Imperial College London
Department of Computing

Methodology for Complex Dataflow Application Development

Nils Voss

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the Imperial College London and
the Diploma of Imperial College London, November 2020

Declaration of Originality

This thesis is a presentation of my original research work. The contributions of others are involved, and every effort is made to indicate these clearly in the references to the literature and in the acknowledgement of collaborative research.

Copyright Declaration

Abstract

This thesis addresses problems inherent to the development of complex applications for reconfigurable systems. Many projects fail to complete or take much longer than originally estimated by relying on traditional iterative software development processes typically used with conventional computers. Even though designer productivity can be increased by abstract programming and execution models, e.g., dataflow, development methodologies considering the specific properties of reconfigurable systems do not exist.

The first contribution of this thesis is a design methodology to facilitate systematic development of complex applications using reconfigurable hardware in the context of High-Performance Computing (HPC). The proposed methodology is built upon a careful analysis of the original application, a software model of the intended hardware system, an analytical prediction of performance and on-chip area usage, and an iterative architectural refinement to resolve identified bottlenecks before writing a single line of code targeting the reconfigurable hardware. It is successfully validated using two real applications and both achieve state-of-the-art performance.

The second contribution extends this methodology to provide portability between devices in two steps. First, additional tool support for contemporary multi-die Field-Programmable Gate Arrays (FPGAs) is developed. An algorithm to automatically map logical memories to heterogeneous physical memories with special attention to die boundaries is proposed. As a result, only the proposed algorithm managed to successfully place and route all designs used in the evaluation while the second-best algorithm failed on one third of all large applications. Second, best practices for performance portability between different FPGA devices are collected and evaluated on a financial use case, showing efficient resource usage on five different platforms.

The third contribution applies the extended methodology to a real, highly demanding emerging application from the radiotherapy domain. A Monte-Carlo based simulation of dose accumulation in human tissue is accelerated using the proposed methodology to meet the real time requirements of adaptive radiotherapy.

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Wayne Luk for all the help and assistance he provided during the course of my research. Only through his advice and encouragement it was possible to create this thesis. He provided invaluable advice on how to tackle technical as well as social problems, organise my research, structure papers and presentations and approach all the people I had to bring together to perform this work.

Similarly, I would like to thank my second advisor and long-time manager at Maxeler Technologies, Prof. Georgi Gaydadjiev. Georgi not only helped me to obtain the unique opportunity to combine my research at Imperial College with my work at Maxeler but also was always there to provide advice and encouragement when needed. We spend countless hours discussing potential research problems and solutions, revising papers and preparing presentations. I am deeply thankful for all of these.

I would also like to thank Maxeler Technologies and especially its CEO and CTO Prof. Oskar Mencer for assisting this research. Without the access to tools, materials and source code most of my contributions would not have been possible. Similarly, I would like to thank all my colleagues at Maxeler for their assistance and help. I would like to especially thank Dr. Stephen Girdlestone, Chris Jones and Simon Tilbury for introducing me to the Maxeler technology stack and FPGA development in general. Additionally, I would like to thank Marco Bacis, Dr. Tobias Becker, Dr. Sotiria Fytraki, Dr. Joost Hoozemans, Dr. Bastiaan Kwaadgras, Pablo Quintana and Ir. Lukas Vermond for contributing directly to my research and helping me with the work for some of my publications. I would like to also thank all the current and past members of the hardware team, which had to deal with my requests for assistance and questions, the compiler team, which also helped me in so many cases and which I had the pleasure of joining for some time, the machine learning team, which I had the pleasure of working with and the Delft office which provided me a refuge on the continent.

I would also like to thank all the academic partners that helped me through fruitful discussions and joint research. I would like to especially thank Konstantina Koliogeorgi, Anna Maria Nestorov and Enrico Reggiani for publishing papers with me. Similarly, I am thankful to the assistance I received from the Joint Department of Physics at The Institute of Cancer Research for my radiotherapy related work. Without the help of Dr. Peter Ziegenhein and Prof. Uwe

Oelfke it would not have been possible to perform this research.

I would also like to express my gratitude to the members of the Custom Computing Research Group at Imperial for the discussions we had and the assistance that was provided.

Finally, I would like to thank my parents and friends for all the assistance they provided over the last few years. You always kept me motivated and focused.

Publications

The following publication contributes to the overview on HLS tools provided in chapter 2:

- Nils Voss, Tobias Becker, Oskar Mencer, Georgi Gaydadjiev, "Rapid Development of Gzip with MaxJ", in *Wong S., Beck A., Bertels K., Carro L. (eds) Applied Reconfigurable Computing. ARC 2017. Lecture Notes in Computer Science, vol 10216.*

The following publication contributes to the methodology described in chapter 3:

- Nils Voss, Bastiaan Kwaadgras, Oskar Mencer, Wayne Luk, Georgi Gaydadjiev, "On Predictable Reconfigurable System Design", in *ACM Transactions on Architecture and Code Optimization (TACO), vol. 18, no. 2, 2021.*

The following publication contributes to the evaluation of the methodology as presented in chapter 3:

- Nils Voss, Marco Bacis, Oskar Mencer, Georgi Gaydadjiev, Wayne Luk, "Convolutional Neural Networks on Dataflow Engines", in *2017 IEEE International Conference on Computer Design (ICCD).*

The following publication contributes to the memory mapping algorithm described in chapter 4:

- Nils Voss, Pablo Quintana, Oskar Mencer, Wayne Luk, Georgi Gaydadjiev, "Memory Mapping for Multi-die FPGAs", in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).*

The following publications contribute to the performance portability best practices described in chapter 4:

- Nils Voss, Tobias Becker, Simon Tilbury, Anna Maria Nestorov, Enrico Reggiani, Oskar Mencer, Georgi Gaydadjiev and Wayne Luk , "Performance Portable FPGA Design", as abstract in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).*

- Nils Voss, Tobias Becker, Oskar Mencer, Georgi Gaydadjiev and Wayne Luk , "Exploring Performance Portability and Scalability", prepared for submission.

The following publications contribute to the evaluation of the methods and algorithms developed in this thesis as described in chapter 5:

- Nils Voss, Peter Ziegenhein, Lukas Vermond, Joost Hoozemans, Oskar Mencer, Uwe Oelfke, Wayne Luk, Georgi Gaydadjiev, "Towards Real Time Radiotherapy Simulation", in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP).*
- Nils Voss, Peter Ziegenhein, Lukas Vermond, Joost Hoozemans, Oskar Mencer, Uwe Oelfke, Wayne Luk, Georgi Gaydadjiev, "Towards Real Time Radiotherapy Simulation", in *Journal of Signal Processing Systems, vol. 92, no. 9, 2020.*

The following publication was created as part of my research using some of the methodology results presented in this thesis but is not further discussed here:

- Konstantina Koliogeorgi, Nils Voss, Sotiria Fytraki, Sotirios Xydis, Georgi Gaydadjiev, Dimitrios Soudris, "Dataflow acceleration of Smith-Waterman with Traceback for high throughput Next Generation Sequencing", in *2019 29th International Conference on Field Programmable Logic and Applications (FPL).*

The following publication was published during my research but is not discussed in this thesis:

- Nils Voss, Stephen Girdlestone, Tobias Becker, Oskar Mencer, Wayne Luk, Georgi Gaydadjiev, "Low Area Overhead Custom Buffering for FFT", in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig).*

Contents

List of Tables

List of Figures

Glossary

ALM Adaptive Logic Module: The hardware unit in which logic resources on Altera/Intel FPGAs are organised. 74

ANSI American National Standards Institute: A private, US based, non-profit organisation developing standards. 60

API Application Programming Interface: The definition of interactions between different software components. It defines which functionalities are available, how they can be accessed as well as the data and its formats. 61, 62, 72–74, 76, 77, 79, 114, 122, 171, 172, 235

ASIC Application-Specific Integrated Circuit: Completely custom and highly optimised integrated circuits for a very specific application. As a result, they are not flexible and can typically be used to execute only one specific task in a specific way. This disadvantage is mitigated by the ability to achieve the best performance and energy efficiency on a given semiconductor technology node. 35, 36, 49, 50, 55, 60, 100, 107, 224

BMM Balanced Memory Mapping: A memory mapping algorithm developed in this thesis. It aims to improve the locality of designs and their timing closure while minimising hardware usage. More details in section 4.2. 22, 150, 153–155, 157, 158, 161–167, 181

BQCD Berlin Quantum Chromodynamics: A popular implementation of Lattice QCD. 22, 44, 133, 143–147, 160, 211–214, 227

BRAM Block Random-Access Memory: One of the main components of Xilinx FPGAs (M20K is the Intel equivalent). Used to store data on the FPGA itself. Usually, a BRAM can

store tens of kbits. More details in section 2.2.1. 47, 53–55, 151–158, 160–163, 166, 196, 197, 202–204, 222, 232

CG Conjugate Gradient: An iterative numerical method used to solve large systems of linear equations of the form $Ax = b$ where A is symmetric positive-definite. 22, 143–146

CNN Convolutional Neural Network: A class of deep neural networks. They are most commonly applied to computer vision problems. CNNs are characterised by the use of convolutional layers. These layers convolve the input data with a set of weights and heavily rely on weight sharing. As a result, CNNs are invariant to shifts in the input data. 19, 44, 88–92, 101, 104, 108, 112, 113, 116, 128, 133, 134, 136, 139, 142, 147, 212, 213, 225, 227

CPU Central Processing Unit: A processing unit which can execute instructions of a computer program. Modern CPUs usually follow the von Neumann architecture. The CPU is able to perform arithmetic, logic, control and I/O operations. 23, 35–39, 41, 43, 44, 46, 47, 50, 58, 61, 62, 66, 70, 73, 74, 82, 86, 87, 90, 94, 98–100, 103, 106–112, 116, 118, 119, 122, 131–133, 149, 168, 175–178, 185, 188–191, 195, 198–200, 204–212, 215–217, 219, 222, 224, 231, 235

DDR Double Data Rate Synchronous Dynamic Random-Access: An implementation of DRAM which transfers data on both the rising and falling clock edges. 52, 62, 66, 75–79, 87, 111, 123, 126, 128, 130, 138–140, 160, 163, 174, 186, 187, 189–192, 195, 198, 199, 204, 205, 217, 233

DFE Dataflow Engine: The definition used by Maxeler to describe their FPGA based compute accelerator cards. These PCIe extension cards combine a large capacity FPGA with parallel, high-bandwidth DDR memory and in some cases network interfaces. 30, 46, 66, 68, 72–75, 78, 79, 93, 94, 120, 139, 144, 170, 172, 178, 200, 215

DIMM Dual In-Line Memory Module: A module which contains multiple individual DDR chips. It can be inserted in a slot on modern computer systems. 75, 76, 139, 187, 190–192, 201

DMA Direct Memory Access: A method to access the main memory of a computer system without direct CPU involvement. This results in reduced load on the CPU and often higher speeds. 66, 71, 120, 125, 206, 209, 222, 231

DPM Dose Planning Method: An implementation of a Monte Carlo technique that simulates the dosimetric effect of high-energy photons in organic materials. More details in section 2.6.3. 96, 97, 186

DRAM Dynamic Random-Access Memory: Very dense volatile semiconductor-based memory. DRAM uses a refresh circuit which regularly rewrites the data in the memory cells to avoid data loss due to a leak of the charge in the tiny capacitors used to implement the individual memory cells. 126

DSE Design Space Exploration: In the context of hardware design DSE refers to a design stage in which different possible implementations for a specific design are considered. Usually there is a trade-off between the required hardware resources, bandwidth requirements, energy usage and the achievable performance or functionality. 42, 83, 84, 91, 100, 101, 139, 141, 203, 204, 222, 223, 227

DSL Domain Specific Language: A programming language which is customised for a specific use case or environment and is not intended for general purpose computing. 57

DSP Digital Signal Processor: A hardware component of modern FPGAs. Mostly used to implement multiplications but also contains adders and in some more recent devices dedicated support for floating point operations. More details in section 2.2.1. 50, 51, 71, 130, 152, 172, 174, 176–178, 180, 197, 202, 204, 212, 213

FF Flip-Flop: A circuit with two states used to store information. In digital circuits they are often used to store data on a clock edge which results in the clock driven synchronisation of the design. As a result, they can be used to facilitate pipelined designs. 50, 51, 75, 196, 197, 204

FFT Fast Fourier Transformation: An algorithm to calculate the discrete Fourier transformation (or its inverse (IFFT)) with reduced computational complexity. 48

FIFO First In, First Out: A strategy for buffering data. The items which are written to the buffer first are also read first. 54, 68, 70, 71, 112, 122, 124, 154, 193, 197, 203

FPGA Field-Programmable Gate Array: Semiconductor devices which are able to implement arbitrary hardware circuits while also being reconfigurable. More details in section 2.2. 7, 14, 16, 17, 21, 23, 35–47, 49–63, 65, 66, 68, 71–76, 81–87, 90, 93, 94, 98–103, 105–112, 114–116, 119–121, 123–134, 136, 137, 139, 140, 147–153, 157, 160, 167–183, 185, 186, 189–194, 198, 200, 204, 207–212, 214–219, 222–227, 229, 231–236

GEMM General Matrix Multiply: In the context of BLAS (Basic Linear Algebra Subprogram) libraries refers to the general form of matrix multiplications. The operation calculates $C = \alpha AB + \beta C$, where A, B and C are matrices and α and β scalars. 215

GPGPU General Purpose Graphics Processing Unit: This term is often used to refer to the usage of GPUs for problems which do not fall into the domain of computer graphics. The GPU is usually used to accelerate a CPU based computing system by offloading parts of an application and the associated data to it. 35

GPU Graphics Processing Unit: A specialised electronic device which can be used to create images intended for the output on a display. GPUs have a highly parallelised architecture with many execution units. Modern GPUs can also be used as GPGPUs. 36, 37, 43, 44, 47, 50, 58, 61, 62, 65, 98–100, 119, 149, 168, 185, 210, 217, 219, 224, 235

HBM High Bandwidth Memory: A DDR based technology to offer higher bandwidth memory to digital circuits. Multiple DRAM dies are stacked on the top of each other in 3D and implemented in a singe package with the digital circuit accessing the memory. This allows for a very wide memory bus leading to higher bandwidth. 128, 235

HDL Hardware-Description Language: A class of languages used to describe the structure and behaviour of digital electronic circuits. They allow the synthesis into a netlist and contain a notion of time. More details in section 2.3.1. 36, 55, 56, 58–60, 64, 81, 85, 100, 219, 220, 223

Heterogeneous Memory This term refers to the usage of fundamental different dedicated memory resources on modern FPGAs. The Xilinx Ultrascale+ architecture introduced URAMs as a new dedicated memory type. This new resource differs fundamentally from the also present BRAMs in terms of capacity, possible aspect ratios and supported feature set (number of ports, dual clock support). Despite these difference for the implementation of many logical memories they can still be used interchangeably. 41, 42, 44, 45, 53, 54, 148, 149, 152, 182, 226

HLS High-level Synthesis: A design process which creates a hardware design from an behavioural algorithmic description. More details in section 2.3.2. 48, 57, 59, 60, 62–64, 79, 81, 83, 84, 90, 100, 216–219, 221, 223, 231, 235

HPC High-Performance Computing: A domain of computing requiring a high amount of memory or computational performance. Often performed on super computers or large computing clusters. 7, 38, 40, 41, 43, 44, 46, 50, 57, 61, 71, 81, 90, 93, 100, 133, 160, 161, 172, 209, 210, 224, 225, 228, 234, 235

I/O Input/Output: The communication interface of a computer system with its external components. From the view of a single device (e.g., FPGA) this can include the communication with other parts of the same computer system (e.g., host computer) but also the interaction with other external devices. 39, 52, 65, 68, 69, 71, 75, 76, 79, 83, 84, 103, 108, 115, 119, 128, 136, 144, 146, 150, 168–170, 175, 177, 200

IP Intellectual Property: In the context of hardware design this usually refers to a pre-designed functional unit which can be integrated in larger designs. Often IP-Cores are provided by chip vendors or third parties. The content of the IP-Cores is the intellectual property of the provider and the usage has to follow a specific license agreement. Often the term is used more freely to refer to reusable functional blocks in a hardware design which are packaged in a way to ease future reuse. 59, 71, 75, 86, 121–123, 150, 163, 168, 171, 203

ISA Instruction Set Architecture: The definition of the interface between software and hardware for instruction-based processors. It defines all instructions which can be used by the

software and provides an abstraction for the functionality of the processor. 44, 168

LLVM-IR LLVM Intermediate Representation: An intermediate representation used by the LLVM compiler infrastructure. A language specific frontend generates LLVM-IR which can then be used to generate instructions for a specific machine. 86

LMEM Large Memory: Maxeler's name for the off-chip memory on their Dataflow Engines (DFEs). Usually implemented using DDR memory. 66, 68, 71, 77

Logical Memory A logical memory is an abstract description of a memory resource. It describes the required number of read and write ports, depth, width and requirements for other hardware features like dual-clock support. Logical memories are used to describe the required behaviour of a memory before they are mapped to a physical memory which will provide this behaviour. 42, 44, 47, 54, 55, 124, 148, 149, 151–158, 161, 181, 232

LQCD Lattice Quantum Chromodynamics: An approach to computationally simulate subatomic particles, based on discretising space and time into a 4D lattice. 143

LUT Look-Up Table: A standard building block of FPGAs used to implement arbitrary logic and arithmetic functions. It has multiple inputs and depending on the status of these inputs an output value is created. The relation between the inputs and outputs is fully configurable. More details in section 2.2. 50, 51, 53, 71, 75, 174, 177, 196, 197, 202

MAC Multiply Accumulate: An operation that first performs multiplication and then adds the result of this operation to an accumulator. 88, 91, 137

Machine Model A machine model provides an abstraction to describe the organisation of a computer. It describes the basic blocks of the computer and how they interact. It defines how data is stored, moved through the system and how operations on the data are performed. 36, 39, 56–58, 62, 64, 66–68, 71, 72, 79, 183, 231

ML Machine Learning: A field of computing looking at algorithms that improve through experience. Usually training data is used to train the algorithm to make predictions on future data. 35, 36, 85, 143

MLS Mid-Level Synthesis: A term recently coined by Microsoft. The idea is to provide a tool which is more appealing to software engineers but not as detached from hardware design as HLS. More details in section 2.3.4. 63

Multi Die FPGA The newer generations of FPGAs contain multiple multi-die FPGAs. In this case a single FPGA is built by connecting multiple individual silicon dies (called SLR in the context of Xilinx devices) on the same package. While the FPGA is presented as a single device to the user, it consists of multiple distinguishable parts with limited interconnectivity. More details in section 2.2.4. 41, 42, 44, 45, 47, 52, 148, 149, 151, 152, 158, 169, 171, 181, 182, 192, 212, 218, 226, 227, 231, 232

NCDF Normal Cumulative Distribution Function: A function which calculates the probability that the value of a random variable X is smaller or equal than x. 93

NN Neural Network: A class of machine learning algorithms which build networks using computational neuron models. 88, 89

OpenCL Open Computing Language: A framework to implement applications which can be executed on heterogeneous parallel computing systems. These systems can consist of a combination of CPUs, GPUs, FPGAs, Digital Signal Processors and other hardware accelerators. More details in section 2.3.3. 57, 58, 62, 63, 69, 86, 87, 221, 235

OpenSPL Open Spatial Programming Language: A standard programming framework for designing spatial computing systems. 64

PCIe Peripheral Component Interconnect Express: A standard for the connection of peripheral devices to a computer. 52, 66, 68, 71, 74–77, 121, 123, 125, 139, 141, 145, 160, 172, 174, 175, 189, 192, 195, 197, 198, 200, 204, 206–209, 233, 234

PE Processing Element: An element which implements a specific function. The parallelism of a design can be increased by adding more PEs which all perform the same functionality. 91, 134–139, 141, 212

Physical Memory A physical memory refers to a memory resource which is physically present on the chip. FPGAs have multiple classes of memory resources. These are memories implemented through logic either by FFs or LUTs and dedicated memory resources like BRAMs. In the case of the Xilinx Ultrascale+ architecture, URAMs are another physical memory resource class. All of these physical memories have different properties in terms of capacity, possible aspect ratios and supported features. 42, 44, 47, 53–55, 124, 148, 149, 151, 153, 155–158, 181, 193, 196, 232

Platform Refers to a specific development target, for example a single FPGA based accelerator card. Even if two cards use the same or a very similar FPGA device and a similar system architecture, they are considered as two different platforms since implementation details (e.g., pinout) differ resulting in required changes to any design targeting both cards. 41, 42, 44–46, 59, 60, 74, 76–79, 82, 84, 93, 101, 103, 107, 118, 120, 125, 149–151, 167, 168, 170–175, 177, 178, 180–182, 192, 200, 218, 222, 227

Programming Language A programming language is a formal language used to describe algorithms which after the necessary transformations (e.g., compilation, assembly) can be executed using a computer. The formal syntax has to be followed to generate a valid program description. A programming language is based around a programming model. 36, 43, 55–61, 63, 72, 74, 85, 100, 217, 231

Programming Model A programming model provides an abstraction for the underlying machine model and thereby helps to manage complexity. It defines how programs for a machine model can be designed. For example, a programming model for a von Neumann machine usually introduces functions, loops and conditionals which can then be mapped to the instructions of the machine by a compiler. Typically, when a parallel machine model consisting of multiple von Neumann machines is considered, a programming model usually represents this parallelism as independent threads or processes which, e.g., use a shared memory (e.g., OpenMP) or send messages over an interconnect (e.g., MPI). 36, 56–58, 60–62, 68, 69, 71–73, 76, 81, 119, 120, 231, 236

QCD Quantum Chromodynamics: The physical theory of strong interactions between sub-

atomic particles. 104, 108, 143, 144

RAM Random-Access Memory: A form of computer memory where the physical location of the data has no or only very limited impact on the time needed to access it. 53, 153, 154, 157, 158, 161, 162

ReLU Rectifier Linear Unit: An activation function used in NNs. It implements the function $f(x) = max(x, 0)$. 89, 134, 137

ROM Read-Only Memory: A memory which can only be read but not written to. 111, 195

RTL Register-transfer-level: An abstraction level used in the development of integrated circuits. The system is modelled through the signal flow between registers. 48, 58–60, 63, 73, 83, 85, 220

SDK Software Development Kit: A collection of tools and libraries used for the software development. 63

SIMD Single Instruction Multiple Data: A computer architecture where a single instruction is applied to multiple data items. 56

SLiC Simple Live CPU Interface: A part of the MaxCompiler toolchain. Used to interface the hardware accelerator on the DFE with an application on the host system. 66, 74

SLR Super Logic Region: Used in the context of multi-die Xilinx FPGAs. Refers to a single silicon die of the device. More details in section 2.2.4. 52–55, 75, 139, 140, 149, 150, 152–154, 156, 160, 162–167, 170, 171, 175–178, 191, 201, 218, 219, 226, 232

SPECT Single-Photon Emission Computed Tomography: A medical imaging technology able to provide 3D data. 100

SSI Stacked Silicon Interconnect: A Xilinx technology to mount multiple FPGA dies on a single silicon interposer. More details in section 2.2.4. 52

TBM2 Threshold Based Memory Mapping: A memory mapping algorithm introduced in this thesis for the comparison between different memory mapping strategies. More details in section 4.2.2. 157, 158, 161, 162, 164–167

TNS Total Negative Slack: In clock based digital hardware design signals are synchronised using FFs. This means that a signal has to be stable on the FF data input at the time of the active clock edge. The slack refers to the time between the signal stabilising and the point in time at which it has to be stable to guarantee correct operation. If the value is negative the timing constraint is violated. The total negative slack is the sum of all negative slack values. 166, 167

TPU Tensor Processing Unit: A class of ASICs for the acceleration of ML applications and especially NNs. The best-known examples for these devices are the TPUs developed by Google. 35, 90

URAM Ultra Random-Access Memory: A memory block introduced with the Ultrascale+ Xilinx FPGAs. The have a larger capacity than BRAMs but have a more limited functionality. More details in section 2.2.4. 47, 53–55, 151–158, 160–163, 166, 196, 202–204, 222, 232

VHDL Very High Speed Integrated Circuit Hardware Description Language: A widely used HDL. 59, 60, 75, 157, 168, 220

VLIW Very Long Instruction Word: A instruction set architecture focused on instruction level parallelism. The compiler groups instructions of a program which can be executed using parallel execution units of the given VLIW processor to accelerate the execution. 56

WRM2 Wastage Reducing Memory Mapping: A memory mapping algorithm introduced in this thesis for the comparison between different memory mapping strategies. More details in section 4.2.2. 22, 158, 159, 161–167

XML Extensible Markup Language: A markup language for hierarchically structured data in a format which is readable by machines and humans. 59

Chapter 1

Introduction

1.1 Motivation

The recent trends of big data, cloud computing, machine learning and race towards exascale computing call for significantly more powerful and energy efficient machines than currently available general-purpose computers. Furthermore, the unavoidable end of Moore's Law supports the need for highly heterogeneous systems which can outperform state-of-the-art Central Processing Units (CPUs) in both performance and energy efficiency.

This development has led to significant commercial and academic interest into alternative computing platforms. As a result, the usage of General Purpose Graphics Processing Units (GPGPUs) is, especially for Machine Learning (ML) workloads, becoming more and more common practice, alternative approaches make use of Application-Specific Integrated Circuits (ASICs), like Google's Tensor Processing Unit (TPU) [59], True North by IBM [97] and specialised coarse grained reconfigurable devices [141] or Field-Programmable Gate Arrays (FPGAs). The use of FPGAs is of special interest, since they promise improved power efficiency and performance compared to GPGPUs and especially CPUs [10, 27, 40, 46, 85, 96], while offering significantly better flexibility and a reduction in development costs by orders of magnitude in contrast to ASICs. Additionally, due to the flexibility offered by FPGAs they enable even smaller companies to differentiate their products from competitors and to highly optimise them to specific

use cases without the need to make the significant investments that would be needed for an ASIC based solution. This trend can be seen especially in the context of big data and ML.

For these reasons multiple industry vendors have started to support the use of FPGAs in a high-performance computing environment. One of the most notable efforts in this direction is the general availability of FPGA based cloud instances in the Amazon AWS EC2 cloud [8]. Other cloud vendors follow the same approach. Microsoft uses FPGAs in the Azure cloud [99], IBM offers SuperVessel in their OpenPOWER cloud [83] and Baidu also has FPGA based cloud offerings [147] Additionally, some server vendors like Dell sell servers with FPGA support directly for on premise use [162]. This development is also in line with the production of multiple dedicated FPGA accelerator cards for the data centre by Xilinx [158].

However, there is still no widespread adoption of FPGAs and while there have been some significant commercial deployments (e.g., at Microsoft [100]) in general they seem to be limited. The main challenge to FPGA adoption is the programming challenge. Traditionally most computers are based on a von Neumann architecture and operate on instructions. This trend is carried forward by Graphics Processing Units (GPUs) which have a different machine model than CPUs but still operate on instructions and have a fixed architecture. In contrast FPGAs do not have a fixed machine model and instead it is up to the designer to define the architecture and how an application is executed. This leads to a fundamental problem in developing a programming model suited to FPGAs.

Traditionally FPGAs were programmed using Hardware-Description Languages (HDLs) which are normally used to implemented integrated circuits and enable the designer to express spatial structures and contain a notion of time. However, these languages are complicated and not well suited for fast development cycles.

To mitigate these issues both academia and industry have tried to develop new development tools to target FPGAs. Some of them tried to apply programming models or programming languages which are normally used for CPUs or GPUs to FPGAs while others tried to develop more modern HDLs or adopted other programming models (e.g., dataflow) for FPGAs. However, often the design methodology and the development process used to program the FPGA

devices is overlooked. Due to their fine-grained control and highly deterministic behaviour, it is possible to predict application performance with very good accuracy, thereby enabling the employment of significantly different design methodologies as compared to the ones used to program conventional computing systems.

Conventional CPU development often uses an incremental approach, where a first implementation is quickly developed and subsequently is gradually improved step by step with the help of different profiling tools. However, there are multiple problems with applying this methodology to FPGA designs. It is very hard to profile an existing FPGA implementation, making it difficult to identify what specific part of the design needs improvement. This is especially true when the initial implementation cannot entirely fit onto the FPGA fabric, or does not reach the required frequency to highlight, e.g., bandwidth limitations. In addition, the identified improvements can easily require a complete redesign of the application especially if they change the way in which data are stored. Finally, the significant amount of time required to generate FPGA bitstreams makes any incremental process practically unusable.

The application of traditional CPU development to FPGA systems is therefore very likely to result in a suboptimal design, failure or delays. This problem is exacerbated by the fact that the majority of developers learn to program software first and thereby pick up the normal software development processes which they will automatically apply to FPGAs as well. The lack of a structured design process for FPGA is therefore one of the main contributing factors contributing to the programmability challenge of FPGAs and hinders especially novice developers.

An additional major factor to the slow adoption of FPGAs is the lack of portability between different devices. For CPUs and GPUs device vendors spent significant effort maintaining backwards compatibility so that existing programs can be executed on new hardware. In addition, one can usually see a performance increase on new devices without many changes as long as the application is not too heavily optimised for a very specific architecture.

These observations lead me to ask the following research questions:

Q1. Does a structured approach for accelerating HPC applications with reconfigurable platforms exist?

Q2. Is there an accurate method able to predict reconfigurable systems' performance prior to the creation of a synthesisable implementation of the reconfigurable sub-system?

Q3. Are there techniques to achieve state-of-the-art performance for a given application and reconfigurable platform?

Q4. If such techniques exist, can a single implementation target different reconfigurable platforms while delivering maximum performance on each?

Q5. What are the issues when the techniques mentioned above are applied to multi-die devices?

To address these questions this thesis proposes a design methodology tailored to FPGA based systems which aims to addresses the common challenges faced by FPGA developers while designing well-performing High-Performance Computing (HPC) applications. The process starts with the initial analysis of the application and its dataset properties. The information obtained is used to accurately model and predict performance and hardware requirements. An architecture which aims to maximise performance is developed based on these results. Overall, the process focuses on discovering potential performance bottlenecks early on and resolving those at the architecture design stage before the hardware implementation is started. This is achieved by careful analysis and prediction of computation and communication patterns throughout the complete system. Furthermore, the methodology contains best practices to support portability between different FPGA designs.

In order to use the methodology, one needs to have or be able to create a working CPU based implementation of the application that one wishes to accelerate using an FPGA. This implementation is important for some of the analysis stages and for debugging. It is for example used to verify that software and FPGA implementations created in the context of the methodology produce the expected output. If possible, a more abstract definition of the application is beneficial. The usage of a specific algorithm always limits the possible freedom for different hardware

implementations. If instead an abstract problem description is used it might be possible to use many different algorithms which might offer different trade-offs in the context of the hardware design. As such this abstract description is not a necessary input to the methodology but might yield additional benefits. The last required input is a representative set of input and output data. This dataset should provide a good representation for the usage of the application once it is deployed. It is used to analyse the dynamic behaviour of the application and test changes to the used algorithm and numeric implementation.

The methodology assumes that Maxeler's static dataflow machine model is used. This model tries to achieve high throughput by automatically generating deep uninterruptible pipelines from a dataflow-based programming model. By using this static execution model, the performance of the resulting FPGAs is highly predictable. This behaviour is used by the methodology to carry out large parts of the design process a-priori before the hardware implementation is started based on a performance model of the hardware design. This predictable performance motivates the development of a special design process tailored to static dataflow-based FPGA systems in order to achieve optimal usage of the available hardware and Input/Output (I/O) resources while keeping the development time and costs acceptable. Especially in the context of exascale computing it is crucial to minimise the amount of data transfers and their relative distances at all system levels so that computation can be performed as local as possible to remain within practically acceptable power budgets. Additionally, it is assumed that the target system consists of a FPGA and a CPU component.

The result of following the proposed methodology is an architecture for a system consisting of a FPGA component which accelerates parts of or a complete application together with a CPU based system hosting the FPGA. This architecture describes the components that have to be implemented on the FPGA and what role they should fulfil. Furthermore, the area requirements and the expected performance of these components are predicted by the performance model together with the required communication bandwidth and the required communication between all components of the system. While following the methodology a software model of the planned FPGA implementation is created which can be used for verification and debugging purposes.

While the methodology is developed around Maxeler's static dataflow technology generalisation to other technologies should be possible, even though it is not discussed in this work. The major requirement in order to apply the methodology to other toolflows is that the performance of the FPGA design is highly predictable.

One major issue faced in the development of any technology, tool or methodology targeted at improving developer productivity is the question how improvements can be evaluated. The only real way to measure productivity and therefor the success of the proposed methodology is to measure how long the development process takes using the methodology in contrast to other methodologies or an unstructured design process. However, one cannot use the same developer to implement the same application using these different processes one after another, since the experience made using one process will influence decisions and development speed on subsequent attempts. Similarly, it is not possible to use different developer since the personal experience and skill might differ significantly and these factors cannot be measured objectively. This would mean that many developers would be required to try the same process on the same application in the hope that the average development time is comparable. This problem is made worse by the fact that designing a complex FPGA application can take many months or even years. As a result, performing a study which accurately measures productivity improvements is very costly and beyond the scope of this thesis. Instead, I focus on measuring performance of the resulting design created by applying the methodology and compare the quality of the result to other designs. The focus is therefor on how good the output created by applying the methodology is in comparison to designs created using other methodologies. Productivity is only reported qualitatively.

1.2 Research Challenges and Contributions

The objective of this thesis is to *propose a development methodology and create the required tool support to enable developers to create state-of-the-art HPC FPGA designs and therewith facilitate wide FPGA adoption.* The focus is on heterogeneous computing systems consisting

of a CPU and FPGA sub-systems, where the FPGA is used as a powerful highly customisable coprocessor. As such the FPGA side of the system handles the time consuming and compute heavy parts of the targeted application, while the CPU performs all remaining parts of the application which might not map well onto FPGAs or will require prohibitive porting efforts.

In my research I discovered that the following three challenges are major obstacles towards large scale adoption of FPGAs in real HPC systems:

1. There exists no development methodology for the design of complex FPGA based systems which reduces the risk of unexpected delay in the development process and provides state-of-the-art performance. Current development methodologies for FPGA based system design rely on a large degree of automation or an iterative design approach. Methodologies relying on highly automated systems struggle to exploit the performance advantages of FPGAs and as a result cannot compete with other hardware platforms. Iterative approaches lead to a very long and risky design process due to the complexity of hardware design, long compilation times and the danger to discover major bottlenecks late in the design process leading to unexpected and often expensive redesign efforts.

2. The large differences between FPGAs even within the same device family prevent portability of hardware designs. Methodology and tool support are required to facilitate portability between different FPGA devices and to deal with emerging hardware features like highly heterogeneous memory architectures and multi-die systems in a package.

3. Current FPGA tools and design methodologies are often not evaluated on representative, complex applications. Instead, simple benchmarks are used, which are not sufficient to evaluate their usefulness in the development of complex applications. Only complex and real applications show the communication patterns that need to be addressed in order to deliver state-of-the-art overall speedup.

To address these major challenges the following three main contributions to reach the objective described at the beginning of this section are made:

1. The development of a methodology for FPGA design based around the accurate prediction of compute performance as well as on-chip area and bandwidth requirements for the interconnect. The methodology also entails the detailed analysis of the target application in terms of compute complexity, data movements and numeric properties. A software model is used as a testbed for numeric and algorithmic explorations as well as a debugging tool. The main target of the methodology is to provide a first-time-right approach in which Design Space Exploration (DSE) is performed before the first line of code targeting the hardware is written and the first hardware implementation manages to fulfil all design goals removing the need for complex design iterations in hardware. The entire methodology is published in [137] and an application developed using the methodology, achieving state-of-the-art performance, is published in [131].

2. Support for portability between different target platforms, which is achieved by first providing tool support for the usage of modern multi-die FPGAs and, second, the integration of methodology steps and best practices to facilitate portability. A memory mapping algorithm is developed to assist in the mapping of logical to physical memory resources. This automated approach solves the problem of mapping to heterogeneous memory resources found on modern FPGAs as well as the prevalence of multi-die FPGAs. The problem of multi-die FPGAs also has to be addressed in the context of portability between different FPGAs platforms. As a result, the methodology is extended accordingly and the portability and performance scalability across FPGA platforms from different vendors is evaluated using a common application from the financial industry. The memory mapping algorithm is published in [132] and the performance portability contribution is prepared for submission.

3. The developed methodology is demonstrated on a real, highly demanding application from radiotherapy. The dose accumulation as a result of radiotherapy is simulated using a Monte Carlo method. This simulation is needed to plan a treatment which limits the dose accumulated in healthy tissue. Modern radiotherapy machines enable the imaging of the patient at the same time as the treatment thus making adaptive radiotherapy [167,168] possible. The first real time dose accumulation simulation is presented which is a major

steppingstone towards facilitating adaptive radiotherapy. This application was developed using the methodology proposed in this thesis and makes use of the memory mapping algorithm as well as the portability methods presented. The radiotherapy simulation is published in [133] and an extended version is published in [138].

In the following three subsections a short overview of the individual contributions and the challenges involved is provided. More details can be found in the related chapters.

1.2.1 Methodology for Reconfigurable System Development

The first contribution of this thesis is *a methodology based on predictive analysis for first-time-right application development*. The contribution focuses on developing a structured design process based on the estimation of system performance to maximise the performance of the overall system according to research questions Q1, Q2 and Q3.

FPGAs present a promising alternative to purely CPU based or GPU accelerated systems in the context of HPC. This is motivated by their ability to implement massively parallel customised data paths. The ability to develop customised solutions for each encountered computing problem reduces overheads found in general purpose computing systems. However, adoption of FPGAs for HPC is still very limited. This is commonly attributed to implementation complexity, required expert knowledge and the resulting long development times and high development costs. While novel programming languages are designed to tackle these challenges there is still a lack of development methodologies tailored to FPGAs.

A design methodology to facilitate structured development of complex applications using reconfigurable hardware is proposed. The target is to enable first-time-right FPGA application development, which means that the initial hardware implementation fulfils all design goals, and no design iterations have to be performed in hardware. The methodology relies on analytical estimation of system performance and area utilisation for a given application and a decoupled controlflow/dataflow reconfigurable system. The targeted application is carefully examined,

and the parts intended for hardware acceleration are reimplemented as a representative software model. Next, with the results of the previous step a suitable system architecture is devised, and its performance is evaluated to determine all bottlenecks. The architecture is iteratively refined, until the final version satisfying the system requirements is obtained which can then be implemented. The methodology is validated based on the design of a widely used Convolutional Neural Network (CNN) (VGG-16) and the Berlin Quantum Chromodynamics (BQCD) application. In both cases it relieved and alleviated all system bottlenecks before the hardware implementation was started. As a consequence, the first implementations of the resulting architectures achieved state-of-the-art performance within 15% of the modelling predictions for these two applications.

The proposed methodology has already been adopted in industry and academia. For example, parts of it were used in [71] and it is used in two master thesis projects ([129] and one yet unpublished) as well as in multiple internal Maxeler projects. It is now also part of the user facing Maxeler documentation [94].

1.2.2 Extensions for Modern FPGAs and Performance Scalability

The second contribution of this thesis is *an extension to the proposed methodology to support portability between multiple target platforms using a unified code base including modern multi-die FPGAs*. It thereby addresses the research questions Q4 and Q5.

Another hindering factor in the adoption of FPGAs for HPC is the difficulty in upgrading from one device generation to the next and the even more complicated migration between FPGA vendors. While CPUs and GPUs sharing the same Instruction Set Architecture (ISA) provide functional portability between targets, changing the target of an FPGA design often involves significant effort. This is further exacerbated by recent trends in FPGA device organisation including the addition of heterogeneous memory architectures and multi-die systems.

To address the challenge of portability an algorithm for mapping logical to physical memory resources on FPGAs is proposed. This algorithm based on a greedy strategy is specifically

designed to facilitate timing closure on modern multi-die FPGAs for static-dataflow accelerators utilising most of the on-chip resources. The main objective of the proposed algorithm is to ensure that specific sub-parts of the design under consideration can fully reside within a single die to minimise inter-die communication. The above is achieved by performing stepwise memory mapping for each sub-part of the design separately while keeping allocation of the available physical resources balanced across the entire device. As a result, the number of inter-die connections is reduced which enables the place and route to successfully finish for all 33 evaluated test cases. The second-best performing algorithm failed on one third of the nine large applications tested to place and route the design. To my knowledge there exists no previous work which tackles the challenge of heterogeneous memory resources in multi-die FPGAs. It was integrated without further modifications into MaxCompiler and is used by default in all releases since version 2017.2.2. As such it was successfully applied to a wide range of commercial and academic projects.

