   rs313373   rs654321   42.55781
...
```

Chapter 5

Evaluation

Parts of this chapter have been previously published in [GWK+15a; GKW+15; KGW+14; WKG+14; KWG+15; GWK+15b; GSK+14; WKH+17].

5.1 Performance Metrics

Performance figures of computing systems are typically represented with a single value, the number of floating-point operations that can be processed per second (FLOPS). Traditionally, it is measured using the linear equation solving package LINPACK [DLP02]. This obviously focuses on multiply/add operations but also includes memory speed, system bus bandwidth, as well as operating system and networking overhead.

In the bi-anually published TOP500.org [T500] supercomputer ranking, the participating systems are constructed from one or more of the following features:

- NVIDIA graphics processors, such as Tesla K20X or
- Intel Xeon Phi manycore accelerators
- High-performance vector processors (i.e. Cray XC40)
- Intel CPUs and IBM Power processors

By the time of this writing, not a single system contains FPGA-based computing devices. As this chapter will show, FPGAs are not inferior regarding raw performance and, in fact, outperform any other architecture when it comes to power consumption. The reason they are not listed becomes obvious when the way FPGAs are used is taken into account. Their configuration is usually precisely engineered to perform a very specific computation and it will not be capable of doing anything else until reprogrammed. It *could* be designed to solve matrices and linear equations but an FPGA cannot be seen as a general purpose computing device and the results would not be representative with respect to the actual power of these devices.

In fact, this argument may also hold for any other architecture, which is a major point of critique when it comes to performance estimations. In graphics processing and 3D rendering in particular, multiplying matrices is a larger part of the computational burden. Hence, graphics processing units are geared towards efficient matrix multiplication, for example by implementing a highly parallel engine that can perform floating-point additions and multiplications very efficiently. Therefore, graphics processing units perform extraordinarily well in the TOP500 supercomputer list. Given these results, the number of floating-point operations per second is not a good performance estimation but merely a measure for the suitability for certain operations - such as the solving of linear equations.

5.1.1 Measures and their Inaccuracies

When evaluating and comparing the performance of a single method across various platforms and architectures, it is therefore reasonable to use a measure that indeed is a good performance indicator *for a particular purpose*. In the case of compound architectures as the one proposed in this work, the performance will be measured by the outcome of its subcomponents and communication speeds. The overall performance will be given in means of elapsed time with respect to a reference data set. While runtimes show very little jitter on GPUs and FPGAs, the host system has more duties than just running the respective application. Instead, user logins are handled, mechanical hard disks spin up and spin down, services respond to network

requests, CPU caches are invalidated by other applications, and so on. To accommodate these disruptions and sources of jitter, the CPU time that has actually been allocated to the process is also given. This allows an extrapolation to 100 % utilization and therefore generating comparable results without dependencies on actual system load.

When it comes to generation of contingency tables, only one measure could be representative for the FPGA's performance: the actual number of tables that can be delivered by the FPGA per second. Assuming perfect determinism, an operation's can be precisely predicted. As FPGAs are not "programmed" but merely "described," the programmer can always guarantee the availability of a certain datum at a certain place within the accuracy of a single clock cycle. Hence, it is easy to predict the number of tables per second. However, the FPGA also reacts to events from outside. Anytime, the memory controller may decide to perform DRAM refresh operations, causing a delay when the FPGA requests data from the memory module. Other sources of timing uncertainty is the host platform itself. The host system moves data to or from the FPGA through various controllers and built-in devices where each adds to the uncertainty. The host application itself may not be able to provide a buffer on time because another application is accessing this very memory region. This behavior cannot be factored out by introducing utilization factors or bus load. Instead, the FPGA fills its small on-board interface buffers and then proceeds to stall the whole processing pipeline until the buffer is finally available. It then takes time to propagate the news through the system in order to resume operation.

While the theoretical throughput of the FPGA design is certainly an indicator of its power, the previously mentioned effects significantly reduce the net data rate that can actually be delivered. In order to assess the transmission efficiency and the FPGA stalling-related delays, additional data is collected.

- On the FPGA, the number of cycles is counted where
  - data is ready to send but the DMA channels are not
  - DMA channels are ready to accept but no data is available (i.e. due to stalling/resumption delays)
  - both the DMA channel and the result channel are transmitting data

- ▻ the generation pipeline is stalled

▻ On the host,

- ▻ the time it takes to read a full buffer
- ▻ the time it takes to supply a new buffer once the previous buffer has been filled
- ▻ the time the application had to wait because no buffer was available that could be supplied

With these numbers, a fairly accurate estimation regarding the table creation speed and communication system efficiency can be given. When it comes to processing the table buffer on the GPU, other measures have proven to be useful. Unfortunately, the NVIDIA CUDA API does only provide a limited set of profiling instrumentation, so some measures are derived from others.

▻ The net transmission bandwidth from and to the GPU

▻ The device utilization measured in active blocks vs. idle blocks

▻ The warp divergence, derived from statistics gathered through side-channel data. This includes coverage information, i.e. from the branches taken in the BOOST filter chain between KSASA, KSA and log-linear methods

Eventually, the host system does a final filtering process where records are written to the file system if they satisfy one of the user-selectable contitions:

▻ The interaction score is larger than a user-supplied threshold

▻ The interaction score is part of the highest k records throughout the whole data set.

Clearly, the runtime of the threshold-based filter is directly dependent on the magnitude of the threshold and the magnitudes of the interaction scores. If a threshold is met or exceeded, the record in question is written

to the (buffered) result output file. The runtime hence not only depends on the decision but also on the file system performance and the underlying storage system. For normalization purposes, the time spend writing to disk is therefore also recorded.

The second filter is based on a fixed-capacity MinMaxHeap implementation where the best k records with respect to the interaction score *so far* are collected (see Section 4.5.6) Other than the threshold-based version with a runtime of $\mathcal{O}(1)$, the heap implementation can only asymptotically process data in constant time but it's Insert method is still in $\mathcal{O}(\log k)$ where k is the maximum heap capacity. To assess the actual efficiency of the heap, two key figures are additionally collected: the number of scores that were not inserted because they were smaller than the current minimum (and the capacity has been reached) and the number of scores that have been accepted for insertion because they were larger the minimal heap element (and the capacity has been reached). Obviously, the number of insertions while the capacity has not been reached yet, is k and does not need to be collected.

Finally, the overall performance is given in terms of runtimes for a reference data set and processed pairs or triplets per second and a performance bias can be given by comparing the average net production rate of the FPGA against the average net processing rate of the GPU. For comparisons with other implementations, runtimes are taken from their respective publications and are interpolated with the expectation that their runtime scales linearly with the number of tables/pairs/triplets (i.e. linear in the number of samples and exponential with the interaction order in the number of SNPs).

5.2 Architecture

In this chapter, in-depth analyses are done on the overall architecture as well as within and between its components. In the second part, the prototype implementation is compared against other hybrid computing platforms, implementations and methods regarding system performance and energy consumption.

Although the FPGAs in use do have a large amount of resources, they are fully exhausted by the application. The resource-wise complexity of an array node is directly anti-proportional to the number of nodes that fit on the device, and, in turn, the generation rates. Therefore, configurations have been created to support specific maximum data set sizes. An overview of these different configurations is given in the following sections.

For comparability, runtimes and resource usage is measured using the maximum supported data set size as well as a *reference data set* that can be processed by all available configurations and consists of exactly 3 000 control samples and 2 000 case samples. The number of SNPs does not affect the table generation and processing rates but merely the total runtime defined by the number of contingency tables to create and process. Hence, this number is not part of the reference and is always given explicitly where applicable.

5.2.1 Throughput

The contingency table stream passes several components and interfaces and is later turned into a result score stream. To assess the potential and actual performance, each of these components and interfaces is examined.

Table Creation on the FPGA

For assessing the rate of contingency table creation, the number of nodes in the systolic array plays a major role along with the frequency the nodes are clocked with and the widths of the genotype lanes. Once the array is under full load, i.e. SNP buffers of all nodes have been filled, each node generates one contingency table per SNP that is streamed along. Assuming a SNP length of 5000 genotypes, with eight genotypes per cycle, a contingency table is created in 5000/8=625 clock cycles. With a clock frequency of 200 MHz, 320 000 tables are created per node and second. In Table 5.1, the generation speeds of all FPGA configurations are shown, along with a short summary. Data rates are given with respect to the reference data set and sample-wise the largest supported data set indicated by "(max)."

Due to the non-trivial ordering of contingency tables in the third-order

Table 5.1. FPGA configuration characteristics

Configuration	2-way 16k	2-way 64k	3-way 8k
Node frequency	200 MHz	200 MHz	200 MHz
Genotype supply	8/s	8/s	8/s
Maximum group size	8 192	32 768	4 096
Node count	1 000	288	180
Array partitions	4	2	3
Rates per node:			
Tables/s (reference)	320k	320k	320k
Tables/s (max)	97.7k	24.4k	195k
Data rate (reference)	4.88 MiB/s	4.88 MiB/s	20.71 MiB/s
Data rate (max)	1.49 MiB/s	0.37 MiB/s	12.64 MiB/s
Total rates:			
Tables/s (reference)	320M	92M	58M
Tables/s (max)	98M	7.0M	35M
Data rate (reference)	4 883 MiB/s	1 406 MiB/s	3 728 MiB/s
Data rate (max)	1 490 MiB/s	107 MiB/s	2 276 MiB/s

approach, tables are accompanied by identifiers as discussed in Sect. 4.2.4 and Figure 4.6. It consists of three fields of 21 bits each and adds an overhead of $3 \cdot 21 = 63$ bit to each table. These fields are added to the data stream in the transmission unit and result in gross data rates of 4 161 MiB/s (reference) and 2 276 MiB/s (max) with a total overhead of 433 MiB/s and 264 MiB/s, respectively.

FPGA/Host Link

Figure 5.1 shows gross data rates that have been measured directly at the PCI Express link level, separated into the DMA channels that have been used in the third-order method.