To further address the challenge of portability design guidelines to assist with performance portability and scalability are proposed. To demonstrate the proposed techniques, a large-scale application used to price Asian options and originally developed for an Intel Stratix-V FPGA platform is ported to several new targets which are based on the Xilinx Virtex UltraScale+ generation. The accelerated application, developed in a high-level framework, is rapidly moved onto the new platforms with minimal changes. The original, unmodified kernel code delivers a 1.74x speed-up due to higher clock frequency achievable on the new platform. Subsequently, using the proposed methodology the application is further optimised to make use of the additional resources available on the larger Ultrascale+ FPGAs, guided by the simple analytical performance model of the proposed methodology. This results in an additional performance increase of up to 7.4x. The speedup normalised to the board capabilities is between 0.95x and 1.35x for all targeted platforms, showing that the capabilities of the platforms are well used. Using the presented framework, I demonstrate rapid deployment of the same application across a number of different platforms that leverage the same FPGA family but differ in their low-level implementation details, the available peripherals and the overall system organisation. To summarise, the same application code can be compiled to use five different platforms: a

Maxeler MAX5C Dataflow Engine (DFE), Amazon EC2 F1, Xilinx Alveo U200 and U250 as well as the original Intel Stratix-V based accelerator card, while efficiently using the hardware capabilities of all platforms.

1.2.3 Methodology Validation using a Real Application

The third contribution of this thesis is *the validation of the extended methodology using a real highly demanding medical application.*

Since measuring designer productivity is a major issue this thesis focuses on exploring if the methodology can be used to successfully develop a complicated HPC application for FPGAs. The performance of the resulting work is used to judge if the methodology helps in the development of well performing designs.

A novel reconfigurable hardware architecture to implement real time Monte Carlo based simulation of physical dose accumulation for intensity-modulated adaptive radiotherapy is proposed. This provides the first step in a long-term effort towards accurate online dose calculation in real time during patient treatment. Adaptive radiotherapy will allow wider adoption of highly personalised patient therapies which has the potential to greatly reduce radiation dose exposure of the patient as well as significantly shorter treatments and greatly reduce costs. The proposed architecture exploits the embarrassingly parallel nature of Monte Carlo simulations by performing domain decomposition and provides high resolution simulation without being limited by on-chip memory capacity. The architecture was developed using the methodology proposed in this thesis by creating a performance model and using it to derive the most promising architecture.

Following the guidelines of the methodology a performance model is created. This is especially challenging, since the number of simulation steps which have to be performed is not fixed but depend on random number generators. This issue provides the major motivation for the selection of this application to challenge the static nature of the performance model (Q2). By simulating typical runs on the CPU, it was possible to identify all the required parameters nec-

essary to build an accurate performance model. The performance model provides an estimate for the Block Random-Access Memory (BRAM) and Ultra Random-Access Memory (URAM) usage as created by the memory mapping algorithm presented in this thesis. As such this application provides a very good and challenging example to verify whether the creation of accurate performance models is feasible for complex real-life applications. Finally, the architecture is evaluated on a Maxeler MAX5C as well as the Xilinx Alveo U250 using the performance portability and scalability methods described in this thesis. An in-depth comparison between predictions and execution results is performed showing that the predicted performance is accurate for the most common configurations and the differences can be clearly explained in the remaining cases. Lessons learned from these discrepancies are discussed. The real time target of processing 100 million randomly generated particle histories per second was achieved using three MAX5Cs. Compared to previously published CPU and GPU implementation this offers a speedup of 4.1x and 8x respectively.

1.3 Thesis Organisation

The background of this thesis is described in **Chapter 2**. It provides an overview on reconfigurable technology, programming frameworks for FPGAs, existing methodologies for performance and area prediction as well as FPGA development and the applications implemented in this thesis. The first contribution is presented in **Chapter 3**. A methodology to develop complex applications on FPGAs is proposed which is based on accurate performance prediction of the final implementation. The methodology is applied to two application examples to evaluate its applicability to real life problems. In **Chapter 4** it is further extended to support modern FPGA devices and portability as well as performance scalability between different device generations. For this an algorithm is developed which automatically maps logical to physical memories on modern multi-die FPGAs. Novel steps in the methodology are developed to ease porting between different FPGA generations and especially when targeting modern multi-die devices. These steps are evaluated on a financial Asian option pricing application. The previous two contributions are applied together to a real, computationally demanding medical

application in the context of adaptive radiotherapy in **Chapter 5**. Using the methodology, a real time capable dose simulation is developed which might enable new treatment methods in the future. Lastly the conclusions and results as well as future research opportunities and a personal outlook are presented in **Chapter 6**. Fig. 1.1 shows how the individual contributions of this thesis link together. The chapters 3 and 4 together build a complete methodology for the acceleration of complex applications using the dataflow abstraction. The resulting methodology is then evaluated on a complex real application in chapter 5.

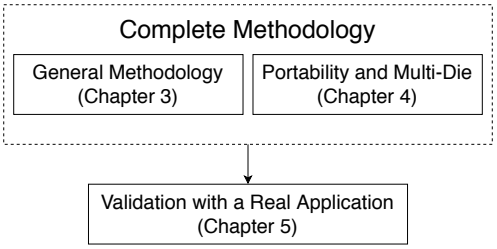

Figure 1.1: Thesis organisation.

Parts of this thesis have been published in [131–133,137,138]. Additionally, during the creation of this work multiple related papers have also been created. [134] describes the acceleration of the gzip compression algorithm using MaxJ and a comparison to a Register-transfer-level (RTL) and High-level Synthesis (HLS) based implementation. The productivity differences discussed in this paper motivated parts of this work and contributed to the background chapter. In [71] the acceleration of the Smith-Waterman algorithm is discussed. For the development of this paper parts of the methodology as described in this thesis were used. Finally, [135] describes an optimised Fast Fourier Transformation (FFT) implementation based on an area saving double buffering approach. This work is not discussed in this thesis but is a building block which can be used in a larger design built using the methodology.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter begins with a brief overview of FPGAs in section 2.2. Their applications, fundamental building blocks and recent trends are discussed. Afterwards section 2.3 discusses different frameworks for FPGA programming and discusses their advantages and disadvantages. Existing methodologies to predict area and performance of FPGA based systems are presented in section 2.4. Section 2.5 discusses FPGA development methodologies. The applications employed to evaluate the contributions of this thesis are presented in section 2.6 and finally section 2.7 provides a summary of this chapter.

2.2 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices able to implement arbitrary hardware circuits while also being reconfigurable. Initially one of the main use cases for FPGAs was the emulation and verification of ASICs. Since FPGAs can implement the same circuits as ASICs it is possible to emulate the ASIC design for testing and verification purposes, reducing design turnaround times significantly. Both tape out and complicated technology mapping are not required which makes design cheaper and faster.

However, the usage of FPGAs was quickly expanded to other use cases most notable networking, signal processing and more recently HPC. The main motivation for this is the massive parallelism supported by the FPGAs combined with the possibility to develop highly customised architectures for the intended use case. Compared to GPUs and especially CPUs, FPGAs support a higher level of parallelism at a lower clock frequency. They often achieve higher or at least comparable application-level performance with lower energy usage. In comparison to ASICs the performance and energy efficiency are often worse using the same technology node, however, the costs of ASICs production are very high. Additionally, the circuit implemented on an ASIC cannot be changed once it is produced, limiting the options for updating functionality later on. From a cost standpoint ASICs are usually only the best choice if enough units are deployed to offset the increased development costs through reduced unit costs or the benefits of improved performance and energy efficiency.

Modern FPGAs often follow a column-based architecture as shown in fig. 2.1. This is the case for the current FPGAs of both major chip vendors Intel and Xilinx. Within these columns different resources are present which can be grouped into three major classes:

1. Freely usable reconfigurable fabric;
2. connectivity to external devices;
3. and clocking infrastructure.

These resources are explained in more detail in the following subsections.

2.2.1 Reconfigurable Fabric

The reconfigurable fabric of the FPGA itself consists of three main user configurable classes. These are logic resources, memories and Digital Signal Processors (DSPs).

The logic resources are made up of Look-Up Tables (LUTs) and Flip-Flops (FFs). A LUT is a structure which is usually implemented as a memory itself. It has multiple inputs and

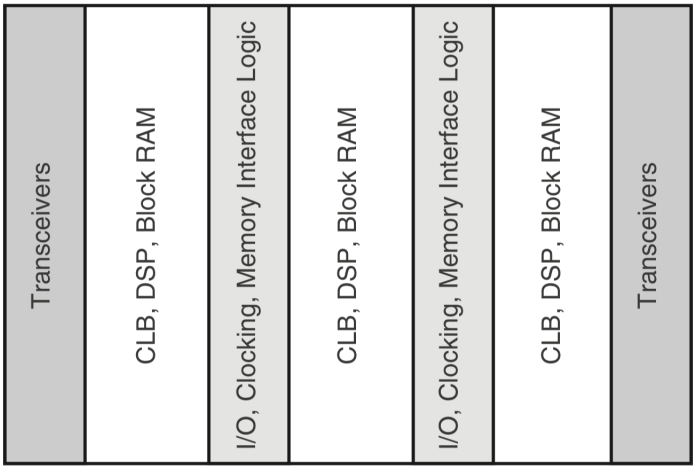

Figure 2.1: Column based architecture of the Xilinx Ultrascale FPGA family [153].

depending on these inputs an output value is created. The input bits form the address for the memory of the LUT and as a result for every combination of input signals an answer can be generated. This allows for the implementation of all possible logic and arithmetic functions. An FF can store a single bit of information based on the input signals read at a clock edge.

The second class of resources are dedicated on-chip memory resources. These resources have a limited number of bits and read and write ports to access the data within the memory. However, in most cases they have highly configurable aspect ratios, meaning that the relation between depth and width can be changed. For example, the same memory might be configured with an aspect ratio which has a width of 36 bits and a depth of 512 entries or a width of 18 bits and depth of 1024.

The last class of resources directly used by the user are the DSPs. These blocks usually consist of dedicated multipliers and additional units, e.g., pre- and post-adders. As a result, DSPs are less versatile than most other FPGA resources, but enable a significantly more efficient multiplication implementation than possible when using LUTs.

Besides these resources directly accessible by the programmer there are also routing resources, which are required to connect the different units in the reconfigurable fabric. These are usually hidden from the programmer and implemented using horizontal and vertical connections across the FPGA. At points where the wires cross each other switches can be used to configure the signal flow.

2.2.2 Connectivity to External Devices

FPGAs also have components dedicated to providing connectivity to other devices. In fact, many FPGAs are specially optimised for high connectivity for example to fit networking applications. While some of this I/O can be configured in more detail by the user, other blocks are preconfigured to support specific protocols or connection types, e.g., high speed networking, Double Data Rate Synchronous Dynamic Random-Access (DDR) memory or Peripheral Component Interconnect Express (PCIe) connections.

2.2.3 Clocking Infrastructure

Besides the normal routing resources most FPGAs also contain dedicated clocking networks. These are similar to the normal routing networks but are dedicated to be used specifically to distribute the FPGA clock across the chip. To complete the clocking infrastructure there are hardware units to generate different clocks based on a single base clock and to help with the clock distribution across the FPGA.

2.2.4 Trends in Modern FPGA Architecture

Apart from the fundamental building blocks described above a few recent trends in modern FPGAs deserve special attention. This section will specifically focus on changes which were introduced in the last three generations of Xilinx FPGAs.

Super Logic Region (SLR)

In order to increase chip area, while keeping yield and production costs in check Xilinx has introduced devices consisting of multiple die, called Super Logic Regions (SLRs), in recent FPGA generations. This is achieved by a technology which Xilinx calls Stacked Silicon Interconnect (SSI), where multiple FPGA die are mounted on a single silicon interposer [148].

As a result, the inter-die communication can only be performed using a limited number of wires on the silicon interposer. For example, the Xilinx VU9P has just above 20,000 inter-die connections in total. However, SLR crossings are only available between neighbouring SLRs. Additionally, the achievable frequency on SLR crossings is often limited especially if the signal is not carefully pipelined. This makes routing between different and especially nonadjacent SLRs challenging and requires special attention.

Memory Resources

With recent advancements in silicon technology the overall memory capacity of FPGAs has increased significantly faster than the availability of other resources. To achieve this chip vendors have introduced heterogeneous memory resources, meaning that more than one dedicated memory resource exists. The Xilinx UltraScale+ FPGAs contain three different physical memory types [154, 155]:

1. Distributed Random-Access Memory (RAM);
2. Block Random-Access Memory (BRAM);
3. Ultra Random-Access Memory (URAM).

Each of the many logic slice of the Xilinx Ultrascale+ FPGAs contains eight 6-input LUTs that can be used to construct a single 512 bit distributed RAM. In the Xilinx documentation this is referred to as SLICEM. Multiple SLICEM can be combined together to form deeper memories, however, this comes with a significant overhead. Individual SLICEMs can be configured to a multitude of different aspect ratios in terms of depth, width as well as the number of read and write ports.

BRAM modules are separate physical hardware memory units. Each BRAM of the UltraScale+ architecture can store up to 36 Kbits of data and can be used as one or two independent memory units. In both cases they consist of two read and two write ports and it is possible to configure different aspect ratios between depth and width. As an example, a 68 bit wide and 850 deep

single port logical memory would occupy two BRAM modules with an aspect ratio of 36x1,024. The number of aspect ratio options is further increased considering all supported read and write port combinations.

Finally, the URAMs represent an additional dedicated memory resource. One URAM module can store up to 288 Kbits of data but has only one single write and one read port. Additionally, URAMs can be used only as 72 bit wide and 4,096 deep memories. They only support a limited subset of the functionality supported by BRAMs and for example cannot be used to implement dual clock First In, First Outs (FIFOs). To summarise, URAMs are the least flexible from all available memory types but usually contribute most to the overall on-chip memory capacity of the Xilinx Ultrascale+ devices.

Logical To Physical Memory Mapping

The mapping of logical to physical resources is one of the basic problems in FPGA tool development. It describes the need to map structures defined by the designer, e.g., memories of an arbitrary size, to the physical hardware resources with limited possible sizes on the device. This often involves using multiple physical resources to implement a single logic resource. In the context of memories this problem is further exacerbated by the recent move to more heterogeneous on-chip memory resources. As such there is a multitude of research trying to address this problem, but it does not cover the changes to the memory architecture of modern FPGAs and the addition of SLRs.

In [49] the authors present an algorithm which maps logical memories to shared physical memories, by taking advantage of dual port functionality. The algorithm reduces resource wastage by letting two logical single port memories share the same physical dual port memory. Additionally, it tries to tile logical memories to use fewer physical memories. This means that different aspect ratios of the physical memories are combined to reduce the overall resource usage. For example, a platform might have physical memories supporting an aspect ratio of 36x512 and another aspect ratio of 18x1,024. A logical memory with a depth of 1,700 and width of 45 bits can be implemented most efficiently by combining memories configured with

both aspect changes (e.g., four memories of aspect ratio 36x512 and two with an aspect ratio of 16x1,024 in this example).

In [121] a technique to improve energy efficiency is presented. The power usage can be reduced by disabling the clock enable in cases where a memory is idle. Logical memories are allocated purely based on their depth. The proposed power optimisations is independent on the used mapping, since it is an optimisation on the hardware level.

An additional problem in the context of memory mapping is the presence of SLRs. Each SLR has only a limited memory capacity which in the case of the Xilinx Ultrascale+ architecture is split between BRAMs and URAMs. If a circuit that is supposed to be implemented in a single SLR for example only allocates BRAMs it might exceed the memory capacity of the BRAMs in that SLR which would lead to using BRAMs in another SLR. However, if that circuit would also use URAMs it would be possible to fully reside the circuit in a single SLR, avoiding SLR crossings and the resulting negative impact on timing closure.

To my knowledge there is no previous work on memory allocation, which takes SLRs into consideration, however, multi-chip partitioning algorithms like [25,92,113] have a similar optimisation target. These algorithms partition the complete design, which provides a possible solution to decide on which SLR a logical memory should be implemented. However, since logical memory resources first have to be mapped to a range of different physical memory resources, they are not directly applicable. Additionally, moving data between SLRs is a different problem compared to moving data between different FPGAs, since it still operates within the same design and only the impact on timing closure has to be considered. Bandwidth and especially latency is less of a concern.

2.3 Programming Frameworks

Since FPGAs were initially mostly used for ASIC emulation historically tools and programming languages used in ASIC design were also applied to FPGAs. This resulted in the widespread usage of dedicated HDLs. They enable a formal description of the hardware circuit under

design. One of their main features is the ability to express time. They do not have a strict programming or machine model in mind but instead allow you to describe a circuit in a well-defined and formal way. Section 2.3.1 will discuss HDLs in more detail.

The low designer productivity and steep learning curve of HDLs has led to large interest into interest into new programming languages and frameworks for more productive FPGA design. This trend has seen further acceleration due to the application of FPGAs to more general use cases. Companies and research institutes working in these fields do not normally work with hardware designers and would prefer to use development tools which behave similar to their normal software development environment. This also usually includes a desire to describe the behaviour of the algorithm to be implemented on the FPGA instead of the circuit used to implement it.

The major challenge to this endeavour is the high flexibility of FPGAs. In fact, one can easily make the argument that there is no machine model for FPGAs since the way in which data and the execution are organised is completely up to the designer. This lack of a machine model also leads to the inability to design a programming model around it which could be used by a programming language.

Over the years many tools and programming languages have been proposed to raise the abstraction level and improve designer productivity. One suggestion is the usage of overlay architectures, e.g., [110]. In this case the targeted FPGA can be configured with a predefined set of overlays. These overlays implement a given architecture and thereby create a machine model which can be abstracted by a programming model and as a result it is possible to construct a programming language which matches the problem. In many cases these overlay architectures construct a microprocessor on the FPGA which uses a Single Instruction Multiple Data (SIMD) or Very Long Instruction Word (VLIW) architecture. As a result, the overlay will always be at a disadvantage compared to devices which use a similar architecture but are directly implemented as integrated circuits. Furthermore, they work against the main advantage of FPGAs which is the opportunity to fully customise a design for a specific use case. For this reason, they are not considered in this thesis.

Another approach is the automatic generation of FPGA designs from domain-specific tools such as Matlab, Simulink or LabView. In this case a subset of the functionality can be synthesised into a hardware design. For different functions of the original programming language or environment highly optimised blocks are defined which only need to be connected. Similarly, there are Domain Specific Languages (DSLs) which create FPGA designs for a specific problem domain. An example of this is MaxGenFD [86] which targets finite difference computations. In both cases the problem of mapping an algorithm to FPGAs is reduced, since the functionality required is limited and a few predefined system architectures can be chosen which are known to map well to the problem domain. In the case of adopting domain specific tools for FPGA programming an existing programming model is reused and a sensible machine model is inferred. In the case of DSLs a machine model well suited to the targeted domain is used to construct a programming model and the DSL itself. Since these domain specific solutions cannot be used for general purpose computing in the context of HPC they are of no further interest in this thesis.

Another widely explored route is the extension of existing sequential languages to support FPGA designs. The target is to provide behavioural synthesis from a sequential algorithm description, which is also known as High-level Synthesis (HLS). In these cases, usually only a subset of the language is supported. Most importantly functionality like pointer arithmetic or dynamic memory allocation is usually not supported, since these techniques prevent a-priori knowledge about the required resources. Another common technique is to extend the syntax, e.g., through pragmas, to provide the developer with the opportunity to provide hints to the tool on how components of the algorithm are supposed to be mapped to hardware. For example, it is possible to define how many loop iterations are supposed to be unrolled and if parts of the design should be executed iteratively or in a pipelined fashion. This means that a programming model developed based on a von Neumann machine model is extended to target FPGAs. This is achieved by providing a set of architectural patterns and options as well as optimisations which can be manually targeted by the designer. HLS will be discussed in more detail in section 2.3.2.

Open Computing Language (OpenCL) also counts as an HLS tool. The major difference here is that the programming model of the original language used is not based on a pure von

Neumann machine model. Instead, OpenCL targeted both GPUs and CPUs from its inception and therefor its programming model has an inherent concept of parallelism. OpenCL uses compute units and processing elements to indicate that a single compute device can execute multiple functions (or kernels in OpenCL language) in parallel. However, fundamentally the programming model is based around a certain memory hierarchy and originally processing elements were assumed to be instruction based. Especially the memory hierarchy does not map very well to FPGAs. OpenCL is discussed in more detail in section 2.3.3.

A tool which deserves some attention is Sandpiper which was recently announced by Microsoft but is not release yet. It claims to be based on both a spatial and an imperative programming model and puts a lot of emphasis on the predictability of the generated hardware circuit. Since it is not released yet it is not possible to discuss the details of the used machine or programming model yet, but a discussion of the already known features can be found in section 2.3.4.

Another approach is the employment of a dataflow-based machine model. An example of this is Maxeler's MaxCompiler which is discussed in section 2.3.5. In this case Maxeler defined its own machine model which is based around static dataflow execution. The programming toolflow is then build around this machine model. The idea behind this approach is that it is easier possible to efficiently express parallelism in the resulting programming model and to map the resulting machine onto the FPGA hardware. Maxeler's toolchain is discussed in great detail in section 2.3.5. Section 2.3.6 compares the different approaches and motivates the usage of Maxeler's toolchain in this thesis.

2.3.1 Hardware Description Languages

The traditional way to design most hardware circuits, including FPGA designs, is based on the usage of HDLs. These languages operate at the Register-transfer-level (RTL). A circuit is described as registers and the combinational logic between these registers. As such the RTL provides a higher abstraction level compared to the logic gate or even more detailed transistor level. As a result, the usage of HDLs delivered a significant improvement in design productivity compared to the previous description of circuits at an even lower level. However, HDLs still

provide a significantly lower abstraction level than most commonly used software languages, significantly limiting designer productivity.

The most commonly used HDLs are Verilog and Very High Speed Integrated Circuit Hardware Description Language (VHDL). VHDL was initially developed to document hardware circuits but was quickly adopted for hardware simulation and verification. Similarly, Verilog was initially developed to facilitate hardware circuit simulation and verification. Both languages contain a synthesizable language subset, which can be used to directly describe hardware circuits on the RTL. As a result, most FPGA synthesis toolchains accept VHDL and Verilog as input.

Chisel is a Scala based hardware description language. The concept behind Chisel is to add modern programming language features to a hardware description language. It is therefore a fundamental different approach to HLS in that one still describes the hardware circuit and not the behaviour of the algorithm. Design is still low level, but the goal is to improve productivity by supporting high-level abstractions in the language [14].

A problem of HDLs is that they often contain FPGA device and vendor specific extensions, Intellectual Property (IP) cores and constraints. The authors of [69] try to mitigate this problem. They propose the usage of user configurable virtual platform interfaces. This enables a user to use a platform independent interface to create a memory while a middleware handles the implementation on a given hardware platform. The approach is limited to memories in the presented work, but the same approach could be extended to other hardware resources. The same authors further develop their contribution in [68] by providing a complete development environment which also supports external FPGA interfaces. This tool flow is based on an application description in Extensible Markup Language (XML) and HDL and automatically maps the application to individual platform resources. The authors focus on portability and not on achieving the best performance on each individual device.

2.3.2 High-Level Synthesis

To mitigate the productivity challenges of HDLs and make hardware design more accessible to software developers many High-level Synthesis (HLS) tools have been developed. These typically attempt to create FPGA designs from conventional programming languages such as C and often require some form of manual intervention in the transformation process. These tools have seen a rapid adoption across industry and academia in recent years. While the productivity advancements are undeniable, there is still an ongoing debate on the quality of results compared to HDL designs. On of the main reasons for this perceived disadvantage compared to HDL is caused by the imperative, sequential programming model used by these tools. All notions of parallelism are added as additional manual hints to the toolchain and implementation options are limited to some predefined design patterns, architectures and optimisations. This section aims to give an overview of commonly used HLS languages and tools.

Vivado HLS is a tool developed by Xilinx. It accepts C, C++ and System-C as inputs and supports arbitrary precision data types. Xilinx claims a 4x speed up in development time and a 0.7x to 1.2x improvement for the quality of result compared to traditional RTL design [160]. However, Vivado HLS is not intended as a simple push-button C-to-FPGA synthesis tool and requires various manual transformations to customise the hardware architecture and achieve well performing designs. The programmer is expected to annotate the source code, usually using pragmas, to direct the tool. These pragmas for example control the unrolling of loops or the pipelining method.

Catapult C creates FPGA and ASIC designs from American National Standards Institute (ANSI) C/C++ and System-C descriptions [1]. Similar to other HLS tools, it requires the designer to perform iterations on the original C-code and manually tweak the hardware architecture in order to achieve a fast implementation.

ROCCC is a C to VHDL compiler designed without extending the C syntax. ROCCC is also independent of specific FPGA platforms. A productivity gain of 15x is claimed in [130].

HeteroCL [79] presents an approach to separate the algorithmic parts from platform-specific

optimisation. The target of the proposed language is to separate individual parts of an FPGA design, e.g., algorithm description, quantisation and memory optimisations from each other. First the algorithm is described in sequential python through usage of the HeteroCL Application Programming Interface (API). Afterwards HeteroCL enables the user to customise the datatypes of variables used before, customise the memory, instruct the compiler how to implement computational functions and which architecture to use. For example, it is possible to define the usage of a systolic array or a stencil.

IBM's *liquid metal* attempts to target heterogenous computer architectures by using a single language to program CPUs, GPUs and FPGAs. It uses a Java based language called *lime*. *lime* is an extension of Java which can be executed on the normal JVM. It was first developed to accommodate FPGA and CPU design and as such Java was extended by additional keywords and abstractions to enable better mapping to FPGAs. These extensions for example include the addition of fixed sized arrays, a support for datatypes with an arbitrary number of bits, the concept of local and global functions and a task-based programming model. The type of hardware the application runs on gets chosen at runtime based on available capacities in the data centre [12, 52].

The Barcelona Supercomputing Center proposes the usage of *OmpSs@FPGA* for the programming of FPGAs in heterogeneous HPC systems. It is based around the OmpSs programming model which uses task-based parallelism and is used as a testbed for future OpenMP features. OmpSs@FPGA uses VivadoHLS to create the FPGA configuration. Its major contribution is the ability to program host and FPGA source code together and describe communication between both using directives. The Nanos++ runtime is used to automatically resolve data dependencies and provides the ability to schedule the same task either on the FPGA accelerator or the CPU [17]. It is also possible to integrate FPGA configurations generated by other tools, e.g., Maxeler's MaxCompiler, which will be discussed in more detail in section 2.3.5 [43]. In that case the FPGA component will be programmed using Maxeler's toolflow and only the integration with the host code is handled by OmpSs.

2.3.3 OpenCL

OpenCL is a standard that aims at providing a single API to target different heterogeneous computing platforms with a special focus on parallelisation and allows a programmer to target different hardware platforms and instruction sets with the same code. Initially it mostly targeted GPUs and multi-core CPUs and was later extended to Digital Signal Processors and FPGAs. OpenCL is a widely used HLS tool and will be discussed in more detail here. While OpenCL does not guarantee optimal performance for the same code on all hardware platforms it does guarantee correct functionality (if no vendor specific extensions are used) [120].

OpenCL uses a machine model in which a compute device, e.g., a single FPGA accelerator card, consists of multiple compute units which can implement multiple processing elements. This allows for a task parallelism in the programming model where a kernel can be executed on each of the processing elements. A scheduler distributes the computing tasks to the processing elements, potentially even on multiple compute devices. Additionally, OpenCL supports data parallelism through vectorised data types.

The machine model also contains a model for the memory of the compute device. Fundamentally it assumes that the memory on the host and the compute device a separate and before any computation can be executed the data has to be copied from the host memory in the global device memory. This global device memory would for example be implemented through DDR memory on the FPGA card. Additionally, there is a concept of constant, local and private memory. Constant memory is global read-only memory, local memory can be potentially accessed by all processing elements in the same compute unit and private memory is only used by a single processing element.

This means that OpenCL has a fundamental different programming model compared to the HLS tools described in the last section. It has for example inherent concepts of parallelism. The disadvantage of this solution is that the machine and programming model is very much tailored to GPUs and significant tool effort is required to provide a mapping to FPGAs. This also motivates the need for pragma extensions to OpenCL to, for example, define the unrolling

of loops. The support for FPGA programming using OpenCL has been added by both Xilinx and Altera (now Intel). This means that both major FPGA vendors have released an OpenCL Software Development Kit (SDK) for FPGAs [4, 119, 159].

The Intel OpenCL compiler supports the core OpenCL 1.0 features as well as extensions, which for example support streaming of data from an Ethernet interface to a compute kernel. It also provides an emulator for functional verification of the created designs in order to speed up the development time. In addition, a detailed optimisation report and a profiler are provided to allow easier development of more efficient designs.

Xilinx provides *SDAccel* which is a programming environment for OpenCL, C and C++. Additionally to the compiler, it also contains a simulator and profiling tools. Xilinx claims to achieve up to 20% better results than with hand-coded RTL designs and 3x better performance and resource efficiency compared to OpenCL solutions from competitors. SDAccel also supports partial runtime reconfiguration on the FPGAs without halting any other computations running on the chip [159].

2.3.4 Sandpiper

Sandpiper is the name of a FPGA programming tool developed by Microsoft. They plan to open source it in the close future and refer to it as a Mid-level Synthesis (MLS) tool [19]. The main idea is to provide a language which is more appealing to software engineers, but not as far detached from the hardware as HLS. This means that the language has to be expressive and imperative but also has to lead to predictable performance. Additionally, they aim to be target independent. Similar to HLS they want to include integration with software development tools and platforms and fast, accurate simulators. The analogy used by Microsoft is that MLS should be similar to C, which hides a lot of the low-level hardware details still present in assembly but exposes the fundamental hardware characteristics as necessary.

To accomplish this Microsoft's Sandpiper language uses a spatial and imperative programming model with C style syntax built around the concept of pipelining. They make parallelism

and memory access, including different types of memory, explicit which leads to a predictable performance. While they admit that more lines of code are required compared to HLS, they still claim a significant productivity improvement over HDLs. On the other hand, the predictable performance and ability to express parallelism in more detail, targeting more versatile architectures and designs, provides a significant advantage over HLS.

2.3.5 MaxCompiler

MaxCompiler is a tool developed by Maxeler Technologies and follows the Open Spatial Programming Language (OpenSPL) standard [3]. The machine model used by MaxCompiler is based on dataflow and systolic array concepts [35,76]. MaxCompiler can be seen as a newer version of the ASC compiler [95] and uses very similar abstractions and concepts. Since the designs of this thesis are implemented using the Maxeler toolchain and extensions to this toolchain are presented as well, this section will focus on this toolchain. First the dataflow and systolic array concepts are introduced and then the Maxeler toolchain will be discussed in detail.

Dataflow

The commonly used von Neumann machine model is not well suited to describe explicit parallelism. Instead, in order to achieve parallelism, in most cases von Neumann machines are replicated so that they can work on multiple execution threads at the same time. To overcome these limitations the dataflow model was proposed. Here a program is represented in the form of a directed graph in which nodes represent operations and the edges the communication between the operations. Each node can execute if all inputs are ready and generates its output as a result. Those outputs can then enable further nodes to be executed. If the inputs to multiple nodes are ready, they can all be executed in parallel. As such the dataflow execution model provides a natural way to describe massive parallelism [11, 33, 34, 57, 60, 61, 73].

The usage of the dataflow model has seen only very limited adoption outside of academia. The main reason for this is that there are very few devices which were built around a dataflow model.

One example for such a device is the dataflow processor presented in [36] which allows for the native execution of dataflow programs. Similarly, a FPGA softcore based on dataflow has been proposed in [107]. The authors of [44] argue that dataflow programming languages could also be used to assist with GPU programming. They motivate this by detailing similarities in the restrictions in terms of shared state memory access and communication between the dataflow model and current GPU architectures.

Systolic Arrays

Systolic arrays were first introduced for some matrix operations in [77] and further developed in [76]. They describe an architecture developed for the design of integrated circuits. The author notes that there are three main problems faced in the context of hardware design. These are the high complexity involved, the need to use parallelism since the gate speed does not increase as fast as transistor density and that I/O bandwidth does not increase as fast as computational performance. The last problem means that the number of bytes loaded per operation has to be reduced in order to scale computational performance. Systolic arrays try to address all these issues.

Systolic arrays consist of multiple cells which are connected to each other. Each cell performs a simple operation and I/O cells on the edge of the array handle the communication with elements outside of the array. This design means, that an array is built from small regular structures with a simple communication and control network, addressing the first identified issue. The communication between the cells is implemented in a pipelined fashion. Data is read by I/O cells and flows through the centrally clocked systolic array where the cells perform their individual operations on the way. This means that cells are operating in parallel addressing the second issue. The last issue can be addressed by systolic arrays by increasing the number of cells in the array. Since each cell performs one operation on the data flowing through it the larger the number of cells in the array the higher the rate of operations per byte read.

The Maxeler System Architecture

Fig. 2.2 shows the general system architecture of the computer systems targeted by Maxeler. The system always consists of a controlflow and a dataflow component. This means that a CPU based host is always working in conjunction with a dataflow component to facilitate the execution of a program.

The dataflow component is realised through what Maxeler calls a Dataflow Engine (DFE). A DFE hosts a dataflow processor which is currently realised using FPGA technology but could also use a dedicated integrated circuit in the future. Additionally, it has access to large amounts of memory which are not part of the processor. Maxeler calls this memory Large Memory (LMEM) and on current DFEs it is implemented using DDR memory. Additionally, the DFE has multiple interfaces to communicate with other devices. First, a DFE has a PCIe connection to connect to the host. Second, a DFE can have MaxRing which is a proprietary interconnect to connect multiple DFE and lastly a DFE can have network interfaces. While the PCIe interface is mandatory the usage of MaxRing and network interfaces is optional.

The host CPU executes a CPU application which can communicate to the DFE using Maxeler's Simple Live CPU Interface (SLiC) runtime library. Additionally, MaxelerOS[1] is running on the host machine which consists of the driver and some additional tools for DFE management. The driver can than communicate with the DFE over PCIe. A Direct Memory Access (DMA) data transfer from the host memory to the DFE is possible. The dataflow program is executed on the dataflow processor of the DFE.

The Maxeler Machine Model

This section introduces the machine model used for the dataflow processor. On a high level the dataflow processor follows the dataflow concept. All interconnects are represented as data producing or consuming nodes in the dataflow graph. Additionally, other nodes can represent complex operations that are executed on the data streamed through the dataflow graph.

[1]Despite the name this is no full operation system. It is a module installed on the host operating system, which is usually CentOS.

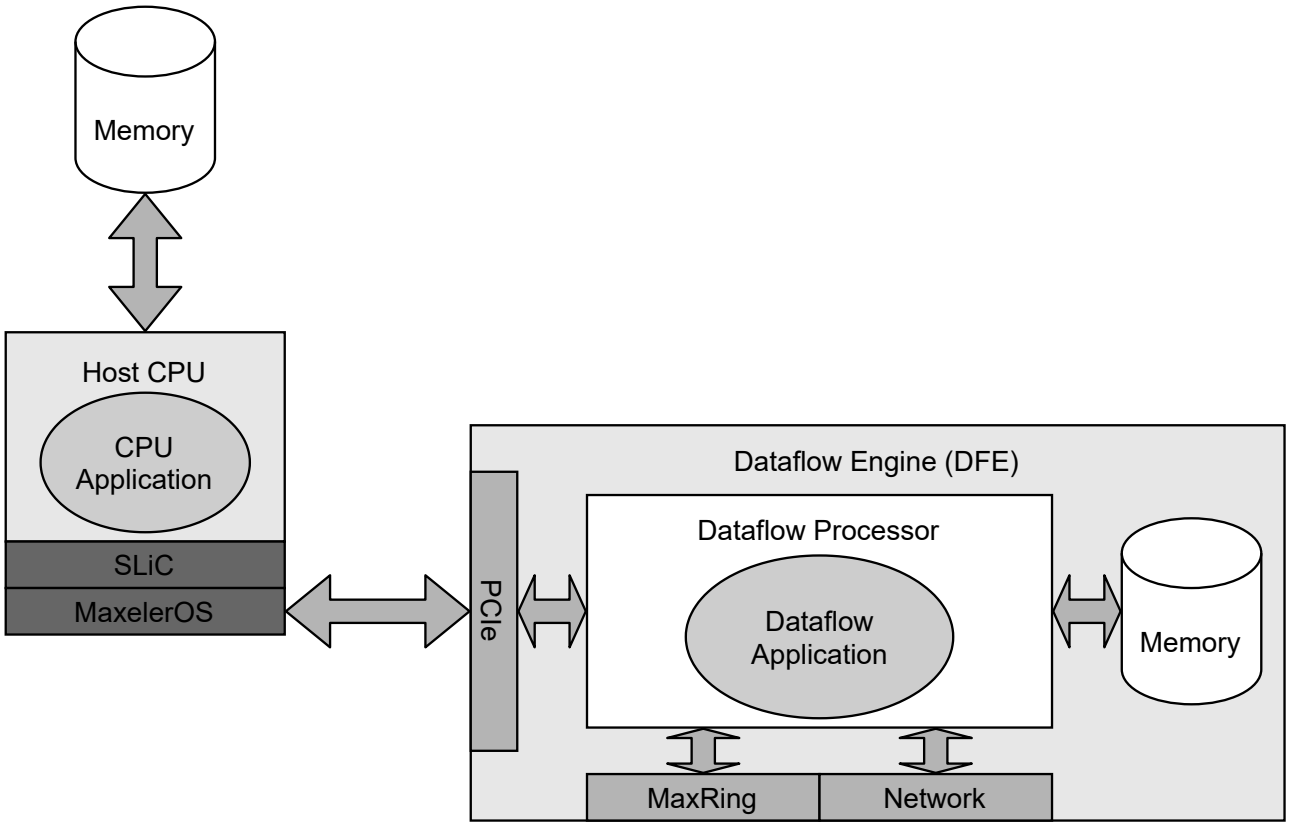

Figure 2.2: Maxeler Computer System Architecture.

As discussed above in the typical dataflow concept nodes only execute when all their inputs have data ready. This is not always the case in the Maxeler machine model. The nodes in this dataflow graph can define which inputs and outputs are currently active to decide if they should execute. Additionally, they can be configured to always accept or send data independently of other execution requirements. This can lead to data loss if a node cannot keep up processing, but this functionality was added to the machine model to support network processing at a constant bandwidth. The resulting execution is fully driven by the presence of data and the configuration of the nodes. Each input or output of a node in the graph can operate on its own clock and at maximum of one datum can be consumed or created per input/output per clock cycle. Different inputs and outputs as well as nodes can share the same clock. It is asynchronous and not explicitly scheduled.

As mentioned above the nodes in this high-level dataflow graph can implement complex operations. Each node can implement a complete functional block with multiple inputs and outputs. Most of the algorithm is supposed to be implemented in a systolic array style execution logic

which would be represented by a single node in this high-level dataflow graph. The systolic array is very deeply pipelined and has a non-interruptible pipeline. It cannot only consist of logical or arithmetic operations but also contains memory blocks to delay or store data within the systolic array. This leads to highly predictable timing characteristics in the context of mapping the resulting structure to the FPGA. It is also possible to define blocks following different design patterns within each node of the high-level dataflow graph. All these nodes can make use of arithmetic and logic functions as well as memory but have to interact with the high-level dataflow graph.

The Maxeler Programming Model

The programming model used by MaxCompiler is based on this machine model. The high-level dataflow graph is represented by a *Manager*. In the Manager the I/O can be configured. The PCIe connection is implemented as incoming FIFOs for data being sent from the host and as outgoing FIFOs for the connections from the DFE to the host. Similarly, MaxRing is implemented based on a FIFO in each direction between the different communication partners. The network connections are also implemented using FIFOs containing the networking packets. The packets are split over multiple cycles and on each cycle new data of a packet arrives.

The last type of I/O node in the Manager handles the communication with LMEM. This node implements a memory controller which Maxeler calls *MCP*. The MCP can handle multiple read and write connections to a block of on-board memory. Each stream consists of two connections which are again implemented using FIFOs. One connection is used to transport the actual data to or from the memory. The width of this FIFO is as wide as the used memory interface. The second connection is used to send a command stream to the MCP. These commands consist of the physical addresses of the on-board memory that are supposed to be accessed and additional commands, e.g., the number of memory words that should be accessed. The MCP does Round-Robin arbitration between the different connections to the memory to make sure no single connection starves. A memory command is only executed if the execution can be atomic. This means that there has to be enough data in the FIFO for write commands and enough space

for read commands. There is no automatic handling of read before write or write before read conflicts, but it is possible to get status streams from the MCP containing notifications of the executed commands.

Additionally, to the I/O nodes the Manager also contains compute nodes. The most commonly used compute node is called a *Kernel*. A Kernel is itself represented by a dataflow graph, which basically means that a node in the dataflow graph of the Manager can contain a complete dataflow graph of itself. The dataflow graph of the Kernel can contain multiple inputs and multiple outputs. Each input or output can be disabled. These I/O nodes interact with the Manager dataflow graph. The other nodes in the dataflow graph implement operations or memories. It is possible to map memories in the Kernel to the host, so data can be written into or read directly from nodes in the Kernel dataflow graph. The nodes in the dataflow graph are connected by edges, which have a data type attached. As a result, the types of all operations are well defined. It is important to stress that the dataflow Kernel used by Maxeler is in scope and usage fundamentally different to the kernel used in OpenCL or the more common term of a computational kernel as often used in software development.

The execution of the Kernel dataflow graph is fully synchronous. This means that if an active input has not incoming data or an active output cannot forward data to the Manager the complete execution stalls. As such not the individual nodes of the Kernel dataflow graph decide about the execution but the node representing the Kernel on the Manager level. This also means that the complete Kernel uses the same clock. Following the programming model, a very deep pipeline from the Kernel dataflow graph is built. This pipeline is not interruptible, so data send in is always processed to the end. Additionally, the pipeline always gets completely filled and there is no concept of pipeline bubbles.

The programming model uses the concept of ticks to abstract from this pipeline. The program presents itself to the user as if everything would happen on a single tick. This means that each individual pipeline stage operates on a different tick once the execution in hardware is started.

To facilitate this static execution pattern the dataflow graph is scheduled. Each node in the dataflow graph has a latency, which describes how many cycles are required to execute its

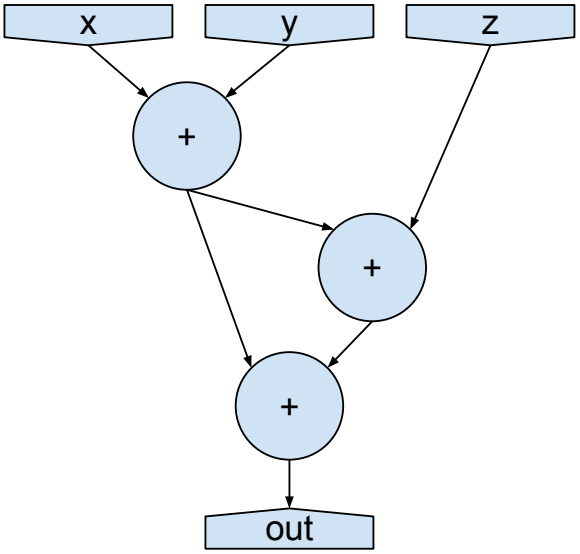

Figure 2.3: Unscheduled dataflow graph.

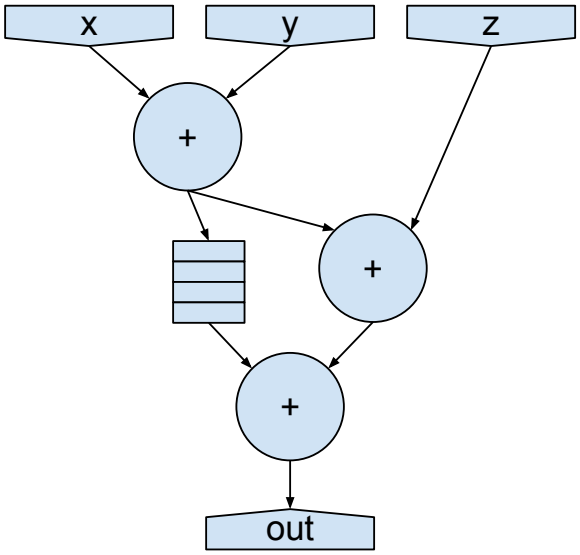

Figure 2.4: Scheduled dataflow graph to enable pipelining.

operation. The scheduler uses these latencies to insert FIFOs as required to ensure that data arrives at the dataflow nodes at the correct time. Fig. 2.3 and fig. 2.4 show how an unscheduled dataflow graph needs to be scheduled to facilitate correct execution once it gets pipelined. Additionally, it is possible to manually add scheduling constraints to access data from previous or future ticks. This allows the implementation of loops in the dataflow graph.

Kernels also have a concept of filling and flushing of the computational pipeline. A fill signal is used to indicate which nodes become active on which cycle as the Kernel gets executed for the first time. This might be important to ensure that, for example, an accumulator only operates on valid data. Similarly, a flush cycle is used to deactivate nodes as the Kernel execution ends.

There are fundamentally three ways to decide when to finish the execution. The first option is to set the number of ticks the Kernel is supposed to execute from the CPU. The second option is to generate a signal within the dataflow graph of the Kernel which is used to trigger the flushing of the Kernel pipeline. Finally, it is possible to disable the flushing logic altogether and keep the Kernel running indefinitely. This is for example often used in network-based applications or the inputs and outputs are disabled at some point by the Kernel which results in a flush like behaviour.

A KernelLite is a more reduced version of the normal Kernel. The main difference is that it does

not match the stalling pattern of a normal Kernel. By default, all operations in the dataflow graph of the KernelLite operate on every cycle. It is possible to stall parts of the KernelLite based on control signals generated in the KernelLite itself.

Finally, ManagerStateMachines provide an abstraction to the programmer which do not map to systolic arrays. They are based on finite state machines but also have the ability to contain memories and some arithmetic operations. All registering, pipelining and scheduling has to be done manually by the programmer.

In this thesis I will mostly use Kernels and in some rare cases ManagerStateMachines. KernelLites are not used. The reason for this is that ManagerStateMachines and KernelLites are mostly used in the context of networking, while this thesis focuses on HPC.

Mapping to FPGAs

After discussing the machine and programming model of MaxJ it is important to consider how these elements are mapped onto the actual hardware to understand design decisions made later on. The Manager is mostly a structure to organise other components. The I/O interfaces will be implemented as IP cores. They usually consist of the normal vendor specific IP cores and some additional control logic developed by Maxeler to integrate the interfaces into remaining toolflow and for example add the DMA engine talking to the driver on the host in the case of PCIe or the MCP in the case of LMEM. The connections between the different nodes in the dataflow graph of the Manager are implemented using FIFOs which will use on-chip memory.

The different nodes in the dataflow graph of the Kernel will be mapped to the appropriate components of the FPGA. This means that memories will be mapped to on-chip memory resources, multiplications to DSPs and the remainder to LUTs. The edges of the dataflow graph of the Kernel are implemented using wires and where FIFOs are inserted again on-chip memory is used. One very important property of this mapping is that the timing properties of Kernels are highly predictable, and Kernels of similar size will always meet timing closure at similar frequencies on the same FPGA device. This is a direct result of the deeply pipelined systolic

array architecture used to implement the Kernel. The mapping for KernelLites behaves in the same way and ManagerStateMachines are similar, but timing characteristics are a bit less predictable. However, ManagerStateMachines are usually very small, so that simple optimisations for the critical path can usually resolve all timing issues.

It is important to note here that there is a direct mapping from every component in the programming model through the machine model to the FPGA hardware. This makes the design of FPGA designs using MaxJ highly predictable.

The Maxeler Programming Environment

Maxeler developed the MaxJ programming language which is used to program DFEs using the programming model described above. MaxJ is a Java based meta language. This means that it uses normal Java syntax but has a few additional keywords and added operator overloading. MaxJ needs a modified compiler to be compiled but can be executed on a normal JVM.

MaxJ makes heavy use of metaprogramming. The program written in MaxJ is not mapped to the FPGA. Instead MaxJ is used to create a program which creates the structures of the Maxeler programming model. This means for example that the MaxJ code generating a Kernel describes a program to generate the dataflow graph of this Kernel. As a result, normal Java structures like for loops and if-else blocks are all evaluated at the compile time of the FPGA configuration. In the case of loops the code generating parts of the dataflow graph in the loop body is executed multiple times, resulting in the same subgraph being added to the dataflow graph multiple times. Similarly, in if-else structures only the branch taken is executed and only the subgraph generated in this branch is added to the dataflow graph.

This is achieved by calling API provided by MaxCompiler. Only calls using this API generate any kind of hardware structure. As a result, Max*Compiler* is a very confusing name, since it is only a library which is used to generate code and automate the FPGA vendor toolflow. The imperative and sequential MaxJ programming language is used to generate highly parallel structures and MaxCompiler does not have to infer parallelism automatically. All decisions

relating to parallelism, pipelining or optimisation techniques are either explicitly made by the programmer or defined by the programming model.

An overview of the complete MaxCompiler toolflow is provided in fig. 2.5. The dataflow program is written in MaxJ to define the Manager and Kernels (and ManagerStateMachines as well as KernelLites as required) using the MaxCompiler API. The resulting Java bytecode is executed to generate the dataflow graphs. MaxCompiler then maps the resulting dataflow graphs to VHDL and starts the FPGA vendor tool to generate a FPGA bitstream. It is possible to configure the vendor tools from the MaxJ code in the Manager. This mostly provides directives to the place and route algorithm to assist with timing closure. The resulting bitstream is packed into a MAX file which also contains some additional information on the content of the bitstream and how the software can integrate with it. It is also possible to create a simulated version of the DFE instead of the bitstream. This is a cycle accurate simulator which runs on normal CPUs and is supposed to help with debugging. While it is faster than an RTL based simulation, it is not optimised for performance and will usually be significantly slower than a normal sequential implementation of the same functionality.

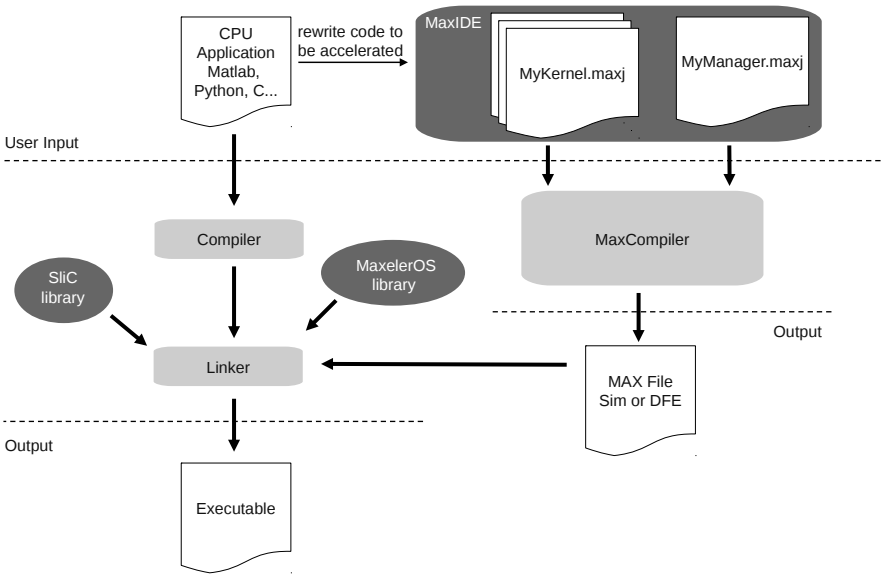

Figure 2.5: MaxJ toolflow [93].

To ease the design and especially optimisation of applications MaxCompiler generates multiple reports showing how the dataflow graph is mapped into hardware. An example for this is the resource annotation, which annotates the resource usage for each line of MaxJ source code, and

the timing report, which highlights timing violations.

The CPU application running on the host uses the SLiC library to interact with the dataflow design. SLiC offers functions to, e.g., initiate PCIe data transfers or access shared memories. Additionally, it has functions for card management, e.g., to allocate a DFE or load a bitstream on it. The SLiC library has a C API but there are also tools to generate interfaces from other programming languages like Python or Matlab.

The CPU application is statically linked with SLiC and the maxfile. Additionally, there is also a shared MaxelerOS library which facilitates the low-lever interaction with the driver. MaxelerOS also provides a demon monitoring card status and tools for debugging and card management. It is no full operating system but a tool which can be installed on CentOS based operating systems. The compiled CPU binary can be executed on any system which has MaxelerOS and DFEs installed.

Dataflow Engines

Five examples for different DFEs are Maxeler's MAX4C and MAX5C DFEs, the Amazon EC2 F1 instance and the Xilinx Alveo U200 and U250 cards. The MAX4C and MAX5C DFEs are developed and sold by Maxeler, while the Amazon EC2 F1 instance is a cloud instance which is available within the AWS cloud [8]. It follows the same system architecture as Maxeler's DFEs and they are fully supported by Maxeler's toolchain. The Xilinx Alveo cards are FPGA cards developed by Xilinx to target the data centre market [158]. Currently there are the U50, U200, U250 and U280 which differ in the used FPGA and the resulting capabilities. All of these cards use the same system architecture as used by Maxeler and the U200 and U250 are both fully supported by the Maxeler tools. The MAX4C DFE is based on the Intel Stratix-V 5SGSD8 FPGA, which is a 28nm device, while all remaining platforms use the Xilinx Ultrascale+ FPGAs realised in 16nm. For the MAX5C and F1 this is a VU9P, for the U200 an XC200 (equivalent to a VU9P) and for the U250 card an XC250 (equivalent to a VU13P).

The FPGA of the MAX4C DFE has 262,400 Adaptive Logic Modules (ALMs), 1,963 27x27

bit multipliers and 50 MBit of on-chip memory. The card uses PCIe Gen2 x8 interface to the host and has six on card 8GB DDR3 Dual In-Line Memory Modules (DIMMs). The Xilinx VU9P FPGA provides 1,182,240 LUTs, twice as many FFs, 6,840 27x18 bit multipliers and 341 MBit of on-chip memory in the form of two different memory resources. On the MAX5C DFE and the Alveo U200 and U250 cards the PCIe interface uses Gen2 x8 even though Gen3 x16 is physically available, while the F1 instance always uses PCIe Gen3 x16. The limitations to Gen2 x8 is due to current limitations in Maxeler's tools. The MAX5C DFE uses three on-card 16 GB DDR4 DIMMs, while the Alveo U200 card and the F1 instance both use four on-card 16 GB DDR4 DIMMs. On the F1 instance, not all resources are available to the user, since roughly 30% of the overall on-chip resources are reserved for the AWS shell which manages the card and provides access to I/O interfaces. The MAX4C, MAX5C and Alveos are used bare metal (i.e., with no shell) in order to fully utilise the available resources. The I/O capabilities of the Alveo U200 and U250 are identical, however, the U250 offers 1,728,000 LUTs, 12,288 27x18 bit multipliers and 455 MBit of on-chip memory. The Xilinx VU9P FPGA (as used in the MAX5C, F1 and U200) consists of three separate die also called SLRs, while the VU13P (as used in the U250) has four SLRs. An overview of all cards is provided in tab. 2.1.

Table 2.1: Overview of the platforms used in this thesis

	LUTs	DSPs	SRAM	PCIe	DDR	FPGA
MAX4C	262,400	1,963	50 Mbit	Gen2 x8	6x8GB DDR3	Stratix V
MAX5C	1,182,240	6,840	341 Mbit	Gen3 x16 (only Gen2 x8 usable)	3x16GB DDR4	VU9P
U200	1,182,240	6,840	341 Mbit	Gen3 x16 (only Gen2 x8 usable)	4x16GB DDR4	VU9P
U250	1,728,000	12,288	455 Mbit	Gen3 x16 (only Gen2 x8 usable)	4x16GB DDR4	VU13P
AWS F1	900,000	5,832	298 Mbit	Gen3 x16	4x16GB DDR4	VU9P

Device Portability in MaxCompiler

An important ability of MaxCompiler is, that it can target multiple different FPGA based accelerators from different vendors. To target different FPGA devices Maxeler generates VHDL and IP cores specific to the chosen device, invokes the correct toolchain and generates the required resource usage and timing reports for the user. The user is only required to write normal MaxJ code without any device specific extensions.

However, isolating the user from other device specific characteristics like the I/O capabilities

and the FPGA architecture is not that easily achievable and often also not desirable. These characteristics include the system architecture of the underlying device (e.g., number of logic resources and multiplier architecture) and target platform (e.g., number of memory channels and I/O bandwidth). For example, a user should be able to make use of the specific port widths of the hardware multipliers or use additional networking ports present on one platform and not the other.

In cases where user input is required to mitigate changes between devices it is crucial to limit the scope of changes and isolate platform specific code blocks from the remaining code base. This is achieved by the difference between the Manager and the Kernel as discussed in the programming model section. As a result, mostly the Manager code has to be changed to deal with device specific characteristics. Kernels are less affected, and the changes needed in Kernels are discussed at the end of this section.

To support portability between different FPGA devices, feature specific APIs for components that might be available on a specific platform were added. These APIs contain for example the functions required to instantiate networking ports, PCIe connections and DDR interfaces but also more generic functionality for example, to instantiate Kernels or ManagerStateMachines. While the former are highly specific to the specific platform the latter are supported by all devices. This API definition can also deal with platform specific differences and for example report port widths or, e.g., in the case of DDR, the number of physical DIMMs available. A simplified example for the PCIe API can be seen in listing 2.1. The API provides platform independent functions to create PCIe interfaces to stream data to and from the host.

Listing 2.1: Simplified example for the ManagerPCIe API.

```
1 public interface ManagerPCIe extends ManagerIO {
2
3   public DFELink addStreamToCPU(String name);
4
5   public DFELink addStreamFromCPU(String name);
6
```

```
7 }
```

As a result, for each platform supported in MaxCompiler there exists an appropriate platform specific Manager using this API. Due to this object orientated approach it is possible to write platform agnostic functions which only operate on the defined APIs.

An example of this is shown in listing 2.2. A Java interface for platform specific Managers is defined which provides access to the functions required to create a compute Kernel and PCIe connections. Within this interface a default implementation is created which instantiates a Kernel and connects it to PCIe and DDR (LMEM). Since the connection to DDR is platform specific the interface only defines a function which provides access to the required hardware interface. This function has to be implemented by the platform specific Manager.

Listing 2.2: Using object orientation to isolate platform specific from generic code.

```
1  public interface ExampleManager extends ManagerKernel, ManagerPCIe {
2
3    default void createDesign() {
4      final KernelBlock kernel = addKernel(new ExampleKernel(
           makeKernelParameters("ExampleKernel")));
5      kernel.getInput("x") <== addStreamFromCPU("x");
6      addStreamToCPU("y") <== kernel.getOutput("y");
7      LMemInterface iface = createLmemInterface();
8      kernel.getInput("z") <== iface.addStreamFromLMem("z",
           MemoryAccessPattern.LINEAR_1D);
9    }
10
11   public LMemInterface createLmemInterface();
12
13   public class ExampleU200Manager extends XilinxAlveoU200Manager
        implements ExampleManager {
14     public ExampleU200Manager(EngineParameters params) {
```

```
15        super(params);
16        createDesign();
17      }
18      @Override
19      public LMemInterface createLmemInterface() {
20        getLMemGlobalConfig().setMemoryFrequency(LMemFrequency.
              LMEM_1200MHZ);
21        return addLMemInterface();
22      }
23    }
24
25    public class ExampleMax4CManager extends MAX4CManager implements
         ExampleManager {
26      public ExampleMax4CManager(EngineParameters params) {
27        super(params);
28        createDesign();
29      }
30      @Override
31      public LMemInterface createLmemInterface() {
32        getLMemGlobalConfig().setMemoryFrequency(LMemFrequency.
              LMEM_800MHZ);
33        return addLMemInterface();
34      }
35    }
36  }
```

In the lines 13 to 23 a platform specific Manager for the Xilinx Alveo U200 card is created. It calls the interface function to create the design and implements the function to create a DDR interface. Similarly, a Manager for Maxeler's MAX4C DFE can be created. It should be noted that the implementations for both cards are nearly identical and mainly consist of Java boilerplate code in this simplified example. However, the creation of the DDR interface is

actually different. The reason for this is that the MAX4C DFE is based on DDR3 technology while the Alveo U200 uses DDR4. As such different memory frequencies are supported. Within the platform specific Manager only supported configurations are offered, but by exploiting the object orientated API based approach to isolate these functionalities, the programmer is able to share as much code as possible between the different platforms to ease maintainability.

A similar, but slightly more manual, approach can be used to isolate other architecture specific features like native hardware multiplier sizes and I/O port widths. MaxCompiler has a fully customisable type system. As such it is possible to define the appropriate types for each platform within the platform Manager and pass them to the Kernels. Similarly, the Kernels can query port widths from the Managers and instantiate functions performing the appropriate aspect change automatically. These functions assist the programmer in developing applications which are deployable on different hardware targets.

2.3.6 Comparison

A comparison between the four main tool options is provided in tab. 2.2. It should be stressed that the classification in terms of possible customisation, designer productivity and expected performance is not the result of an elaborated objective study, but my subjective opinion based on functional considerations, my personal experience and the literature I have read during my research. Obviously especially performance can differ significantly between use cases. However, it is also clear that the machine model the HLS languages were originally developed for introduces additional challenges to the toolchains.

To provide an example for these challenges one can consider the simplified implementation of a von Neumann architecture in fig. 2.6. Fundamentally, all instructions have to be loaded from memory. Additionally, the data the operations are executed on have to be loaded from memory as well and the resulting data are written back to memory in the end. While caches and registers can alleviate some of the disadvantages of this architecture the languages still assume that data are constantly stored and loaded which will result in increased memory bandwidth requirements. It is normal to store intermediate results of a computation in an

Table 2.2: Comparison between the different Programming Frameworks.

Framework	HDL	Sequentiell HLS	GPU based HLS	MaxCompiler
Machine model	None	Von Neumann machine	Compute device with hierarchy of compute units and processing elements as well as predefined memory hierarchy	Custom dataflow based machine model
Programming model	Based on spatial structures and notion of time	normal imperative programming model with annotations to direct hardware compilation	Data and task based parallelism and annotations to direct hardware compilation	Custom dataflow based programming model build around Managers and Kernels
Connection between source code and hardware implementation	Direct. The hardware structure is specified in the source code	Indirect. The behaviour of an algorithm is described by the program and the annotations only direct the tool. The tool infers the resulting structure of the hardware	Indirect. The behaviour of an algorithm is described by the program and the annotations only direct the tool. The tool infers the resulting structure of the hardware	Mostly direct. There is a direct relation between every source code line written and the resulting hardware implementation. The remaining properties can be inferred from the machine model.
Possibility for Customisation	All designs possible. Only limited by the targeted hardware / technology	Limited. A set of design patterns, optimisation schemes and architecture options can be used	Limited. A set of design patterns, optimisation schemes and architecture options can be used. Focus on processing elements which are mapped to the FPGA	High. While the machine and programming model is mostly aimed at high throughput implementations using systolic arrays, it is possible to implement completely arbitrary functionality. Additionally, the architecture within the systolic array structure is highly customisable
Designer Productivity	Low	Very High	High. A bit harder to use compared to sequential HLS, but still easily accessible for software developers	Medium. While MaxJ is a modern software language fundamental hardware design principles need to be understood. Greater space of architectural options requires more knowledge by the designer. Metaprogramming and programming model not as widely known
Expected Performance	High. HDLs enable many low level optimisations. However, due to the complexity of the design system level optimisations can be challenging	Good. HLS tools will be able to efficiently implement the predefined design patterns and provide tool support for optimisations. However, it will, for example, be difficult or impossible to optimise memory access patterns on and off chip to maximise bandwidth and reduce the number of memory accesses per operation	Good. HLS tools will be able to efficiently implement the predefined design patterns and provide tool support for optimisations. However, it will, for example, be difficult or impossible to optimise memory access patterns on and off chip to maximise bandwidth and reduce the number of memory accesses per operation	High. While MaxCompiler will in some cases introduce slight overheads due to the machine model, the designer has access to some low level optimisations and some will be performed automatically. Additionally, the higher level of abstraction will enable rapid system level optimisations
Predictability	Area usage will be easy to predict due to direct relationship between source code and hardware. Achievable frequency is hard to predict and depends on implementation by the designer	Low. Only possible once implementation is complete	Low. Only possible once implementation is complete	Very high. Area usage can be predicted since implemented hardware is predictable and controllable by designer. The systolic array architecture makes achievable frequency predictable as well

array when writing an imperative sequential program. The HLS toolchain has to identify these intermediate results to avoid the need to store them in memory. In contrast the systolic array structure used in the Kernels of Maxeler's dataflow-based programming model is designed to address precisely this issue. As a result, the toolchain design is simplified, and the quality of result is less dependent on how good the compiler is at optimising the code.

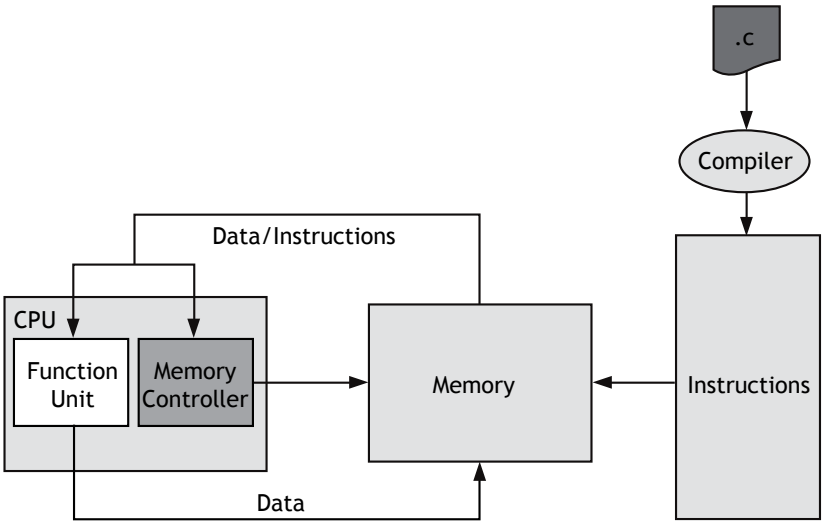

Figure 2.6: Control-flow system using von Neumann architecture.

MaxCompiler attempts to find a balance between the fine-grained control of HDLs and the programmability improvements of HLS. The target is to automate as much as possible, without impacting the ability of the designer to control the actual hardware generation. In order to address this balance, usually there is a direct relation between every line of MaxJ and the generated hardware. This enables to predict the final system in terms of performance and resource requirements which will become very important in later chapters of this thesis.

This design decision is the main reason for the usage of MaxCompiler in this thesis. It enabled me to quickly design architectures in great detail and map them onto the FPGA without being limited to only the architectures supported by the toolchain. I was able to quickly implement even complicated applications. The above effort would have required significantly more development time to achieve the same while using one of the HDLs.

A second reason is the integration of MaxCompiler into current European research projects which explore the use of FPGAs in HPC systems of the future. Most notable the EuroEXA

project [2] which develops a testbed for future FPGA based exascale machines. One of the main differences in the employment of FPGAs in that project, is that the dependence onto CPUs is reduced and FPGA are usable as a standalone device [80, 81]. Due to the integration of MaxCompiler into this approach the applicability of the work in this thesis is not only provided for current but also potential future systems.

The third and final reason is that Maxeler provided full source code access to MaxCompiler to me. As a result, I was able to make changes to a commercial FPGA toolchain to implement and test ideas I developed during my research.

2.4 Performance and Area Prediction Methodologies

For the development of FPGA applications, it is crucial to predict performance and area usage. Only then is it possible to assess if the porting to FPGA will yield the required benefits to justify the required system and development costs. Especially area usage predictions are important to avoid a case in which at the end of a long development project the design does not fit into the device potentially resulting in the need for a complete redesign.

The authors of [50] propose a methodology, which can be used to predict the achievable speed-up for a given application on a specific FPGA platform. The target is to quickly perform a cost benefit analysis for a specific acceleration project and provide information necessary to decide whether a given acceleration project is worth doing.

Their methodology focuses on predicting host to accelerator communication time as well as the time needed to perform the computation. The prediction of the communication time assumes an either single or double buffered architecture, where a working set is written to a buffer, processed and then written back. The computational performance is based around the number of operations that are needed to process a given element and the number of operations that can be performed on the FPGA per cycle. However, the computational performance is only modelled to a certain level of accuracy and then an inefficiency factor is used to avoid overly optimistic predictions.

The authors also mention the need to predict the resource usage and provide some guidelines on how this might be achieved, but do not present any examples where this was done. The prediction of the speed-up for the three presented use cases has an error of up to 40%.

In [116] a framework for the performance and power prediction of standard cell-based accelerators is presented. It uses the intermediate representation generated by a just-in-time compiler to capture the algorithm described in a C program. This is then used to generate a dynamic data dependence graph to represent the accelerator. The authors mention that a problem of this approach is that representative input data are needed to capture the intermediate representation and operations like system calls and dynamic memory allocation are not considered.

The tool automatically optimises this graph and analyses data dependencies to capture an accurate model of the accelerator. The framework includes simulators for memory and cache behaviour, but it is not possible to influence the optimisations performed manually or describe a target architecture. Instead, the C algorithm has to be changed and the framework performs DSE for a set of parameters like loop unrolling factors and frequencies.

The tool is validated against RTL and HLS designs and the reported error for the tested benchmarks is less than 1% for the execution time, less than 5% for the power prediction and less than 6.5% for area prediction.

In [91] a similar technique is used, to estimate the resource usage for HLS tools. The input is C/C++ code with a selected number of HLS pragmas, like loop pipelining. The tool uses the application trace to create a dynamic data dependence graph which is then optimised. Analytical equations are used to predict the resulting area usage with an error of less than 10%. Still the tool makes multiple assumptions, like usage of only single precision floating-point arithmetic. As such only a limited number of operations is required. The port width of on-chip memory is ignored and only the total number of bits is observed. The proposed tool does not model I/O, but only the FPGA fabric itself.

The authors of [166] follow a similar approach. The resource usage and execution time of applications synthesised using HLS tools are predicted, by supporting a limited number of

pragmas and applying analytical formulas. In contrast to [91] more pragmas are supported, and more detailed analytical formulas are used to calculate the hardware usage. However, the logic usage is ignored completely.

The authors spent considerable effort to predict the latency of the application with a high degree of accuracy. In order to do this most optimisations performed by the used HLS tool are modelled, including scheduling of the operations. While this technique leads to a low error, it will not adapt well to different HLS tools or even new versions of the same tool. I/O bandwidth limitations are not considered.

The developed tool is integrated in a tool flow to automatically perform DSE. A brute force approach is used to consider all possible combinations of pragmas, and the performance prediction is used to identify the most promising design point.

In contrast in [32] machine learning is used to predict hardware usage and timing characteristics for HLS toolchains. Different machine learning methods are used to reduce the error of the HLS toolchain estimations significantly, but the method requires a completely developed design as input.

In [30] the normal Roofline model [143] is extended to also support FPGAs. This is achieved by for example considering area requirements. The model assumes that FPGA implementations always make use of processing elements. The area usage for one processing element is measured by performing place and route. Additionally, the performance of a single processing element can be measured. With linear interpolation the maximum number of processing elements possible can be calculated which also provides the potentially best computational performance. The authors note that linear scaling is a simplification.

The I/O bandwidth is measured in bytes per second that can be transmitted. Together with the computational performance this provides the maximum achievable performance on a given platform. The same authors test their model on more use cases and with a different HLS tool in [31].

To summarise there is a multitude of approaches to predict system performance and resource

utilisation of varying complexity. Most of them focus on specific use cases and assume a certain architecture or toolchain. Some use advanced predictive analysis techniques like ML, while others use a simpler linear equation. However, with increasing complexity of the implemented design it becomes harder and harder to understand what causes performance bottlenecks or high resource usage. As a result, it is not easily possible to improve the system design to improve performance or to save resources. Some of the methods require a complete or partial hardware design leading to design iterations in hardware, if the performance is supposed to be improved. To alleviate these problems this thesis will propose to use simple equations to predict system performance and resource usage at a high degree of accuracy.

2.5 Development Methodologies

To assist programmers with the design of FPGA based applications multiple development methodologies have been proposed.

In [111] a methodology based on hardware description languages is presented. It focuses on a top-down approach, starting with a specification, potentially even in an HDL, and leading towards a physical implementation via the stages of architectural design and logic design. The methodology puts specific emphasis on specification, documentation and testability via simulators at every stage in the design process. It is mentioned that larger blocks of a specific design can be reused in other designs. The paper focuses in large parts on available tools, their pitfalls and shortcomings as well as design guidelines but there is no guidance provided on the general system design.

The authors of [101] propose a methodology specifically designed for industrial control systems. The methodology is independent of the used language but requires a functional description of the developed controller on the system, behavioural, RTL or synthesis as well as physical level. The reason for this is the ability to test, debug and verify the system under development at every stage of the development process, in which the implementation is incrementally refined.

The methodology itself has three major parts. The first is the already mentioned algorithm

refinement, which for example includes a transformation into fixed-point arithmetic and potential algorithmic transformations used to reduce area usage or increase performance. The second part is the ability to reuse modules between different designs for example in the form of IP cores.

The final part is called "A^3" methodology. Most importantly it refers to a process with the goal of finding the optimal solution within the system constraints. These constraints include the physical limitations of the hardware device, like timing characteristics and area constraints but also the requirements that the controller needs to meet operational requirements in order to produce a working system.

This is achieved by transforming the arithmetic part of the controller into a dataflow graph, then multiple possible hardware implementations for this dataflow graph are implemented and synthesised, thus enabling the designer to pick the best solution. The possible implementations are mostly differentiated by the fact that hardware units, like adders and multipliers, are reused for different operations needed by the algorithm.

While this proposed solution might be feasible for industrial controllers and very small dataflow graphs, it is not feasible for dataflow graphs of sufficient size, since the number of possible implementations increases exponentially with the algorithm complexity. Performing place and route for all these possible design points will require a significant amount of time.

In [139] a design methodology based on OpenCL is discussed. It is similar to the normal CPU based methodology, where an initial algorithm implementation is incrementally improved, using tool reports and profiling tools. The authors of [140] provide another methodology for OpenCL based FPGA application optimisation. The OpenCL code is parsed using a LLVM based frontend to collect the LLVM Intermediate Representation (LLVM-IR), which is then analysed. The tool estimates computational performance based on the latency of loops and the level of parallelism and predicts memory access time using a drastically simplified memory efficiency model. The resource usage of the design is measured by running place and route.

The authors embrace a CPU based methodology of incremental improvement, where an ini-

tial algorithm is improved. They mention by themselves, that for example a memory bound algorithm might require a complete rewrite to resolve this bottleneck. The proposed tool provides four simple metrics, which are supposed to guide the programmer on how to improve the existing application. These metrics are the utilisation of the resources, the utilisation of the memory bandwidth as well as the balance between different computational kernels and the size of the dataset used.

The tool is evaluated on a few simple use cases like matrix multiplication and k-means, where the runtime is predicted with high accuracy and the proposed metrics show the bottlenecks of the design. As such the authors manage to achieve multiple orders of speed-up on their base implementation by following the suggestions made by their tool.

In [103] an execution environment similar to the OpenCL approach is used. The target application is written on a fully functional level without any expression for parallelism and is then automatically transformed into multiple variants in an intermediate representation. The roofline model is used to automatically explore the design space and implement the pareto-optimal implementation.

Similarly [70] automates the complete design process including design space exploration using a performance model consisting of analytic equations and machine learning models. The tool uses an intermediate representation which contains a set of parameterizable architectural templates. Overall, the approach is based on parallel patterns which can be used to describe the parallelism within the application by the programmer. Again, the authors assume an architecture in which all initial data as well as results are copied into DDR. No direct communication between the FPGA fabric and other components is possible. The performance model uses analytical equations and additional machine learning models to also model the optimisations and decisions made by the synthesis tool. They test their tool on multiple computational kernels and achieve speedups between 0.1x and 16.73x. The authors only use small benchmarks to validate their work and even for those simple examples the speedup over CPUs is often limited. The time to transfer the initial data into DDR memory is ignored.

All these approaches might be feasible for small applications. For bigger applications, place

and route time as well as the need to rewrite the complete application, based on bottlenecks discovered late in the design process, will become prohibitive.

2.6 Applications used for Evaluation

To evaluate the techniques, tools and methodology developed in this thesis different real applications are employed. In this section a quick introduction for each of them is provided summarising the reasons for selecting this specific application, the desired functionality as well as the current state-of-the-art.

2.6.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of Neural Networks (NNs) which contain convolutional layers. Convolutional layers perform a convolution between the input data and a set of weights having the same dimension, called a kernel, to produce the input to the activation function. As a result, convolutional layers can extract proximity-based features.

CNNs are often used in the context of image processing. The convolutional layers are used to extract features from an image and have shown good results to common problems including recognition and detection tasks. The kernels are usually shared for all pixels of the same input channel. This parameter sharing leads to an equivariance to translations, which is often desired in image-based problems. Additionally, the parameter sharing reduces the overall memory requirements to store all the required weights. It should be noted that CNNs often require significant compute performance to calculate all the Multiply Accumulate (MAC) operations required for the convolutions [15].

Eq. 2.1 shows how the output of a (2D) convolutional layer is calculated, where Z represents the output, X the input, K the weight kernel and b the bias. Additionally, i is the output number, j and k are the x- and y-positions, l represents the input number and m and n are

the x- and y-offsets in the convolution [**?**].

$$Z_{i,j,k} = \sum_{l,m,n} X_{l,j+m,k+n} K_{i,l,m,n} + b_i \tag{2.1}$$

Another commonly used technique in CNNs is called pooling. Pooling combines multiple values of the input into a single output value. While this aggregation of information has the risk of data loss it also provides spatial invariance and reduces the complexity of the following layers.

Finally, the last part of a network is often composed by a standard NN, implemented as the weighted sum of the inputs plus a bias term, where the weights and the bias represent the learned coefficients. These last layers are used to perform the final classification/recognition of the input.

In order to introduce nonlinearities into the CNN model, activation functions are usually applied to the output of the convolutional layer. Possible activation functions include sigmoid, SoftMax, the Rectifier Linear Unit (ReLU) and arctangent. In particular, the ReLU function is applied after the convolutional layer, helping to improve generalisation of the network [163].

One widely used CNN is VGG-16 [118], which has 13 convolutional layers and 3 fully connected layers. The convolutional layers are distributed into five different groups. In each of those groups the input size and the number of outputs stays constant and between two groups pooling is applied. All layers apart from the last fully connected layer use the ReLU function as activation function, while the last layer uses SoftMax. The input sizes and the numbers of outputs can be found in table 2.3.

All convolutional layers of VGG-16 use precisely the same filter size of 3 by 3 data elements. This simplifies any potential hardware implementation.

In recent years there has been significant commercial and academic interest in high performance implementations of CNN applications. As a result, I decided that the implementation of the widely used VGG-16 CNN provides a good opportunity to test the development methodology proposed in this thesis. There are two main reasons for this decision. First, since the proposed

Table 2.3: VGG-16 layer properties

Layer Group	Input Size	Outputs
conv1	224x224	64
conv2	112x112	128
conv3	56x56	256
conv4	28x28	512
conv5	14x14	512
fc6	25,088	4,096
fc7	4,096	4,096
fc8	4,096	1,000

methodology targets HPC systems selecting a common HPC use case provides a good opportunity for evaluation. Second, due to the high interest into implementing CNNs on FPGA based platforms there is a wide body of current related work which can be used for comparison. This enables a fair comparison on a similar set of know optimisations.

To enable this fair comparison, I only consider work which was developed in the same time frame (2017) as the implementation presented in this thesis. The progress in the field of CNN acceleration in the last few years has been significant. Newer work will outperform the CNN implementation proposed in this thesis and the related work presented in this section by multiple factors. It will therefore not be helpful to evaluate the proposed methodology but only to show how badly the proposed implementation has aged.

Interest into high performance implementations of CNNs increased during the last few years. The most notable work is the TPU chip, which is specifically designed for neural networks [59]. The TPU was developed by Google for their machine learning needs and offers up to 200x speed-up compared to conventional CPUs. The major TPU advantages are the short computation latency and up to 10x cost reductions compared to other solutions.

The remainder of this section compares different FPGA implementations for CNNs. I compare the achieved performance while executing a CNN (VGG-16 if not stated otherwise).