It can be seen that the transmission speed breaks down at regular intervals. These intervals tend to become smaller as the application proceeds reaching their minimum at the very end. This can be explained by the way

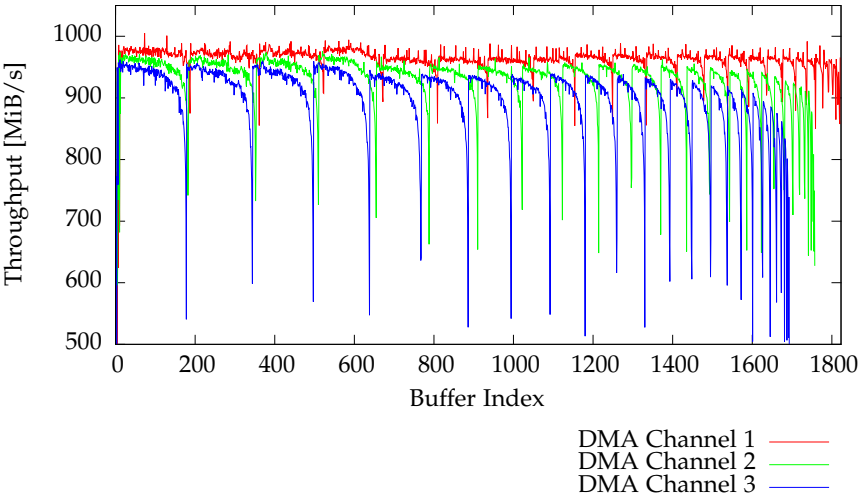

Figure 5.1. Channel-wise FPGA/host link speeds (third order)

the systolic array processes data. When throughput is at a high level (between negative spikes), both SNP buffers are filled and each node generates one table every SNP. Once the full set of SNPs has been streamed, streaming restarts but nodes have to replace their SNP buffer contents before new tables can be emitted. Every new streaming sequence, the SNP buffer replacement frequency rises as the combination space becomes smaller (see Sect. 4.2.3). Hence, the intervals become smaller as the SNP buffers are replaced more frequently culminating at the last few SNPs where only very few tables are generated per streaming sequence. Data throughput is sampled once every 256 MiB. This results in the "between" throughput apparently becoming lower. However, this effect is likely visible because of the sequencing frequency rising over the throughput sampling frequency. Hence, spikes are averaged and lower the measured throughput. Regarding the data rates given in the configuration overview in Table 5.1 and the overall construction, node-wise lower generation speeds are implausible.

Although the measured transmission speeds already suggest it, Fig-

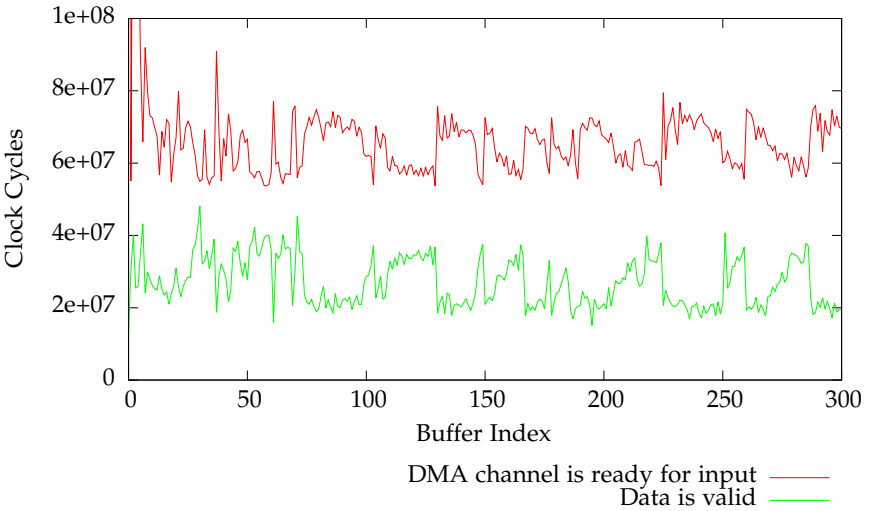

Figure 5.2. FPGA/Host link saturation

ure 5.2 shows another important statistic. On the FPGA, the clock cycles have been counted where each DMA channel signals that it is able to accept a new data word ("DMA input ready") and the number of clock cycles where data is available for sending. These counters are reset when a new buffer commences and are moved to the FPGA's configuration space to be read by the host as side-channel data. Measurements show that the average buffer transmission time on the three-way method is 317 ms. Therefore, the average buffer time in clock cycles assuming 250 MHz interface clock, is 80×10^6. For simplicity, only statistics for the first DMA channel are shown but the other two channels perform similarly.

Both graphs appear to be mirrors of each other. When both the DMA channel is *ready* for input and the table output displays valid data, transmission is started and the DMA channel changes its state to *not ready*. Hence, the DMA channel "ready count" graph essentially drops by the "data valid" counter value during transmission. Still, the DMA channel is ready to a large portion of the buffer cycle. Although no "transmission capacity" can

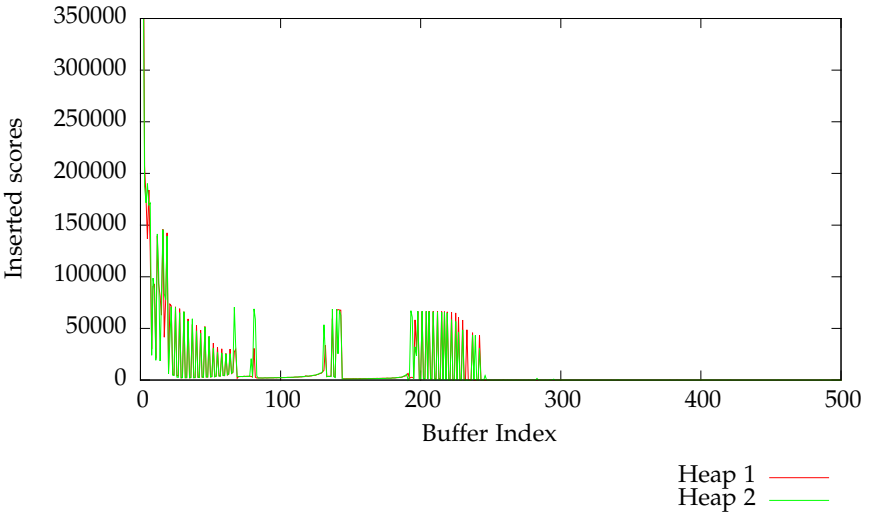

Figure 5.3. Min-max heap result insertions

be deduced from these data as a single transmission may lead to several *not valid* emissions, it does allow to conclude that the transmission between the host system and the FPGA is not fully saturated.

Result Processing

The host application can, if desired by the user, output only the highest ranked results among the whole data set. The chosen data structure, a MinMaxHeap, has been modified to restrict the maximum number of stored elements and is therefore independent of the data set size (see Section 4.5.6). Instead, the required storage is bounded by the number of items that the user desires to keep. Although insertion into the MinMaxHeap is fast, even $\mathcal{O}(\log k)$ becomes significant when a very large number of items are queued for insertion. By extending the heap with a fixed size property, the runtime can be reduced to asymptotically constant time by simply comparing the to-be-inserted item to the lowest recorded score in the heap and only

performing the actual insertion if it is larger than the current minimum. To prevent lock contention in busy situations, every result processor thread keeps a thread-local heap. When done, all heaps are merged into a single heap which is then written to disk. Figure 5.3 depicts the number of heap insertions for the first two heaps. The heap capacity used to generate data for this diagram is 10^6. It can be seen, that the very first buffers (with a table volume of approx. 8×10^6 tables) fill the heaps to a large extent while the actual insertion counts become less and less. Around the 80th buffer, only 0.00625 % of all contained tables are submitted to insertion. The anomalies around buffers 140 and 210 can be explained by single-SNP effects that raise the signal strength of all combinations where that SNP participates in. After buffer 250, no more than 20 tables (or 0.00025 %) per 8×10^6 tables are inserted, reducing the measured CPU load to less than 10 % on the result processing threads. The full profiling run on the reference data set yields approx. 5 300 buffers whereof the first 500 buffers are shown in the diagram.

Joint Application Performance

From Figures 5.4 and 5.5, it can be seen that FPGAs and GPUs, at least in the configurations used in this application, perform equally well on both tasks. Especially in the third-order method, the average FPGA generation speed matches the average GPU processing speed within a margin of 130 MiB/s. In Figure 5.5 though, the variations in the GPU application speed are much larger. This can be explained through varying degrees of warp divergence. GPU-based threads are organized in *warps* where each thread in the warp is supplied with the same instruction stream. More specifically, this disallows threads in a warp to diverge in the branches taken, for example, in a conditional clause or a loop. If threads in a warp require a different path to be taken, all other threads are halted. Hence, all diverged branches are executed sequentially. In the case of the second-order filter chain, most results are discarded by the KSASA pre-filter. In data sets that contain SNPs that have a strong influence by themselves, all combinations that include that SNP may present an increased score. In turn, this might cause larger numbers of high-ranked consecutive combinations where more pairs happen to proceed further into the filter chain into the

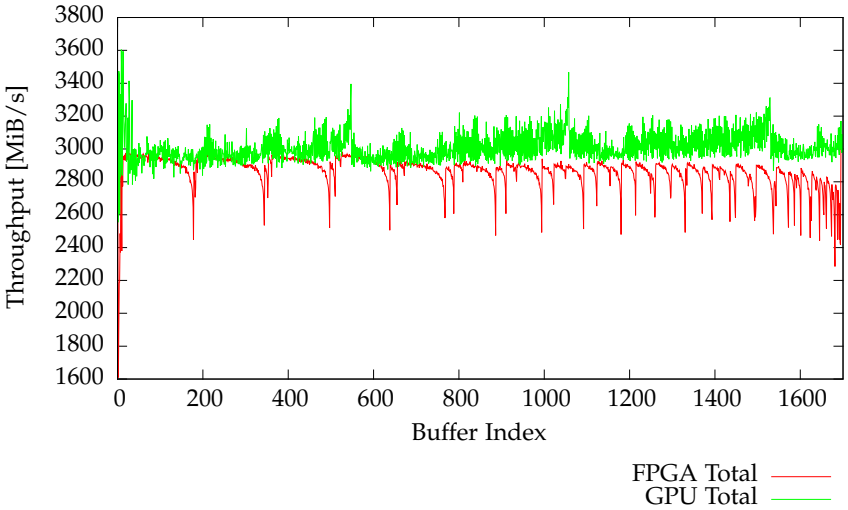

Figure 5.4. FPGA generation speeds vs. GPU processing speed (third order)

Table 5.2. FPGA table generation and GPU processing speed comparison

	Second-Order BOOST	**Third Order** Mutual Information
FPGA average	2 749 MiB/s	2 799 MiB/s
GPU average	2 502 MiB/s	2 929 MiB/s

computationally more expensive tests. Due to the lack of branching in the third-order method, warp divergence is essentially not happening while the second-order method performs operation where the runtime is heavily data dependant, causing throughput figures with higher variance.