One notable work is *Caffeine* [164], in which the authors demonstrate the implementation of a general framework based on a matrix multiplication architecture. The proposed architecture uses an HLS generated systolic array which exploits data locality thanks to a weight-major

mapping technique for both convolution and fully connected layer execution. Their implementations, integrated with the deep learning framework *Caffe*, obtained 354 GOPS throughput when configured as VGG-16 on the complete network and 488 GOPS if only the convolutional part is considered.

A different approach is reported in [89]. The authors perform an analysis of the CNN computation loop structure and compare different optimisations (in particular loop unrolling, tiling and interchange). Their analytical model based on the loop order and parameters is then used to perform the DSE. The best design found in the work uses loop unrolling of the external loops of the convolution layer. This allows to reuse the weights and minimise memory accesses and size requirements, and to achieve 645 GOPS performance.

In [142], the authors propose an end-to-end automation flow for systolic array design synthesis. A 2D systolic array structure improves the timing and the data reuse of the design and is obtained from the analysis of the nested loops implementing the considered algorithm. By performing a two-phase DSE (first generic and then platform-specific), the authors are able to map an arbitrary user-defined CNN algorithm to few pre-designed templates. As a result, an example VGG-16 design achieves 1.17 TOPS.

Adyonat et al. present another accelerator for CNNs called DLA [13]. In their work, they describe a methodology to reduce bandwidth requirements through the use of stream-buffers in conjunction with batching, and to reduce the number of MAC operations by using the Winograd transformation [145]. These optimisations allow the proposed work to obtain a 1.38 TFLOPS running the AlexNet network.

Zhang et al. [165] propose a 2D interconnection between the different Processing Elements (PEs) in the design, enhanced by a 2D dispatcher and a shared buffer technique to prevent memory duplication and optimise the use of on-chip memory and required bandwidth. The proposed work was thus able to reach a 1.79 TOPS throughput running VGG-16.

In [87] the authors use the 2D Winograd algorithm (instead of the 1D proposed in [13]) and buffer the needed data in a line buffer. The architecture performs the 2D Winograd over a tile

with a fixed stride, in order to compute multiple results at the same time. In this way, the authors have been able to implement an automatic tool flow and reach 2.94 TOPS with their VGG-16 implementation and 3.04 TOPS if only the convolutional layers are considered.

Tab. 2.4 provides an overview of the related work. For the Intel based platform the number of multipliers used is provided and not the number of used DSPs. This is the case since on this Intel platform a single DSP can be used to implement two multiplications for the datatypes used in the work that is considered here. Additionally, in order to compare the performance between different devices easier I added a normalised performance figure. For this the performance in TOPS is divided by the number of multipliers and the frequency. This results in the average number of operations executed per multiplier on one cycle. One can see that most of the related work has a relatively similar normalised performance between 1.1 and 1.7 operations per multiplier. However, the work presented in [87] manages to outperform the other related work significantly due to the usage of the 2D Winograd algorithm.

Table 2.4: CNN Performance comparison.

	[164]	**[89]**	**[142]**	**[13]**	**[165]**	**[87]**
Implemented Network	VGG-16	VGG-16	VGG-16	AlexNet	VGG-16	VGG-16
Device	Xilinx VX690T	Intel GX1150	Intel GT1150	Intel GX1150	Intel GX1150	Xilinx ZVU102
Precision	16 bits fixed	8-16 bits fixed	8-16 bits fixed	16 bits shared exponent floating-point	16 bits fixed	16 bits fixed
Freq (MHz)	150	150	231.85	303	385	200
Logic cell (K)	300	161	313	246	-	600
SRAM (Kb)	$1,248 \times 18$	$1,900 \times 20$	$1,668 \times 20$	$2,487 \times 20$	$1,450 \times 20$	$1,824 \times 18$
Multipliers	2,833	3,036	3,000	2,952	2,756	2,520
TOPs	0.488	0.645	1.17	1.38	1.79	3.04
Normalised Performance OP/Multiplier	1.148	1.416	1.682	1.543	1.686	6.032

2.6.2 Asian Option Pricing

An option is a financial instrument which represents a contract between at least two parties. Within a predefined time horizon the holder of an option has the right, but not the obligation, to buy or sell the asset the option is referring to at a predefined price. In the case of Asian options, the outcome is depending on the average price during the predefined time horizon and

not only on the prize at the valuation date. As a result, calculating the value of an Asian option is more challenging than for other options.

The evaluation of the option payoff requires an approximation, as the probability distribution of the variable defining the option payoff at valuation date has no closed-form solution. Curran's approximation algorithm [29] is widely used in the financial industry to calculate the value of an Asian option due to its high accuracy. The algorithm calculates the expected option payoff at multiple averaging points. This means that in order to successfully price an Asian option prices at multiple averaging points for multiple possible market scenarios have to be calculated. This is compute intensive and acceleration is desirable as financial institutions carry out option pricing repeatedly and over large datasets.

In [106] a dataflow architecture of the Curran approximation algorithm for a MAX4C DFE is presented. I use this implementation of Asian option pricing to evaluate methods and best practices to support multiple different FPGA platforms from a single code base. Asian option pricing is highly relevant in the context of risk calculations which itself is a prominent use case in the HPC domain. As such this application provides a good example of a typical HPC workload. Additionally, the usage of this application was highly practical, since I had full access to the source code and documentation and did not need to implement another application from scratch. This allowed me to focus on the process of porting the application to multiple platforms.

Implementing the Curran approximation algorithm involves accelerating the calculation of a Normal Cumulative Distribution Function (NCDF), the most compute-intensive part in the approximation model. The authors note that mapping this computation to an FPGA is very resource intensive due to the reliance on floating-point computations and non-elementary functions such as exponential, square roots, and logarithms. As a first step, they carried out several algorithmic and numeric optimisations which include reordering of computations, changing the approximation model for the NCDF, and replacing floating-point with fixed-point calculations, such that the output satisfies an application-specific accuracy requirement of 9 decimal places.

Next, they developed a dataflow architecture which consists of a chain of five asynchronous Kernels, each implementing different stages of the Curran's algorithm. Stages two and four can

benefit from parallelism, as each stage carries out loop computations related to the number of averaging points n in the option. However, attempting to fully unroll these loops is challenging because the number of averaging points is a runtime parameter that changes for different option settings. Also, for large numbers of n, loop unrolling will be limited due to FPGA resource constraints. Hence, the authors developed a parametric design where loops are partially unrolled by a factor of k and an arbitrary number of averaging points is supported at runtime.

Fig. 2.7 shows an overview of the architecture. Each Kernel is annotated with the main tasks it fulfils and it also shows the communication pattern between the different Kernels. The Kernels one, three and five operate independently of the number of averaging points used for the calculation, which also means they operate for a fixed number of cycles for each scenario and option. Kernels two and four are configurable in size through the parameter k and the number of cycles they are executed for needs to be adjusted based on the number of averaging points, since k averaging points are processed per cycle.

The degree of parallelism k is a design parameter in the dataflow implementation that can be easily tweaked to maximise performance and resource utilisation according to the targeted device. For the targeted MAX4C DFE a parallelism of $k = 13$ was found to be the best value. The accelerated Asian option pricer was benchmarked in a Value-at-Risk application that evaluates 5,000 scenarios with 10,000 options each. The authors achieve speed-ups of 278.3x and 9.2x comparing to baseline software versions that were executed on a 48 core CPU either single-threaded or multi-threaded, respectively [106].

2.6.3 Monte Carlo Based Dose Simulation for Radiotherapy

Radiotherapy is a commonly used treatment for various cancer types. High doses of radiation are used to kill cancer cells. Modern radiotherapy relies on an intensity modulation technique that aims to deliver high dose gradients to cancerous tissues while sparing the surrounding healthy organs as much as possible. This is achieved by setting up a therapy treatment plan which takes into account the anatomy as well as the clinical case and dose delivering machine. In order to validate and optimise such therapy plans, the expected spatial dose distribution

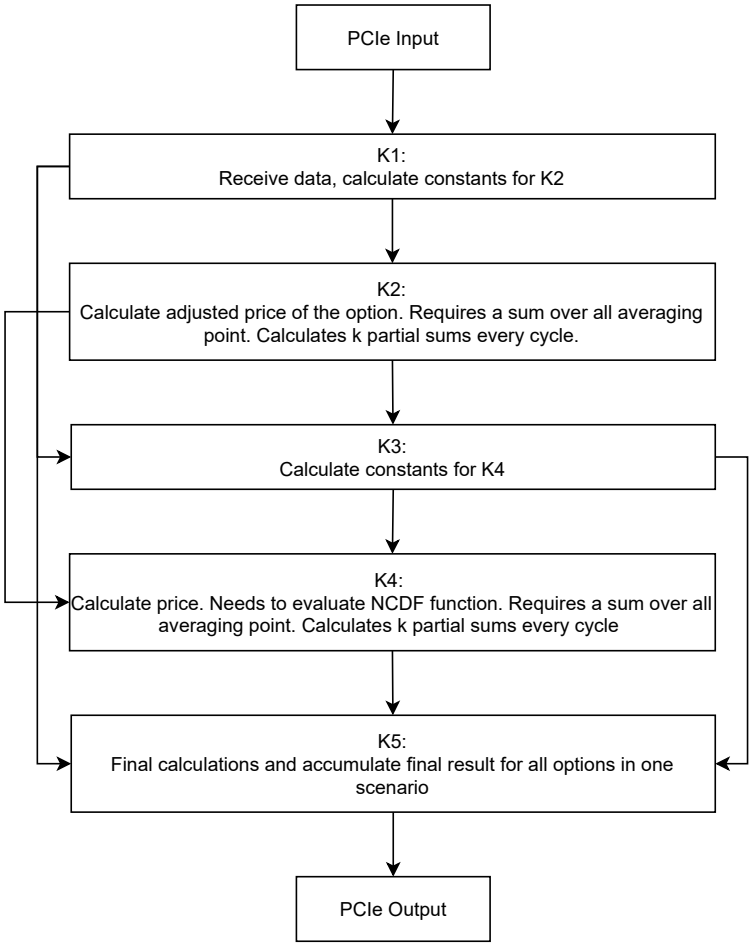

Figure 2.7: Architecture of the Asian option pricing application proposed in [106].

within the patient has to be simulated before the actual treatment. This is often implemented by Monte Carlo methods which simulate the pathway of millions of radiation particle trajectories as they enter the patient body. These simulations are highly accurate on the one hand but require relatively long computation times on the other hand.

Using Monte Carlo simulations to calculate the dose distribution in radiotherapy is widely considered to be the most accurate method. The software simulates particle interactions and calculates the dose deposition along the trajectories following fundamental physical laws. However, this accuracy comes at a cost, since a significant amount of particles need to be simulated to achieve statistically significant results.

This work will focus on the Dose Planning Method (DPM) [115] implementation of a Monte Carlo technique that simulates the dosimetric effect of high energy photons in organic materials. This algorithm is specifically optimised for radio therapy. DPM provides implementations for all relevant photon-matter and electron-matter interactions that occur in radiotherapy. High efficiency is achieved by optimising the physical interaction description as well as their implementation on modern processors. The authors distinguish between hard interaction processes which have to be calculated analogously and soft interactions which can be accumulated and only simulated once over a certain distance. Especially the latter technique reduces the simulation time of electron interactions significantly.

In DPM the patient data is represented in a so-called patient cube. This cube is discretised into multiple voxels with a given resolution (e.g., 1 cm cubed). Each voxel contains information on its material, e.g., tissue, bone or water, and the accumulated dose. During the simulation photons and electrons travel through this patient cube and the relevant interactions that occur are simulated to calculate the dose deposited.

The deposited dose is calculated as follows. Initially particles are generated with a direction and energy and multiple fuel values. While a particle moves along its trajectory fuel is used up and once it runs out an interaction occurs. Depending on the type of fuel a different interaction subroutine is executed. This can be a hard inelastic scattering interaction modelled according to Moller, an elastic particle scattering using the Hinge theory or the bremsstrahlung

interaction modelling a change in speed of charged particles. These subroutines calculate the radiation deposited in the voxel in which the interaction occurs as well as the new values for the fuels, the particle energy and the new trajectory. Once the energy of the particle falls below a threshold it gets absorbed leading to further dose deposition.

In order to reach statistically significant results more than 100 million particles have to be simulated according to medical experts [26]. Additionally, for the use case of adaptive radiotherapy it is important to finish the complete simulation in less than one second to adjust for natural patient movements like breathing. Alg. 2.3 provides a simplified pseudocode description of the DPM algorithm.

Algorithm 2.1: Simplified pseudocode for the DPM algorithm

```
1  input: n = number of particles to simulate
2  begin
3    for 0 to n:
4      CreateParticle(E)
5      InitialiseConstants(E)
6      InitFuels(E)
7      while E not absorbed:
8      begin
9        // The following function decides if an interaction occurs, and if so, which one
10       BurnFuel(E)
11       ModelContinuousEnergyLoss(E)
12       DepositDoseBasedOnContinuousEnergyLoss(E)
13       if E.energy <= threshold:
14         AbsorbParticle(E)
15         DepositDoseFromAbsorption(E)
16       end if
17       MoveParticleAlongTrajectoryThroughPatientCube(E)
18       if Moller interaction occured:
19         MollerInteraction(E)
20         DepositDoseFromMoller(E)
21       end if
22       if Hinge interaction occured:
23         HingeInteraction(E)
24         DepositDoseFromHinge(E)
25       end if
26       if Bremsstrahlung interaction occured:
27         BremsstrahlungInteraction(E)
```

```
28          BremsstrahlungGeneratePhoton(E)
29          BremsstrahlungRefuel(E)
30          DepositDoseFromBremsstrahlung(E)
31        end if
32        if E.energy <= threshold:
33          AbsorbParticle(E)
34          DepositDoseFromAbsorption(E)
35        end if
36      end while
37    end for
38  end
```

This application is selected to evaluate the overall development methodology. Due to the stochastic nature of the Monte Carlo simulation and the significant impact on the resulting execution time this application is a worst-case scenario for the analysis and prediction-based development methodology. As such it will be helpful to discover if the methodology can handle even this more complicated case and if there are any weaknesses in the current methodology that it will highlight. Furthermore, the application has a complicated memory access and communication pattern which complicates the implementation on a FPGA. This will help to evaluate if the methodology can really support the development of complex applications.

Due to the practical relevance of Monte Carlo dose simulation and the high computational requirements related to it a considerable amount of research has focused on accelerating it. This includes algorithmic improvements as presented in [18, 62, 63, 88, 115, 146]. There are also multiple studies which use GPUs to accelerate the workload, e.g., [55] and [125]. In these cases, speed-ups of up to multiple 100x are reported in comparison to CPU code. However, the authors of [82] and [54] report that this performance advantage is actually a lot smaller, if realistic test cases are considered and the comparison is performed against optimised CPU code. In those cases, the speed-up of GPU over CPU implementations is closer to 2.5x.

In addition to the GPU implementations, also CPU based implementations were proposed. Examples for these can be found in [24], [126] and [168]. The latter manages to finish the dose simulation in less than a minute and outperforms well-known GPU implementations.

To facilitate adaptive radiotherapy and the required real time dose simulation, the work in [168]

was further expanded in [167] by adding support for cloud computing. The authors propose to use the scalability of cloud-based systems to create a bigger cluster of cloud instances to perform the simulation. They manage to reduce the runtime of Monte Carlo dose simulations to values between 1.1 and 10.9 seconds depending on the specific configuration. Additionally, they make use of encryption to facilitate privacy for the medical data transferred into the cloud. However, cloud-based solutions have the disadvantage of requiring a fast and stable internet connection in the hospital to be usable for reliable medical treatment.

In [39] the authors propose an FPGA implementation for Monte Carlo based dose simulation. They simulate photons and electrons, where the initial photons are generated by an external source and sent to the FPGA. Afterwards, the dose is calculated and accumulated in the patient cube. However, the patient cube voxels are only saved in on-chip memory, limiting the resolution of the patient cube to 64 voxels in each dimension. A speed-up of up to two orders of magnitude compared to a CPU implementation is claimed.

In [27] a methodology to develop FPGA based mixed precision Monte Carlo designs is presented. The authors propose an analytical model to determine the optimal precision and resource allocation for a given Monte Carlo simulation. They combine an FPGA and a CPU to achieve the desired accuracy while using reduced precision. As a result, they report speed-ups of up to 4.6x, 7.1x and 163x compared to state-of-the-art GPU, FPGA and CPU designs respectively.

A run-time adaptive FPGA based Monte Carlo architecture is proposed in [122]. The authors note that the ability to use custom number formats tailored to the Monte Carlo simulation is one of the main advantages of FPGAs compared to CPUs and GPUs. As a result, they propose an architecture which monitors and reconfigures the used number representation at runtime to fully exploit this advantage. On average this method manages to achieve an increase in throughput of 1.35x compared to a conventional implementation.

The authors of [84] present a domain specific language for the development of Monte Carlo simulations which targets FPGAs and GPUs. They report a 3.7x speed-up compared to CPUs for the generated FPGA designs. The advantage of this work is that the user only needs to describe the Monte Carlo simulation using a high-level framework based on LaTeX equations to

obtain the FPGA design.

A significant amount of other related work exists, in which different Monte Carlo simulations are accelerated using FPGAs. This includes image reconstruction for Single-Photon Emission Computed Tomography (SPECT) [67], pricing of Asian options [114], simulation of electron dynamics in semiconductors [105] and simulation of biological cells [161].

2.7 Summary

This chapter introduces the architecture of current FPGAs and provides an overview of programming languages in addition to performance and area prediction as well as development methodologies. Reconfigurable chips provide a highly efficient and flexible option for application acceleration. They provide energy and performance benefits to GPUs and especially CPUs, while providing a lower cost to market compared to ASICs. As such this technology provides a great fit for HPC.

This chapter also discusses the programming challenges of the traditional HDL design flow. An introduction into multiple HLS tools is provided and the Maxeler's dataflow computing toolflow, which will be used throughout this thesis, is discussed in more detail.

Additionally, an overview of different performance and area usage prediction solutions is presented. It is shown that all of the presented solutions have severe shortcomings, by either assuming a specific system architecture, having a low accuracy, too high complexity or relying on specific tools. Later in this thesis, I will present simple equations to perform these predictions with a high degree of accuracy providing valuable insight into the systems behaviour.

Furthermore, different methodologies for FPGA development are discussed. They either perform a brute force design space exploration, which requires a very high amount of place and route runs or are based on normal CPU methodologies. As such they contain severe shortcomings for the design of large-scale applications, which is made possible due to recent advances in semiconductor technology. Most notably they require too much place and route time for DSE

and might require the complete rewrite of an application based on bottlenecks, found only at the end of the design process. This thesis will propose a methodology which is based around area and performance predictions to derive an optimal architecture. As a result, bottlenecks are discovered before the first line of hardware code is written and DSE can be performed based on the performance model without needing a place and route tool.

Finally, the applications employed to evaluate the contributions of this thesis are presented. The concept of CNNs is explained and the state-of-the-art in CNN acceleration is summarised. This thesis will show how the methodology developed in this thesis can produce acceleration results in line with state-of-the-art results. Additionally, the Asian option pricing application as an example for a typical financial application is presented. This thesis will show how the existing dataflow-based implementation can be extended to target a wide range of target platforms. Finally, the Monte Carlo based dose simulation for radiotherapy represents a real time critical medical application. The methods and tools developed in this thesis will be used to create an FPGA accelerated version of this application.

Chapter 3

Methodology for Reconfigurable System Development

3.1 Introduction

This chapter addresses the challenge of a missing development methodology for the design of complex FPGA based systems which minimises the risk of delay introduced through design iterations in hardware and provides state-of-the-art performance. To accomplish this, it presents a methodology for application acceleration using FPGAs and applies it to two real applications. The aim of the methodology is a first-time-right design process, delivering state-of-the-art performance for highly complex applications. This means that the hardware implementation is only done once and all major design improvements are finished, before the hardware implementation is started removing the need to perform time consuming design iterations in hardware. The proposed methodology provides guidelines orthogonal to any algorithm and dataset specific optimisations. It uses an analytical model to highlight bottlenecks in a given architecture based on these algorithm properties and optimisations. This knowledge can be used by the designer to resolve the bottlenecks with a new architecture working towards maximising the utilisation of all system resources as highlighted by the analytical model. The combination of many different individual best practices into a coherent, structured methodology based around

performance modelling and a software model is the main novel contribution of this chapter.

It is important to stress that the methodology does not aim at the highest performance ever achievable, but rather facilitates the best performance for a given set of known optimisations to meet design requirements. New optimisations might be discovered or designed and older ones can become obsolete. Nevertheless, the proposed methodology does not focus on individual optimisations; it only guides the design process. As such, no changes to the methodology are needed as the set of considered optimisations evolves. When new techniques arise an update of the design might still be desired. In such cases, the methodology will assist the designers to rapidly evaluate the impact of these techniques and enable a precise cost benefit analysis. Similarly, the proposed methodology enables designers to quickly evaluate the relevant system characteristics of different hardware platforms.

Traditional CPU development methodologies usually follow an incremental approach, whereby a first implementation is gradually improved step by step with the help of different profiling and debugging tools. However, there are multiple problems with applying this methodology to FPGA based designs. (Problem 1) It is difficult to profile an existing FPGA implementation, making it difficult to identify what specific part of the design needs improvement. This is especially true if the initial implementation cannot entirely fit onto the FPGA fabric or does not reach the frequency required to highlight certain issues, like bandwidth limitations. (Problem 2) Incremental hardware changes can easily require a complete redesign of the existing application, which might render nearly all previously undertaken development efforts useless. This is especially the case if the fundamental handling and storage of data are reconsidered to work around bandwidth or memory capacity limitations. (Problem 3) The process to generate an FPGA bitstream is extremely time consuming and can take days. Combined with the high complexity of FPGA programming this results in prohibitive development times.

It is however possible to avoid these problems altogether. Given that the performance of FPGAs is highly predictable, especially for deeply pipelined designs able to hide I/O latency, it is possible to save a lot of time and developer effort by performing a large part of the design process a-priori, before even a single line of hardware code is written. The proposed

methodology provides a guide through this process.

The proposed methodology is applied to a CNN and a Quantum Chromodynamics (QCD) simulation to demonstrate its advantages.

The proposed methodology consists of four main parts:

1. accurate analysis of the targeted application;
2. design of a representative software model;
3. modelling of architectural candidates; and
4. development of an application specific architecture based on the results of the above parts.

Fig. 3.1 shows how these four parts interact with each other. All individual steps are explained in more detail in subsequent sections. The methodology involves an iterative process, which requires incremental refinement until the final result cannot be further improved. The natural starting point is the analysis of the original application (Part 1). Based on the analysis of the application an initial partitioning in hardware and software components is performed (Step a). The parts selected to be moved onto hardware are modelled in software (Part 2). Additionally, an initial architecture is designed (Part 4) which informs the representative performance model (Step d, Part 3). The performance model predicts the achievable performance as well as area and bandwidth requirements. It relies on the analysis of the original application (Step c), which is further improved by making additional measurements of, for example., data movements and numerical properties. These are hard to obtain in the original application and as a result the software model (Step b) is used instead. The performance model is used to refine the architecture by identifying bottlenecks, e.g., bandwidth limitations which can be addressed by changes to the architecture (Step e). However, the architectural choices determine the components and structure of the model, since the performance model is based on a given architecture (Step d). Next, the software model can be used to verify algorithmic and numerical changes of the architecture (Step f).

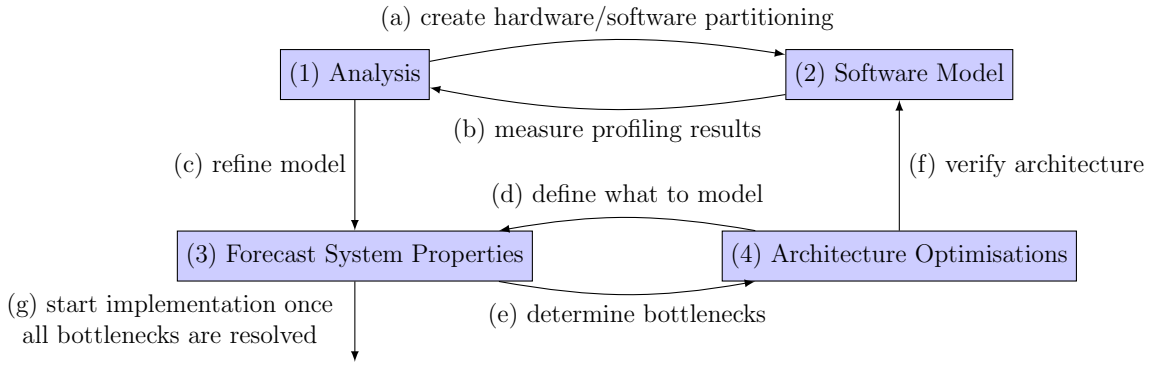

Figure 3.1: Design methodology breakdown and its four parts and seven steps.

Usually all of these steps are continuously repeated in an iterative fashion, where, for example, a change of the architecture causes changes to the software model and the performance model. Similarly, the performance model might highlight that additional measurements of the initial application are necessary. This iterative process is also shown in alg. 3.1. The programming of the FPGA only starts, once the bottlenecks highlighted by the performance model are resolved and full utilisation of all system resources is achieved (Step g).

Algorithm 3.1: Iterative refinement using the methodology (Numbers and letters in brackets do not describe specific transition, but only refer to the different parts and steps described above).

```
1  begin
2    perform initial analysis of original application (1)
3    create initial partitioning in hw and sw (a)
4    create first initial software model, performance model and architecture (2, 3, 4)
5    while bottleneck existing
6      identify bottlenecks using performance model (e, 3)
7      resolve bottlenecks with new architecture or hw/sw partitioning (4, a)
8      verify new architecture in updated sw model (f)
9      perform further analysis as necessary (b, c)
10     refine performance model based on updated analysis and architecture (d, c)
11   end
12   start implementation (g)
13 end
```

As a result, the analysis (Part 1) combined with the software (Part 2) and the performance model (Part 3) erases the need to profile the FPGA implementation (Problem 1) since bottlenecks are discovered beforehand. Needs to perform redesigns late in the design process (Problem

2) will be avoided, since the architecture (Part 4) is based on the performance model (Part 3), moving this issue to design time. This, combined with the availability of functional simulators for verification, also removes the need to perform place and route for multiple different design points (Problem 3).

In order to use the methodology a working CPU implementation of the application is needed. It is used for debugging and verification throughout the development and is crucial to ensure that the correct functionality is implemented. It is possible to create such an implementation before the methodology is used if needed. Furthermore, it might be beneficial to have a more abstract description of the application, which might make it easier to explore alternative algorithms or implementation strategies. This is not strictly required, since in many cases it can be also inferred from the CPU implementation, but it can be of advantage to achieve better results. Finally, it is necessary that a representative set of input data is available. Again, this is required to ensure that changes made to the implementation, especially on the numeric aspects, still leads to correct results.

The methodology is based around Maxeler's toolflow as discussed in section 2.3.5 and assumes a system architecture consisting of a reconfigurable accelerator connected to a CPU. This is mostly required to perform accurate prediction of the area requirements, performance and achievable frequency of the implemented design. Additionally, it means that at least significant parts of the application can be implemented in a deeply pipelined fashion and that the pipeline is not interruptible. If the chosen architecture includes a pipeline that has to be interrupted this would have to be modelled as well and is not discussed here. If the application is purely control based or latency and not throughput is of major concern the methodology is not a good fit (even though it might still provide some interesting insight).

The methodology produces a detailed architecture which can be used to implement a FPGA design. This includes a description of the major building blocks, how they work and interact, what tasks each of them performs, the area required, performance achieved and the numerics used. The implementation itself is not part of the methodology.

While the analysis of the initial application is usually only application dependent, all the other

steps might be device dependent. For example, the compute and communication capabilities of specific accelerators might be different and, as a result, the developed architecture might need to circumvent different bottlenecks. However, in general, the same performance model and architecture can be used for different devices as long as they are similar enough in terms of compute and communication ratios. In other cases, it is usually possible to reuse most of the work leading to the system design for the initial target platform. Even though this chapter covers the case where the target system is already selected, it is also possible to identify a system for optimal performance or energy efficiency by adjusting the performance model and the architecture for different target systems under consideration. Similarly, it is possible to quickly evaluate the performance impact a new device might have on the complete system and the changes required to the architecture and implementation to achieve this.

While the methodology assumes the usage of Maxeler's static dataflow model, to ease prediction, it should be possible to generalise the methodology even further, as long as accurate system prediction is possible. Similarly, the toolchain used for programming the target device has to provide enough fine-grained control over the hardware to the programmer, to allow for an accurate implementation of the developed architecture. An example of a tool which can potentially benefit from this methodology is Sandpiper [19,20], since it is based around explicit control over parallelism, memory access and is promising predictable performance. The main challenge here is to achieve predictable timing closure, since otherwise the error of the performance predictions can increase significantly. Even though this thesis will focus on FPGAs the methodology is also applicable to ASICs and supports highly heterogeneous systems with multiple, potentially different, accelerators.

For simplicity FPGA to FPGA communication is not explicitly considered in the description of the methodology presented in this chapter. However, the same steps to handle any kind of CPU to FPGA communication can be applied in the same way to FPGA to FPGA communication.

The main contributions of this chapter are as follows.

- A methodology for first-time-right development of complex, FPGA based systems, delivering the forecasted performance;

- The evaluation of the application of the proposed methodology using two real and complex applications: a CNN and QCD.

The rest of this chapter is organised as follows. Section 3.2 describes the analysis of the original application and what information is needed for a successful application acceleration process. In section 3.3 the purpose of the software model is discussed and details on how it should be developed are provided. Section 3.4 explains how the performance and behaviour of the FPGA and the overall system can be estimated. In section 3.5 usual trade offs encountered in the architectural definition phase are described. The methodology is evaluated in section 3.6 with the example of a CNN and a QCD application. Section 3.7 summarises the chapter.

3.2 Application Analysis

The first step, as depicted in fig. 3.1, is the analysis (Part 1) of the original CPU application in order to identify its compute and data intensive parts and build some understanding of the challenges which will be faced in the architecting stage. If no CPU implementation is available, it is necessary to create a runtime optimised implementation at this stage in order to enable the analysis. However, in most real-world applications there will be an existing reference application, which has been developed for CPUs, before operational requirements created the need for further acceleration.

In classical general-purpose computing, the view on the performance characteristics is often limited to analysing instructions and their order of execution, as well as the assignment to execution units. This view, however, is not optimal for applications on massively parallel architectures like FPGAs. Instead, it is crucial to also understand the required data movements in the system. As a result, a strategy where data placement and movement are optimised at all levels of the system has to be developed. Only then is it possible to make maximal use of the available computing resources by providing undelayed access to the working dataset to mitigate limited I/O bandwidths.

The target of the control/dataflow analysis is twofold. First this step will help to make an initial assessment of which parts of the application should be ported to FPGAs, since in most cases it is not feasible and/or desirable to port the entire application. However, following Amdahl's Law [9] the designer needs to be careful to address a significant part of the execution time, in order to make sure that the code remaining on the CPU will not dominate the runtime. For example, if the part of the application that contributes 90% of the execution time is ported onto the FPGA the maximum theoretical achievable speedup is 10x, due to the remaining CPU parts of the application.

The second target is to collect all the information needed to model the performance of the accelerated application and consequently make correct design decisions. This includes the amount of data that is needed and when it is needed as well as which operations are executed and in what order.

3.2.1 Static and Dynamic Code Analysis

In order to identify the parts of the application that can benefit most from hardware acceleration, various profiling tools like gprof [45] or Intel Advisor [53] can be used. With their assistance, it is possible to annotate the execution time of all functional blocks within the application. These measurements should be made using multiple sets of realistic data in order to approximate the envisioned real-world applications as accurately as possible. Erroneous test data selection can invalidate the entire porting process, if the compute pattern differs significantly in the real operational setting.

However, not only the computational complexity, but also the data requirements are essential to decide on an optimal partitioning of the algorithm onto the available computing resources. One has to keep in mind that the bandwidth between the different subsystems, e.g., CPUs and FPGAs, is often limited and frequently becomes a major bottleneck. As a result, it might be beneficial to also port parts of the code with small contributions to the overall runtime onto the FPGA, as a consequence of the need to reduce the amount of data that needs to be transferred.

As such, the size of the data structures and their accessing patterns need to be well understood. In the case of statically allocated memory this might be trivial, but in many cases the size of the allocated data structures depends on the input dataset. However, most profiling tools only have limited support to monitor the data structure size and as a result it is often only possible to add debugging statements into the original application. Since those statements might have a significant impact on execution times it is necessary to perform the runtime and data analysis separately. Similarly, access patterns to those data structures should be analysed. Sometimes only a tiny fraction of the data is accessed even though a reference to the complete data structure is provided.

Based on the findings of this analysis it is possible to create the initial partitioning between the CPU and FPGA. However, once the performance is modelled (Part 3), as it will be described in section 3.4, it might be necessary to revisit and reconsider this partitioning. This is the case if bandwidth limitations between components of the system exist or individual components are under- or over-utilised.

The parts of the application selected for porting to the FPGA require further analysis to provide sufficient information for the later performance modelling stage (Step c). This analysis stage can either be performed on the original application or on a software model (Step b, Part 2) as described in section 3.3.

Within the parts of the application selected for porting, the operation count for each operation type needs to be obtained. This information is needed in order to estimate the area usage of the circuit which will be implemented (see section 3.4.1 for more details). The process of counting operations can be performed manually or with performance analysis tools.

Control structures like if statements and loops typically introduce additional challenges. In order to accurately model the area requirements for multiple branches, which are present as a result of an if-else statement or a switch case construct, it is necessary to count the operations in each branch separately. In general, it will be necessary to implement all the branches in hardware separately, however, in certain conditions optimisations can be performed. It is not necessary to port branches onto the FPGA, if they are never active for the representative

input data set. Additionally, it is beneficial to remove as many branches as possible by code restructuring. For the remaining branches the probability of taking or not taking the branch should be captured for more accurate modelling later on.

For loops it is important to determine the maximum and, in some cases, the average number of iterations for which the loop is active. In order to determine these numbers accurately it is again crucial to have a large and representative test dataset.

The second part of this analysis aims at the identification of data movements and memory access patterns. In case the data used on the FPGA cannot be generated on the FPGA itself, it has to be sent over from the host[1]. For these transfers it is important to determine the amount of data that will be transferred, but also if the data stay constant or change on regular basis, since in the first case it might be possible to store the data in Read-Only Memories (ROM) while in the second case a regular retransmission might be necessary depending on the application control flow.

In most real-world applications it is often not possible to store the entire working dataset in the very high bandwidth on-chip memory of the FPGA. As a result, even the data which are only generated and used on the FPGA, including temporary results, need to be analysed. In these cases, again, not only the size of the data structure but also its access pattern needs to be examined. The access pattern is of special interest, since it has a significant impact on the read and write rates the on-board DDR memory can achieve, if the data need to be stored there. This will be further discussed in section 3.4.4.

3.2.2 Loopflow Graphs

Counting operations for a given set of functions is typically easier than the observation of data movements, which are often harder to analyse, represent and understand. For this reason, it is beneficial to create a loopflow graph, which will help to visualise some of the most important results of the previous analysis. Loopflow graphs are generated manually using the results of the

[1]The CPU side of the system

analysis and provide a good overview of the results collected. Loopflow graphs focus on the loops in the program and how they interact with each other. Since in general most of the execution time is spent in loops this provides a good level of abstraction. Each loop is represented by a rectangle and the number of nested loops is annotated by a factor. Additionally, the operation count can be annotated inside each rectangle.

The dataflow between the loops is depicted using directed arrows, where the width of the arrow alludes to the amount of data that needs to be transferred. Consequently, the computational and data requirements can be easily visualised together, which helps to identify the portions of the code which should be moved to the FPGA (Step a).

The loopflow graph for the VGG-16 CNN [118] in fig. 3.2, provides a good example on how this graph can help to perform the code split between CPU and FPGA. One can easily spot that the operations needed to compute the fully connected layers, depicted in the box at the bottom, are two orders of magnitude less than for the convolutional layers, depicted by the remaining boxes. Additionally, only a tiny amount of data needs to be transferred between the convolutional and the fully connected part of the network. As such, splitting between the convolutional and the fully connected part of the network becomes the obvious choice.

The loopflow graph captures the data transfer and compute requirements during the complete execution of the application. As a result, it does not necessarily capture the dynamic properties of the system. If, for example, data transfers or certain compute passes have highly dynamic properties and only operate at certain points in time it might be beneficial to generate multiple loopflow graphs for these different phases. It is important to consider, that FIFOs can often be used to smooth over data transmission happening in bursts. If this technique is used a constant resource usage can be achieved, however if the amount of data that is sent in a burst pattern is too large this has to be considered in the performance model (Part 3) and architecture (Part 4) later on. This is usually achieved by modelling the computation in different phases in which the different components are active or inactive.

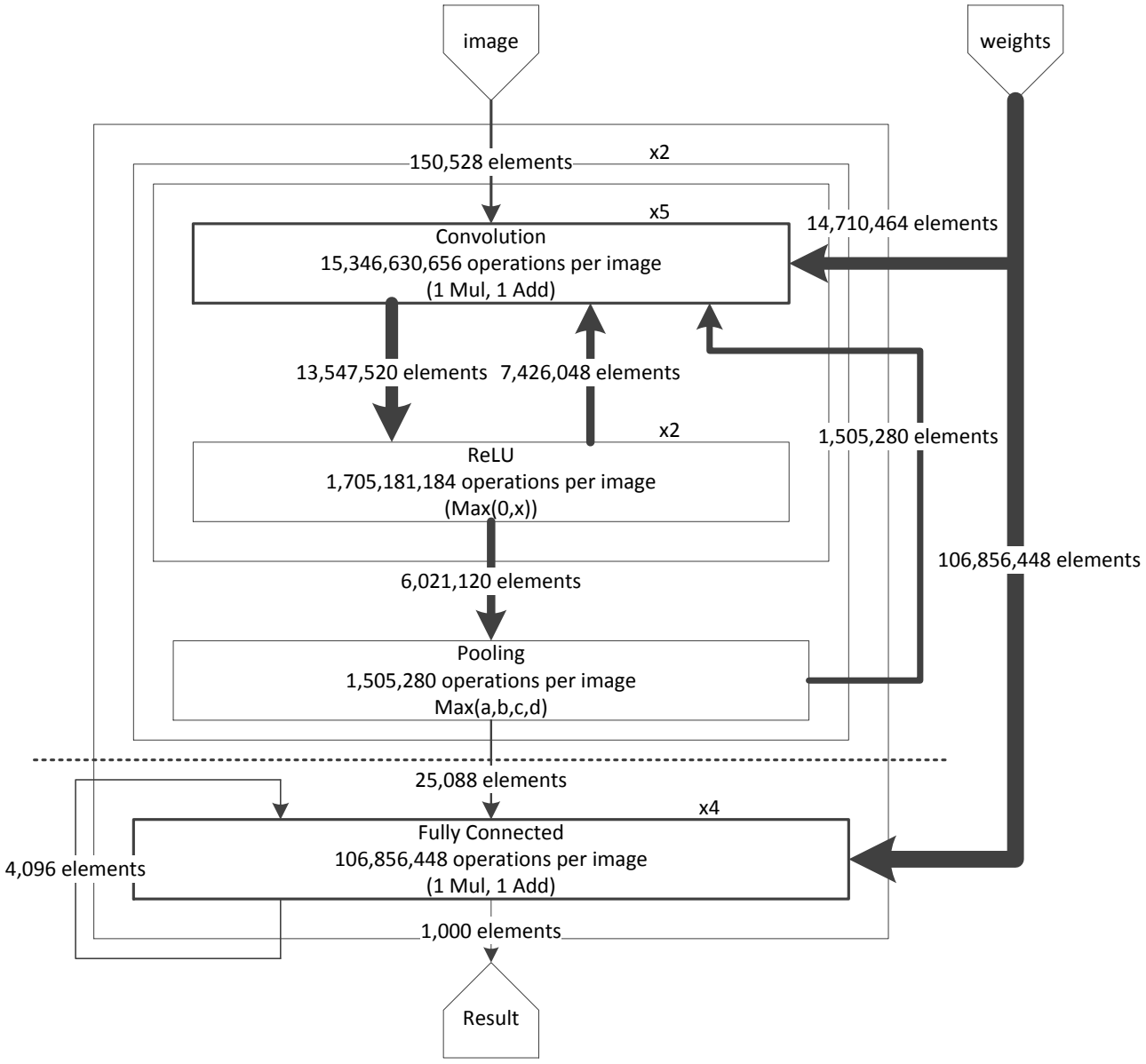

Figure 3.2: VGG-16 CNN Loopflowgraph. Boxes represent loops and their internal operation types and counts. Arrows show data movements, thickness hints transferred data amounts.