In the BOOST implementation, the throughput averages of FPGAs and GPUs differ slightly more than in the Mutual Information implementation. Here, the average GPU processing speed is approximately 247 MiB/s lower than the average FPGA generation speed as shown in Table 5.2.

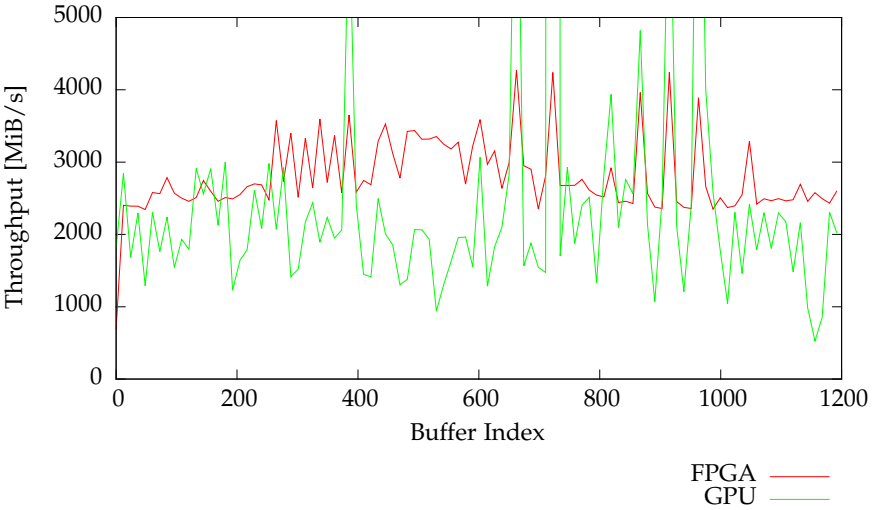

Figure 5.5. FPGA generation speeds vs. GPU processing speed (second order)

These results show that the computational power with respect to their specific problems, GPUs and FPGAs are on par and can be combined to develop a high-efficiency hybrid computer system. It is clear, that the FPGAs involved process a workload that is close to their optimal problem structure, counting and pipelining being two of the most suited tasks when it comes to reconfigurable logic. The same conclusion can be derived for GPUs. These devices contain special hardware modules to accelerate operations on floating-point numbers and allow multiprocessing in a fixed but tunable and large computation grid, combined with efficient on-chip scheduling algorithms. Both parts combined with a potent host system to control the data flow yields a system with a high performance-per-watt ratio and low amount of idle resources. The following section displays and discusses the proposed hybrid computer under the aspect of energy efficiency.

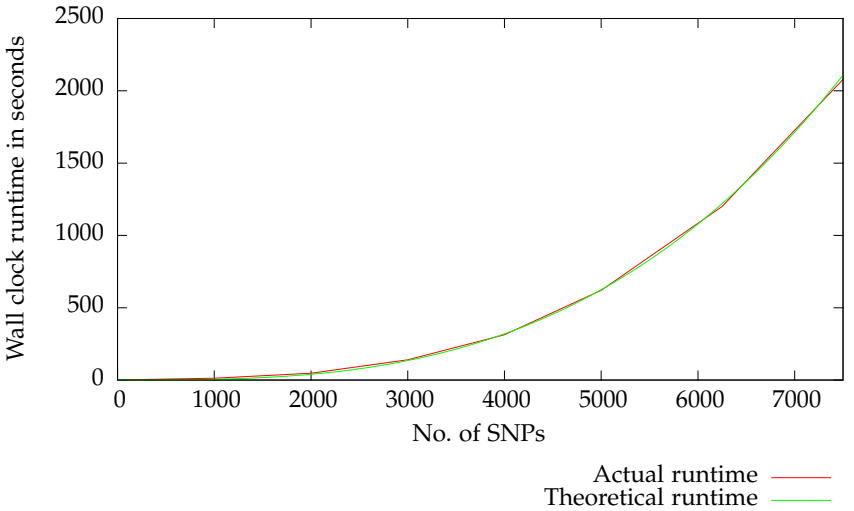

Figure 5.6. Theoretical runtime against actual runtime in third-order interaction

5.2.2 Runtimes

Figure 5.6 shows how the runtimes of third order interaction search rises with the number of SNPs. For reference, a function plot for the theoretical runtime as computed by contingency table transmission rates has been added. It can be seen that they only differ in a few seconds in runtime. The difference can be explained by input data loading times as well as result processing and writing. Furthermore, the parallel loading scheme as well as the MinMaxHeap-based approach on result selection reduces the runtime overhead to negligible values. The graph data has been collected from a data set consisting of 5 000 samples and SNPs in the range from 500 to 7 500 SNPs in steps of 500 SNPs. The result processing was set to gather the one million best results in a MinMaxHeap, using two threads for loading and eight threads for filtering. All threads have been bound to a specific CPU core to limit context switching overhead and cache invalidation.

5.2.3 Energy Consumption

In the era of "green computing," bare computational power is still the most important aspect of a computing platform but rising energy consumption is also becoming a problem. Especially in the domain of bioinformatics and personal medicine, algorithms and applications cannot always be processed at academical or commercial data centers, mainly due to restrictions when it comes to sensitive patient data. Therefore, hospitals, labs and research groups are often required to use local installations that cannot meet the requirements of computing clusters, in terms of energy supply and housing.

Additionally, energy consumption is tightly coupled to thermal discharge. As described earlier, thermal discharge has already significantly influenced he evolution of CPUs (see Sect. 2.1.1) and this trend is likely to continue, making it harder to further increase the "performance density." The development of heterogeneous systems made of efficient and specialized components does not only offer large amounts of computational power but also increases the performance-per-watt ratio, leading to more sustainable developments.

As isolated energy consumption of computer peripherals cannot be easily measured, the tool set provided by the chip manufacturer Xilinx Inc. is able to estimate the worst-case power consumption of the device based on the compiled VHDL output and device feature mapping, using the Xilinx Power Estimator (XPE). As there is no scheduler involved in FPGA operation, it always operates under full load with only slight changes related to the PCI Express transmission interface. The only other (non-negligible) power consumer on the FPGA boards are the DDR3 memory modules. The idle and load consumption have been taken from their manufacturer's data sheets. The results are displayed in Table 5.3.

The results clearly show that the performance presented earlier can be delivered by regular desktop computers equipped with common-off-the-shelf graphics hardware, consumer-grade power supply units and FPGA accelerators. With a peak power of 530 Watts, no high-end cooling system is required, instead the hybrid computer could be placed under any clinician's or lab technician's desk.

Table 5.3. Power consumption of hybrid computer components

Device	Power		Estimated by
	idle	load	
Host System	124 W	242 W	Energy meter
NVIDIA GeForce 780 Ti	10 W	259 W	System monitor
Xilinx KC705 Board			
Kintex 325T (2-way)	8.0 W	8.0 W	XPE
Kintex 325T (3-way)	11.2 W	11.2 W	XPE
DDR3 memory	0.5 W	2.3 W	[Mic10]
Alpha Data Virtex Board			
Virtex 690T (2-way)	17.9 W	17.9 W	XPE
Virtex 690T (3-way)	17.3 W	17.3 W	XPE
DDR3 memory	0.9 W	5.1 W	[Mic11]
Total (Virtex, 3-way)	152.2 W	523.4 W	
Total (Virtex, 2-way)	142.5 W	511.3 W	

5.3 Competing Systems

To estimate how well the hybrid computer prototype performs, it is compared against several other systems and methods. First, conventional CPU-based approaches are analyzed. When evaluating large workloads, whole clusters and cloud services are often used. Also, a limited number of hybrid computers are shown, although none of them exploits the benefits of *two* acceleration platforms but only use either FPGAs or GPUs.

It is usually impossible to find a data set where all other publications agree upon. Hence, runtimes are extrapolated, assuming that the runtime scales linearly in the number tests to be performed. Realistically, all computations incur enough overhead that no method truly scales linearly. Lacking other methods of comparison, this might be the only fair and representative solution. Runtimes that have been extrapolated are marked with an asterisk (*). Furthermore, none of the other methods are run locally as other approaches may use hardware that is not available for individual measurement. Instead, runtimes are solely taken from their respective publications

and scaled to key data set sizes.

Energy consumption of competing systems is rarely publicized. To provide energy measures, hardware details from publications are looked up in data sheet archives. Then, the Thermal Design Power (TDP) is taken to give a rough estimate of the power under load, assuming the programming has been done to utilize as much available performance as possible. Hence, these results may not represent the actual state of energy efficiency but give an indicator that allows comparison among other platforms.

Furthermore, especially regarding the Mutual Information method, no publications are known that specifically implement this measure. As this work aims at establishing *platforms* instead of *methods*, comparison is done throughout a multitude of tools that may not always produce results of the same quality but at least claim to find second-order and third-order interactions by exhaustively searching the space of SNP tuples.

5.3.1 Second Order Interaction

In Table 5.4, several implementations of the same method are shown and compared to each other with respect to the processing throughput in million tables per second and the total energy consumption in kWh. The reference data set used contains of approx. 5 000 samples and 500 000 SNPs (WTCCC) and the method parameters are comparable to the original BOOST implementation's defaults. For reference, the original implementation has been included.

The bottom half of Table 5.4 contains the original CPU-based implementation of BOOST and the later published GPU-based version. The cluster-based reference implementation on a 32-node Intel Xeon E5 system delivers 46.3 million tables per second, a speed-up of 138 over the original CPU-based implementation when comparing whole systems. Assuming that the original implementation does not use multiple threads and furthermore, the cluster nodes each use 8 threads as supported by the processor, the per-core speed-up becomes approx. 0.6. This may be explained by the cluster node interconnects. Even with high-performance Infiniband links as they are used in the cluster system, a severe loss of computational power is to be expected. On the other hand, these results show that the original im-

Table 5.4. Performance comparison of different BOOST implementations

Architecture	Runtime	Throughput	Energy	Source
FPGA Cluster 128× Spartan 6-LX150	6 min	348.0 MT/s	0.07 kWh	[GWK+15a]
Multi-GPU Node 4× NVIDIA GTX Titan	10 min	208.8 MT/s	0.15 kWh	[GWK+15a]
Hybrid System *Virtex 7-690T, NVIDIA GTX 780 Ti*	*12 min*	*158.0 MT/s*	*0.10 kWh*	[KWS+16]
Multi-GPU Node 4× NVIDIA Tesla K20m	15 min	139.2 MT/s	0.18 kWh	[GWK+15a]
Single GPU NVIDIA GTX Titan	37 min	56.4 MT/s	0.15 kWh	[GWK+15a]
CPU Cluster 32× Intel Xeon E5-2660	45 min	46.3 MT/s	2.28 kWh	[GWK+15a]
FPGA Kintex 7-325T	51 min	41.3 MT/s	0.44 kWh	[Wie16]
Single GPU original GBOOST	2 h 52 min*	12.1 MT/s	(unknown)	[YYW+11]
CPU original BOOST	115 h 44 min*	0.3 MT/s	(unknown)	[WYY+10]

plementation is indeed well-written and optimized and is therefore suitable for reference in comparisons.

Both multi-GPU systems implement the BOOST measure in single precision. In their dynamic workflow, data is split into work units and distributed among the systems' GPUs on-demand. When comparing the "4× NVIDIA GTX Titan" node against the single GPU node, a speed-up nearly linear to the number of devices can be observed (3.7). As the GTX Titan is specified with a power-under-load of 250 W, but the whole single-node system draws only 243 W, it can be assumed that neither the host system, nor the GPU is able to exploit its full potential, though the margin seems to become narrower on the four-GPU version.

It can further be seen that the hybrid-parallel prototype system (table row in italics) achieves a speed-up of 527 over the original CPU-based implementation [WYY+10] and 13 over the original GPU-based implemen-

Table 5.5. Performance comparison of different exhaustive pairwise interaction detectors

Method	Architecture	Runtime	Throughput	Energy
iLOCi [PNI+12]	FPGA Cluster	4 min	520.8 MT/s	0.05 kWh
multiEpistSearch [GKW+15]	2×NVIDIA Tesla K20m 1× Intel Xeon Phi	10 min	210.2 MT/s	(unknown)
BOOST	*Hybrid System*	*12 min*	*158.0 MT/s*	*0.05 kWh*
EpiGPU [HTW+11]	NVIDIA GeForce GTX 580	2 h 55 min	11.9 MT/s	(unknown)
iLOCi [PNI+12]	2× Intel Xeon quad-core (unspecified model)	19 h	1.8 MT/s	4.94 kWh

tation [YYW+11]. An optimized version running on a compute cluster node with four NVIDIA GTX Titan, performance-wise comparable to the NVIDIA GTX 780 Ti used in this work, performs only 24 % better than the hybrid system but uses four computation devices instead of two. When compared to an FPGA cluster consisting of 128 Xilinx Spartan 6-LX150 FPGAs, 50 % of the table throughput is achieved. However, a single LX150 device contains a fifth of the logic resources of the Xilinx Virtex 7-690T. Additionally, an implementation of the contingency table generation and evaluation on a Xilinx Kintex 7-325T FPGA takes 51 min. This FPGA is technically equivalent to the Virtex 7-690T FPGA used in the prototype but only possess half its resources. However, the single-FPGA implementation achieved less than a quarter of the hybrid performance. It can therefore be concluded that a hybrid system with a GTX 780 Ti and a Virtex 7-690T can deliver more performance than a system with two of these graphics cards or two FPGA boards of this kind. Assuming that the performance scales linearly in the number of computing devices used, the hybrid prototype processes the data set in 35 % less time than a hypothetical Multi-GPU node with two NVIDIA GTX Titan devices. Regarding energy efficiency, only the FPGA cluster solution beats the hybrid prototype.

In Table 5.5, several two-way interaction analysis implementations are

compared that exhaustively cover the full WTCCC data set without prefiltering. By far the fastest method is iLOCi, a model-free method that measures differences in the *linkage equilibrium (LD)* of SNPS, i.e. the "equivalence" of SNPs [PNI+12]. Hemani et al. created EpiGPU, an OpenCL-based GPU-assisted approach to perform Fisher's dependency tests with 4 degrees of freedom [HTW+11; Lom07]. The second best performance in this survey is delivered by multiEpistSearch, a logistic regression-based model similar to BOOST, implemented on GPU arrays using the Unified Parallel C++ compiler extension [GKW+15] and a novel block-oriented work distribution scheme. Unfortunately, energy consumption rates were not available for all methods.

Comparing different methods by performance, though, may only be useful in initial screenings of newly acquired data sets, as they significantly differ in sensitivity and specificity, or in general, the quality of results.

5.3.2 Third Order Interaction

Evaluation with three-way epistasis detection methods is not well-established, probably due to the heavy computational burden. Therefore, a novel measure based on information theory methods has been developed, the Mutual Information measure. A CPU-only OpenMP-based parallel implementation has been used to provide a performance reference to determine the speed-up. The reference system features an Intel Core i7-4790K with four physical cores and 2-way simultaneous multithreading enabled (eight virtual cores, see Sect. 2.1.1). The clock frequency is 4.0 GHz and the system supports 32 GiB DDR4-SDRAM.

Table 5.6 shows the results for a data set of 5 000 samples and the displayed number of SNPs. It can be seen that the overhead of setting up the FPGA/GPU pipelines is negligible as the hybrid-parallel solution outperforms the CPU-only version after just over 100 SNPs and converges to a speed-up of approx. 72 in higher SNP numbers.

One noteworthy implementation using a measure similar to the Mutual Information used in this work, is GPU3SNP [GS15]. The authors use a configuration of four NVIDIA GTX Titan graphics accelerators to exhaustively evaluate all SNP 3-tuples using Mutual Information in 22 hours for

Table 5.6. Three-way interaction comparison

SNPs	Host Runtime	Hybrid Runtime	Speed-up
100	0.8 s	3.2 s	0.25
250	9.7 s	4.7 s	2.0
500	1 min 10 s	6.9 s	10.1
1 000	7 min 39 s	8 s	57.4
2 000	1 h 2 min	53 s	70.7
5 000	16 h 14 min	13 min 36 s	71.6
10 000	5 d 10 h	1 h 49 min	71.8

50 000 SNPs and 1 000 individuals. Assuming linear decrease in runtime w.r.t. to the number of samples in the data set, the hybrid-parallel prototype system's evaluation runtimes can be extrapolated to take 45 hours and 20 minutes for the same data set, based on the runtime for 5 000 SNPs and 5 000 samples. However, earlier analyses suggest that data channel capacity between the FPGA and GPU becomes a bottleneck when a large number of tables is generated in a short time frame (see Sect. 5.2.1). In case of larger sample numbers, more time passes between the creation of two tables, reducing the actual data rates. Thus, interpolating from a data set run of 40 000 samples instead of 5 000 down to 1 000 samples, yields a runtime of 31 hours by factoring out the transmission speed limit. The hypothetical Multi-GPU node consisting of two NVIDIA GTX Titan boards as used in the BOOST evaluation would take 44 hours, resulting in a 42 % higher speed when using the hybrid architecture instead, a similar speed increase as observed in the two-way interaction set-up.

To assess the viability of the hybrid prototype system with respect to a system solely consisting of FPGAs, the full analysis pipeline, i.e. creation of contingency tables and evaluating the Mutual Information measure, has also been implemented on an FPGA. Unfortunately, the 3-way configuration presented in Table 5.1 only occupies less than 50 % of the FPGA's available resources as a fully occupied device would require higher data rates than the Generation 2 PCI Express link is currently capable of. Therefore, a Mutual Information pipeline could be added without reducing the number

of nodes in the systolic array, making the standalone version precisely as fast as the hybrid prototype [Wie16].

Although the transmission link is an inherent component and apparently a weakness of the hybrid architecture, using a more recent PCI Express interface on the FPGA side such as Gen 3 or Gen 4, would certainly support the assumption that the previous runtime interpolation of 31 hours is valid. Upgrading the link from Gen 2 to Gen 4 almost quadruples the available data rate. When taking the table construction and processing rates from earlier chapters (i.e. Tables 5.1 and 5.2), the bottleneck is unlikely to remain in the link. This further allows better exploiting the FPGA's resources to create systolic arrays with more than the double amount of nodes, making the hybrid prototype viable again with respect to the standalone FPGA version.

Unfortunately, aside from GPU3SNP, no other exhaustive search tool with comparable methods for third order interaction search is available for comparison although the authors of BOOST explicitly state that their method is generalizable to higher orders of interaction than two [WYY+10].

Regarding the energy consumption, GPU3SNP fails to be evaluated because of stability issues in longer runs but as both the hardware and software architectures are similar to the four-node BOOST GPU system shown in Table 5.4, similar rates can be expected. These similarity reasons also apply to the hybrid-parallel prototype. The differences in energy consumption have been shown to be very small as seen in Table 5.3.

Chapter 6

Conclusion

6.1 Summary of Results

A working hybrid-parallel architecture has been developed with focus on applications in bioinformatics, mainly large-scale analysis on genome-wide association studies. A general concept in computer science is to divide a problem into smaller, more manageable problems. This divide-and-conquer approach is also a central thought in the adaption or development of new methods and tools for the hybrid system. Specialized peripherals, freely configurable FPGAs, highly parallel GPUs and flexible CPUs, are used to create a heterogeneous architecture where every component is assigned a sub-problem that is specifically designed for the target platform in question.