3.3 Software model

The software model (Part 2) is a simplified software implementation of the parts of the application which are to be ported onto the FPGA. It is not intended to be optimised for speed, but instead it should be an accurate representation of the intended FPGA implementation. This means that it should be as close as possible to the planned FPGA architecture and use the same API. For example, it should mirror changes to the algorithm, to verify correct functionality. It serves three different purposes.

1. Insight into the selected code and easier measurement of profiling results (Step b);
2. Testbed for verifying numeric and algorithmic changes (Step f); and
3. Debugging reference for the FPGA implementation (Step f).

In order to achieve the first goal of the software model the algorithm can be implemented as a simple, not optimised code, helping with the analysis described in section 3.2.1, and as a result the lessons learned from this implementation are used to refine the performance model and architecture. In the software one would not profile runtimes, but for example memory access patterns or the number of loop iterations executed. Similarly, the software model will be used to quickly evaluate different algorithmic and numerical options, which then feed back into the architecture and performance prediction. As a result, the software model is typically co-developed with those two components.

It is important to integrate the software model back into the original application. This ensures that the planned API between the original application and the accelerated modules is sufficient to recreate the original functionality. It also enables the verification of the software model by comparing the results of the original software and the application using the software model.

Using the same API for the software model as for the FPGA implementation also creates an easy-to-use debugging tool for the FPGA implementation. The reason to not only verify against the original application is that the software model is supposed to undergo the same

numeric and algorithmic changes as the FPGA implementation. As such it is easy to check if an unexpected result for a given input dataset is due to an error in the FPGA design, side effects or fundamental problems with the selected algorithm or its numerical properties. Numerical and algorithmic problems are significantly easier to resolve in the software model than in an FPGA implementation. Additionally, this provides an FPGA mock-up implementation for earlier integration into the host application.

While the creation of the software model and especially its continuous improvement during the design process represents an overhead in terms of design effort, it avoids the need to make these design iterations in hardware. It is widely accepted that software development, especially if code is not optimised for performance, is easier to accomplish than hardware development, meaning that it is easier and cheaper to perform these design iterations in software.

3.3.1 Numerical Analysis

One of the most crucial and often also hardest steps in the process of porting an application onto FPGAs is the change from floating-point to custom fixed-point arithmetic. This is due to fixed-point number formats having a limited dynamic range, which is fixed at compile time. As such the numerical properties of the algorithm to implement need to be well known, which is especially complicated regarding operations that have a big impact on the value range, e.g., multiplications or exponential functions.

However, in order to maximise the usage of the available hardware resources this is usually necessary, since for example, a floating-point addition needs roughly one order of magnitude more logic resources than a fixed-point addition [149,151]. The reason for this is that exponent alignment requires expensive barrel-shifters before and after each floating-point operation.

The usage of fixed-point operations will, in most cases, allow significantly more operations to be placed on the FPGA. However, if a design is mostly constrained by the I/O bandwidth and there is no option to remove this bottleneck, floating-point precision might still be sufficient. This can be determined using the performance model (Part 3).

In general, it is of importance to consider the precision required by the final application. In many cases the algorithm or model used by the application has inherent limitations in terms of achievable precision. This means that it is not necessary to use datatypes which can represent numbers with a higher precision. Additionally, one can consider reducing the achievable precision of the application and accept a slightly larger error for additional performance gains. The selection of the correct precision and reduced precision implementations are common optimisation techniques in FPGA design.

The first step in order to generate a fixed-point implementation is to understand the numerical properties. This can be achieved using tools which collect data during the execution of the software. Usually, the easiest way to integrate these tools is by instrumenting the software model which is supposed to use the same number representation as the final design. As in section 3.2.1, it is crucial to select a representative input data set to ensure correctness of the finished system.

An example for such a tool is Maxeler's proprietary value profiling library. This library offers tracing variables, which behave like normal floating-point types, but are able to record the value distribution. This is implemented by having an array of integers representing the possible exponents of the floating-point value under observation. Each time the observed variable is updated, the exponent value is read, and the according member of the array is incremented.

It is possible to dump this array to disk at any given time using a function call and additionally it is possible to turn monitoring on and off. Fig. 3.3 shows the value distribution of the result of the convolution operation in a CNN during training. Each training batch is recorded as a separate iteration and 20,000 iterations are then aggregated to generate the presented heat-map, showing the value distribution over the complete training period. Using these data educated decisions can be made regarding possible fixed-point types.

Using the Maxeler library it is not only possible to instrument CPU code to generate data for visualisation, but it is also possible to use the Maxeler hardware simulator to generate similar histograms based on the actual behaviour of the accelerator.

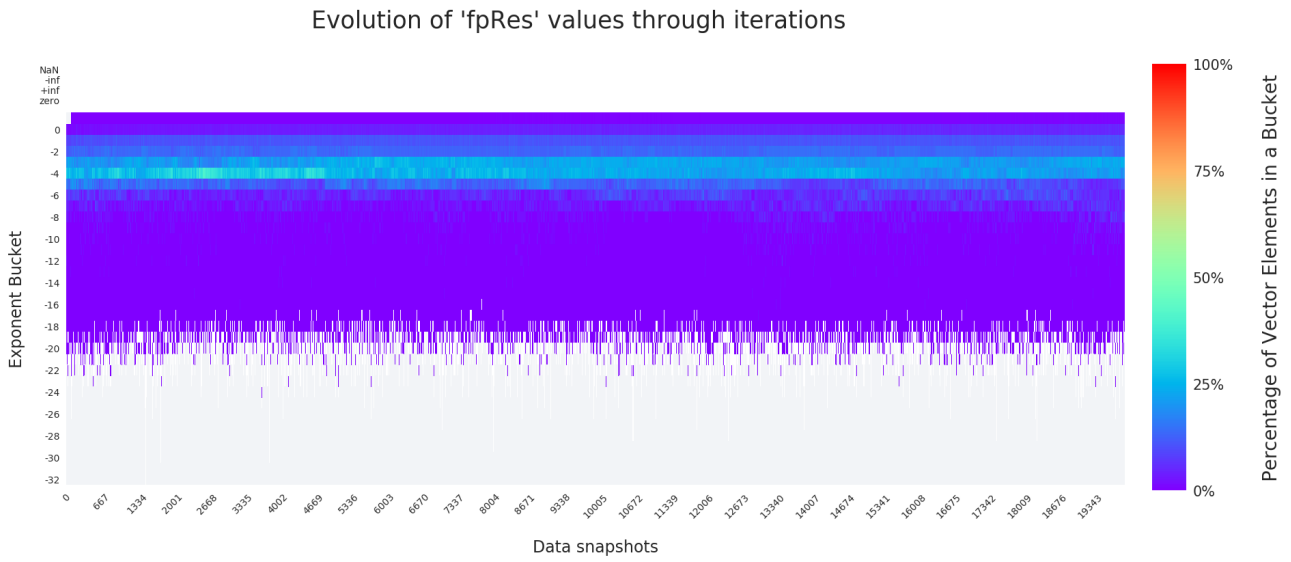

Figure 3.3: Heatmap of the exponent distribution for a convolutional layer during training. The y-axis shows the exponent of the floating-point values the variable holds. The x-axis represents different data snapshots, which are collected over the execution time. The colour indicates the percentage of occurrence for the individual exponent buckets.

3.3.2 Bit-level Accurate Fixed-Point Simulation

Using the results of the numerical analysis the software model is adjusted to use fixed-point arithmetic and deliver a functionally correct fixed-point implementation. It is important to note that choosing minimal data width has the potential of reducing resource utilisation. The functional correctness is verified by comparison with the original software using a representative data set.

The problem of floating-point to fixed-point conversion is widely covered and there are many tools available addressing this problem [16, 21, 22, 28, 64–66, 117]. However, in the scope of this work a small library to simulate fixed-point datatypes was developed, to provide an easy-to-use drop-in floating-point replacement.

The library is designed to work in conjunction with the Maxeler toolchain. As a result, it supports all three rounding modes which are supported by MaxCompiler, which is one of the main differences compared to other fixed-point libraries, which normally do not support different rounding modes. The type creation is based on C++ templates. As such any number of integer and fraction bits is possible in combination with all rounding types, but each type

is differentiable at compile time. All common arithmetic operations are supported as well. Internally the library uses 128 bit unsigned integers and scales them appropriately to convert between different datatypes.

Additionally, to the functionality as a fixed-point simulator the library also allows the logging of overflows. Overflows are logged with a stack trace and a user can attach additional information, e.g., a loop counter, as supportive information from within the application source code. Further it is possible to log the precise values after every calculation and every assignment if required.

Performing the fixed-point conversion at an early stage of the design process enables much larger acceleration structures, improves performance model precision and avoids complex debugging of numerical problems in hardware.

If the dynamic range of the algorithm is so large that the required data width causes problems, further strategies can be used. One option is the usage of a method similar to denormalised floating-point, where a shift value, similar to an exponent, is used in addition to the bits representing the significand. This reduces the resource usage if either the shift value can be applied to multiple values at the same time (this is also called block floating-point [102]) or if renormalising is only needed sporadically and not after every operation. This solution is for example often used to solve partial differential equations for the oil and gas industry, where a wavefield decays over time, but all values in the wavefield can share the same shift value [108]. A different solution is to handle overflows on the CPU, if they only occur rarely.

3.4 Forecasting System Properties

With the results of the code analysis (Part 1) and the initial version of the software model (Part 2), an accurate performance model (Part 3) capturing the implementation of the planned architecture (Part 4) on the selected platform can be built. In order to achieve this, the hardware and bandwidth usage, as well as the execution time for a problem of a defined size, are estimated. A preliminary speed-up expectation is thus obtained.

The performance model enables the design space exploration but also emphasises potential problems and bottlenecks (Step e) of the architecture before the implementation is started. Since the architecture tries to alleviate the bottlenecks highlighted by the performance model (Step d), an iterative improvement of performance model and architecture is often necessary. As such, it is possible to perform design space exploration early on in the acceleration process, enabling cost-benefit analysis and resulting decision making.

Additionally, it is possible to perform a cost-benefit analysis for the different feasible architectures and design points identified. This means that dependent on the requirements, the available resources and the achievable speedup figures, the best architecture can be selected, or the project can be reconsidered, if the costs exceed the benefits. In my experience it is usually possible to reach this point very early in the design process, significantly reducing the risk of the FPGA porting project to fail under great cost.

Estimating the benefits of the project that can be achieved given a certain architecture is straight forward using the performance model, since it accurately predicts the expected speedup. A sufficiently experienced developer can judge how complicated it will be to implement a given architecture. Using the performance model an optimal balance between FPGA and CPU resources in the final system can be found and consequently the costs of the final system can be estimated with high accuracy. This provides all the numbers necessary to perform the aforementioned cost benefit analysis.

Since FPGAs consist of essentially predictable building blocks, it is possible to estimate the performance for a given architecture very accurately using only a few simple equations. This is especially true for statically scheduled, deeply pipelined applications as created by MaxCompiler, which can hide potential I/O latencies. In contrast, on a CPU or GPU, performance improving components like caches and branch predictors greatly limit the ability to predict the performance of a given application. Even though fine grained models and simulators are developed to simulate these side effects, their accuracy is still often limited [51, 112].

The performance model presented here is based around Maxeler's programming model and requires the ability to accurately predict resource usage, performance of components and achiev-

able frequencies. It does not use specific constructs of the programming model. As such other toolchains with similar properties can be used as well. Furthermore, it is necessary that the application can be accurately analysed and split into phases in which computational and data transfer patterns are similar. If the latter is not the case the accuracy of the resulting model might be worse and worst-case assumptions have to be made.

The time it takes to process a given workload on an FPGA (T_{tot}) can be represented as the sum of the time it takes to initialise the FPGA (T_{init}), for example to set up DMA requests, set registers or to fill the computational pipeline and the time the actual execution takes (T_{exec}). This is shown in equation 3.1.

$$T_{tot} = T_{init} + T_{exec} \tag{3.1}$$

If the workload is sufficiently large, which is normally one of the prerequisites for FPGA acceleration, the execution time is supposed to dominate over the initialisation time so that it can be safely ignored. The precise initialisation time depends on the used platform and framework. For example, the initialisation time of a Maxeler system is usually between 1 and 100 ms. Additionally, it is possible to take the time for reconfiguration into account. If regular reconfiguration is not required during execution, this can be modelled as an additional part of the initialisation. Again, the time required for reconfiguration is dependent on the platform and the size of the FPGA configuration memory and is usually between 100 and 2000 ms. For example, reconfiguring Maxeler's MAX4C DFE requires roughly 100 ms while the newer MAX5C DFE requires more than 2 seconds for reconfiguration, due to the bigger chip size and a different physical implementation.

In a streaming dataflow design the execution time is the maximum of the time it takes to perform the computations performed by the algorithm (T_{comp}, section 3.4.2), the time it takes to transfer the data between host and FPGA (T_{comm}, section 3.4.3) and the time required to transport data between FPGA and on-board memory (T_{mem}, section 3.4.4). As a result, the longest latency dominates the overall execution time of the accelerated task (eq. 3.2) and

becomes the primary focus.

$$T_{exec} = max(T_{comp}, T_{comm}, T_{mem}) \tag{3.2}$$

It should be noted that this is only the case if all resources are utilised at a constant rate with respect to time. This, however, may not always be the case, e.g., it might be that all communication to the host has to happen before the compute can start. When the utilisation is non-constant, but its moving average is, contiguous flow can be mimicked by sufficiently large buffers smoothing the effect over time. If this is not possible the execution has to be split into multiple parts, in which each part represents on state of the application (e.g., certain data transfers and computations active or not active). For each of those parts an individual performance model can be used to calculate the resulting execution time. The execution time of these parts as has to be treated as an additive term to the overall runtime.

3.4.1 Predicting Area Usage

In order to determine the time requirements for the computation one first needs to find out what degree of parallelism is achievable for a given device. This is mainly limited by the hardware resources available on the FPGA fabric.

In general, the FPGA hardware resources are used for three different purposes. First, to implement the arithmetic operations, second the scheduling of operations and finally other IP modules, e.g., the PCIe, memory controllers, etc.

In order to predict the area usage for arithmetic operations it is first necessary to find out the amount of hardware resources required to implement a simple operation on the target FPGA device. Those figures can be determined either by using automated tools, finding appropriate tool and FPGA vendor documentation (e.g., [6, 7, 149–152]) or by creating micro applications and running them through the vendor tools. The overall area usage can be estimated as the area cost of a single operation multiplied by the number of operations to be implemented on

the fabric.

Additionally, some tools might provide an API to query this information. For example, Maxeler's MaxCompiler toolchain provides an API, which provides area estimations for single operations and memories, based on the used datatypes and current configuration. This configuration includes for example the level of pipelining as well as the currently used rounding mode.

The scheduling is implemented using FIFOs, which usually use on-chip memory resources, which will be discussed in more detail later in this section, or registers.

For IP modules, the resource requirements can typically be predicted using micro benchmarks. In general, it is recommended to assume a slightly higher memory and logic usage, in order to keep some safety margins especially for scheduling but also additional control logic.

Conditional statements

On CPUs it is only necessary to execute the selected branch of a conditional statement. However, in hardware it is necessary to implement logic which can deal with all possible branches. The final result is then selected from all computed alternatives using a multiplexer. As such the resource requirements for the different branches need to be accumulated together with the cost of the multiplexer.

In cases where only one branch is active at a time, resources can be time shared. If for example two branches perform a multiplication, one can use multiplexers on the inputs of the multiplier instead of on the output. This can be done manually or automatically using tools like Maxeler's Kernel Merger [136].

As described above it is beneficial to collect data on the likelihood of each branch being taken. This can be used to decide on the used implementation strategy and if it might be required to model individual stages of the computation separately. A major factor here is the size of the branch and if it is present in an inner loop or at the top level of the application. In the case of inner loops, it is usually necessary to implement both branches, while branches on the top level can lead to completely different designs. Having completely different designs might not

be highly problematic if there is no dependency between different data items processed and the likelihood of each branch being taken is known. Then the system can be built to include all required designs separately and replicated in such a manner that the overall data set can be computed without any design being idle for a significant amount of time.

Dealing with loops

In general, loops either have to be fully unrolled, partly unrolled or not implemented at all. If a loop is unrolled, the operation count has to be multiplied with the unrolling factor. Unrolling is especially recommended, if dependencies between loop iterations exist. If the loop cannot be fully unrolled due to area limitations, the loop is normally implemented in a pipelined fashion. This means that the loop is only partially unrolled and then the full loop is computed iteratively across multiple cycles. Again, the operation count has to be multiplied by the number of unrolled loop iterations to generate an accurate resource model. While in the case of for-loops it is usually straight forward, to determine how many loop iterations need to be implemented on the surface of the reconfigurable fabric, for while-loops this is not the case. If possible, all while-loops should be transformed into loops with a fixed number of iterations. If this is not possible the loop cannot be completely unrolled. In order to estimate the computation time one can assume the average number of iterations for the average case and the maximum number of iterations for the worst case.

Predicting on-chip Memory Usage

On-chip memory is normally used for three different main purposes:

1. in operations and IP-blocks, e.g., DDR memory, FPGA to FPGA communication or PCIe;
2. for on-chip buffering or reordering of data; or
3. for scheduling of the Kernel's dataflow graph.

Predicting the area usage for the first case is straight forward, since it can be estimated with micro benchmarks. The same can be done for the second case, however it is also possible to model it using eq. 3.3, where the number of memory blocks required (n_{mem}) is calculated as the product of the width (w) required divided by the native width of the on-chip memory read and write ports, the depth (d) required divided by the possible depth of the hardware unit and the number of read ports (p) required divided by the number of read ports physically present.

$$n_{mem} = \left\lceil \frac{w_{req}}{w_{hardware}} \right\rceil \times \left\lceil \frac{d_{req}}{d_{hardware}} \right\rceil \times \left\lceil \frac{p_{req}}{p_{hardware}} \right\rceil \tag{3.3}$$

The on-chip memory resources of FPGAs can often be used in different aspect ratios. For example, the same memory block might be configured with an aspect ratio with a width of 18 bits and a depth of 1,024 or with a width of 9 bits and a depth of 2,048. It is possible to tile logical memories to fit into these physical on-chip memories of different size. The total hardware cost of the memory is the sum of the costs for each individual tile.

Predicting operation scheduling resources accurately is nearly impossible, since this would require deep knowledge of the scheduling algorithm used by the toolchain or manual scheduling. However, it is possible to identify the largest FIFOs needed to access data from previous cycles and treat them as normal buffers. The amount of memory resources required by the smaller FIFOs is highly dependent on the application. As such only a rough estimate is possible.

3.4.2 Predicting the Compute Performance

After having estimated the area usage, one can predict the achievable level of parallelism and the number of data items processable per clock cycle, ignoring bandwidth limitations.

The time needed to compute a set of data items can be calculated as shown in eq. 3.4, by dividing the number of items that need to be processed by the product of the targeted frequency and

the number of items processed per cycle.

$$T_{comp} = \frac{n_{items}}{items\_per\_cycle \times f} \tag{3.4}$$

The frequency cannot be precisely predicted. However, if sufficiently deep pipelining is applied and no more than 80% of the chip resources are used, the frequency for different designs using the same FPGA can be safely estimated based on previous experience or experiments using artificial designs to fill up the chip if a similar implementation scheme is used. The systolic array style implementation scheme used by MaxCompiler for example results in a highly predictable frequency. A Maxeler FPGA card using an Altera Stratix V FPGA usually achieves 200 MHz if the chip is only filled up to 80%.

3.4.3 Predicting I/O Bandwidth Usage

The bandwidth between the host and the FPGA depends on the physical interconnect used. For most acceleration platforms the interconnect is PCIe. PCIe is built by bidirectional links, meaning that the full bandwidth can be used in both read and write direction at the same time. For example, second generation PCIe can transport up to 4 GB/s over an eight-lane link. The full speed of the link is usually not achievable due to overhead introduced by the protocol and encoding schemes. In the case of DMA using MaxCompiler a bandwidth of 3.5 GB/s can be achieved. If data is transported in other ways, the achievable bandwidth might be even lower and needs to be measured with a benchmark.

As a result, the communication time between host and accelerator can be estimated using the maximum data sizes transferred in either direction (S_{in} and S_{out}) divided by the bandwidth (BW), as written in eq. 3.5.

$$T_{comm} = \frac{max(S_{in}, S_{out})}{BW} \tag{3.5}$$

3.4.4 Modeling On-Board Memory Behaviour

On-board memory for FPGA accelerators is mostly based on DDR memory. The bandwidth to this memory is unidirectional, meaning that the bandwidth is shared between both directions (read and write). As such the time it takes to transfer data between on-board memory and the FPGA can be calculated as shown in eq. 3.6.

$$T_{mem} = \frac{S_{write} + S_{read}}{BW \times efficiency} \tag{3.6}$$

The parameter *efficiency* represents the proportion of the theoretical DDR bandwidth achievable for a given data set. Only large linear access patterns allow efficient memory access. Fig. 3.4 shows the achievable memory efficiency based on the amount of data which are accessed in one linear read for DDR4 memory as used on an FPGA card. Different DDR based memory technologies behave in a similar manner. The term burst refers to the native word width of the DDR memory.

The location in memory at which the data are stored also has significant impact on memory bus efficiency. The address space is divided into multiple ranks[2]. Due to the fact that if one rank is currently accessed the next rank can already be prepared, accessing data from different ranks will make sure that efficiency is improved. If only one burst at a time is used the efficiency will be halved, but if more bursts are accessed this penalty will be significantly reduced. For this reason, it is beneficial to distribute different chunks of data across the available address space.

3.4.5 Comparison to Roofline Model

The roofline model [143] is a commonly used tool to estimate system performance, perform design space exploration and identify optimal architectures. It was initially developed for floating-point programs executed on multicore architectures and is based around *operational intensity*, which is a measure describing the number of computations per byte accessed from

[2]All Dynamic Random-Access Memory (DRAM) chips sharing the same chip select [98]

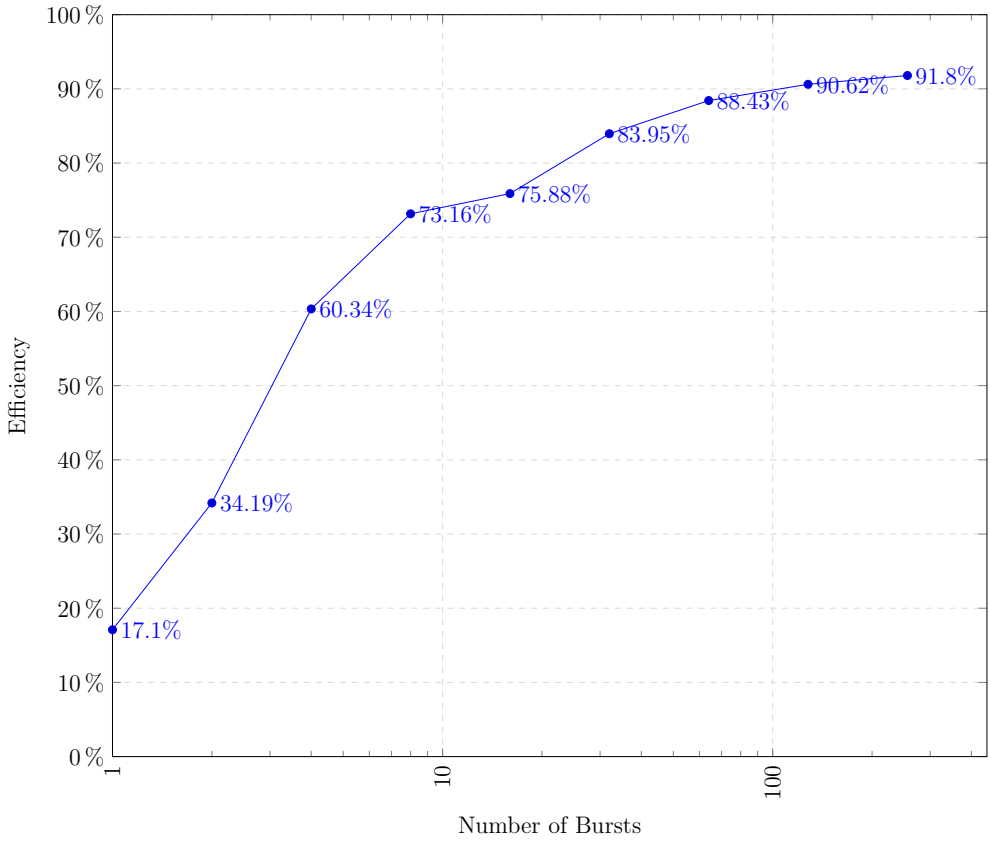

Figure 3.4: DDR4 memory efficiency of an FPGA card.

memory. The roofline model is a 2D chart, where both axes are logarithmic. The x-axis in this chart is the computational intensity, which represents the number of operations executed on each byte accessed from off-chip memory. The y-axis represents the performance in floating-point operations per second.

Using these two metrics one can add two ceilings, the roofline, to describe the maximum attainable performance of the system. The first ceiling, on the top right, is based around the maximum achievable performance in floating-point operations per second, while the second ceiling, on the left, is limited by the memory bandwidth. This roofline describes the maximal attainable performance given a specific processor at different levels of operational intensity. To evaluate an intended implementation one can identify the operational intensity of the implementation and determine the maximum achievable performance by comparing to the roofline. Additionally, it is possible to add multiple ceilings to tune the roofline model to different algorithm characteristics. For example, one can add multiple memory bandwidth ceilings based on

memory efficiency or multiple compute ceilings based on different operations and datatypes.

The roofline model was expanded to be also applicable to FPGAs in [31] and [30]. Fig. 3.5 shows an example roofline model for a VGG-16 CNN which will be discussed in more detail in section 3.6.1. It can be used to quickly provide an overview and a visualisation of the communication to computation ratio of an architecture and the resulting performance impact.

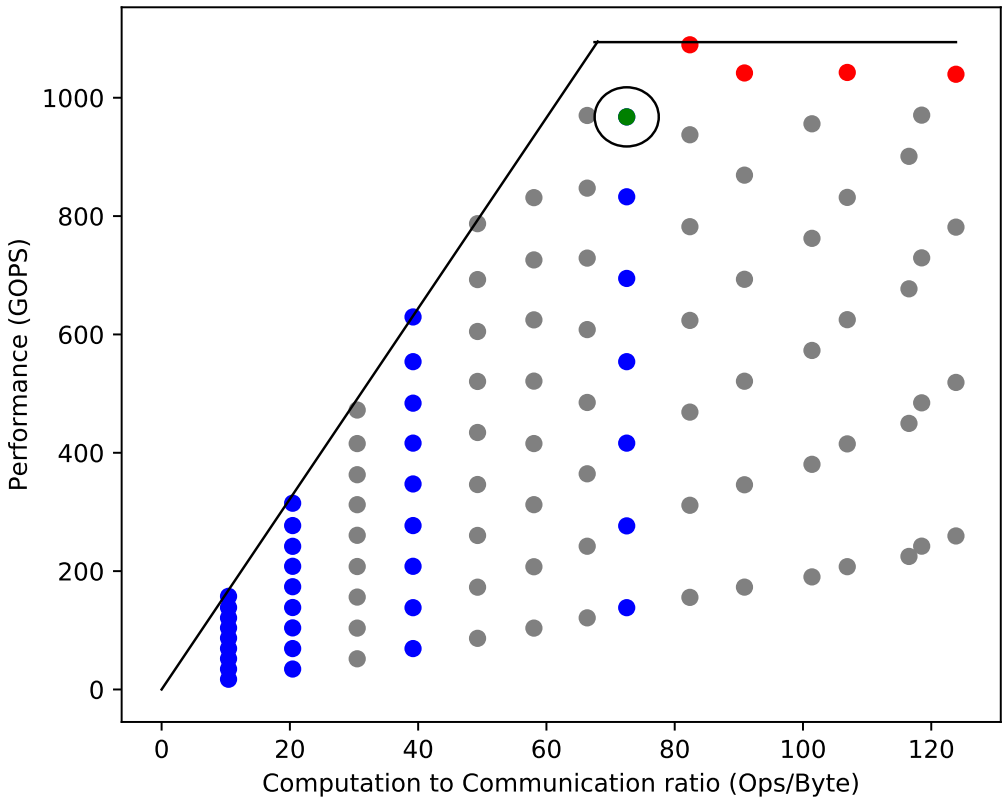

Figure 3.5: A roofline model for a chosen architecture of VGG-16 CNN. Individual dots represent potential design points. The green dot is the chosen design, blue dots are valid design points, red dots are not feasible, e.g., due to on-chip memory requirements, and grey dots introduce significant design complexity and are ignored as a result.

One can see how the roofline model can provide a good overview of the possible design points and, as a result, ease decision making, however, it also has a few shortcomings. First and foremost, the model can only focus on one I/O bandwidth at a time. In reality, systems are not only limited by external memory but also host communication bandwidth. Furthermore, some systems might additionally have heterogeneous memory architectures, e.g., High Bandwidth Memory (HBM) and DDR, or accelerator to accelerator communication. Integrating those into the simplified roofline model is often not achievable.

Factors like on-chip memory capacity are not considered and would need to be addressed separately. Another problem is that for FPGA based systems the computational roofline inherently includes the frequency the computation is run at. However, the achievable frequency is highly dependent on the routing complexity and the overall resource usage. The proposed methodology works around this problem by using MaxCompiler with its very predictable frequency properties. Finally, the roofline model is based on the assumption that the overall system performance is constrained by one computational kernel focused on a specific type of computation. Even though this is the case in many scenarios, especially highly complex applications benefit most from implementing multiple computational kernels on the same accelerator. These computational kernels might even operate on different datatypes. In those cases, it will prove to be problematic to determine consistent computational ceilings.

To conclude, the roofline model provides a good tool to quickly perform design space exploration for a given architecture, however, the more detailed performance model described in this section provides a lot of additional information. This additional information can, in many cases, prove crucial in the discovery of better architectures. It is possible to convert a detailed performance model into a roofline model but not the other way round.

3.5 Architectural Optimisations

The details of developing an efficient architecture (Part 4) go beyond the scope of this thesis. However, some general concepts should be explained here, since only the development of a good architecture promises optimal results.

In general, the target is to achieve a balance, where all system resources are fully and equally used, and all bottlenecks of the system as indicated by the performance model (Step e) are removed (Step g). If, for example, all the memory bandwidth is utilised but only 50% of the chip resources are in use, a better architecture should try to save memory bandwidth. This will allow an increase of the degree of parallelism, using more of the chip resources and reducing the overall execution time.

The techniques described below are very general and can be applied to applications developed using other methodologies as well. The main advantage of the proposed methodology is the detailed knowledge about bottlenecks as provided by the performance model (Part 3). While other methodologies and performance prediction tools, e.g., the roofline model, might also provide knowledge on the existence of certain bottlenecks the detailed performance model also provides information on where optimisations might be most beneficial. For example, it informs on the size of all memories or which functions use, e.g., DSPs or a lot of memory bandwidth and as a result one can quickly judge which problems to tackle first. This insight into the details of the design enables fast design iterations.

3.5.1 Improving Bandwidth Utilisation

Reducing the memory-to-accelerator or host-to-accelerator bandwidths are similar. There are three main strategies to achieve this. Firstly, one can buffer more data on-chip, reducing the need for streaming it back and forth. Secondly, customised compression can be used. This could be as simple as changing the data width (e.g., from 32 bit to 12 bit integers). However, more advanced compression algorithms, e.g., run length encoding, might provide additional benefits. Thirdly, data sequences with predictable values can be generated on the FPGA, avoiding transmission altogether.

In the case of on-board memory, it is often also possible to increase memory efficiency, e.g., by reordering data on the chip just before it is written or after it is read in order to create linear access patterns. Reordering operations to reduce the amount of data that need to be moved is also an option. Additionally, it is possible to optimise the access into the different ranks of the DDR memory, as discussed in section 3.4.4, by using an optimised memory layout.

3.5.2 Reducing Area Usage

Usually the most important step, in order to make efficient use of the available hardware resources, is the selection of the datatype and width. It is usually inefficient to use floating-point

arithmetic, but even if fixed-point arithmetic is used, smaller data widths use less resources.

Otherwise, it is possible to replace area expensive operations with cheaper ones. One option to achieve this is by considering alternative implementations of a given algorithm. Even if those implementations are less efficient on a CPU, they might be cheaper to implement in hardware. For example, sorting networks are often used in hardware, while on CPUs other algorithms, e.g., merge sort, are usually preferred. Moreover, one could consider moving calculations to the CPU or precompute some values, if possible.

The on-chip memory usage can easily be reduced by moving buffers into on-board memory. If double buffering[3] is used, it is sometimes possible to recreate the same functionality and read-write speed with only one memory by developing a custom addressing logic. This custom addressing logic has to ensure that only elements which are already read are overwritten and usually involves switching between multiple complex addressing patterns [135].

3.5.3 Overlapping Host and Accelerator Execution

Usually, applications are not fully ported onto FPGAs, but instead less compute and more control heavy code parts remain on the CPU. The simplest way of integrating the FPGA into the CPU implementation is to call the FPGA as a function. As shown in fig. 3.6 a), this leads to an inefficient usage of the available hardware resources, where only one of the resources is fully used at any time.

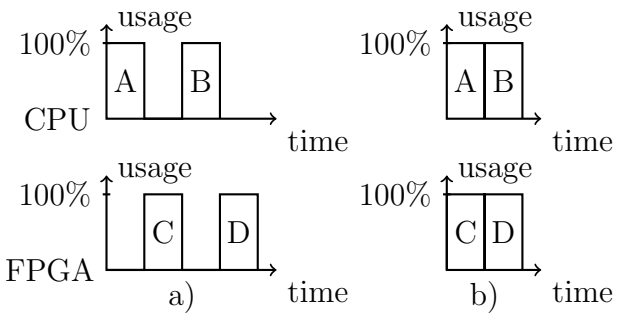

Figure 3.6: Suboptimal vs overlapped use of resources.

[3] A technique with two copies of the buffer is used to resolve write before read dependencies. One is written, while the other is read.

A better way to handle this integration is to overlap the execution as shown in fig. 3.6 b). One way to achieve this is by calling the FPGA in a separate thread. If the CPU and FPGA calculations are independent, then it is easy to parallelise them. Otherwise, it is recommended to have the CPU and the FPGA work either on different parts of the same dataset or completely disjoint tasks.

3.6 Evaluation

To properly evaluate the proposed methodology, one would need to compare the complete development process of an application using the proposed methodology and other methodologies. Then one could compare the quality of result in terms of achieved performance, maintainability and area usage as well as the designer productivity usually measured in development time. The problem with this kind of evaluation is to isolate the impact of the methodology on these metrics. For example, if the same developer implements the same application multiple times using multiple methodologies the designs created later will benefit from the experience made earlier on. Similarly, if different designers are used to perform the evaluation one would need to consider the individual skill and experience of each developer. These factors are hard to quantify and as a result the only way to measure designer productivity with any kind of certainty in the results is by designing a bigger study where for each application and methodology pair multiple designs are created by different engineers. Performing such a study is very resource intensive and beyond the scope of what was achievable in this thesis. Similarly, it is not possible to compare the designer productivity based on related work since most publications do neither state the time it took to develop the design nor the used methodology.

As such, I decided to compare the methodology to the development processes currently used by comparing the quality of the resulting implementations against published work. While this does not consider designer productivity, it does show that the methodology is able to produce state of the art designs, which is an important consideration in the selection of a development methodology. This comparison does not manage to compare the proposed methodology

against a specific methodology, but only the different not described processes which were used in the creation of the specific designs used for comparison. I assume that this presents a fair representation of how FPGA designs are currently created within academia and industry.

In this section the proposed methodology is evaluated on two realistic applications. One is the acceleration of the inference of the VGG-16 [118] CNN discussed in section 3.6.1 and the second is the acceleration of BQCD which is discussed in section 3.6.2. The latter was independently performed by Maxeler Technologies using the methodology described in this chapter.

VGG-16 is selected as a use case since it is a very commonly used CNN and there is very high interest in research on efficient CNN implementations on FPGAs. As such there is a well-defined state-of-the-art with highly optimised and current designs. This enables a good evaluation of the quality of results achievable using the proposed methodology. BQCD is selected to show how the methodology can help on a very large and complex HPC application. To my knowledge there exists no other FPGA based implementation of BQCD.

3.6.1 Convolutional Neural Network

The details of the analysis (Part 1) and software model (Part 2) for the CNN are not described here, since their results are already encapsulated in fig. 3.2. The loopflow graph also motivates the split of the application (Step a). In this case the convolutional layers are ported onto the FPGA, while the fully connected layers stay on the CPU.

First the architecture (Part 4) is introduced, and a series of applied optimisations is presented. Afterwards the performance model (Part 3) and the design space exploration based on it are shown. Finally, the resulting implementation and its performance compared to related work are discussed.

Architecture

In this section first an initial architecture (Part 4) is defined to determine what to model (Step d) in the performance model (Part 3). There are two fundamentally different options to implement a CNN on FPGAs. The first option is the implementation of a fully streaming architecture, in which hardware is dedicated to each layer separately and all layers are computed in parallel. However, this architecture often requires a prohibitive amount of on-chip memory, since on-board memory does not provide the required bandwidth. The second option is the usage of a generic hardware unit, often called a PE, which is able to perform the necessary computations for all layers. In this case only parallelism within a layer is exploited.

To estimate the on-chip memory usage of the fully streaming architecture, it is possible to calculate the memory needed to store the results of the first layer. If a resolution of 224x224 pixels is assumed, the 64 output channels of the VGG-16 CNN produce in total 3,211,264 elements of data. This means that approximately 3MB of on-chip memory is required if each element occupies one byte of memory.

As a result, it is not feasible to implement the fully streaming architecture (Step e). Instead, the PE based architecture shown in fig. 3.7 is selected. Each PE performs the convolution operations, accumulation over input channels and the ReLU activation functions [163]. To save on-board memory bandwidth, all PEs operate on the same input channel producing different output channels, but the units performing pooling and writing data to on-board memory are shared between PEs.

Optimisations

In this section a series of different optimisations to the proposed architecture are introduced. The main goal of the applied optimisations is to fully utilise all available arithmetic resources in addition to maximising usage of the on-chip memory.

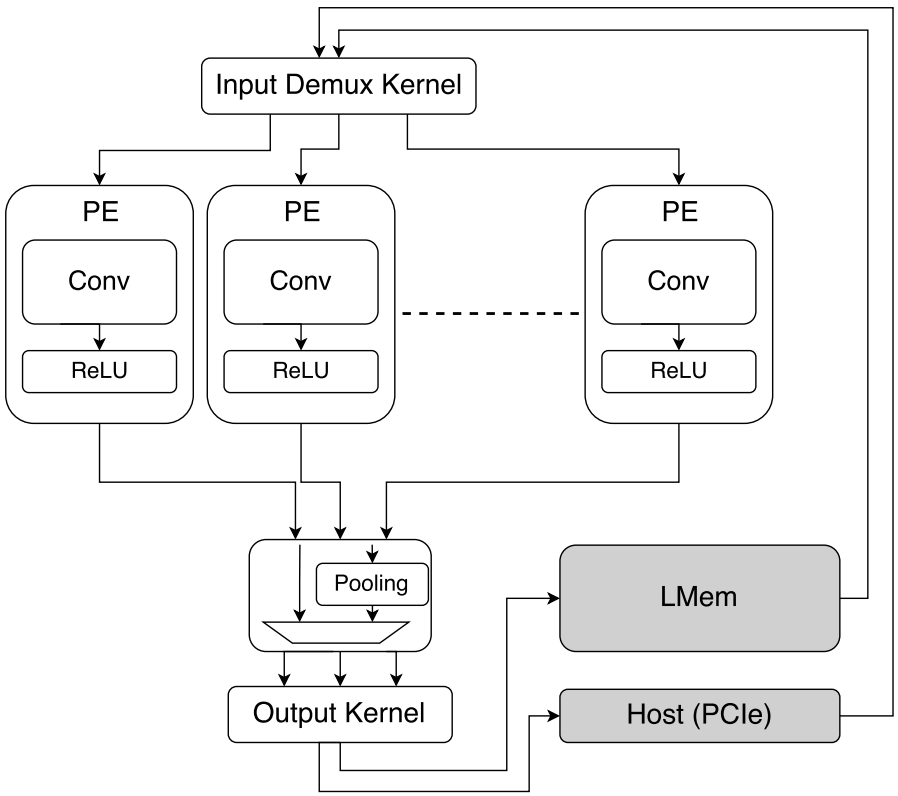

Figure 3.7: Convolution design architecture, PE connectivity.

Processing Elements count The number of used PEs defines directly the required resource utilisation and memory bandwidth. In particular, the more PEs are used the more on-chip memory is needed, but on the other hand the memory bandwidth requirements are reduced, as less pixels have to be processed per cycle. In addition, more PEs mean that less iterations have to be performed to generate all outputs. To simplify control logic and to avoid stalling, the number of PEs is a divisor of the specific number of outputs.

Processing of Multiple Inputs in Parallel An option to narrow the PEs' memory write port width is to process multiple inputs in parallel. While this has no direct impact on the required off-chip memory bandwidth, it requires more weights to be loaded at the same time.

Multiple parallel inputs can be used to make it easier to match the aspect ratios of the available on-chip memory or to fit the output size of each layer better to the available PEs.

Datatype Customisation CNNs are ideal for custom data representations [42, 124, 128]. It is possible to successfully use very low precision for inference as shown for example in [127] and also in training as shown in [109]. However, these very low precision types usually also lead to changes to the network architecture.

Because of the specific CNNs characteristics only fixed-point datatypes are considered in this work. The precise fixed-point type used is determined by simulating the system (with a device simulator or using software libraries for fixed-point computation) and checking the classification results on a test set against a reference floating-point version. A different option to optimise the trade-off between the used area and the achieved performance is to use asymmetric arithmetic. For example, the weights can be represented with less bits than the actual network data. This also helps when the port widths of the available hardware multipliers are asymmetric.

Modelling Host to Accelerator Communication

First the I/O communication time will be predicted, using eq. 3.5. For this the amount of data needed to be transferred to and from the FPGA would need to be determined. This consists of the weight and input image data.

The weight data are constant across iterations and as a result only need to be transmitted once. As such they can be neglected since the one-time transmission is of no consideration as soon as thousands of images are processed.

In cases when it is not possible to compute all output channels of the first layer in parallel, it is necessary to retransmit the input or buffer it in memory. Eq. 3.7 shows the situation when the data is retransmitted. n_{out} is the number of output channels of the first layer, n_{PE} the number of PEs and n the number of images transferred.

$$T_{comm} = \frac{n \times max(S_{in} \times \frac{n_{out,L1}}{n_{PE}}, S_{out})}{BW} \tag{3.7}$$

Eq. 3.7 becomes eq. 3.8, to estimate the number of items that can be processed in a given time.

To obtain the number of input items per second, T_{comm} is set to one second.

$$n = min\left(\frac{BW \times T_{comm}}{S_{in} \times \frac{n_{out,L1}}{n_{PE}}}, \frac{BW \times T_{comm}}{S_{out}}\right) \tag{3.8}$$

Forecasting Accelerator Computational Latency

Similarly, the accelerator runtime can be estimated as shown in eq. 3.9, where n again denotes the number of input images, OPS_{needed} the number of operations performed per input image and $OPS_{available}$ the number of operations that can be performed on the FPGA for the developed architecture within a given amount of time.

$$T_{comp} = \frac{n \times OPS_{needed}}{OPS_{available}} \tag{3.9}$$

The maximal number of processed elements, as limited by the on-chip computational capabilities, is given by eq. 3.10.

$$n = \frac{OPS_{available} \times T_{comp}}{OPS_{need}} \tag{3.10}$$

Convolutional layers mainly consist of MAC operations; hence the total number of MAC operations and the FPGA capacity are needed to estimate T_{comp}. This assumes the implementation of the ReLU and pooling functions inside the PEs as proposed in this architecture.

Estimating On-Chip Memory Usage

Depending on the on-chip memory size, it might be feasible to buffer all the input and output channels to each layer on-chip. However, on smaller chips, or for layers having either more or bigger output channels, this is not feasible. Similarly, there are networks like ResNet-101 or other residual networks, which reuse data from earlier layers [47], and therefore require even more buffer space. In those cases, it is necessary to store the results produced by the computation of each layer in on-board memory. Only this case will be modelled here.

Each PE works on a separate output channel, hence all the data produced by each PE need to be written back to memory.

Based on eq. 3.3, eq. 3.11 describes the amount of required on-chip memory, where n_{PE} is the number of PEs, p the number of output pixels created per PE per cycle and w the datapath width. Furthermore, $elem_{out,layer}$ presents the number of elements in a given output channel. MEM_{width} and MEM_{depth} are the memory port width and memory depth.

$$n_{mem} = \left\lceil \frac{n_{PE} \times p \times w}{w_{hardware}} \right\rceil \times \max_{layer} \left(\left\lceil \frac{2 \times elem_{out,layer}}{p \times d_{hardware}} \right\rceil \right) \tag{3.11}$$

The first term in eq. 3.11 represents the requirement to write all results produced by the PEs into memory on every cycle, while the second term represents the need to store the complete output channel calculated by each PE on-chip, employing double buffering. Only one read port is needed, removing the last term of eq. 3.3.

Double buffering is needed, to avoid writing all outputs to external memory at the same time. The latter would require a prohibitively large buffer to the on-board memory in order to smooth over this burst while consuming enough data at each cycle. Double buffering additionally improves the efficiency of the DDR memory if each output can be written back separately, since it enables data access in long continuous bursts.

Estimating On-Board Memory Bandwidth Utilisation

The amount of data that needs to be read to load the weights is shown in eq. 3.12, where n_{in} and n_{out} represent the number of inputs and outputs per layer and $S_{weights}$ represents the size per weight filter.

$$S_{weights,total} = \sum_{l=0}^{layer} n_{in,l} \times n_{out,l} \times S_{weights,l} \tag{3.12}$$

The amount of memory that need to be accessed for the data per layer can be estimated by eq. 3.13, where n_{in} represents the number of input channels of layer and n_{out} the number of output channels. S_X represents the size of one input channel, S_Z the size of one output channel

and n_{PE} the number of PEs and therefore output channels processed in parallel.

$$S_{layer} = S_X \times n_{in} \times \left\lceil \frac{n_{out}}{n_{PE}} \right\rceil + S_Z \times n_{out} \tag{3.13}$$

The boundary can be estimated as shown in eq. 3.14, where again the efficiency will be close to the maximum since all memory accesses are linear and most of them will use the maximum number of bursts.

$$n = \frac{BW_{mem} \times efficiency \times T_{mem}}{\sum_{layer} S_{layer} + S_{weights,total}} \tag{3.14}$$

Design Space Exploration

Using these equations from the performance model (Part 3) it is straight forward to perform DSE. The targeted hardware is a MAX5C DFE using the Xilinx VU9P FPGA. This FPGA consists of three separate die, called SLRs, which are mounted on the same silicon interposer. The communication between those die is a significant bottleneck.

To get around this problem, all three SLRs are treated as individual FPGAs. Each SLR is connected to one separate DDR DIMM and the PCIe bandwidth is shared.

The size of the images fed into the CNN is fixed to 224×224 images and as a result $150,528$ pixels need to be sent into the FPGA and $100,352$ elements need to be received. Assuming that 16 PEs are used the number of images that can be processed within one second can be calculated using eq. 3.8 as shown in eq. 3.15.

$$n = min(\frac{4GB/s \times 1s}{401{,}408B \times \frac{512}{16}}, \frac{4GB/s \times 1}{602{,}112B}) = 334 \tag{3.15}$$

The design is heavily constrained by the number of on-chip multipliers. As such only modelling of the multipliers will be presented here. If each of the 16 PEs processes 14 pixels per cycle, the number of multipliers used can be calculated as shown in eq. 3.16, where n_{filter} represents the

number of elements in each weight filter. This fits within one SLR, which has 2,280 multipliers.

$$n_{mul} = n_{filter} \times n_{PE} \times n_{pixelspercycle} = 9 \times 16 \times 14 = 2{,}016 \tag{3.16}$$

The number of convolutions required to compute one image using the whole network can be calculated as shown in eq. 3.17. The result is obtained, by accumulating the multiplication between the input channel count, the output channel count and the number of elements in the input channel for each layer.

$$OPS_{needed} = \sum_{l=0}^{layer} n_{in,l} \times n_{out,l} \times elem_{in,l} = 1{,}705{,}181{,}184 OPS \tag{3.17}$$

For the whole network 1,705,181,184 convolutions need to be computed for each image. The proposed architecture can perform, as just calculated $16 \times 14 = 224$ convolutions per cycle. Using eq. 3.10, one can calculate the compute bound processing speed based on the frequency as shown in eq. 3.18. For a frequency of 250 MHz this would mean that 32.8 images can be processed per cycle. The 250 MHz are determined based on previous experience with this version of MaxCompiler and the used FPGA.

$$n = \frac{224 \frac{OPS}{cycle} \times f \times 1s}{1{,}705{,}181{,}184 OPS} \tag{3.18}$$

It was decided that the data between layers should be buffered on-board. As a result, the DDR bound performance can be estimated using eq. 3.19. Per image, 355 MB of pixel data and 33 MB of weights need to be transported. The memory efficiency is estimated as 0.85 for this case.

$$n = \frac{14GB/s \times 0.85 \times 1s}{388MB} = 33 \tag{3.19}$$

As a result, the overall expected performance of all three SLRs is 99 images per second for all three SLRs combined. Fig. 3.8 shows other possible design points, where the equations are evaluated for the different degrees of parallelism and the area usage is predicted to discard all

invalid design points. Blue dots in the figure are considered in the DSE (chosen design point in green). Red and grey dots were not considered because of resources and complexity constraints.

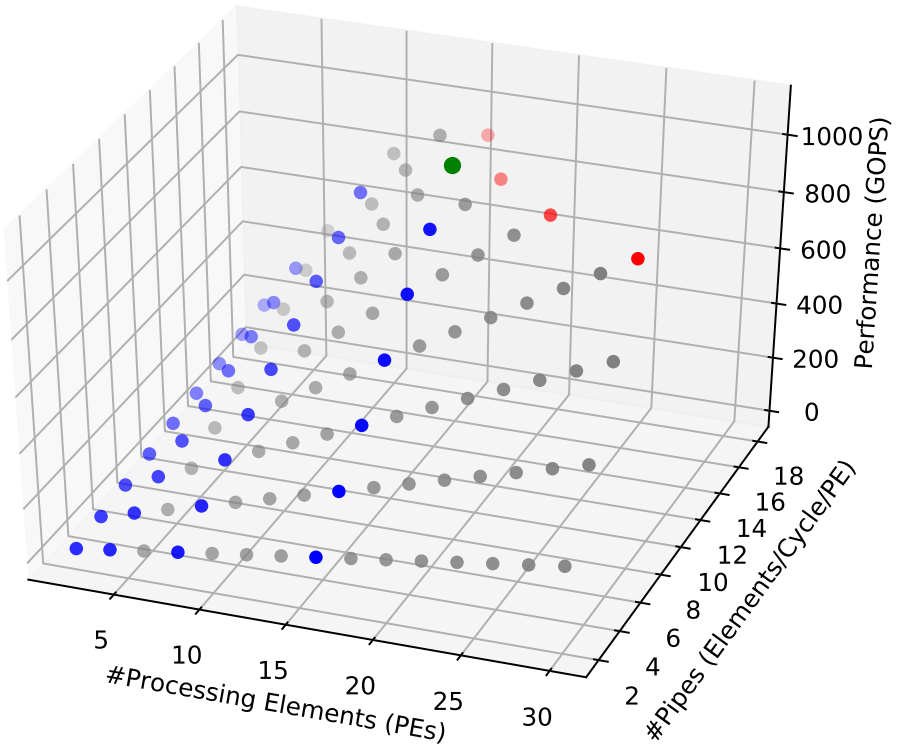

Figure 3.8: Design space of the proposed architecture for the VGG-16 network. Individual dots represent potential design points. The green dot is the chosen design, blue dots are valid design points, red dots are not feasible, e.g., due to on-chip memory requirements, and grey dots introduce significant design complexity and are ignored as a result.

It is possible to also represent the results of the design space exploration in a roofline model. This is shown in fig. 3.5. In this case only the memory bandwidth with full efficiency and the optimal computational performance for multiply-adds at 250 MHz frequency are considered to determine the ceilings. In comparison to fig. 3.8, the roofline model includes memory bandwidth, but does still not include on-chip memory requirements, which would increase with additional PEs, or PCIe bandwidth. As such, the more detailed performance model is required to determine invalid design points due to prohibitive on-chip memory usage.

Experimental Results and Discussion

Table 3.1: CNN Performance comparison.

	[164]	[89]	[142]	[13]	[165]	[87]	*This Work*
Implemented Network	VGG-16	VGG-16	VGG-16	AlexNet	VGG-16	VGG-16	VGG-16
Device	Xilinx VX690T	Intel GX1150	Intel GT1150	Intel GX1150	Intel GX1150	Xilinx ZVU102	Xilinx VU9P
Precision	16 bits fixed	8-16 bits fixed	8-16 bits fixed	16 bits shared exponent floating-point	16 bits fixed	16 bits fixed	18-27 bits fixed
Freq (MHz)	150	150	231.85	303	385	200	240
Logic cell (K)	300	161	313	246	-	600	788
SRAM (Kb)	$1,248 \times 18$	$1,900 \times 20$	$1,668 \times 20$	$2,487 \times 20$	$1,450 \times 20$	$1,824 \times 18$	$3,128 \times 18$
Multipliers	2,833	3,036	3,000	2,952	2,756	2,520	6,057
TOPs	0.488	0.645	1.17	1.38	1.79	3.04	2.45
Normalised Performance OP/Multiplier	1.148	1.416	1.682	1.543	1.686	6.032	1.685

Following the results of the design space exploration, the actual implementation was performed (Step g). Tab. 3.1 extends tab. 2.4 and shows hardware utilisation as well as performance of the proposed implementation in comparison to other state-of-the-art designs. Benchmarking of the CNN shows that the implemented design delivers 84.5 images per second at 240 MHz. This means that the error of the performance model is less than 15%. If a frequency of 240 MHz is used in the model the error decreases to less than 10%. This remaining difference can be explained by an overestimation of the on-board memory efficiency.

Comparing the performance of the proposed implementation with other state-of-the-art results of the year it was created in (2017), one can see that the proposed design is faster than [165] but slower than [87]. If the normalised performance is considered as well, one can see that the proposed implementation is on par with [142] and [165] and again slower than [87] and faster than the remaining designs. The difference in normalised performance between the proposed implementation and [87] is significant. The reason for this is that in the authors of that work use the Winograd efficient filtering algorithm [145] which reduces the number of multiplications necessary for each convolution. As such they can use the multipliers on the device with a significantly better efficiency.

It should be stressed that the methodology does not promise the best performance achievable in general, but only the state-of-the-art performance given a set of optimisations. For the architecture presented in this chapter the Winograd optimisation was not considered. One

example of a design using Winograd was still included in the comparison to show how the proposed methodology leads to a design which either outperforms or is on par with related work using the same set optimisations but is itself outperformed once a wider optimisation space is considered. Especially in the space of ML the huge research interest leads to rapid improvements and many novel optimisation approaches. For example, more recent work presented in [74] achieves more than 3.5 times as many operations per second per DSP at only two thirds of the frequency. As a result, every design developed using the proposed methodology only delivers a snapshot of the achievable performance considering the current known optimisations and in fields with such rapid progress regular re-evaluation of the architecture might be required. Due to quick performance estimation enabled by the proposed methodology it is possible to judge if an update to a design yields sufficient advantages to justify the required effort. For the same reason, a comparison to more recent related work is not provided, since the rapid progress in ML acceleration would only show the limitations of this specific implementation, but not contribute to the evaluation of the overall methodology.

3.6.2 Berlin Quantum Chromodynamics

The methodology was applied to a widely used Quantum Chromodynamics (QCD) application. QCD is the physical theory of strong interactions between subatomic particles, and Lattice Quantum Chromodynamics (LQCD) is an approach to computationally simulate such particles, based on discretising space and time into a 4D lattice [144]. Berlin Quantum Chromodynamics (BQCD) [104] is a popular implementation of LQCD. It natively supports timing its compute steps and a generated output file includes a detailed timing report that made utilities like gprof unnecessary (Part 1). Since the majority of the BQCD compute time was spent in its Conjugate Gradient (CG) solver, Maxeler focused on porting this (Step a). Overall BQCD contains roughly $200,000$ lines of source code illustrating the complexity of the application, motivating the need for a more elaborate design process and explains the inclusion in this evaluation.

The results presented below are compared with BlueGene/Q system with 32,768 cores. This

system needs 4.04 ms for one iteration on a problem size of 8,388,608 points in the spacetime grid. One DFE needs 5.34 ms for a problem of size 131,072. If linear scaling is assumed (which is valid at least for the DFE), 388 cores of the reference system achieve the same performance as a single DFE. Similarly, the complete system could be replaced by a system using 64 DFEs.

A CG algorithm [48] iteratively solves an $M\mathbf{x} = \mathbf{b}$ equation for the unknown vector $\mathbf{x}$, where M and $\mathbf{b}$ are known. The algorithm relies on repeated multiplication of M by conjugate vectors. In the case where M is sparse, rather than performing explicit matrix-vector multiplication, it is more efficient to infer the effect of M when applied to a vector $\mathbf{p}$ by referencing only selected elements of M.

In the case of BQCD, each lattice point contains a so-called spinor which, for the purposes of this thesis, is a block of 12 complex numbers, and the vectors are enumerations of the spinors of all the lattice points. The matrix M consists of repeated application of so-called Wilson $\not{D}$ (d-slash) and clover operators (both sparse) to the lattice of spinors. Rather than explicitly constructing M and computing a matrix multiplication, BQCD applies $\not{D}$ and clover to a lattice of spinors four times in succession for each CG iteration. For the application of $\not{D}$ the program needs to make reference to the current output site's neighbouring spinors (in 4D) and, furthermore, one so-called gauge matrix (3×3 matrices with complex elements) per neighbour.

Architecture

During performance modelling (Part 3), it became clear that the QCD application would be bound by on-board memory bandwidth (Step e) and, hence, most design decisions were taken in order to minimise on-board memory I/O (Part 4). The resulting architecture is displayed in fig. 3.9. Spinors, gauges and clover matrices are read from on-board memory and streamed into the CG algorithm. Each CGKernel performs a $\not{D}$ and clover operation and some perform additional $a\mathbf{x} + \mathbf{y}$ operations required by the CG algorithm. Some spinor vector results are output by CGKernel0, whereas the result of the matrix multiplication $M\mathbf{p}$ is output by CGKernel3. These results are streamed back to on-board memory to be used by the next CG iteration. CGKernels 0 and 1 accumulate the square norm of certain spinor vectors, which are fed to

the CGControlKernel at the end of a CG iteration, which uses these numbers to compute the scalars required for the next CG iteration. Note that to save chip space, the CGKernels process a lattice site only every 4 cycles. Since a CGKernel needs to reference all neighbours in 4D, its latency is effectively $2L_XL_YL_Z \times 4$, where L_i denotes the number of sites in direction i, and 4 the cycles needed per site.

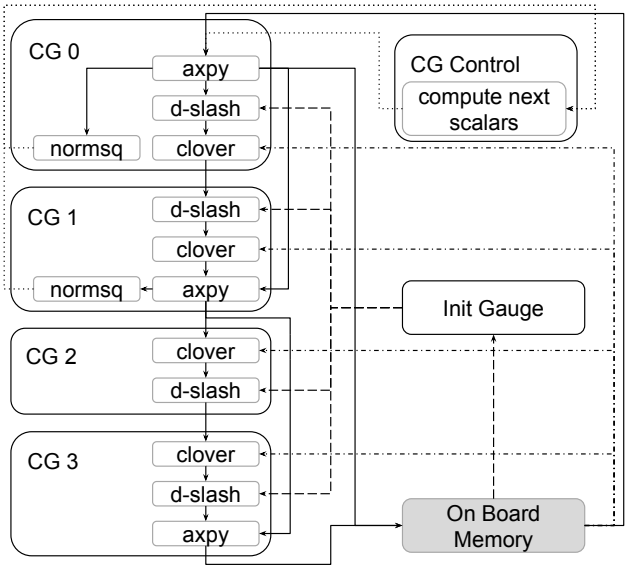

Figure 3.9: BQCD design architecture.

Performance modelling

Architecture (Part 4) performance (Part 3) can be modelled as follows (Step d). Eq. 3.20 and 3.21 show compute bound and memory bound times to solution.

$$T_{comp} = \frac{4L_XL_YL_Z(L_T + 11)}{f} \tag{3.20}$$

$$T_{mem} = \frac{L_XL_YL_Z\,(2,064 + 480L_T) \times 3\mathrm{B}}{BW \times \mathit{efficiency}} \tag{3.21}$$

The theoretical time-to-solution from eq. 3.22 is plotted in fig. 3.10 in yellow. The BQCD initialisation time is assumed to be 10 ms, while time for data transfer via PCIe is 11.7 ms.

$$T_{exec} = \max\left(T_{comp}, T_{mem}\right) + T_{PCIe} + T_{init} \tag{3.22}$$

Experimental Results and Discussion

Fig. 3.10 depicts the experimental results. Generally, the estimated results represent measurements very closely. However, at high frequencies, a discrepancy in time-to-solution of around 10% is observed. This discrepancy is due to the assumption that data accesses to on-board memory can be averaged over the run. In reality, on-board memory I/O varies during a CG iteration, being lower for halo sites than when computing ordinary sites. This means that there will be a transitional frequency range where part of the execution is on-board memory bound and the other part is compute bound. By treating the halo and core compute separately, whilst assuming the "flushing" cycles after each CG iteration to be compute bound, a more accurate model is created (the grey line in fig. 3.10). It should be noted, that more simplified models, e.g., the roofline model, will not be able to accurately predict this behaviour.

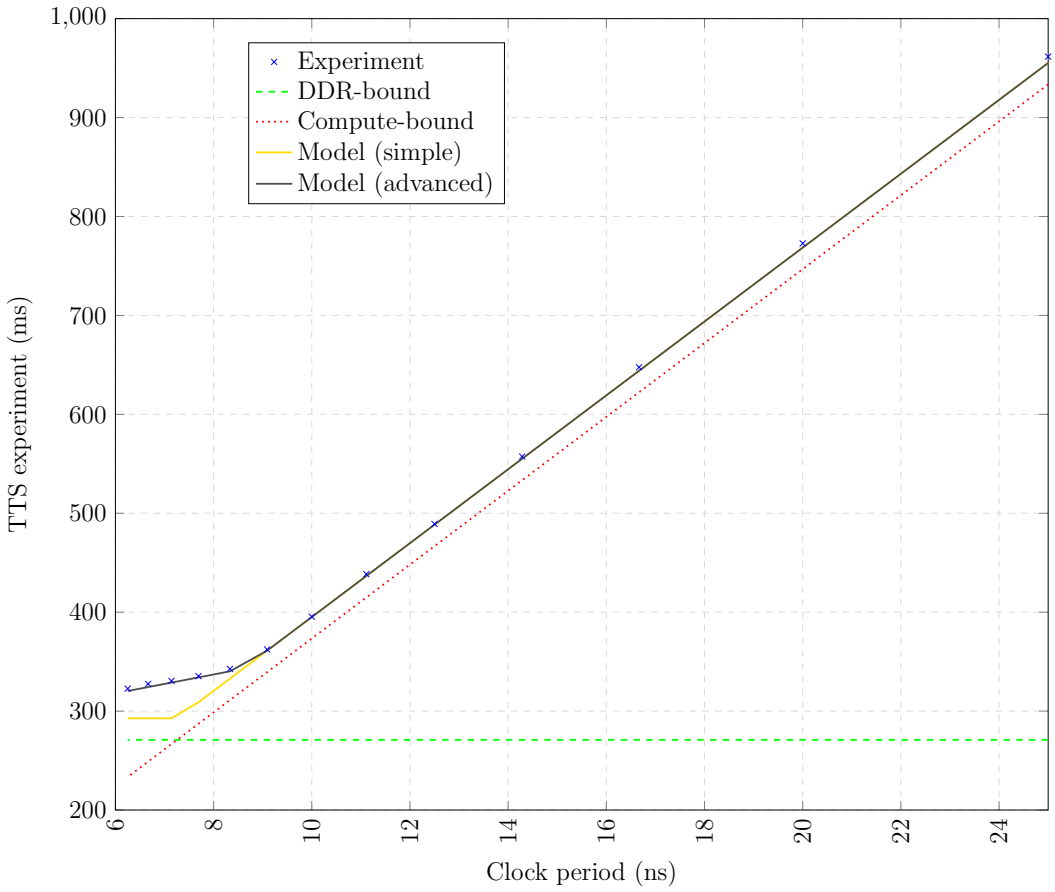

Figure 3.10: BQCD CG time-to-solution.

3.7 Summary

This chapter presented a methodology for the development of complex applications using FPGAs. It facilitates application-centric modelling to accurately predict system performance and to help with the identification of the best system architecture before the first line of hardware code is written. As such, complicated and time-consuming design iterations on the actual hardware design are omitted. The precise prediction of the final system properties is available early in the development process facilitating early business and overall system design decisions.

The proposal involves four phases, analysis, software modelling, forecasting of system properties and architectural development, which have to be co-developed in an iterative fashion to gain optimal results. Two case studies based on a CNN and BQCD were presented, demonstrating that following the proposed methodology yields state-of-the-art performance. In both cases the predicted speedup remained within 15% of real measurements and as intended the first implementation achieved specification targets. This high accuracy is achieved by using a highly predictable toolchain. Additionally, both applications have a predictable execution pattern which benefits the analysis used in the methodology. An application with a less predictable execution pattern is discussed in chapter 5.

The methodology has already seen initial adoption both within Maxeler itself as well as in academia and has received positive feedback. For example, the work presented in [71] uses parts of the methodology. Additionally, two master thesis projects used the complete methodology successfully ([129] and one yet unpublished). Maxeler has integrated the methodology proposed in this thesis into their tool documentation to teach it to internal and external developers [94].

Chapter 4

Extensions for Modern FPGAs and Performance Portability

4.1 Introduction

This chapter addresses the challenge of portability between different FPGA devices. In order to accomplish this, the methodology from the last chapter will be extended to support multiple current FPGA devices using the same code base. The challenge of portability is divided into two major aspects.

1. The heterogeneous memory of modern multi-die FPGAs; and
2. The missing methodology and tool support for portability between devices.

This means that tool support to efficiently target modern multi-die FPGAs with heterogeneous memory architecture and support for performance scalability are added. In both cases the proposed solution strategy is applied to relevant real-world applications.

The first challenge addressed in this chapter is the automatic mapping from logical into physical memory resources. This has become a lot more challenging for modern FPGA devices. The reason for this is twofold. Xilinx introduced additional memory types with their latest chips

[156], increasing the diversity of the available memory resources a FPGA programming tool has to manage. Additionally, Xilinx' biggest FPGAs consist of multiple silicon die on the same interposer, also known as SLRs [148]. One example of such a device is the VU9P which powers the Amazon EC2 F1 instances. In order to facilitate timing closure, it is beneficial when a hardware structure described by the designer can fully reside within a single SLR to avoid the usage of slow and scarce inter-SLR connections. Since the number of individual memory resources within each SLR is limited, efficient logical to physical memory mapping is important. The reason for this is that over allocation of a specific memory resource, e.g., allocating more of one of the physical memory resources than available within a single SLR, will automatically lead to a design in which the hardware structures span across multiple SLRs. Such a SLR crossing will hinder timing closure and often produce slower designs. As a result, it is important for an FPGA programming tool to allocate different memory resources so that the programmed structure resides as much as possible within a single SLR.

While it is possible to manually resolve this challenge within the methodology, this chapter proposes a greedy algorithm which automatically allocates hardware memory resources for user defined memories. As a result, this step of the methodology can be fully automated. This algorithm considers modern technology trends like SLRs and increasingly heterogeneous on-chip memory resources. The aim of the algorithm is to balance the allocation of different memory resources for individual sub-parts of the design and minimise the number of inter-SLR connections, which will facilitate timing closure and reduce routing congestion. The algorithm targets the latest Xilinx technology; however, the algorithm is generally applicable to systems with similar properties.

The second challenge addressed in this chapter is performance scalability between different FPGA based devices and different FPGA generations. Due to the high flexibility of FPGAs, it is possible to heavily customise and optimise designs. The resulting implementation is typically specific to the chosen target platform. This severely limits its portability to other FPGA-based target platforms.

In contrast CPU and GPU vendors spend significant effort to ensure forward compatibility

between devices. As such the same binary can be executed on new devices, often even offering a small speed-up. Additionally, binaries can be optimised with low designer effort by recompiling with a compiler, which is able to tune for the selected platform. As a result, the absence of performance scalability for FPGAs is severely impacting their adoption due to the need to spend significant designer effort in upgrading to new devices.

The optimisations used in the FPGA implementation include various types of computational patterns such as pipelining and parallelisation, numeric optimisations, e.g., custom fixed-point, and various types of memory access schemes and buffering. Other optimisations take platform specific information such as the total amount of resources or I/O bandwidth into account. Many of these optimisations capture aspects of the target platform and hence, target-specific optimisations become intertwined with application code. Combined with differences in the vendor tools and IP core libraries this severely limits the portability of the optimised application code to a different target device.

As a result, a solution to the problem of performance scalability requires not only tool support to provide abstractions for the device dependent components of the targeted platform, but also support by a development methodology and best practices to deal with changes in compute to communication ratios. It is crucial that it is possible to share a single code base between multiple platforms, to ensure that future functional changes and bug fixes can apply to all supported platforms without further problems. As such an isolation of platform dependent settings and configurations is desirable to ensure maintainability.

This chapter proposes best practices which can be integrated into the methodology introduced in the last chapter to ease portability between FPGA platforms. The proposed methodology steps are evaluated by porting an existing financial application to a wide range of platforms and measuring the achieved speed-up.

The contributions of this chapter are as follows:

- A simple greedy Balanced Memory Mapping (BMM) algorithm, which optimises timing closure by reducing the number of SLR crossings;

- A careful evaluation of three proposed memory mapping algorithms based on a set of use cases for small, medium and large workloads;
- And best practices to achieve performance scalability between FPGA platforms;
- As well as an evaluation of these best practices using a financial application.

The remainder of this chapter is organised as follows. In section 4.2 the mapping of logical to physical memories for multi-die FPGAs is discussed and the proposed algorithm is described in section 4.2.1. Section 4.2.2 provides an evaluation using a rich set of different workloads. The topic of performance scalability is discussed in section 4.3. The proposed best practices are explained in section 4.3.1. An evaluation of these best practices is provided in section 4.3.2 and a summary in section 4.4 concludes the chapter.

4.2 Memory Mapping Algorithm for Multi-Die FPGAs

Generally, a good memory mapping algorithm should:

1. use as few resources as possible; and
2. facilitate timing closure.

In order to address the first objective, the utilisation of the available memory resources needs to be considered. As shown in eq. 4.1, the utilisation of a given physical memory resource (BRAM or URAM) is the maximum utilisation of its valid aspect ratios. The utilisation of each aspect ratio is the ratio between the user defined logical memory size and the product of the physical memory unit size (both in #bits) and the required number of physical memories. To minimise hardware wastage the memory type with the best utilisation is selected.

$$\max \left(\frac{logical\ memory\ size}{unit\ size * \#units} \right)_{aspect\_ratio=1}^{N} \tag{4.1}$$

This thesis will call a group of hardware resources with high interconnectivity placed in close proximity on the FPGA fabric a *design unit*. Design units can be specified explicitly by the user, implicitly by using language structures or by creating a high-level floorplan model. As such a design unit could, for example, contain all resources of a single computational unit, e.g., a Maxeler Kernel. Alternatively, if a floorplan is used, a design unit could also contain all the resources which are mapped to a certain SLR. In order to reduce SLR crossings and as a result aid timing closure, it is of major concern to balance the allocation of different memory resources between different design units. Consider a case where a specific design unit requires more BRAMs than available in a single SLR. As a result, BRAMs from neighbouring SLRs have to be allocated, increasing SLR crossings and routing congestion as a result. As such redirecting some of the memories to URAMs is beneficial even though this introduces overheads in terms of allocated memory bits. In short, balanced allocation between BRAMs and URAMs is expected to improve SLR locality of individual design units. In the authors experience SLR crossings and the related routing congestion limit timing closure. As a result, the major goal for a timing optimised multi-die aware memory mapping algorithm is avoidance of unnecessary SLR crossings.

The problem addressed by this algorithm only occurs for memories using current devices. This is caused by the heterogenous memory architecture used by the modern FPGAs. The problem discussed above only occurs because there a BRAMs and URAMs and the same logical memory can be mapped to both resources. As such mapping decision influence if a design unit can reside within an SLR or not. This is not the case for, e.g., DSPs since one cannot decide to map a multiplier to one type of DSP or the other (since there only is one). If FPGA vendors should decide to introduce other heterogenous resources to the FPGA the algorithm can probably be adopted to those resources as well.

Another aspect in the context of this algorithm is general placement locality. E.g., if a DSP is connected to a memory one could consider which resources should be used to make the connection between them as short as possible. However, this aspect is beyond the scope of this work, since it needs significantly more knowledge of other mapping and placement decisions. It should probably be integrated with the placement algorithm in the FPGA vendor tools.

However, due to the need to route SLR crossings through the silicon interposer leading to long signal runtimes and their limited availability they usually have a larger impact on timing closure than routes within the FPGA die itself.

4.2.1 Algorithm Description

To address the above the following Balanced Memory Mapping (BMM) algorithm is proposed. BMM runs per design unit and its input is the list of logical memories. In a few cases a logical memory can only be mapped to a specific physical memory, due to specific hardware feature requirements, e.g., dual clock domain support. Such special logical memories are handled first to ensure mapping to the appropriate hardware resources.

Afterwards another pre-processing stage decides which of the remaining logical memories should be mapped to distributed RAM. Since the logic resources used to implement distributed RAM are less scarce, this mapping can be based on a simple heuristic and does not need to consider SLRs. This heuristic is based on the BRAM utilisation calculated using eq. 4.1. It is not needed to test URAM utilisation, since URAMs have a significantly larger capacity than BRAMs and because of the aspect ratio restrictions will never achieve a better utilisation for a given logical memory than BRAMs. The decision on mapping to distributed RAM uses a simple BRAM threshold. When the BRAM utilisation of a logical memory unit is lower than 1/8 it will be mapped to distributed RAM. Otherwise, the algorithm decides between URAMs and BRAMs on a later stage. The BRAM threshold value was deduced by considering the BRAM aspect ratio with the smallest possible depth of 512 and a width of 36 bits. The above datapath width was chosen based on the knowledge that 18 bits is typically not sufficient for small memories implemented in distributed RAM. If a 36 bits wide logical memory, matching BRAM widths, is considered, it will be mapped to distributed RAM when its depth is 64 or less. It should be noted that a depth of 64 matches a distributed RAM aspect ratio. Comparing utilisation provides a simple metric which accounts for both, width and depth. The reason for this low utilisation requirement is that the capacity of available distributed RAM is very small in comparison to the other memory resources. Additionally, logic resources are also used

to implement arithmetic and other user design features. As such saving logic resources is in general considered beneficial. However, in the context of static dataflow designs with many shallow FIFOs distributed RAM has to be used. For those FIFOs BRAM utilisation will be very low.

When beneficial, the BRAM threshold value can be customised for each specific design depending on the overall logic utilisation. For example, when a design requires an exceptional amount of logic resources it is possible to skew the balance towards BRAMs by decreasing the threshold. As a result, the best BRAM threshold value is application specific; however, the setting used here achieved good results for all applications and benchmarks used in the evaluation.

After the first memory mapping stage all memories not mapped to distributed RAM are grouped by design units. Each design units' memories are stored in a global list, sorted by decreasing URAM utilisation. This list is used by the memory allocation algorithm to decide if a particular logical memory should be implemented as BRAMs or URAMs.

Fig. 4.1 shows the BMM algorithm in more detail. The algorithm uses a score, which is initialised to zero and is used to decide in which order the memories are allocated. The score is based on BRAM cost and used to keep track of the balance between BRAMs and URAMs. As a result, when the algorithm selects to map a given logical memory to BRAMs the score is increased by the number of BRAMs allocated. When URAMs are selected, the score will be decreased as explained next. Since the score is based on BRAM cost a factor relating BRAM to URAM cost needs to be found. This is achieved by using the ratio between the available BRAM and URAM modules. Consequently, if a logical memory is mapped to URAMs the score will be decreased by the product of this ratio and the number of URAMs needed to implement the logical memory resource. This procedure is repeated for each design unit until the corresponding list of unmapped memories is emptied. As a result, the proposed algorithm will perform best in the case of a single design unit per SLR. This will avoid segmentation and the consequent suboptimal mapping resulting in slightly increased memory utilisation. However, when the toolflow does not support floor planning, the above is highly unlikely and resources have to be grouped into design units based on other properties. The evaluation

presented here assumes the less advantageous later option, where design units are implicitly inferred by language structures.

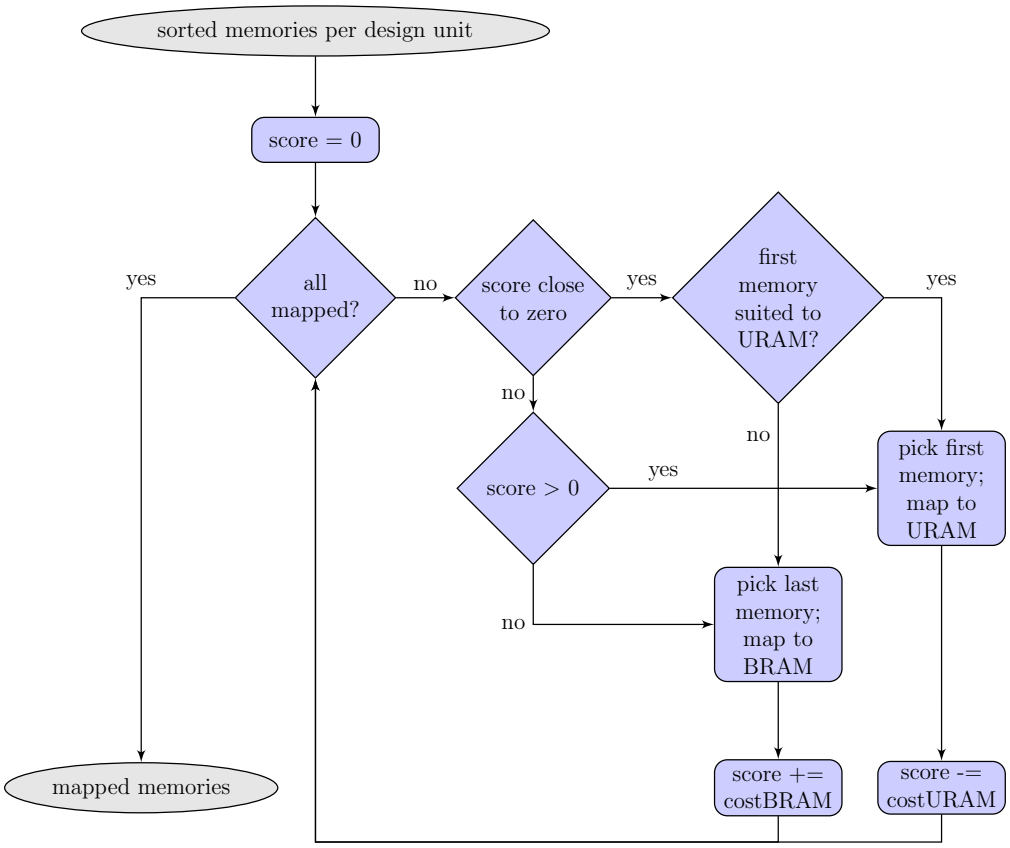

Figure 4.1: BMM algorithm with greedy, score-based proportional mapping.

Within the mapping loop of the BMM algorithm there are three possible cases which are handled separately. If the score is close to zero, the allocation between BRAMs and URAMs is considered as balanced. In this case the next unmapped logical memory is picked from the beginning of the list. Since the list is ordered by URAM utilisation, highest URAM suitability of the selected unmapped logical memory is ensured. The memory will be actually mapped to URAMs only when its utilisation is bigger than the URAM threshold. Otherwise, a new memory from the end of the list, hence suited to BRAMs, is selected and mapped to BRAMs. This URAM threshold together with the score determines when a logical memory is mapped to URAMs and can be modified by the designer. In the authors experience a URAM threshold of 0.6 provides good results. This means that logical memories with at least 60% URAM utilisation are directly mapped. The URAM threshold ensures that design units with only a single logical memory benefit from the best suited physical memory resource.