In the case of gene-gene interactions in two and three dimensions, it has been shown that such a heterogeneous system is very efficient when compared to more conservative architectures. By the use of large systolic data structures, FPGAs can be programmed to generate contingency tables for later evaluation on GPUs, where higher arithmetics and transcendental functions are required for calculation. Through careful process orchestration by the CPU, both accelerators are able to work concurrently and move data to and from each other in very efficient ways. The hardware foundation is explicitly selected to perform high-throughput transfers with current system bus controllers and beneficial PCI Express lane configurations. All peripherals are tied together by a custom driver infrastructure that makes heavy use of advanced Linux kernel features and hardware-assisted data transfers while providing a usable interface for applications, such as the presented two-way and three-way interaction analysis toolkit.

The suitability of the presented hybrid-parallel architecture is supported

by the results presented in Chapter 5. In the case of two-way interaction analysis, the processing throughput of the proposed prototype has been shown to deliver results 35 % faster than two similar graphics cards while performing 25 % better than two similar FPGA boards. In third-order interaction, it is still 42 % faster than a two-GPU node but only as fast as a single-FPGA solution with the transmission link being the limiting factor. Since the introduction of the Alpha Data Virtex 7-690T boards, better PCI Express endpoint modules ("drivers" on the FPGA) became available that support Generation 3 and Generation 4 systems, doubling and quadrupling available net data rates. These developments will certainly lift the transmission barrier. As the FPGA system in question only consumes less than half of the resources available, fully exploiting the device will make the prototype faster than a two-FPGA system, or at least delivering on-par performance, given higher transport capabilities.

In general, the development and construction of the given hybrid-parallel architecture prototype yields a platform that supports applications in bioinformatics, specifically the analysis of gene-gene interactions in an exhaustive way. The performance figures show a drastic decrease in runtime when compared to more homogeneous and conservative architectures such as single-CPU nodes, CPU-based clusters, and even GPU clusters. Even then, the whole system fits into a single standard ATX desktop chassis without any expensive server-grade hardware or power supply requirements, besides a GPU and FPGA card. Instead of moving sensitive patient data to off-site data centers for computation, data sets can now be analyzed in-house by any lab technician due to simple platform interfaces, consumer-grade hardware requirements and low energy consumption.

6.2 Future Work

Although the development process outlined and discussed in this work is targeted towards a proof-of-concept system, a highly potent system has evolved. However, during research and development, some optimizations became clear that would have gone beyond the scope of this work and therefore were not implemented.

6.2.1 Direct Memory Access

Direct Memory Access is a core technology that this prototype system could not have been developed without. It allows transferring data between memories without involvement of the CPU, besides set-up and tear-down, and therefore frees valuable computational resources for the application's needs. In DMA scatter/gather transfers, the memory buffers that are scheduled for transmission are stored in lists where each entry represents a 4 096-byte block of memory, also called a *memory page*. These lists have a fixed capacity and are processed and re-filled in a circular buffer-style. While the CPU does not need to transfer the data by itself, it does have to prepare these scatterlists and feed them to the DMA controller. The presented applications use a number of buffers that are typically 256 MiB in size, or 65 536 pages per buffer. With transmission speeds measured in GiB/s, refilling these lists on time becomes measurable part of the runtime.

Newer hardware platforms, processors and operating systems support structures that are called *Huge Pages* on Linux, *Super Pages* on BSD and *Large Parges* on Windows. Generally, these are regular memory pages that support sizes of 4 KiB (the default size), 2 MiB and 1 GiB per page, drastically reducing the effort of list management, provided that these page sizes are supported by all connected peripherals.

Currently, data buffers are moved off the FPGA to the host's main memory using a DMA transaction, and then another DMA transaction is responsible for moving the same block of data from the main memory to the GPU's memory. Although both the NVIDIA CUDA interface and the Alpha Data interface explicitly allow transfers to "peer devices," no transmission protocols have been published. Reverse engineering techniques could be used to develop a protocol and remove the burden of buffer management entirely from the host, and reducing the number of required DMA transactions from two to one. Several research groups have already published findings regarding the direct data transmission between FPGAs and NVIDIA GPUs, but none of them achieve the data rates required and presented in this work, yet [BRF14; Sus14; Gil15].

6.2.2 Mutual Information Deficiencies

Mutual Information is a method used in information theory to measure the dependence between the entropies of two random variables. In this work, a single random variable has been assigned to the joint probability distribution of all SNPs that are used in the test. While this allows efficient computation of the dependency between the SNP tuple and the case/control variable, it is not only the combined effect of all SNPs that is shown, but for a "good" score, it is sufficient for a single SNP to have a high dependence (primary effects) or two of three SNPs (secondary effects). It is not always desirable to allow primary and secondary effects overshadowing a possible tertiary effect. Additionally, when a low number of best list entries is selected, a single SNP with a strong influence may be present in a large number of results, raising the score for an otherwise unremarkable SNP combination.

Although the primary and secondary effects could be removed from the final result by separately calculating the mutual information of all 1-tuples and 2-tuples of a 3-tuple and subtracting these, implementing a different method where primary and secondary effects play a lesser role in the result, might be a better solution. A promising method is *Information Gain*, where not only positive influence is modeled, but also negative (i.e. suppressing) influence. However, the Information Gain measure is much more complex than Mutual Information and hardly being able to be implemented on FPGAs in a resource-efficient way. Apart from the more precise measurement results, this also basically removes the possibility of standalone FPGA solutions, emphasizing the role of this hybrid-parallel system with respect to third-order interaction analyses. Research on Information Gain applications in third-order interaction is currently conducted and publications prepared.

6.2.3 Scaling Prototypes

An obvious follow-up target of this work is the turning of the prototype into a full productive system where multiple FPGAs, GPUs and CPUs are working together. Although this possibility has already been considered in the design of the respective drivers and software packages, data

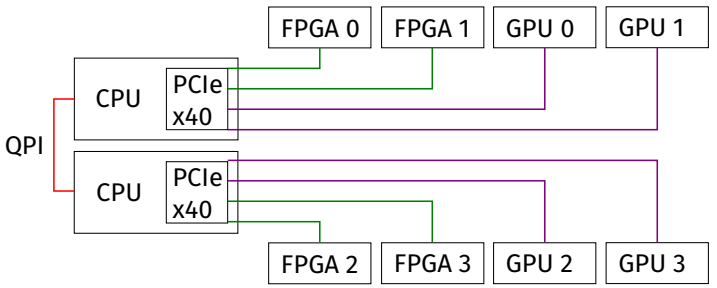

Figure 6.1. PCI Express lane layout in a multi-processor system

flow management becomes much more complex. An increased number of data moving threads will inevitably increase the lock contention ratio, i.e. the timer share that threads spend on waiting for a shared resource to become available. Currently, with only a few producing threads and a single consumer, the buffering queue could be implemented using standard queue structures from the C++ Standard Template Library and semaphores. A larger number of producers and consumers in high-performance environments might require more sophisticated data structures. One notable candidate for implementation is the Parallel Flat-Combining Synchronous Queue, where both producers and consumers do not actively operate on the work queue anymore but are serviced by a dedicated thread pool by dispensing *slot tickets* [HIS+10]. These tickets can then be used to access *slots* in the queue without further synchronization with other, concurrently accessing threads, virtually eliminating lock contention on these central data structures.

Aside from the pure software implementation, multiple computing devices raise issues in the PCI Express connectivity. If a GPU requires 16 lanes, four GPUs require 64 lanes. Unfortunately, current processors do not provide more than 40 lanes. An obvious solution is to use a multi-processor system for more PCI Express bandwidth, as shown in Figure 6.1. Even in a dual-processor set-up though, 40 lanes are not sufficient to deliver maximum performance to all connected devices. In a shared FPGA/GPU solution as presented in this work, data only flows between FPGAs and

GPUs. It is therefore sufficient to downgrade the GPU link from 16 lanes to 8 lanes, the highest lane width current Xilinx FPGAs support natively.

As Figure 6.1 suggests, additional care has to be taken to not cross CPU domain boundaries. For example, if FPGA 1 transmits data to GPU 2, the PCI Express connection is routed through the CPU interconnect QPI (Intel QuickPath Interconnect). The QPI is a shared medium where the various CPU peripherals communicate with each other, i.e. memory controllers, thread management units, I/O devices and PCI Express controllers. It does not only limit the maximum bandwidth for concurrently running (and boundary-crossing) FPGA/GPU transfers but may also severely reduce the system performance through bus contention, such as critical main memory transfers. For a successful implementation on a multi-device hybrid-parallel architecture, the exact CPU and PCI Express topologies have to therefore be examined. Additionally, great care has to be taken to bind execution threads that service a particular FPGA or GPU to the responsible CPU to avoid costly thread migration between CPUs and/or physical cores.

Currently, a follow-up system to this prototype is being installed at the Institute of Clinical Molecular Biology, Kiel University, generously funded by the Prof.-Dr.-Werner-Petersen-Stiftung. It features four high-end NVIDIA Tesla P100 accelerators, four Xilinx UltraScale Kintex 7-115, the largest devices in the Kintex series, two 8-core Intel Xeon processors clocked at 3.2 Ghz and 256 GiB DDR4-SDRAM. The presented BOOST method, Mutual Information and Information Gain measures are currently being implemented.

6.2.4 Other Applications

Analysis of genome-wide association studies is only one computationally demanding area in bioinformatics, while the potential applications of such a hybrid system are manifold. Ongoing projects include the following, where standalone FPGA implementations have already been developed [Wie16]:

Imputation refers to the statistical inference of factually unobserved genotypes. When using microarrays as explained in Section 3.1.6, only a limited set of SNPs is actually acquired. These, however, supply suppos-

edly enough information to *guess* SNP variants in non-typed regions based on statistical evaluations.

Sequence Alignment is the process of aligning two or more DNA sequences to each other to find the position with the most identical overlaps. Before GWAS are conducted, the actual SNP locations on the genome has to be known beforehand. One method to find these locations is the alignment of a large number of sequences to a reference genome. If a location is found were all (or many) sequences match except for a single base pair, a SNP might have been found. Naïve sequence comparison is typically done in quadratic runtime with respect to the string lengths and therefore presents a demanding problem.

Appendix A

Command-Line Options

A.1 `adboost` Help Output

```
Usage: adboost [options] <SNP input file>

Regular options:
  -? [ --help ]                         produce this help message
  --version                             prints version information
  --export-snpdb arg                    exports the SNP database to the given
                                        file
  -f [ --format ] arg (=plink_transposed)
                                        use this input data format (only needed
                                        if not importing the native format).