It should be mentioned that the URAM threshold in most cases has a limited impact, due to the algorithm capability to balance allocation between BRAMs and URAMs within the same design unit. However, the URAM threshold becomes important for certain rare edge cases. These consist of designs composed of multiple design units with only one or two memories of significant size. The default value of 0.6 tries to find a good balance with the target of decreasing the overall memory wastage. However, it might be necessary to adapt the URAM threshold manually in those rare edge cases.

In the case of a positive score more BRAMs than URAMs are used and again the logical memory from the beginning of the list is mapped to URAMs without considering the URAM threshold. Finally, in the case of a negative score a memory from the list end will be mapped to BRAMs.

Each time a memory is mapped to a specific resource the score is adjusted as described above. It will be increased in the case of mapping a logical memory to BRAMs or decreased in the case of mapping to URAMs. As a result, BRAMs and URAMs are allocated at comparable rates with respect to the overall availability.

By ordering the memories based on URAM utilisation it is ensured that always the memories most suited to URAMs or BRAMs are mapped first. This greedy strategy ensures that as many memories as possible are mapped to their best suited physical memory resource therefore saving area and addressing the first objective of the algorithm. To further improve memory resource utilisation, it is possible to combine the proposed algorithm with already existing memory allocation approaches, e.g., by tiling logical memories and making use of dual port memories as in [49]. However, it is necessary that all additional optimisation algorithms do not disturb accurate estimation of the number of physical memory resources that have to be allocated. Additionally, the second algorithm objective is fulfilled by allocating memories to URAMs and BRAMs at the same rate and hence minimising SLR crossings, which causes routing congestion reduction and aids timing closure.

4.2.2 Evaluation

In order to evaluate the proposed BMM algorithm place and route on multiple designs for the Xilinx VU9P FPGA using Vivado 2017.4 is performed. In all cases Maxeler's MaxCompiler is used and only the memory selection is influenced to enforce VHDL with the desired memory macros instantiated. This means that the design will be completely the same and only the used physical memory resources are changed. As a result, when the designs satisfy the same timing constraints, the achieved throughput remains the same as well as the logic and arithmetic resources utilisation. For all designs the set of implementation strategies able to achieve the best results in meeting a specific frequency is used. In cases where designs could not fit on the chip, the synthesis results on area utilisation are reported to emphasise the reason why designs failed to fit. For all experiments the URAM and BRAM thresholds suggested in the last section are used. First the memory mapping algorithms used to compare against are described. Later the used test cases are introduced, and finally experimental results are provided.

Algorithms

To evaluate the proposed BMM algorithm, three other mapping algorithms are used in the comparison below.

In the first case all mapping decisions are left to the Xilinx Vivado toolchain by using the *XPM_MEMORY* core and setting the *memory_style* to *auto*. Vivado, however, currently only uses distributed RAM and BRAMs [157]. Since URAMs are not yet supported by Vivado for the sake of fair comparison I introduce two additional algorithms. They both implement traditional memory mapping approaches and extend them with URAM support to form a realistic comparison base line.

The first additional algorithm, called Threshold Based Memory Mapping (TBM2), uses the same mapping to distributed RAM as BMM, but it simply uses the same fixed URAM threshold as in BMM to decide which memories should be mapped to URAMs or to BRAMs. As a result, if the URAM utilisation for a logical memory is above 0.6 it will be mapped to URAMs otherwise

BRAMs will be selected. This algorithm implements standard techniques which, for example, only consider the depth of a logical memory as in [121]. It aims only at efficient utilisation of each individual physical resource.

Additionally, I introduce the Wastage Reducing Memory Mapping (WRM2) algorithm. This third and final algorithm, as depicted in fig. 4.2 improves on the Threshold Based Memory Mapping (TBM2), by alleviating over usage of a single resource when other memory resources are still available. First, it uses the same mapping to distributed RAM as described before. Next, all remaining unmapped memories are ordered by URAM suitability as with BMM. In contrast to the BMM this ordered logical memory resources list is global and not on a per design unit basis.

The memories on that list are then allocated using the same URAM threshold of 0.6 as for the previous algorithms. In addition, Wastage Reducing Memory Mapping (WRM2) also keeps track on how much of the available resources are already allocated. When more than 80% of one physical memory resource is allocated, the remaining memories will be mapped to the other resource type. If both resources exceed 80% this limit will be increased in steps of 10%. For example, when 80% of BRAMs are used, the algorithm will only map to URAMs until the overall URAM usage also exceeds 80%. Due to ordering by suitability, when mapping to URAMs, the algorithm will only pick memories from the front of the list and consequently, when mapping to BRAMs it considers memories from the back. As a result, this algorithm tries to maximise physical memory utilisation and always aims at mapping logical memories to the best suited physical memory resource while not over-allocating resources. This approach allows the study of the area overhead introduced by the proposed BMM algorithm.

To the best of my knowledge no alternative multi-die aware memory mapping algorithm with the target to facilitate timing closure exists that can be used for direct comparison.

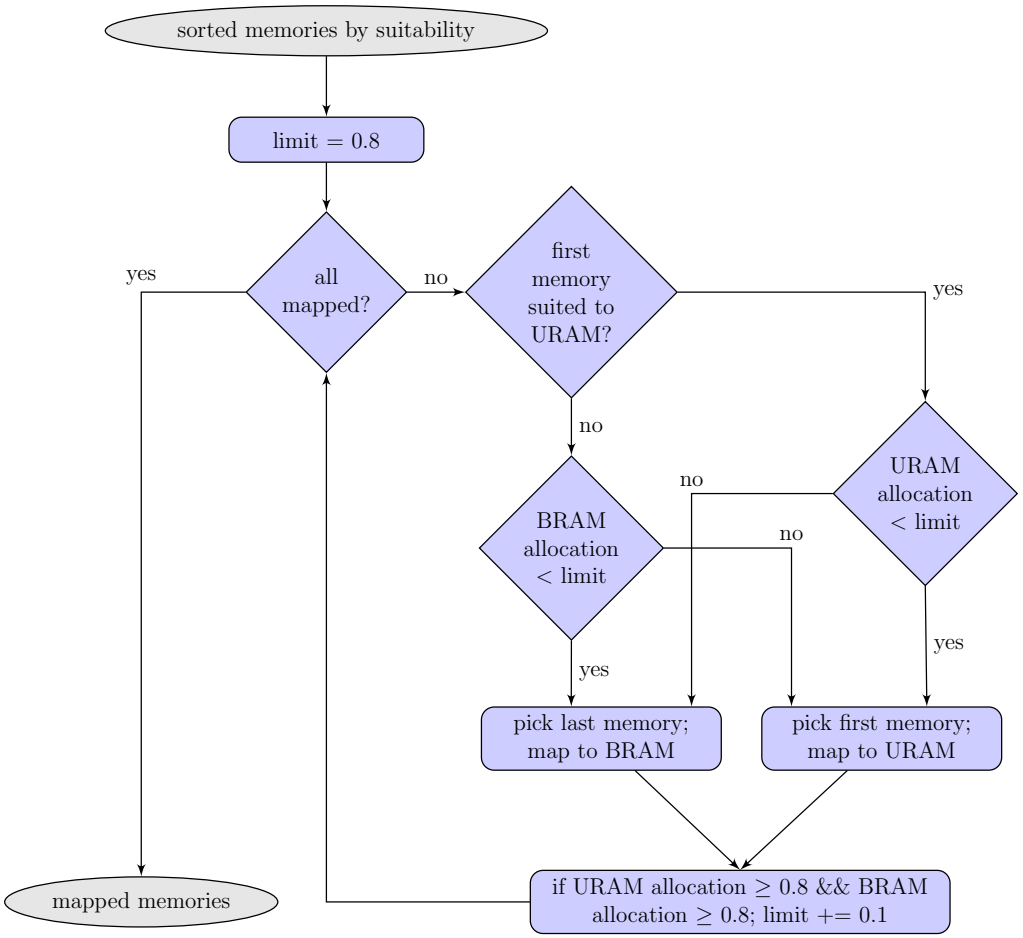

Figure 4.2: WRM2 algorithm with greedy, global area optimised mapping.

Test Cases

The four algorithms in this study are applied to three different sets of use cases. The first set consists of 16 small benchmarks (S1-S16), which only occupy a small area on the VU9P and can comfortably reside within a single SLR.

The second set of use cases consists of eight different medium sized designs (M1-M8). These examples occupy more than a single SLR, but still do not fully fill up the VU9P chip. M1-M5 are small synthetic examples, while M6-M8 are different versions of an FPGA based implementation of a real application, SPECFEM3D [72]. SPECFEM3D is a widely used HPC workload, which simulates different geophysical events, like wave propagation through different materials.

Finally, the last set of use cases consists of nine real applications (L1-L9). These applications represent real HPC workloads, which typically use most of the available on-chip resources, span multiple SLRs and make extensive use of the PCIe and DDR interfaces and hence decrease the overall number of available BRAMs and URAMs.

L1-L5 are machine learning applications. L1 is a fully connected network, while L2-L5 implement two convolutional neural networks with and without Winograd transform and all incorporate three copies of the same network implementation. L2 and L3 do not use Winograd and are based on the design presented in section 3.6.1. They differ only in forcing the design copies to specific SLRs using placement constraints or not. L4 and L5 use Winograd and follow L2 and L3 in their placement constraints.

L6 is a nanoscale material simulation application called Quantum ESPRESSO [41] another widely used HPC workload. L7 is an FPGA implementation of BQCD [104] a quantum chromodynamics application as presented in section 3.6.2. L8 implements the ocean engine of NEMO [90] a commonly used weather simulation tool. Lastly, L9 is a dense matrix-matrix-multiplication[1], a main building block of many HPC applications.

The smaller synthetic test cases are included to study how the proposed algorithm behaves for smaller applications which do not make use of all on-chip resources even though this was not its

[1] https://github.com/nilsv/Dense-Matrix-Multiplication

typical optimisation target. The use cases M6-M8 as well as L1-L8 are real world applications which are deployed in HPC environments and as a result represent the primary target for the design of the presented algorithm. Especially M6-M8 as well as L6-L8 represent a significant portion of the workloads running currently on HPC systems.

Results

Fig. 4.3 and fig. 4.4 show the BRAM and URAM usage for the small use cases. It can be observed that all algorithms apart from BMM use the exact same number of BRAMs and URAMs. Since the BMM algorithm tries to find a balance between allocating BRAMs and URAMs, it maps some of the logical memories to URAMs. As a result, the overall memory usage in allocated bits is increased by 38%. As this may seem quite high, the total number of allocated memory blocks and therefore the expected power consumption stay close. All four algorithms allocate the same amount of distributed RAM.

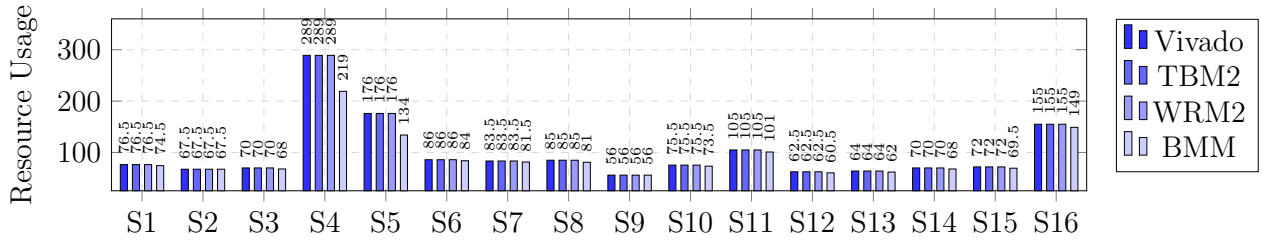

Figure 4.3: BRAM usage for the four different algorithms on the test set of small applications.

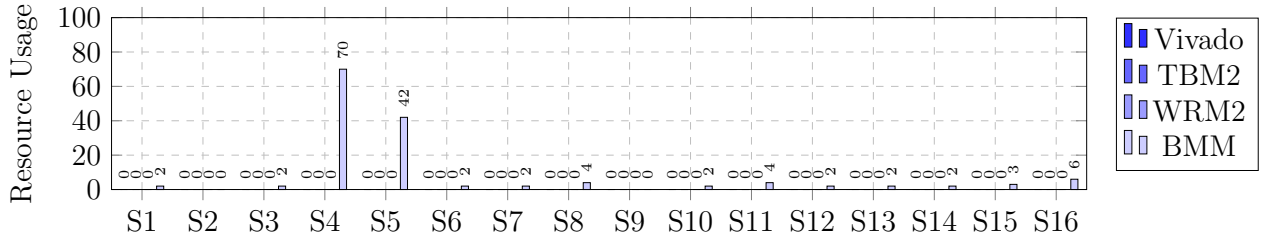

Figure 4.4: URAM usage for the four different algorithms on the test set of small applications.

The memory resource usage for the set of medium sized use cases is shown in fig. 4.5 and 4.6. The standard Vivado algorithm does not map any logical memories to URAMs. As a

result, M8 does not fit into the chip, due to a significant overallocation of distributed RAM. The simple TBM2 algorithm and WRM2 both reached the same mapping decisions, since no single resource exceeded 80%. The proposed BMM algorithm again uses URAMs and BRAMs more balanced than the other algorithms. As a result, the overall number of allocated bits is increased by 85% compared to the WRM2 algorithm. However, it should be noted that using large amounts of a particular resource usually makes timing closure harder. The allocation of distributed RAM is similar between all algorithms. Only for the use case of M8 the standard Vivado algorithm allocates significantly more distributed RAM.

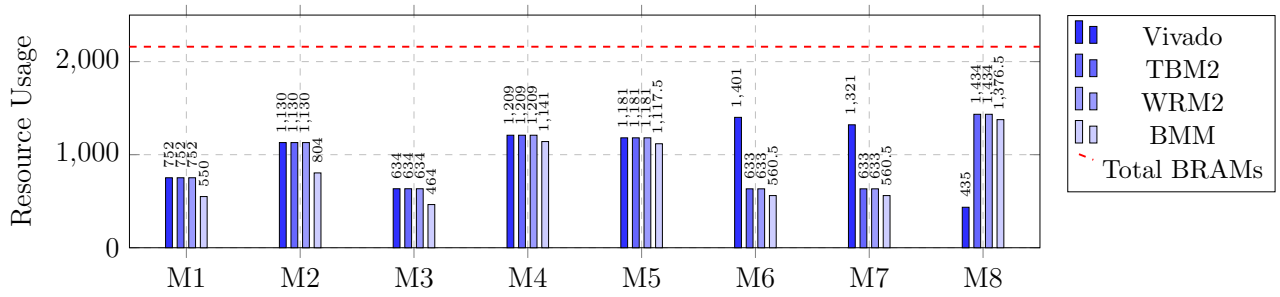

Figure 4.5: BRAM usage for the four different algorithms on the test set of medium applications.

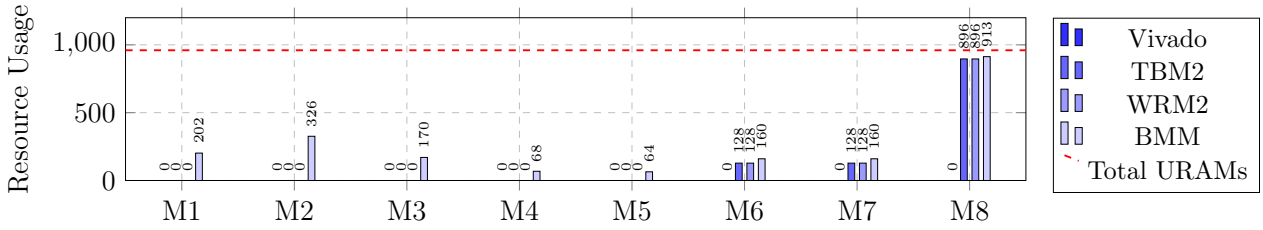

Figure 4.6: URAM usage for the four different algorithms on the test set of medium applications.

Finally, the resource usage for the large test cases is shown in fig. 4.7 and 4.8. Vivado and the simple TBM2 algorithm both fail to place and route test cases L2, L3, L6, L8 and L9. Additionally, the high resource usage for the standard Vivado algorithm prevents successful place and route of L7.

L2 and L3 designs instantiate three copies of the same large design unit. In the case of L2, Vivado placement constraints are used to force each design unit in a single SLR, while L3 has no placement constraints. The WRM2 algorithm has difficulties with designs like the above.

It maps one of the three design units mainly to BRAMs and the other two mainly to URAMs. This causes all three design units to span across multiple SLRs. As a result, L2 fails to meet its placement constraints and L3 fails due to high routing congestion, even though the total count of allocated BRAMs and URAMs is similar between WRM2 and the proposed BMM algorithm. Only the latter finds a mapping, facilitating successful place and route completion.

L9 provides a similar test case as above consisting of three individual design units. WRM2 again allocates resources unevenly between the three design units, mapping one entirely to URAMs and the other two mostly to BRAMs. However, the mapped workload is less complex making successful place and route possible, even though the number of SLR crossings is increased by a factor of 2.6.

The mapping behaviour for designs with multiple big design units was the main motivation behind the proposed BMM algorithm. Since both memory resources are allocated at the same rate, it is guaranteed that no single resource is heavily overused. Only when the overall memory usage of a design unit is larger than a single SLR capacity, multiple SLRs will be used. This ensures that the toolchain does not limit place and route of valid, well designed architectures.

In the case of L8 the WRM2 algorithm also fails to facilitate successful place and route completion. Here WRM2 makes use of all available URAMs, which leads to a violation of a placement constraint introduced by the Xilinx DDR IP core causing place and route to fail. This could be potentially avoided by predicting the BRAM usage more accurately, since only 85% of BRAMs are used. In general, it can be noticed that WRM2 requires precise estimations in order to avoid overallocation of a single resource.

To summarise, only the proposed BMM algorithm manages to successfully generate place and route results for all large designs, even though the memory usage in number of bits is on average 13% higher. However, in cases where the memory usage of both physical resources approaches levels above 80% the BMM and the WRM2 algorithm both allocate a similar amount of BRAMs and URAMs.

Fig. 4.9 shows the average number of SLR crossings across multiple implementation strategies

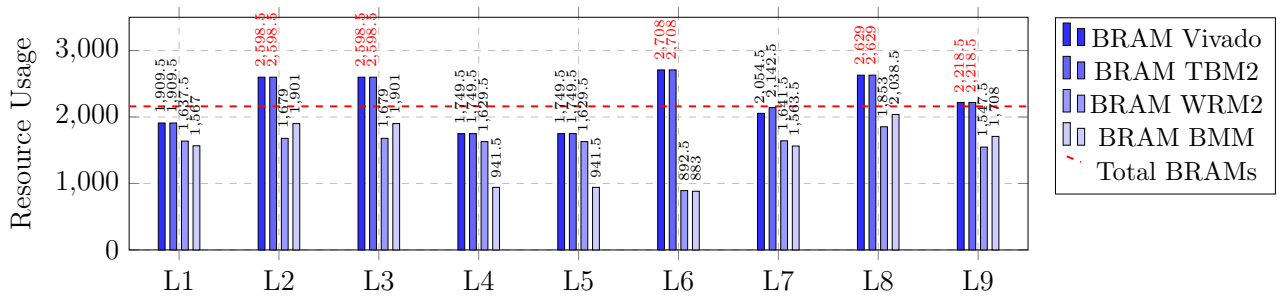

Figure 4.7: BRAM usage for the four different algorithms on the test set of large applications.

for all medium and large use cases and all four mapping algorithms. The main target of the mapping algorithm is to improve the timing closure behaviour of the design. The problem is that this is very hard to accurately measure especially on a bigger set of large applications. It would be desirable to consider the frequency for which each individual design for each frequency would meet. There are two major problems with this. First, the variance between different place and route runs and implementation strategies[2] is large. Second, it would be required to dial the frequency for each application manually for each algorithm until the design meets timing. While Maxeler was able to provide me with a cluster of servers to record the experimental values presented in this chapter performing that many place and route runs would have used more cluster time than made available to me.

As such, I report the SLR crossings since the strongly correlate with the timing closure behaviour. The reason for this can be found in their limited nature, their comparatively long routing delay and their fixed position leading to further routing constraints. Additionally, as discussed above, the algorithm attempts to reduce SLR crossings as a way to improve timing closure and this comparison shows how well the algorithm performs for this intermediate goal.

In order to compare the average number of SLR crossings between algorithms, only those cases where place and route finished successfully are considered since this is necessary to obtain a figure on SLR crossings. This means that cases in which those algorithms performed especially poorly are not taken into account, creating a bias against the proposed BMM algorithm. In general, there is no straight forward way to include the test cases currently not taken into

[2]The optimisation goals for the place and route algorithm.

account. One could for example assume that in the not routable cases all SLR crossings are fully used, which is not realistic and would create unnecessary bias in favour of the proposed BMM. However, even the current selection shows the advantages of the proposed algorithm.

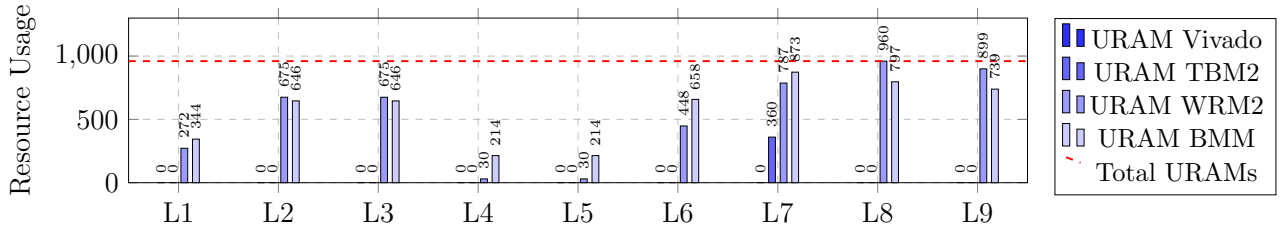

Figure 4.8: URAM usage for the four different algorithms on the test set of large applications.

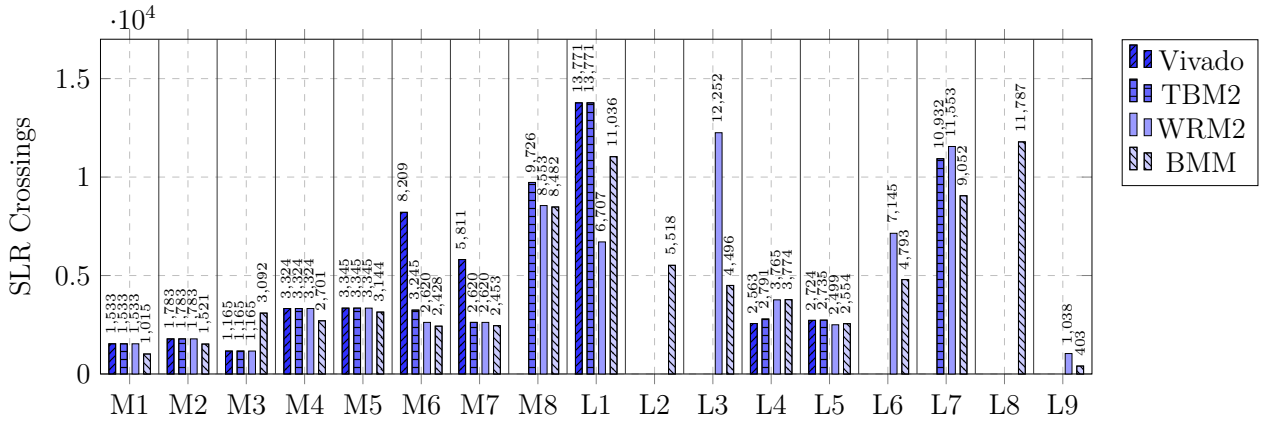

Figure 4.9: Number of SLR crossings for the different mapping algorithms on the medium and large test set. Missing bars corresponds to failed builds.

For the medium use cases the proposed BMM algorithm achieves a reduction in the average number of SLR crossings by 46%, 11% and 6% compared to the standard Vivado, the TBM2 and the WRM2 algorithms respectively. For the three large cases successfully routed by Vivado and BMM on average the same number of SLR crossings was created. However, in the six other test cases Vivado failed to finish place and route. In comparison to TBM2 and WRM2 the usage of the BMM algorithm leads to an average reduction in number of SLR crossings by 7% and 52% respectively.

L3 provides a good example on how the proposed mapping algorithm can reduce the number of SLR crossings. L3 consists of three copies of the same design. This also means that each of those copies has the same types of memories and additional most memories in the design have

the same size. The WRM2 algorithm now maps all of these memories to the most suitable resource, BRAMs, until no more BRAMs are available. The remaining one will be allocated to URAMs. In this case this means that one of the three design copies will mostly use BRAMs while the other two will mostly use URAMs. The design copy mostly using BRAMs will need to span across all three SLRs since no single SLR has enough available BRAMs to accommodate it. The other two copies will also need to span across two SLRs since not enough URAMs are present within one SLR (due to the low URAM suitability). This results in the difference of nearly 3x in terms of SLR crossings between WRM2 and BMM. While this presents a worst-case scenario to WRM2s it was not deliberately constructed as such but is the result of the implementation of VGG-16 as discussed in section 3.6.1.

Lastly, fig. 4.10 shows the Total Negative Slack (TNS) for test cases in which at least one algorithm produced a TNS value while failing timing closure. Additionally, at least two algorithms produce TNS values to facilitate comparison. In the case where a design is very congested TNS values are often not generated by Vivado. As a result, it is only possible to draw meaningful conclusions from the five test cases shown in the figure. The depicted TNS is the average over multiple implementation strategies.

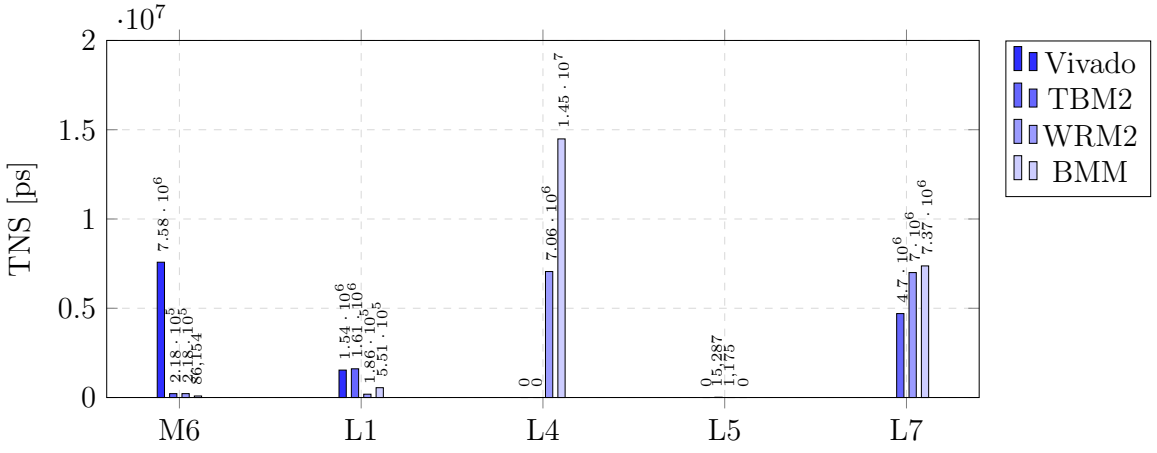

Figure 4.10: Average TNS for test cases, where more than one algorithm produced a non-zero TNS. Missing bars corresponds to failed builds.

By comparing fig. 4.9 and fig. 4.10 one can observe that there is a strong correlation (not a causation) between the number of SLR crossings and the average TNS. For example, in the case of M6 the standard Vivado algorithm creates by far the most SLR crossings, resulting in the

worst average TNS. Accordingly, BMM creates the least amount of SLR crossings and achieves the lowest TNS.

The same correlation holds true for L1 and L4. However, in the case of L7 BMM creates the highest average TNS even though the number of SLRs crossings is the lowest. The reason for this is that for one of the implementation strategies the TNS is 50,114,977 ps. This single outlier impacts average TNS significantly. If this implementation strategy is excluded the average drops to 1,265,720 ps, which would be the lowest average TNS for this use case. It was sadly not possible to identify the precise frequencies each application can achieve for each of the test cases due to the limitations described above.

As a result, it can be concluded that comparing the number of SLR crossings is a good metric to estimate the impact of memory mapping algorithms on TNS. The advantage of this is that it provides an easier to compare metric, generating more data points in a smaller value range, whereas TNS can be prone to outliers. Similarly, it is hard to take into account if for a few implementation strategies no TNS value is generated, since the design cannot be routed.

More importantly, for the test cases considered in this comparison only the proposed BMM managed to successfully place and route all designs. In contrast the standard Vivado algorithm failed on M8 and all large applications apart from L1, L4 and L5. This means it only managed to be successful on three out of the nine large applications. TBM2 only managed to additional place and route M8 and L7, meaning that it still fails for more than half of all large applications. The only algorithm getting close is WRM2 which only fails on L2, L3 and L8, which improves the failure rate to one third. All in all, the proposed BMM algorithm provides a significant and noticeable improvement over all other algorithms tested.

4.3 Performance Portable FPGA Designs

Generally speaking, when moving between different FPGA platforms there are typically three major challenges:

1. Different vendor tool chains;
2. Different vendor and platform-specific programming constructs as well as hardware IP cores; and
3. Different FPGA device architectures, characteristics and platform capabilities.

If one switches the FPGA chip vendor usually a different toolchain has to be used. While all commonly used toolchains support a set of widely used languages like VHDL and Verilog the required design description might still differ profoundly. For example, the way in which timing and placement constraints are applied is usually not portable between toolchains. Similarly, the build process might differ significantly.

Even if the vendor is not changed, porting to a newer FPGA often requires significant changes to the design. For example, it is usually necessary to use newer IP cores.

Finally, new device generations often also introduce new hardware features or changes to the chip architecture. It might not be possible to simply instantiate an existing design on the new hardware platform without either decreasing the area usage of the design or exploiting the new architectural features. This can also include changes to I/O like the usage of different memory or interconnect technology, which go hand in hand with changes to the required port widths and data alignments.

In contrast on conventional CPU and GPU technologies programs can be easily moved between devices using the same ISA. Usually chip vendors also guarantee backwards compatibility, where programs can be executed on all future devices and only the usage of new instructions requires changes to the binary.

Especially the first two challenges are solved through more advanced tool support which abstracts these details away from the user. There are multiple FPGA programming tools available, including MaxCompiler, which address these issues.

If the area and memory capacity of the newly targeted FPGA increases or at least stays the same and the tools resolve the first two portability challenges it is possible to port an application

to the new device without significant changes. In this case it is only necessary to deal with the changes to the I/O interfaces and introduce the required aspect changes if hardware port widths change (BP1). However, this only provides functional correctness and not good performance. It might even worsen performance compared to older devices, if I/O cannot be used efficiently or high clock frequencies are no longer achievable due to device architectural changes, e.g., new multi-die architectures.

Moving to smaller or otherwise more restricted devices requires changes to the application to reduce the area usage. As a result, techniques used for performance scalability have to be applied as described in the next section.

4.3.1 Performance Scalability

A performance model as described in section 3.4 is a necessary part of any process delivering performance scalability (BP2). It can be used to quickly analyse the performance of a given architecture for different devices and also highlight which changes might be necessary to circumvent potential bottlenecks.

Most FPGA applications contain a set of standard optimisations, including loop unrolling or the instantiation of multiple copies of the same computation. In these cases, it is easily possible to parameterise the degree of parallelism and therefore steer hardware usage (BP3). Similarly, data path widths can be adjusted based on the underlying hardware substrate for efficient use of, e.g., hardware multipliers (BP4). By using an accurate performance model, it is possible to predict the resulting area usage, memory and I/O requirements of a design point. This enables design space exploration of architectural options without waiting for time consuming place and route jobs.

If the computational abilities increase faster than the available I/O bandwidths, it is necessary to make further changes to the design to take advantage of this. In most designs on-chip memories are used to buffer bigger chunks of data for fast access. An example for this is tiling (BP5), which is, e.g., often used in the context of linear algebra. In these cases, a bigger tile size

can reduce the pressure on the I/O system. If the on-chip memory capacity increased similarly to the compute capability the additional on-chip memory capacity can be used to resolve I/O bottlenecks (BP6). For example, considering the MAX4C DFE and the Alveo U200 cards, the memory bandwidth of the U200 card only increased by roughly 10% while the on-chip memory capacity increased nearly seven-fold. As such it is important that the application designer makes the size of these on-chip buffers configurable.

It should be noted that there are also cases where such a simple solution is not applicable, e.g., if data are truly streamed and not buffered at all. In those cases, changes to the implementation are required. In the best-case additional compression and decompression blocks are sufficient, which can be added without requiring significant changes to the remaining code base. As a result, it is possible to still maintain code sharing between different platforms. In the worst case a fundamental change to the architecture is required which will limit code sharing options between platforms.

Similarly, changes to the overall memory capacity including on card memory have to be considered. If the memory capacity of the new target device is big enough to hold the same amount of data as the current device no further changes are necessary. However, it might be possible to transfer a bigger working data set to the FPGA if the memory capacity increases, potentially reducing communication overheads. If the capacity decreases and falls below the required capacity a domain decomposition has to be introduced (BP7). In that case the dataset is split into smaller parts which are processed individually.

An issue often encountered with big, modern Xilinx FPGAs is the internal split into separate SLRs. If SLRs are ignored, timing closure and routing congestion can become challenging. As a result, it is important to limit the number of connections between different SLRs to mitigate this problem.

In MaxCompiler, the easiest way to mitigate this is by making sure that every Kernel can fully reside within a single SLR and only the connections between different Kernels or Kernels and I/O cross SLRs. There are two straight-forward ways to achieve this. First, it is possible to split a design into multiple Kernels, where each individual Kernel is small enough to fit within

one SLR (BP8.1). A second option is to create a copy of the design for each SLR (BP8.2).

In the first case it is often challenging to ensure that all individual Kernels can be scalable in terms of size at a similar rate to fill up all individual SLRs. Especially if the size or count of SLRs might change between current or future targeted platforms this might introduce a significant challenge to performance scalability.

However, in most cases it is possible to simply create copies of the same design for each SLR. This is especially recommended if it is possible to split the overall computation into smaller either completely or partial independent units, which require no or only limited intercommunication and can all be completed using the same hardware functions. This practice provides an easy way to achieve performance scalability between different FPGAs.

Usually, the size of different SLRs is similar, however, in some cases certain SLRs might have less usable space than others, e.g., due to the need to instantiate certain IP cores close to a physical pin out. In these cases, the different copies of the design can be instantiated with differing degrees of parallelism. Alternatively, the first suggested approach of splitting the design into multiple separate units might be more promising. Since the use of chiplets and multiple die becomes more prevalent in the semiconductor industry, one can expect that future platforms will also include multiple die.

All of the best practices described in this section fall in the category of design parameterisation. They describe which aspects are most important to parameterise and together with the Manager API developed by Maxeler and my usage of it as shown in section 2.3.5 they propose a way towards performance scalability for FPGA designs. This means that if an existing design is not already parameterised, parameterisation has to be added in order to achieve scalability. Otherwise, it might be possible to only achieve portability and not scalability as long as the design fits into the new device.

The more aspects of the design are configurable the higher the achievable scalability. For example, only if the size of memories and, as a result, the resulting memory bandwidth requirements are scalable one can avoid being constantly memory bound. The same is true for other

components like the usage of DSPs.

4.3.2 Evaluation

The best practices and methods proposed in section 4.3.1 and the tool support presented in section 2.3.5 are now applied to the Asian option pricing application introduced in section 2.6.2 which was originally developed for a MAX4C DFE to evaluate their usefulness. This application was selected due to its relevance as a typical HPC workload and my ability to use the existing source code. This allowed a focus on applying the best practices without the need to develop a new application from scratch.

Application porting

The Asian option pricing application is now ported from the original MAX4C DFE to the four additional target platforms, following three individual steps:

1. Adapt the application to the new Manager API;
2. Apply the discussed best practices; and
3. Add platform Managers for all new platforms.

In order to add platform portability to the application the existing Manager code has to be migrated to the new Manager API first. A single platform generic function can be created which instantiates the design by creating Kernels and wiring them up with each other and PCIe. Additionally, a platform dependent function is created which configures the Quartus toolchain for the Intel Stratix V based MAX4C DFE. This function contains up to ten lines of additional code, which can be shared between all platforms using the same generation of FPGA devices. The resulting platform Manager for the MAX4C DFE is shown in listing 4.1.

Listing 4.1: The new platform specific Manager code for the MAX4C DFE.

```
1 public class AsianMax4Manager extends MAX4CManager implements
    AsianManager {
2     public AsianMax4Manager(AsianEngineParameters params) {
3         super(params);
4         setDefaultStreamClockFrequency(params.getFrequency());
5         setupMax4(params, getBuildConfig());
6         createDesign(params);
7     }
8 }
```

It is now possible to achieve application portability for a new platform with minimal effort. For example, the Manager shown in listing 4.2 adds support for the Xilinx Alveo U200 card. This only requires the addition of one new function to configure the Xilinx Vivado toolchain. In this case no further optimisations for the specific platform are performed. As a result, the MAX4C and U200 design will both achieve the same performance at the same design frequency. It should be noted that apart from these few lines of Manager code and the toolchain configuration the remaining code base is completely shared, and no further changes are required.

Listing 4.2: A Manager for the Alveo U200 which achieves performance portability.

```
1 public class AsianU200Manager extends XilinxAlveoU200Manager
    implements AsianManager {
2     public AsianU200Manager(AsianEngineParameters params) {
3         super(params);
4         setDefaultStreamClockFrequency(params.getFrequency());
5         setupUltrascale(params, getBuildConfig());
6         createDesign(params);
7     }
8 }
```

It is now possible to build the design for both platforms. The selection between both can be handled as a command line argument for the build script. In both cases the same number of

loop iterations is unrolled which means the parallelism factor is set to $k = 13$. The results can be seen in tab. 4.1. At the same frequency the design on the MAX4C and the Alveo U200 achieve the same performance, fulfilling the target of performance portability. Furthermore, it is possible to increase the frequency from 200 MHz to 350 MHz which results in a 1.74x speedup without any further changes.

Table 4.1: Comparison of the designs for the different target platforms. Speedup is provided relative to the MAX4C implementation. The resource usage percentage is based on the available resources of the given FPGA.

Platform	Design Copies	Parallelism k	Frequency	Time (s)	Speedup	Normalised Speedup	Logic[3](%)	DSPs (%)	BRAMs[4](%)	URAMs (%)
MAX4C	1	13	200	15.13	1x	1x	258,209 (98.40%)	1,436 (73.15%)	1,205 (46.94%)	-
U200	1	13	200	15.13	1x	0.57x	214,569 (18.15%)	2,628 (38.42%)	933 (21.60%)	59 (6.15%)
U200	1	13	350	8.69	1.74x	0.57x	214,611 (18.15%)	2,628 (38.42%)	933 (21.60%)	59 (6.15%)
U200	3	10	350	3.79	3.99x	1.31x	535,374 (45.28%)	6,372 (93.16%)	2,157 (49.93%)	189 (19.69%)
U250	4	14	350	2.04	7.42x	1.35x	874,035 (50.58%)	11,184 (91.02%)	3,471 (64.56%)	323 (25.23%)
MAX5C	3	10	350	3.91	3.87x	1.27x	536,143 (45.35%)	6,372 (93.16%)	2,164 (50.09%)	189 (19.69%)
F1	3	5	250	10.48	1.44x	0.78x	544,320 (60.48%)	3,855 (66.10%)	1,967 (58.54%)	356 (44.5%)
F1	1	18	250	8.62	1.75x	0.95x	423,501 (47.06%)	3,471 (59.52%)	1,506 (44.82%)	170 (21.25%)

[3]In the case of the MAX4C (Intel Stratix V) I count ALM usage, while in all other cases (Xilinx Ultrascale+) I count LUT usage.
[4]In the case of the MAX4C (Intel Stratix V) I count M20Ks usage, while only in all other cases (Xilinx Ultrascale+) I count actual BRAM usage. There is also no URAM equivalent for the MAX4C

As a next step the previously discussed best practices are applied to the code base to achieve a further speedup. The first step is to adjust the performance model (Part 3) of the application for the new platform (BP2). For the Asian option pricing application, no DDR memory access is required. As such it is possible to fully focus on the PCIe bandwidth and area usage.