                                        Allowed formats: plink_transposed,
                                        plink_binary,boost,native
  -o [ --output ] arg                   write SNP interaction data to this file
                                        instead of stdout
  -t [ --threshold ] arg (=15)          set threshold for KSA test
  --threshold-ll arg (=30)              set threshold for log-linear test

Hidden options (only visible in debug mode):
  --debug                           produce lots of debug output
  --assume-fpgas arg (=-1)          set number of FPGAs to use (-1 for all)
  --assume-gpus arg (=-1)           set number of GPUs to use (-1 for all)
  --convert-only                    do only format conversion, don't do actual
                                    contingency table processing
  --dump-buffers                    dump received FPGA and GPU buffers to disk
  --timeout arg (=5000)             Timeout for FPGA transmissions (in ms)

This is version 1.00, compiled on Oct 10 2016, 11:10:22
Send bugs to Jan Christian Kaessens <jka+bugs@informatik.uni-kiel.de>
```

A.2 ad3way Help Output

```
Usage: ad3way [options] <SNP input file>

Regular options:
  -? [ --help ]                         produce this help message
  --version                             prints version information
  --export-snpdb arg                    exports the SNP database to the given
                                        file
  -f [ --format ] arg (=plink_transposed)
                                        use this input data format (only needed
                                        if not importing the native format).
                                        Allowed formats: plink_transposed,
                                        plink_binary,boost,native
  -o [ --output ] arg                   write SNP interaction data to this file
                                        instead of stdout
  -n [ --result-count ] arg (=1000)     set number of best results to keep

Hidden options (only visible in debug mode):
  --debug                           produce lots of debug output
  --assume-fpgas arg (=-1)          set number of FPGAs to use (-1 for all)
  --assume-gpus arg (=-1)           set number of GPUs to use (-1 for all)
  --convert-only                    do only format conversion, don't do actual
                                    contingency table processing
  --dump-buffers                    dump received FPGA and GPU buffers to disk
  --timeout arg (=5000)             Timeout for FPGA transmissions (in ms)

This is version 1.00, compiled on Oct 10 2016, 14:15:39
Send bugs to Jan Christian Kaessens <jka+bugs@informatik.uni-kiel.de>
```

Appendix B

Curriculum Vitae

M.Sc. Jan Christian Kässens

Date of birth:	October 3rd, 1984
Place of birth:	Eckernförde, Germany
Nationality:	German

Education

09/1995–07/2004	Jungmann Gymnasium Eckernförde, Abitur

Employment

10/2004–07/2005	Albert-Schweitzer-Schule, Eckernförde, Mandatory Social Service
07/2005–01/2007	syscOm automatisierungssysteme, Groß Wittensee, apprenticeship with focus on systems programming

since 02/2008	Freelancing with focus on application development
10/2012–03/2013	SciEngines GmbH, Kiel, software and hardware development, focus on systems and driver programming
04/2013–09/2016	CAU Kiel, Department of Computer Science, Technical Computer Science Group, research assistant, focus on high-performance applications in bioinformatics
since 10/2016	CAU Kiel, Institute of Clinical Molecular Biology, Bioinformatics Group, focus on high-performance applications in bioinformatics

Study and Research

04/2007–09/2010	CAU Kiel, Computer Science, Bachelor's thesis "Model-based Detection of Highlights in Dense Light Field Samples". Minor in psychology, cognition and learning
04/2010–12/2012	CAU Kiel, Computer Science, Master's thesis "A Distributed Shared Memory Core for FPGA Clusters". Minor in media pedagogics
since 03/2013	CAU Kiel, Computer Science, Dissertation "A Hybrid-parallel Architecture for Applications in Bioinformatics"
08/2013–02/2016	Research assistant in excellence cluster project "A Power-saving Benchtop Machine for Ultra-fast Genetic Data Analysis and Interpretation in the Inflammation Clinic"
07/2013–02/2016	Leading scientist in ZIM/AiF project "Development of a hybrid-parallel computing architecture for bioinformatics"

Teaching

04/2008 - 09/2009	Teaching assistant for "Organisation und Architektur von Rechnern" (*Organization and Architecture of Computer Systems*)
10/2009 - 09/2011	Student assistant at the Department of Computer Science (IT department)
10/2011 - 03/2013	Student assistant at the Media Education Instutute (IT department)
04/2013 - 09/2016	Head teaching assistant for "Rechnergestützter Entwurf digitaler Systeme" (*Computer-aided design of digital systems*)

Mentoring

07/2015	Annika Pooch (Bachelor's Thesis), "Entropieanalyse und Kompression von DNS"

Publications

- Jan Christian Kässens, Jorge González-Domínguez, Lars Wienbrandt, and Bertil Schmidt, *UPC++ for Bioinformatics: A Case Study Using Genome-Wide Association Studies*, 15th IEEE International Conference on Cluster Computing, Sep 2014.
- Jorge González-Domínguez, Bertil Schmidt, Jan Christian Kässens, and Lars Wienbrandt, *Hybrid CPU/GPU Acceleration of Detection of 2-SNP Epistatic Interactions in GWAS*, 20th International European Conference on Parallel and Distributed Computing (Euro-Par 2014), Lecture Notes on Computer Science, vol. 8632, pp. 680–691
- Lars Wienbrandt, Jan Christian Kässens, Jorge González-Domínguez, Bertil Schmidt, David Ellinghaus, and Manfred Schimmler, *FPGA-based*

acceleration of detecting statistical epistasis in GWAS, Procedia Computer Science, vol. 29 (2014), pp. 220–230.

- 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, vol. 29 no. 4, pp. 506–510

- Jan Christian Kässens, Lars Wienbrandt, Jorge González-Domínguez, Bertil Schmidt, Manfred Schimmler, *High-Speed Exhaustive 3-locus Interaction Epistasis Analysis on FPGAs*, Journal of Computational Science, vol. 9, pp. 131–136

- Michael Wittig, Jarl Anmarkrud, Jan Kässens, Simon Koch, Michael Forster, Eva Ellinghaus, Johannes Hov Roksund, Sascha Sauer, Manfred Schimmler, Malte Ziemann, Siegfried Görg, Frank Jacob, Tom Karlsen, Andre Franke, *Development of a high resolution NGS-based HLA-typing and analysis pipeline*, Nucleic Acids Research, 43(11):e70, 2015

- 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, 12 (5), 2015, pp. 982–994

- Jan Christian Kässens, Lars Wienbrandt, Jorge González-Domínguez, Bertil Schmidt, Manfred Schimmler, *Combining GPU and FPGA Technology for Efficient Exhaustive Interaction Analysis in GWAS*, Proceedings of the 2016 IEEE 27th International Conference on Application Specific Systems, Architectures and Processors, 2016, pp. 170–175

- Sven Gundlach, Jan Christian Kässens and Lars Wienbrandt, *Genome-Wide Association Interaction Studies with MB-MDR and maxT multiple testing correction on FPGAs*, Procedia Computer Science, 2016, vol. 80, pp. 639–649

- Lars Wienbrandt, Jan Christian Kässens, Matthias Hübenthal, and David Ellinghaus, *Fast Genome-wide Third-order SNP Interaction Tests with Information Gain on a Low-cost Heterogeneous Parallel FPGA-GPU Computing Architecture*, Procedia Computer Science, 2017, manuscript accepted for publication

Professional Activities and Memberships

- Member of the Program Committee of the *Workshop on Parallel Computational Biology (PBC 2015)*. Krakow, Poland
- Member of the Program Committee and Reviewer of the *ACM Platform for Advanced Scientific Computing (PASC 2016)*. Lausanne, Switzerland
- Reviewer for the *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*
- Reviewer for the *International Journal of High Performance Computing (IJHPC)*
- Member of the Program Committee and Reviewer of *The International Conference on High Performance Computing & Simulation (HPCS 2017), Workshop on Exploitation of high performance Heterogeneous Architectures and Accelerators (WEHA 2017)*

Bibliography

[LDD3] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. "Linux Device Drivers". In: 3rd ed. O'Reilly, Jan. 2005. Chap. 8.

[Z97] *Intel 9 Series Chipset Family Platform Controller Hub Datasheet*. Intel Corporation. June 2015.

[04] *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor*. White Paper. 2004.

[12] *NVIDIA Kepler GK110 Architecture Whitepaper*. NVIDIA Corporation. 2012.

[15] *Leveraging Power Leadership at 28 nm Xilinx 7 Series FPGAs (WP436)*. Xilinx Inc. Jan. 2015.

[4790] *4th Generation Intel Core Processor Family Datasheet*. Intel Corporation. Mar. 2015.

[ADK+12] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. "An Efficient Unbounded Lock-Free Queue for Multi-core Systems". In: *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, 8 27-31, 2012. Proceedings*. Ed. by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis. Berlin, Heidelberg: Springer, 2012, pp. 662–673. ISBN: 978-3-642-32820-6. DOI: 10.1007/978-3-642-32820-6_65.

[AJL+15] Brude Alberts, Alexander Johnson, Julian Lewis, David Morgan, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell*. 6th. Garland Science, 2015.

[ALI+00] Kati Asumalahti, Tarja Laitinen, Raija Itkonen-Vatjus, Marja-Liisa Lokki, Sari Suomela, Erna Snellman, Ulpu Saarialho-Kere, and Juha Kere. "A Candidate Gene for Psoriasis near HLA-C, HCR (Pg8), is Highly Polymorphic with a Disease-associated Susceptibility Allele". In: *Human Molecular Genetics* 9.10 (2000), pp. 1533–1542. DOI: 10.1093/hmg/9.10.1533. eprint: http://hmg.oxfordjournals.org/content/9/10/1533.full.pdf+html. URL: http://hmg.oxfordjournals.org/content/9/10/1533.abstract.

[ASS+86] M. D. Atkinson, J.-R. Sack, N. Santori, and T.Strothotte. "Min-max Heaps and Generalized Priority Queues". In: *Communications of the ACM* 29.10 (Oct. 1986), pp. 996–1000.

[BAS03] Ravi Budruk, Don Anderson, and Tom Shanley. "PCI Express System Architecture". In: Addison-Wesley, 2003. Chap. 6, pp. 252–282.

[BCC+07] P. R. Burton et al. "Genome-wide Association Study of 14,000 Cases of Seven Common Diseases and 3,000 Shared Controls". In: *Nature* 447.7145 (June 2007), pp. 661–678.

[BCP+04] Sebastian Bonhoeffer, Colombe Chappey, Neil T. Parkin, Jeanette M. Whitecomb, and Chrostos J. Petropoulos. "Evidence for Positive Epistasis in HIV-1". In: *Science* 306.5701 (2004), pp. 1547–1550.

[Bel53] David Arthur Bell. *Information Theory and Its Engineering Applications*. Sir Issac Pitman & Sons, Ltd., 1953.

[BK08] Pankaj Bhambri and Amit Kamra. "Computer Peripherals And Interfaces". In: Technical Publications, 2008. Chap. System Resources, pp. 1–35.

[BO03] Randal E. Bryant and David O'Hallaron. *Computer systems: a programmer's perspective*. Pearson Education, 2003.

[BRF14] Ray Bittner, Erik Ruf, and Alessandro Forin. "Direct GPU/FPGA Communication via PCI Express". In: *Cluster Computing* 17.2 (2014), pp. 339–348. ISSN: 1573-7543. DOI: 10.1007/s10586-013-0280-9. URL: http://dx.doi.org/10.1007/s10586-013-0280-9.

[CC10] Brian Charlesworth and Deborah Charlesworth. *Elements of Evolutionary Genetics*. Roberts & Company Publishers, 2010.

[CCM94] Svante Carlsson, Jingsen Chen, and Christer Mattsson. "Heaps with Bits". In: *Algorithms and Computation: 5th International Symposium Proceedings*. Ed. by Ding-Zhu Du and Xiang-Sun Zhang. Vol. 834. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 288–296.

[CCT+15] Christopher C. Chang, Carson C. Chow, Laurent CAM Tellier, Shashaank Vattikuti, Shaun M. Purcell, and James J. Lee. "Second-generation PLINK: Rising to the Challenge of Larger and Richer Datasets". In: *GigaScience* 4.1 (2015), p. 7. ISSN: 2047-217X. DOI: 10.1186/s13742-015-0047-8. URL: http://dx.doi.org/10.1186/s13742-015-0047-8.

[CLR+07] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2nd. The MIT Press, 2007.

[Cor04] Intel Corporation. *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor*. White Paper. Mar. 2004.

[CSM12] Yupeng Chen, Vertil Schmidt, and Doublas L. Maskell. "A hybrid short read mapping accelerator". In: *BMC Bioinformatics* 14.67 (2012).

[CT06] Thomas M. Cover and Joy A. Thomas. "Elements of Information Theory". In: 2nd. John Wiley & Sons, Inc., 2006. Chap. 2, pp. 13–20.

[Dar01] Frederica Darema. "The SPMD Model: Past, Present and Future". In: *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2001, pp. 1–. ISBN: 3-540-42609-4. URL: http://dl.acm.org/citation.cfm?id=648138.746808.

[Deu96] Peter Deutsch. *GZIP File Format Specification 4.3*. RFC 1952. Internet Engineering Task Force, Network Working Group, May 1996.

[DGN+88] Frederica Darema, David A. George, Alan Norton, and Gregory F. Pfister. "A Single-program-multiple-data Computational Model for EPEX/FORTRAN." In: *Parallel Computing* 7.1 (1988), pp. 11–24.

[DLP02] Jack J. Dongarra, Piotr Luszczeck, and Antoine Petitet. *The LINPACK Benchmark: Past, Present, and Future*. Tech. rep. University of Tennessee, Department of Computer Science, Knoxville, July 2002.

[DS182] *Kintex-7 FPGAs Data Sheet: DC and Switchting Characteristics*. v2.15. Xilinx Inc. Nov. 2015.

[DS890] *UltraScale Architecture and Product Overview*. v2.10. Xilinx Inc. Nov. 2016.

[DWL+12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. "From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming". In: *Parallel Computing* 38.8 (2012). {APPLICATION} {ACCELERATORS} {IN} {HPC}, pp. 391–407. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2011.10.002. URL: http://www.sciencedirect.com/science/article/pii/S0167819111001335.

[ECX+11] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. "Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling". In: *SIGARCH Comput. Archit. News* 39.1 (Mar. 2011), pp. 319–332. ISSN: 0163-5964. DOI: 10.1145/1961295.1950402. URL: http://doi.acm.org/10.1145/1961295.1950402.

[Fis83] Joseph A. Fisher. "Very Long Instruction Word Architectures and the ELI-512". In: *SIGARCH Comput. Archit. News* 11.3 (June 1983), pp. 140–150. ISSN: 0163-5964. URL: http://dl.acm.org/citation.cfm?id=1067651.801649.

[Fly72] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960.

[Fly95] Michael J. Flynn. “Computer Architecture: Pipelined and Parallel Processor Design”. In: Computer Science Series. Jones and Bartlett Publishers, Mar. 1995. Chap. 1, pp. 54–56.

[Fog16] Agner Fog. *Instruction Tables*. Tech. rep. 4. Technical University of Denmark, 2016.

[GFB+04] Edgar Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[GHF+06] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. “Synergistic Processing in Cell's Multicore Architecture”. In: *IEEE Micro* 26.2 (Mar. 2006), pp. 10–24. ISSN: 0272-1732. DOI: 10.1109/MM.2006.41. URL: http://dx.doi.org/10.1109/MM.2006.41.

[Gil15] Alexander Gillert. “Direct GPU-FPGA Communication”. MA thesis. Technische Universität München, 2015.

[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: *The International Journal of High Performance Computing Applications* 29.4 (2015), pp. 506–510. DOI: 10.1177/1094342015585846. eprint: http://dx.doi.org/10.1177/1094342015585846. URL: http://dx.doi.org/10.1177/1094342015585846.

[GMY+14] Xuan Guo, Yu Meng, Ning Yu, et al. “Cloud Computing for Detecting High-order Genome-wide Epistatic Interaction via Dynamic Clustering”. In: *BMC Bioinformatics* 15.1 (2014), p. 102. ISSN: 1471-2105. DOI: 10.1186/1471-2105-15-102. URL: http://www.biomedcentral.com/1471-2105/15/102.

[Goo01] David S. Goodsell. “The Molecular Perspective: Ultraviolet Light and Pyrimidine Dimers”. In: *The Oncologist* 6.3 (June 2001), pp. 298–299.

[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". In: *BMC Genomics* 14.3 (2013), pp. 1–18.

[GS15] Jorge González-Domínguez and Bertil Schmidt. "GPU-accelerated Exhaustive Search for Third-order Epistatic Interactions in Case–control Studies". In: *Journal of Computational Science* 8 (2015), pp. 93–100. ISSN: 1877-7503. DOI: http://dx.doi.org/10.1016/j.jocs.2015.04.001. URL: http://www.sciencedirect.com/science/article/pii/S1877750315000393.

[GSK+14] Jorge González-Domínguez, Bertil Schmidt, Jan C. Kässens, and Lars Wienbrandt. "Hybrid CPU/GPU Acceleration of Detection of 2-SNP Epistatic Interactions in GWAS". In: *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, 8 25-29, 2014. Proceedings*. Ed. by Fernando Silva, Inês Dutra, and Vítor S. Costa. Springer International Publishing, 2014, pp. 680–691.

[GWK+15a] Jorge Gonzalez-Dominguez, 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 Trans Comput Biol Bioinform* 12.5 (2015), pp. 982–994.

[GWK+15b] Jorge González-Domínguez, Lars Wienbrandt, Jan C. 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 (Oct. 2015), pp. 982–994.

[HIS+10] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. "Scalable Flat-Combining Based Synchronous Queues". MA thesis. Ben-Gurion University, Tel-Aviv University, and Sun Labs at Oracle, 2010.

[Hoe12] Jesse Hoey. *The Two-Way Likelihood Ratio (G) Test and Comparison to Two-Way Chi Squared Test*. 2012. arXiv: 1206.4481 [stat.ME].

[HTW+11] G. Hemani, A. Theocharidis, W. Wei, and C. Haley. "EpiGPU: Exhaustive Pairwise Epistasis Scans Parallelized on Consumer Level Graphics Cards". In: *Bioinformatics* 27.11 (June 2011), pp. 1462–1465.

[IA32] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. June 2016.

[IBS12] Ra Inta, David J. Bowman, and Susan M. Scott. "The "Chimera": An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform". In: *International Journal of Reconfigurable Computing* 2012 (2012).

[Ill16] Illumina Inc. *Human Infinium DNA Microarrays Information Sheet*. Online, accessible at http://www.illumina.com/applications/agriculture/consortia.html. Pub. No. 370-2015-003. June 2016.

[Ing56] Vernan M. Ingram. "A Specific Chemical Difference Between the Globins of Norml Human and Sickle-cell Anaemia Haemoglobin". In: *Nature* 178.4537 (Oct. 1956), pp. 792–794.

[Int93] *8237A High Performance Programmable DMA Controller (8237A-5) Datasheet*. Intel Corporation, Sept. 1993.

[JLS11] Jestinah Mahachie John, François Van Lishout, and Kristel Van Steen. "Model-Based Multifactor Dimensionality Reduction to Detect Epistasis for Quantitative Traits in the Presence of Error-free and Noisy Data". In: *European Journal of Human Genetics* 19 (2011), pp. 696–703.

[Kau93] Stuart A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993.

[KDH+06] Eric J. Kelmelis, James P. Durbano, John R. Humphrey, Fernando E. Ortiz, and Petersen F. Curt. "Modeling and simulation of nanoscale devices with a desktop supercomputer". In: vol. 6328. 2006, pp. 4–12. DOI: 10.1117/12.681085. URL: http://dx.doi.org/10.1117/12.681085.

[Ken01] Steven L. Kent. *The Ultimate History of Video Games: From Pong to Pokémon*. Three Rivers Press, 2001.

[KGB+14] J. Kocz et al. "A scalable hybrid FPGA/GPU FX correlator". In: *Journal of Astronomical Instrumentation* 3.1 (2014), p. 1450002.

[KGW+14] J. C. Kässens, J. González-Domínguez, L. Wienbrandt, and B. Schmidt. "UPC++ for Bioinformatics: A Case Study Using Genome-Wide Association Studies". In: *IEEE CLUSTER*. 2014, pp. 248–256.

[KL10] Wolfgang Kastl and Thomas Loimayr. "A Parallel Computing System with Specialized Coprocessors for Cryptanalytic Algorithms". In: *Sicherheit 2010: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 5. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 5.-7. Oktober 2010 in Berlin*. 2010, pp. 73–84. URL: http://subs.emis.de/LNI/Proceedings/Proceedings170/article5766.html.

[KL51] Solomon Kullback and Richard A. Leibler. "On Information and Succiciency". In: *Annals of Mathermatical Statistics* 22.1 (Mar. 1951), pp. 79–86.

[KMS+07] M. A. Khan, A. Mathieu, R. Sorrentino, and N. Akkoc. "The Pathogenetic Role of HLA-B27 and its Subtypes". In: *Autoimmun Rev* 6.3 (Jan. 2007), pp. 183–189.

[KTL+12] Kiran Kasichayanula, Dan Terpstra, Piotr Luszcek, Stan Tomov, and Shirley Moore. *Power Aware Computing on GPUs*. Tech. rep. Innovative Computing Laboratory, University of Tennessee Knoxville, USA, 2012.

[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 (2015), pp. 131–136.

[KWS+16] J. C. Kässens, L. Wienbrandt, M. Schimmler, J. González-Domínguez, and B. Schmidt. "Combining GPU and FPGA Technology for Efficient Exhaustive Interaction Analysis in GWAS". In: *2016 IEEE 27th International Conference on Applič ation-specific Systems, Architectures and Processors (ASAP)*. July 2016, pp. 170–175. DOI: 10.1109/ASAP.2016.7760788.

[Lam79] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *ACM Transactions on Computer Systems* 28.9 (Sept. 1979), pp. 690–691.

[Lew04] Ricki Lewis. *Human Genetics: Concepts and Applications*. 6th. McGraw-Hill, 2004.

[LMA+16] Jing Li, James D. Malley, Angeline S. Andre, Margaret R. Karagas, and Jason H. Moore. "Detecting Gene-gene Interactions using a Permutation-based Random Forest Method". In: *BioData Mining* 9 (2016), p. 14.

[Lom07] Richard G. Lomax. *Statistical Concepts: A Second Course*. 3rd ed. Lawrence Erlbaum Associates Inc., Apr. 2007.

[Mah08] Brendan Maher. "Personal Genomes: The Case of the Missing Heritability". In: *Nature* 456 (Nov. 2008), pp. 18–21.

[Man10] Teri A. Manolio. "Genomewide Association Studies and Assessment of the Risk of Disease". In: *New England Journal of Medicine* 363.2 (2010), pp. 166–176.

[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.

[MBH+02] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. *Hyper-Threading Technology Architecture and Microarchitecture*. Tech. rep. Intel Corporation, 2002.

[Mic10] Micron Technology Inc. *MT8JTF12864HZ-1GB 204-Pin DDR3 SODIMM*. Data sheet. Rev. G 5/13 EN. 2010.

[Mic11] Micron Technology Inc. *18KSF1G72HZ-8GB 204-Pin DDR3L SODIMM*. Data sheet. Rev. I 8/15 EN. 2011.

[NCT+16] Marco S. Nobile, Paolo Cazzaniga, Andrea Tangherloni, and Daniela Besozzi. "Graphics processing units in bioinformatics, computational biology and systems biology". In: vol. 182. Briefings in Bioinformatics 4. July 2016, pp. 1–16.

[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 (1970), pp. 443–453.

[OGK+12] Daniel A. Orozco, Elkin Garcia, Rishi Khan, Kelly Livingston, and Guang Gao. "Toward High-Throughput Algorithms on Many-Core Architectures". In: *ACM Transactions on Architecture and Code Optimization* 8.4 (2012), Article No. 49.

[Ori07] Nikki Mirghafori Oriol Vinyals Gerald Friedland. *Revisiting a Basic Function on Current CPUs: A Fast Logarithm Implementation with Adjustable Accuracy*. Tech. rep. International Computer Science Institute, University of California, June 2007.

[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.

[Phi08] Patrick C. Phillips. "Epistasis – The Essential Role of Gene Interactions in the Structure and Evolution of Genetic Systems". In: *Nature Reviews. Genetics* 9.11 (2008), pp. 855–867.

[PMV] Albert Pool, Martin Mares, and Michael Vaner. "The PCI ID Repository". retrieved on 2016-09-20. URL: http://pci-ids.ucw.cz/.

[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* 13.7 (2012), pp. 1–15.

[PS10] Mihaela Pertea and Steven L. Salzberg. "Between a Chicken and a Grape: Estimating the Number of Human Genes". In: *Genome Biology* 11.5 (2010), p. 206. ISSN: 1474-760X. DOI: 10.1186/gb-2010-11-5-206. URL: http://dx.doi.org/10.1186/gb-2010-11-5-206.

[PS81] David A. Patterson and Carlo H. Sequin. "RISC I: A Reduced Instruction Set VLSI Computer". In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 443–457. URL: http://dl.acm.org/citation.cfm?id=800052.801895.

[Sew] Julian Seward. *bzip2 Documentation*.

[SR00] Michael S. Schlansker and B. Ramakrishna Rau. *EPIC: An Architecture for Instruction-Level Parallel Processors*. Tech. rep. Compiler and Architecture Research, HP Laboratories Palo Alto: Hewlett Packard, Feb. 2000.

[Sus14] David Susanto. "Parallelism for Computationally Intensive Algorithms with GPU/FPGA Interleaving". MA thesis. Technische Universität München, 2014.

[T500] "Global Supercomputing Capacity Creeps Up as Petascale Systems Blanket Top 100". In: *TOP500 - The List* (Nov. 2016). URL: https://www.top500.org/.

[TAOCP] Donald Ervin Knuth. *The Art of Computer Programming*. Vol. 4, Fascicle 1. Addison-Wesley Professional, 2009.

[TB14] Andrew S. Tanenbaum and Herbert Bros. *Modern Operating Systems*. 4th ed. Prentice Hall, Mar. 2014.

[TE14] Matthew B. Taylor and Ian M. Ehrenreich. “Higher-order Genetic Interactions and their Contribution to Complex Traits”. In: *Trends in Genetic* 31.1 (Sept. 2014), pp. 34–40.

[TKF+11] Philippe E. Thomas, Roman Klinger, Laura I. Furlong, Martin Hofmann-Apitius, and Christoph M. Friedrich. “Challenges in the Association of Human Single Nucleotide Polymorphism Mentions with Unique Database Identifiers”. In: *BMC Bioinformatics* 12.Suppl 4 (2011), S4.

[TOG13] *The Open Group Base Specifications*. 7th ed. vol. “System Interfaces”. The Open Group. Oct. 2013.

[TT08] Barry N. Taylor and Ambler Thompson, eds. *The International System of Units (SI)*. 2008th ed. Vol. NIST Special Publication 330. An English translation of the Convocation de la Conférence générale des poids et mesures, (25e réunion). National Institute of Standards and Technology, U.S. Department of Commerce, Mar. 2008.

[UG473] *7 Series FPGAs Memory REsources User Guide*. v1.12. Xilinx Inc. Sept. 2016.

[UG479] *7 Series FPGAs DSP48E1 Slice*. v1.9. Xilinx Inc. Sept. 2016.

[Vaj11] András Vajda. “Programming Many-Core Chips”. In: 1st ed. Springer US, 2011. Chap. 1.2.

[Vet15] Kishore Veturi. *Intel Pentium Processor, Statistical Analysis of Floating Point Flaw: Intel White Paper*. Tech. rep. Intel Corporation, 2015.

[Wet16] Kris A. Wetterstrand. *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)*. Dec. 2016. URL: https://www.genome.gov/sequencingcostsdata.

[Wie16] Lars Wienbrandt. *FPGAs in Bioinformatics*. Kiel Computer Science Series 2016/2. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, Kiel University, 2016.

[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*. Vol. 29. Elsevier, 2014, pp. 220–230.

[WKH+17] Lars Wienbrandt, Jan C. Kässens, Matthias Hübenthal, and David Ellinghaus. "Fast Genome-Wide Third-order SNP Interaction Tests with Information Gain on a Low-cost Heterogeneous Parallel FPGA-GPU Computing Architecture". In: *Procedia Computer Science*. manuscript accepted for publication. 2017.

[WLo11] Yue Wang, Guimei Liu, and Mengling Feng others. "An Empirical Comparison of Several Recent Epistatic Interaction Detection Methods". In: *Bioinformatics* 27.21 (2011), pp. 2936–2943.

[WWC05] Daniel M. Weinreich, Richard A. Watson, and Lin Chao. "Perspective: Sign Epistasis and Genetic Costraint on Evolutionary Trajectories". In: *Evolution* 59.6 (2005), pp. 1165–1174. ISSN: 1558-5646. DOI: 10.1111/j.0014-3820.2005.tb01768.x. URL: http://dx.doi.org/10.1111/j.0014-3820.2005.tb01768.x.

[WYY+10] 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.

[X200] *Intel Xeon Phi Product Brief*. Intel Corp. 2016.

[YC14] W. Yang and C. Charles Gu. "Random Forest Fishing: a Novel Approach to Identifying Organic Group of Risk Factors in Genome-wide Association Studies". In: *European Journal of Human Genetics* 22.2 (Feb. 2014), pp. 254–259.

[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.

[ZP02] David Zhang and Sankar K. Pai, eds. *Neural Networks and Systolic Array Design*. Series in Machine Perception and Artificial Intelligence. World Scientific Publishing Co. Pte. Ltd., July 2002.

[ZSX+16] Fusheng Zhou et al. "Epigenome-wide Association Data Implicates DNA Methylation-mediated Genetic Risk in Psoriasis". In: *Clinical Epigenetics* 8.1 (2016), p. 131. ISSN: 1868-7083. DOI: 10.1186/s13148-016-0297-z. URL: http://dx.doi.org/10.1186/s13148-016-0297-z.