The area usage can be predicted by counting the number of operations for the design and running micro benchmarks to get figures for the area usage for each individual operation. Since, the old performance model already included operation counts, only the area usage for each operation on the new platforms using micro benchmarks has to be measured. In this case all additional platforms are based on the Xilinx Ultrascale+ technology so the area usage will be constant across all of them. As a result of this the DSP and LUT usage can be calculated as shown in eq. 4.2 and eq. 4.3 respectively.

$$dsps = 444 + k \times 168 \tag{4.2}$$

$$luts = 63{,}482 + k \times 9{,}181 \tag{4.3}$$

In terms of I/O bandwidth the amount of data that has to be transmitted for each scenario and each option has to be analysed. Per option 67B and per scenario 654KB have to be sent from the host to the FPGA. Additionally, 8B have to be read from the accelerator per option. Since 5,000 scenarios on 10,000 options are run in total 3.12GB of data have to be transmitted. This can be achieved in roughly one second using PCIe Gen2 x8 or in a quarter of that time using PCIe Gen3 x16 at slightly higher hardware costs.

The number of cycles that are required to finish the calculation can be computed as shown in eq. 4.4. Using this equation, it is possible to derive the compute time for a given frequency.

$$n_{cycles} = n_{scenarios} \times \left\lceil \frac{n_{average\_points}}{k} \right\rceil \times n_{options} \tag{4.4}$$

Using this information, it is possible to further optimise the design for the additional platforms. As a first step one has to ensure that all data are properly aligned, if the width of the PCIe bus changes. To accomplish this the PCIe port width is passed to the Kernels communicating with the CPU and an automatic aspect change is added (BP1). The CPU code to allocate data has to be changed accordingly.

The second step is to make efficient use of multiple SLRs. As discussed in section 4.3.1 there are two fundamental strategies to deal with this problem. One can either try to split the design into equally sized parts (BP8.1) or replicate the design multiple times (BP8.2).

In the case of the Asian option pricing application there are good arguments for the application of both solution strategies. For many components of the design the resource usage stays constant independent of the unrolling factor. As a result, keeping a single design and only unrolling the loop more is more resource efficient. However, a high unrolling factor results in low hardware utilisation, if only a limited number of average points is used. The reason for this is that each loop iteration deals with one average point. As an example, if k is set to 29 and one calculates 30 average points one would need to make two passes where only one of the 29 hardware pipelines is used in the second pass. Also unrolling is only applied in two out of the five Kernels. As a result, it will not be possible to distribute the design proportionally across the bigger FPGAs,

e.g., the FPGA used in the Xilinx Alveo U250 card which has four SLRs.

Creating multiple copies of the design can be considered as an overhead. Since DSPs are the limiting resource in this design, each additional design instance uses hardware resources which could otherwise be used to unroll 2.5 additional loop iterations. It was decided to create multiple design instances (BP8.2), since the overhead is small in comparison to the expected benefits in terms of achievable frequency and ease of development.

To implement this optimisation the Manager code is further modified. The `createDesign()` function has to be called in a loop to create multiple instances of the design. Additionally, the naming of Kernels and interfaces has to be changed accordingly, to ensure that all names are unique. As a result, it is now possible to configure the number of copies via a simple command line argument for the build environment. The result of these changes can be seen in listing 4.3.

Listing 4.3: The Manager for the Alveo U200 achieving performance scalability.

```
1  public class AsianU200Manager extends XilinxAlveoU200Manager
       implements AsianManager {
2      public AsianU200Manager(AsianEngineParameters params) {
3      super(params);
4          setDefaultStreamClockFrequency(params.getFrequency());
5          setupUltrascale(params, getBuildConfig());
6          for (int i = 0; i < params.getDesignCount(); i++) {
7              createDesign(params, i);
8          }
9          addMaxFileConstants(params, 16);
10     }
11 }
```

In this case it is also necessary to modify the CPU code to split the overall workload into multiple parts, which are equally distributed onto the different copies of the design. Using MaxCompiler it is possible to add C/C++ preprocessor definitions to an automatically generated header file

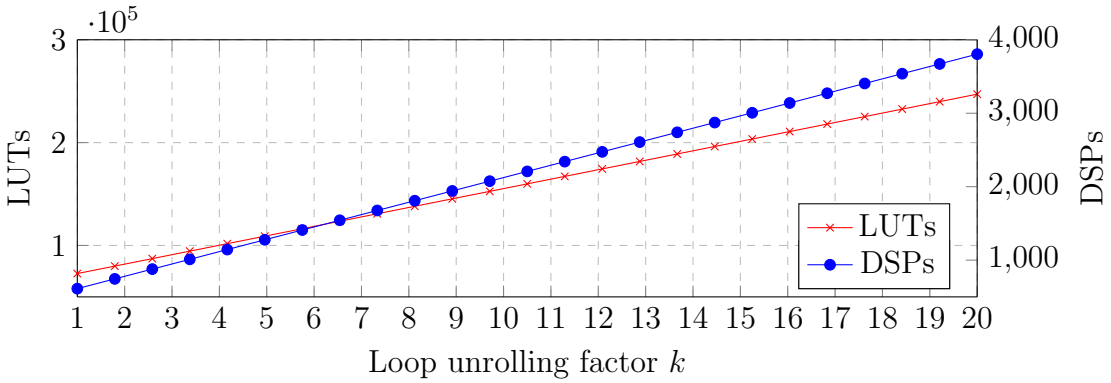

Figure 4.11: LUT and DSP usage predicted by the performance model for one design instance of the Asian option application implemented on Xilinx Ultrascale+ technology.

which is needed to integrate the bitstream into the host application. This enables the automatic synchronisation of the design count between bitstream and CPU code, by passing the number of design instances to the CPU code which can then automatically distribute the workload accordingly. In listing 4.3 the `addMaxFileConstants()` function fulfils this role. As a result, the same CPU code base can be used for all platforms.

At this point it is possible to build performance scalable designs for the current platforms targeted, MAX4C and Alveo U200. This can be expanded to the other targets, Alveo U250, MAX5C and Amazon F1, by adding the Managers accordingly. No further changes to the code base are required.

To rapidly perform design space exploration one can use the previously developed performance model (BP2). Fig. 4.11 shows the area usage in terms of LUTs and DSPs for a single design instance dependent on the loop unrolling factor k (BP3). This can be seen as the area usage for a single SLR and since SLRs have approximately the same size it can be used to determine the most reasonable values for k. It is now only necessary to try these values and figure out the highest achievable frequency by running place and route. Even though it is still required to run multiple place and route runs per targeted platform, overall, the possible number of required runs is decreased significantly by using the performance model. For the valid design points of this application and the targeted platforms the I/O bandwidth is no bottleneck, so it is not necessary to apply further optimisations to change the compute to communication ratio.

The achieved performance and area utilisation for all designs are shown in tab. 4.1. Max-

Compiler 2019.1, Vivado 2018.3 and Quartus 13.1 are used for building and 5,000 scenarios on 10,000 options are executed as a benchmark. In most cases it was possible to use a significant part of the FPGA resources while still achieving timing closure at high frequencies. In the case of the MAX4C DFE more than 98% of the logic resources are used and timing is met at 200 MHz. For the MAX5C DFE as well as the Alveo U200 card the DSP usage is the highest at more than 93% and the design meets timing at 350 MHz achieving a speedup of nearly 4x compared to the MAX4C design. Porting from the MAX5C DFE to Alveo U200 does not require any architecture changes and delivers almost the same performance, since the platform architecture is similar. In the case of the Alveo U250 more than 91% of the DSPs are used at a frequency of 350 MHz resulting in a speedup of 7.4x compared to the MAX4C baseline.

In the case of the Amazon EC2 F1 instance the resource usage and performance improvement are smaller. This is due to the presence of the AWS shell which takes up some of the chip resources. Hence, a lower number of resources is available to the user application as illustrated in fig. 4.12. In the two SLRs also containing the AWS shell, the available fabric is densely used. Especially DSPs are used to nearly one hundred percent. As a result, it is not possible to achieve a higher frequency with the proposed architecture or increase the parallelism further, due to resource shortage in these two SLRs. The re-optimised dataflow architecture provides a parallelism of $k = 5$ and delivers a speed-up of 1.5x over the original MAX4C design.

There are two potential options to improve performance in this case. It would be possible to increase the parallelism of the design mapped to the SLR not used by the AWS shell. However, this would also require additional changes to the CPU code to distribute the workload onto the different designs to address this imbalance. For this work I decided that the achievable performance advantage would not justify the additional design complexity. The second option is not to create multiple instances of the design, but only one with a higher parallelism. In the cases of the other cards, it is difficult to split the five dataflow Kernels of the Asian option pricing application across three or four SLRs of the same size. Here one can make use of the fact that most of the area is used in two Kernels while the others remain smaller. As a result, the tool can potentially spread the design a lot better across the three available SLRs of which two are already partially occupied. The existing code can be utilised to create this

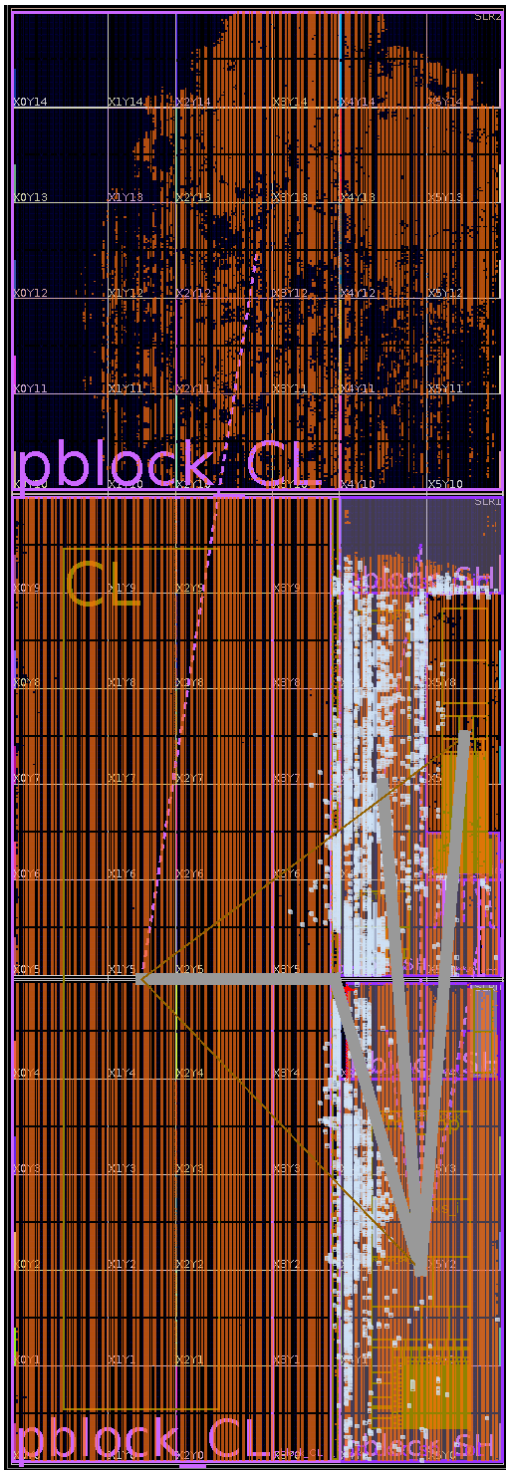

Figure 4.12: Chip image of the F1 bitstream. SLR0 and SLR1 (the two SLRs at the bottom) are highly congested since they share resources with the AWS shell.

bitstream by instantiating only a single design instance with a parallelism factor of $k = 18$. The resulting design achieves a speedup of 1.75x compared to the initial MAX4C design. The lower frequency achieved for this design can be traced back to the fixed interconnect to the AWS shell. In both of these cases a performance increase can again be achieved without changing the actual implementation of the FPGA application but only its configuration.

To provide a fair comparison between the different platforms which also considers the increased area and better timing characteristics of the newer platforms a normalisation of the speedup is needed. To achieve this, I assume linear scaling with the achievable frequency and area. Additionally, I only consider DSPs, since this is the resource which limits the acceleration on the Ultrascale+ based platforms. Since the MAX4C design is actually limited by the logic resources this choice is not perfect, but also considering logic would lead to larger errors, since only 50% of these resources are used on the Ultrascale+ devices. For the datatype used in the application two DSPs of the Ultrascale+ FPGAs are needed for a single multiplication while only one DSP of the Stratix V is required for the same operation. As a result, the normalised speedup is calculated by linearly scaling the measured time on the MAX4C with the frequency and the number of DSPs divided by two. The normalised speedup is shown in tab. 4.1.

One can see that the unoptimised designs for the U200 card achieve a normalised speedup of only 0.57x showing that a lot of the additional hardware capabilities are not used. In contrast the optimised designs for the U200, U250 and MAX5C achieve a normalised speedup of around 1.3x. Since all three cards achieve a similar normalised speedup, we can conclude that they are used similarly well. The speedup larger than one can be explained by the fact that the DSPs of the MAX4C are not the limiting resource. In fact, nearly 30% are unused nicely matching the normalised speedup of 1.3x. For the F1 designs it was not possible to make optimal use of the available chip area. This can be traced back to the problems the Maxeler tools have with meeting timing on this platform due to the static interconnect of the shell which means that more of the chip has to remain unused to achieve timing closure. All in all, one can see that the proposed techniques managed to make full use of the Ultrascale+ based cards and managed to get close to achieving this also for the F1 instance.

It should be stressed that it was not necessary to modify any Kernel code to achieve these results and only the Manager code was changed. Furthermore, the required code size per individual platform is in the order of tens lines of code. As such maintaining this application for current and future platforms, e.g., other cloud providers or the Alveo U50 and U280, is greatly simplified without degrading performance. Building for different platforms is achieved by simply passing different command line parameters to the build system.

4.4 Summary

In the beginning of this chapter the Balanced Memory Mapping (BMM) algorithm is presented, which maps logical to physical memories. The proposed algorithm aims at balancing allocation between different physical memory resources in partitioned large designs, to facilitate locality in multi-die FPGAs. The proposed algorithm is compared to three memory mapping algorithms representing different optimisation goals and commonly used methods, including the standard Xilinx Vivado memory mapping algorithm, using 33 different benchmarks and real applications. Only the proposed algorithm managed to successfully produce place and route results for all test cases while the second-best performing algorithm had a failure rate of one third for large and complex applications.

The proposed algorithm and threshold values were shown to work well in the case of static dataflow applications. The algorithm is used by MaxCompiler since version 2017.2.2 and was successfully employed in multiple research and industrial projects.

In the second part of this chapter methods and best practices to achieve performance scalability employing the methodology developed in this thesis are presented. It is shown how a real application can be modified to migrate it to new platforms and which design decisions can be made to easily generate efficient implementations for a wide variety of platforms. A speedup of up to 7.4x on the newer device generation is achieved, by replicating designs and fine tuning the used parallelism. Additionally, a speedup figure normalised to the additional device capabilities shows that the devices are well used and the new hardware capabilities are not wasted. For

this, only tens of code lines had to be added for each new targeted platform, while keeping the remaining code base unchanged.

Fig. 4.13 provides an overview of the methodology as extended in this chapter. The memory mapping algorithm is conditionally used for multi-die FPGAs containing heterogeneous memory resources to obtain a hardware implementation. If the memory mapping algorithm is used the performance model (Part 3) has to consider this. Additionally, the design of the architecture (Part 4) has to consider the best practices for performance portability.

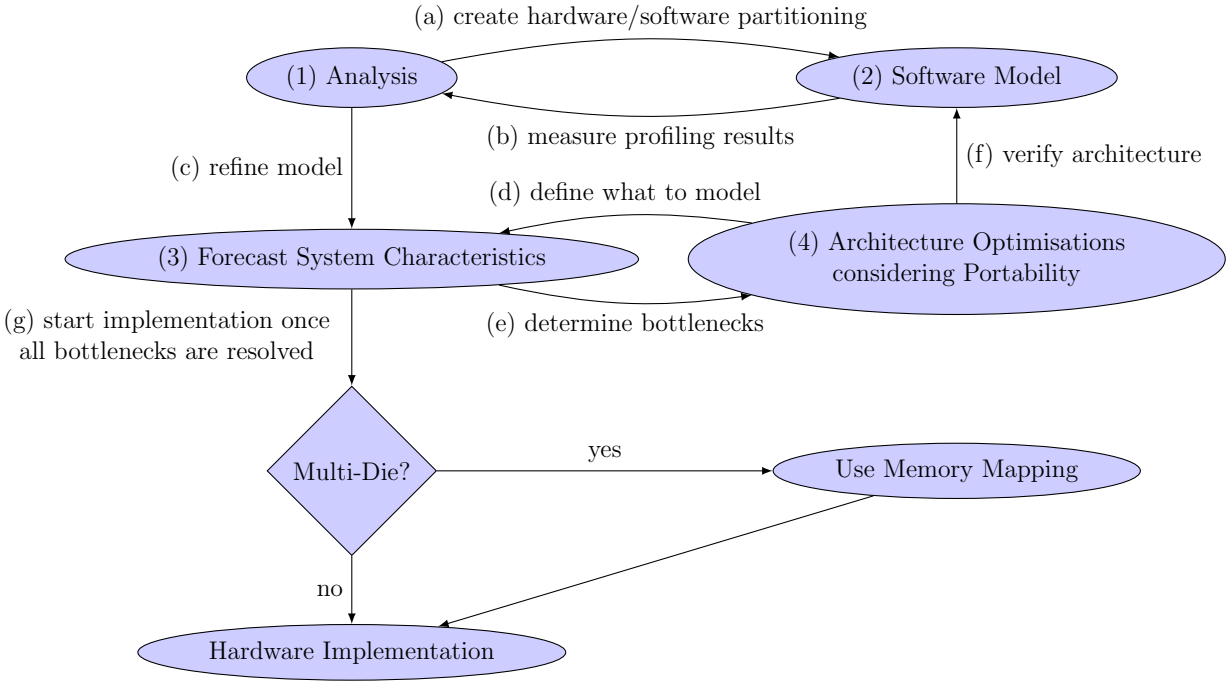

Figure 4.13: Extended design methodology including the contributions in this chapter.

Chapter 5

Methodology Validation using a Real Application

5.1 Introduction

To address the challenge of weak evaluations of FPGA tools and methodologies on simplistic use cases, this chapter assesses the methodology and the tools developed in the last two chapters on a real-world application for validation and evaluation. This application is a Monte Carlo based simulation of dose accumulation in the context of radiotherapy.

Simple benchmarks are not sufficient to evaluate system level performance effectively or to predict application performance as is exemplified by the discussions on the widely used LINPACK benchmark [37,38,75]. The authors of [23] show that simple synthetic benchmarks provide only poor indications of application performance and only the combination with significant application analysis will lead to meaningful predictions. Similarly, the authors of [123] use genetic algorithms for predicting application performance based on memory bandwidth. However, this requires a-priori knowledge about the specific bottleneck of the system under study, which might be possible for von Neumann architectures, but does not map well to FPGAs with their high degree of freedom in terms of system architecture design and their undefined machine model. As such for this thesis I selected applications which required a significant amount of development

effort to ensure sufficient complexity to draw meaningful conclusions. This for example includes realistic, application specific data transfers between hosts and accelerators. Only in this way it is possible to validate that forecasting the system properties correctly highlights potential system bottlenecks which can be alleviated by developing a suitable architecture. This is a major difference to previous work, which often focuses on a few oversimplified (predominantly synthetic) examples. The complete methodology is evaluated using the application presented in this chapter and additionally the applications presented in section 3.6 provide insight into the methodology without additional portability support.

The additional contribution of this chapter in comparison to the evaluations presented in the last two chapters is that the complete methodology including the performance scalability considerations is evaluated. The previous evaluations focused on certain aspects of the overall methodology. Additionally, the application used for evaluation in this chapter presents a worst case due to its highly dynamic execution pattern and high reliance on random number generators to influence the execution. This enables a thorough evaluation highlighting potential problems of the methodology and especially performance prediction.

Radiotherapy is a commonly used treatment for various cancer types. High doses of radiation are used to kill cancer cells. Modern radiotherapy relies on an intensity modulation technique that aims to deliver high dose gradients to cancerous tissues while sparing the surrounding healthy organs as much as possible. This is achieved by setting up a therapy treatment plan which takes into account the anatomy as well as the clinical case and dose delivering machine. In order to validate and optimise such therapy plans, the expected spatial dose distribution within the patient has to be simulated before the actual treatment. This is often implemented by Monte Carlo methods which simulate the pathway of millions of radiation particle trajectories as they enter the patient body. These simulations are highly accurate. On the other hand, they require relatively long computation times.

Historically, these long computation times were not a problem. However, modern treatment machines in addition to radiation delivery, also allow imaging of the patient during treatment [78]. Real time dose simulation would allow patient treatment adjustments in real time. This is

advantageous since, e.g., in the case of prostate or lung cancer target tissue might significantly move between imaging and treatment or even within one treatment session. The usage of real time imaging techniques will enable doctors to adapt to these changes and facilitate accurate dose delivery. This would minimise dose accumulation in healthy tissue and therefore reduce the damage caused. Additionally, it will be possible to significantly reduce the number of treatments per patient by delivering a higher energy at shorter time due to more targeted radiation. While this would decrease the overall treatment costs and improve treatment quality for the patient, it is crucial to ensure very high accuracy to compensate for the high energy delivery. As a result of this, the simulation has to be repeated regularly based on new measurements. According to medical experts, the time required for the simulation of the system has to stay below one second to facilitate real time updates [26].

To solve the computational challenge of real time dose simulation, different technologies have been proposed which utilise CPUs or GPUs on local or cloud-based systems. However, in the case of CPUs and GPUs the size of the machine required to meet the real time target is prohibitive. In the case of cloud-based systems privacy concerns, bandwidth requirements and latency issues as well as the need to guarantee service quality during treatment provide major challenges for practical deployment.

In this chapter, the usage of FPGAs to address these problems is discussed in order to build the first real time radiotherapy simulation system. The target is to perform the complete Monte Carlo simulation in less than one second. There is a long history of accelerating Monte Carlo simulations using FPGAs. The inherent parallelism of Monte Carlo simulations allows high speed-ups on FPGAs. Additionally, FPGA implementations are highly predictable making them especially suited for real time applications. Finally, the compute density of FPGA based systems is typically superior, allowing for placement directly in the medical facility. As a result, FPGAs are an excellent fit for the problem of real time dose simulation.

The main contributions of this chapter are as follows:

- A dataflow architecture for Monte Carlo based dose accumulation simulation;

- An analytical model to estimate hardware usage and accurately assess performance;
- Evaluation of the architecture and model using implementations based on a Xilinx VU9P FPGA and the Xilinx Alveo U250; and
- A discussion of the methodology developed in this thesis based on its usage for multiple real and complex applications.

The remainder of the chapter is organised as follows. Section 5.2 will present the architecture used for the FPGA implementation. The performance will be modelled in section 5.3. Section 5.4 will present and evaluate implementations of the proposed architecture. In section 5.5 I discuss the methodology presented in this thesis based on the lessons learned from the applications it was applied to. Finally, section 5.6 will conclude the chapter with a summary.

5.2 Architecture

For simplicity the full capability of DPM as described in section 2.6.3 is not completely implemented. For example, this work focuses only on the simulation of electrons, does not consider Bremsstrahlung and only uses water as material within the patient cube. However, these simplifications have no impact on the feasibility of the architecture and adding the actual implementation will add only minimal overhead. More importantly it does not impact the ability to evaluate the proposed methodology using this application. For the full feature set the following changes are required. Bremsstrahlung is an additional form of particle interaction and as a result only needs more area. Different materials can be implemented as added on-chip memory initialised from DDR. Additionally, interaction equations will need to select different material coefficients dependent on the current voxel. All in all, the simulation Kernel area will slightly increase and a bit more on-chip memory as well as DDR bandwidth will be required.

One of the major challenges involved in implementing the dose accumulation simulation is the memory access into the patient cube. This is due to the stochastic paths an electron takes through the patient cube. As a result, the position of the memory access into the patient cube

to accumulate the dose is also stochastic. The performance model (Part 3) helps to determine the impact on the runtime. For each voxel in the patient cube 36 bit have to be accessed. Eq. 5.1 calculates the fraction of useful data one can expect to get by reading a single burst from one DDR DIMM. Using one DIMM already represents the best case, since more DIMMs would increase the burst size. The efficiency factor for reading a single burst is determined using fig. 3.4. The resulting achievable memory bandwidth is calculated as shown in eq. 5.2.

$$efficiency = \frac{useddata}{burstsize} = \frac{36bit}{512bit} = 0.07 = 7\% \tag{5.1}$$

$$BW_{act} = BW \times efficiency = 45GB/s \times 0.07 \times 0.171 = 0.54GB/s \tag{5.2}$$

It hast to be considered that it is not only necessary to read the voxel but also to write it back after a dose was deposited at the position of the voxel. It is possible to calculate how many voxel positions can be accessed per second as shown in eq. 5.3. Since to achieve statistically significant results 100 million electrons have to be generated and a runtime of less than a second is targeted this speed is not sufficient. This is even more the case if one considers that for each electron multiple voxels have to be accessed before the electron gets fully absorbed. For this reason, the patient cube has to be buffered on-chip.

$$VoxelPerSecond = \frac{BW_{act}}{2 \times S_{voxel}} = \frac{0.54GB/s}{2 \times 36bit} = 64,550,339\frac{1}{s} \tag{5.3}$$

As such the performance model (Part 3) has successfully discovered an initial bottleneck (Step e) and it is now possible to perform a design iteration to adjust the architecture. The next consideration concerns the storage of the patient cube in on-chip memory. Eq. 5.4 calculates the required on-chip memory if one considers a patient cube with a resolution of 256 voxels in all three dimensions. Comparing this result to the available on-chip capacity of the available devices as shown in tab. 2.1 shows that implementing the patient cube in on-chip memory is also not possible. As such the architecture has to be adjusted further (Step e).

$$S_{patient\_cube} = x \times y \times z \times S_{voxel} = 256 \times 256 \times 256 \times 36bit = 576Mbit \tag{5.4}$$

To avoid the limitation of the patient cube resolution, it was decided to decompose the patient cube into multiple subdomains, where each subdomain fits into on-chip memory. Since only water is considered as a material, the on-chip patient cube buffer only needs to store the dose. If other materials are also used one would need to also store the material type which can be encoded in usually two bits. Due to on-chip buffering of the patient cube, fully random memory access can be performed without impacting performance.

The buffer containing the patient cube is implemented in a Kernel. Additionally, this Kernel contains the arithmetic to perform the actual simulation of the electrons and the calculation of the emitted dose. As described above, the simulation of the electron decides which interaction occurs. Based on this, the emitted dose is calculated, and the values of the electron can be updated. The updated electron moves into a new direction and has updated energy and fuel values. The energy and fuel values determine which interaction occurs and when an electron gets absorbed. In the CPU implementation, a while loop is executed for each externally generated electron, which repeats these steps until the energy of the electron is depleted. This is shown in fig. 5.1, which depicts the loopflow graph of the original application and summarises the results of the analysis of the original application (Part 1).

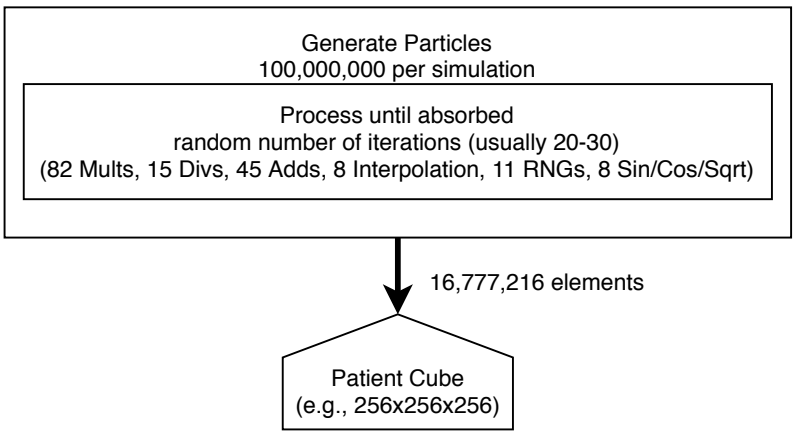

Figure 5.1: Loopflow Graph of the CPU implementation.

However, in the presented case, the Kernel accepts new electrons, evaluates the interaction and outputs the updated electrons on every cycle. Since the Kernel is deeply pipelined, a loop

implementation is not feasible. To circumvent this, the processing order of electrons differs between CPU and FPGA. This is a valid transformation, since all electron interactions are fully independent due to the embarrassingly parallel nature of this Monte Carlo simulation. As a result, the data arbitration and loop logic are handed off to a different component, which also handles the transport of electrons between subdomains.

Fig. 5.2 shows the simplified architecture (Part 4) of the application. An *External Particle Generator* Kernel generates new electrons and sends them to the *Particle Distributor* ManagerStateMachine. The particle distributor has three inputs, one from the *External Particle Generator*, one from DDR and another from the Kernel containing the subdomain buffer and interaction simulation. Additionally, it has outputs to DDR and to the particle simulation Kernel. This Kernel sends the patient cube back to the host via PCIe once the dose is calculated and forwards the updated electrons to the particle distributor.

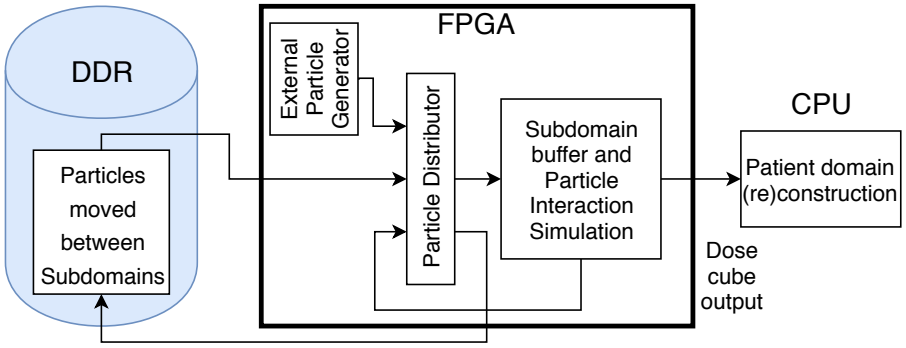

Figure 5.2: The simplified architecture of the dose accumulation simulation.

The particle distributor handles the arbitration of the electrons and controls the simulation Kernel. It decides when the simulation Kernel is going to switch to the next subdomain. Additionally, it makes sure that only electrons which are in the current subdomain are sent to the Kernel. If they belong to a different subdomain, they are buffered in DDR and read as soon as the Kernel switches to the correct subdomain.

The amount of data that has to be stored for each electron approximates the DDR4 burst size. To simplify the memory layout, it was decided to pad the electron data structure to 512 bits. For each subdomain the same memory capacity is reserved. However, this decision reduces the number of electrons that can be buffered in the DDR memory. Since around 100 million

electrons need to be generated for statistically significant results and each of those electrons can create multiple additional electrons, this has to be considered.

In the worst case we need to allocate enough memory for each subdomain to buffer the complete 100 million electrons. Eq. 5.5 calculates the capacity of memory required per subdomain. This means in a single DIMM, which has a capacity of 16 GB we could only allocate enough memory for two subdomains. Even if the complete on-board memory capacity of the Xilinx Alveo U250 is considered the number of subdomains is limited to ten subdomains as shown in eq. 5.6. This limitation has to be addressed through further changes to the architecture (Step e, Part 4).

$$S_{electron\_buffer} = n_{electrons} \times S_{electrons} = 100,000,000 * 512bit = 5.96GB \tag{5.5}$$

$$n_{sub\_domains\_U250} = \frac{S_{U250\_on\_board}}{S_{electron\_buffer}} = \frac{64GB}{5.96GB} = 10.7 \tag{5.6}$$

Since it is not possible to buffer all electrons for a specific subdomain in the allocated on-board memory block if the complete simulation is run the overall simulation is split into multiple batches. Within each batch the simulation runs through all subdomains, which means that each subdomain of the patient cube is processed multiple times. However, it was decided that the architecture could be further simplified by sending the current part of the patient cube back to the CPU once processing of the current batch is finished. As a result, it is not required to accumulate the dose of multiple batches on the FPGA, which removes the requirement to buffer the results of the simulation on-card once processing for the current subdomain is finished. As such, the patient cube is implemented in a double buffered fashion. Therefore, when processing of a subdomain finishes the buffers can be switched and the now inactive half can be streamed out and set to zero in preparation for the next subdomain.

As a result of splitting the patient cube into subdomains, a problem occurs if an electron updated by an interaction moves into an already processed subdomain. Since multiple batches are used, it is possible to buffer these electrons in DDR for the next batch. However, on the last

batch this is not possible. To address this problem, another output to the particle distributor is added which sends those electrons back to the CPU when the last batch is currently processed. Since the number of electrons sent back is orders of magnitude smaller than the total, it is possible to simulate those electrons on the CPU. Additionally, the simulation of the electrons sent back is started as soon as they arrive to overlap the execution on the CPU and on the FPGA as described in section 3.5.3.

In the proposed architecture, DDR memory is used only to buffer electrons. Potentially, the number of electrons which have to be buffered in DDR is very large. Eq. 5.7 calculates how many electrons can be written to the buffer per second if three DIMMs are used[1]. The efficiency factor is selected for the worst case where each individual electron might move to a different subdomain. Additionally, it would also be necessary to read the electrons again reducing this number further. Even though one can expect that not all electrons have to be buffered the number of electrons which can be buffered would be limited significantly. Furthermore, this equation assumes that all three DIMMs are used. This limits architecture options later on to handle the presence of SLRs. As such, the access patterns need to be considered to optimise the achievable bandwidth. By using long continuous memory access one can get closest to the theoretical peak memory bandwidth.

$$ElectronsToBufferPerSec = \frac{BW * efficiency}{S_{electron}} = \frac{45GB/s \times 0.171}{512bit} = 129,100,677\frac{1}{s} \quad (5.7)$$

Reading electrons from DDR is inherently linear, since one can simply read all electrons buffered for a specific subdomain sequentially. However, the access pattern on the write side is not linear. Since the direction of electrons after interaction is based on random number generators, it is likely that each electron is written to different parts of the memory. To alleviate this problem an additional ManagerStateMachine is added, which has small on-chip buffers for each subdomain. Multiple electrons are accumulated in these on-chip buffers and only when they

[1]Note that this is double the figure as calculated in eq. 5.3 since in both cases one burst gets accessed at a time

are full the complete buffer is written to DDR. Additionally, they can be flushed by the particle distributor to make sure that all electrons for the current subdomains are written to memory, so that they can be read again for processing. It was decided to make these buffers hold sixteen electrons, which limits the required on-chip memory capacity but already manages to achieve up to 90% of the peak bandwidth (see fig. 3.4). By packing all individual buffers into a single on-chip memory one can also increase the on-chip memory utilisation. Each individual buffer has a unique address range in the bigger on-chip memory and since only one particle can be received per cycle there is no possibility for write port conflicts. By ensuring that read and write patterns are linear the on-board memory bandwidth is improved significantly.

The area required for the simulation of a single electron is small compared to the area available on modern FPGAs (see section 5.3 for the detailed calculations). As such, one cannot only rely on the pipeline parallelism but also needs to exploit algorithm level parallelism to use all available chip resources. In this work the inherent parallelism of the Monte Carlo simulation is exploited on two levels.

The first additional level of parallelism creates multiple instances of the entire design. The motivation for this can be found in the targeted platform (see section 5.4), which are FPGA accelerator cards based on the Xilinx Ultrascale+ architecture. The big devices used in this work consist of multiple individual die and interconnectivity between these die is limited. As such it is often a good idea to treat those die like separate FPGAs (see BP8.2 in section 4.3.1). On the platforms used here, each die is connected to one DDR4 DIMM and as a result implementing one design on each die is straight forward. The individual designs only share the PCIe controller and are otherwise completely independent.

The second additional level of parallelism enables the processing of multiple electrons in parallel within the same simulation Kernel. Parallelising the computation in the Kernel itself is accomplished by simply duplicating the dataflow graph. However, the patient cube buffer has to be shared to save on-chip memory resources. As a result, one needs to consider potential memory access conflicts. To decrease the likelihood of such events, each xy plane of the cube is implemented as a separate memory instance. Thus, z individual memories holding the data

of one xy plane are created. This will also help with timing closure, since big on-chip memory structures often have problems routing control signals between memory columns.

Another ManagerStateMachine is introduced which checks the electrons coming from the particle distributor for memory access conflicts. Only if the z position of the electrons is different, meaning different physical memories are used, or they access the same memory position, all electrons are sent to the simulation Kernel. Otherwise, only a conflict free subset is forwarded. To avoid starving one input, a round robin scheme is used to prioritise all inputs fairly. Since it is non-trivial to parallelise the particle distributor, it was decided to instead create one instance of the particle distributor for each electron processed in parallel. This also means that the on-board memory space has to be equally split between each particle distributor. The overhead introduced by this is negligible, but the implementation complexity is significantly reduced.

The final architecture for a single die where the Kernel processes two electrons per cycle is shown in figure 5.3. All arrows, apart from the Kernel output sending the dose cube to the host, represent electrons. These connections use FIFOs.

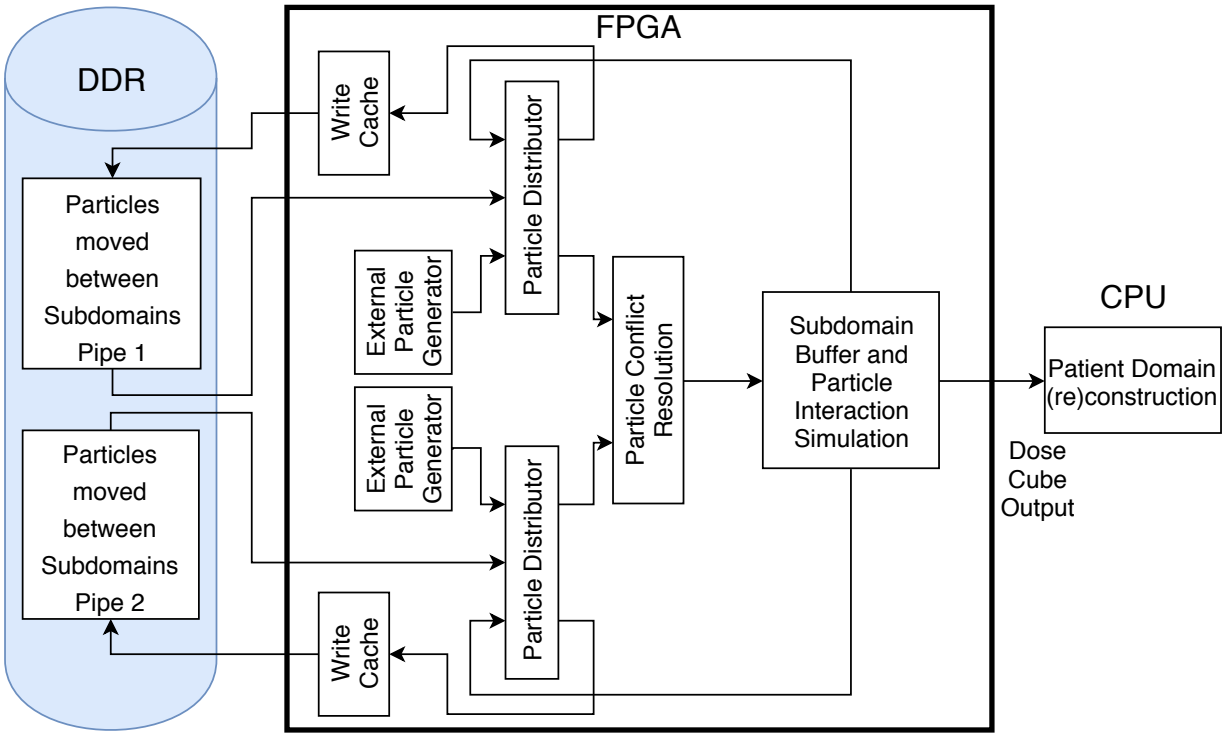

Figure 5.3: The architecture of the dose accumulation simulation for a single FPGA die if the Kernel processes two electrons on every cycle.

Fig. 5.4 shows the loopflow graph of the application after all the proposed changes for a patient

cube size of 256 in all dimensions, split into subdomains of size 128 in x dimension and 64 in all others. Additionally, the execution is done in two batches. In comparison to fig. 5.2, one can see, that the algorithm structure is more complicated, but enables further parallelisation and more optimal usage of the hardware.

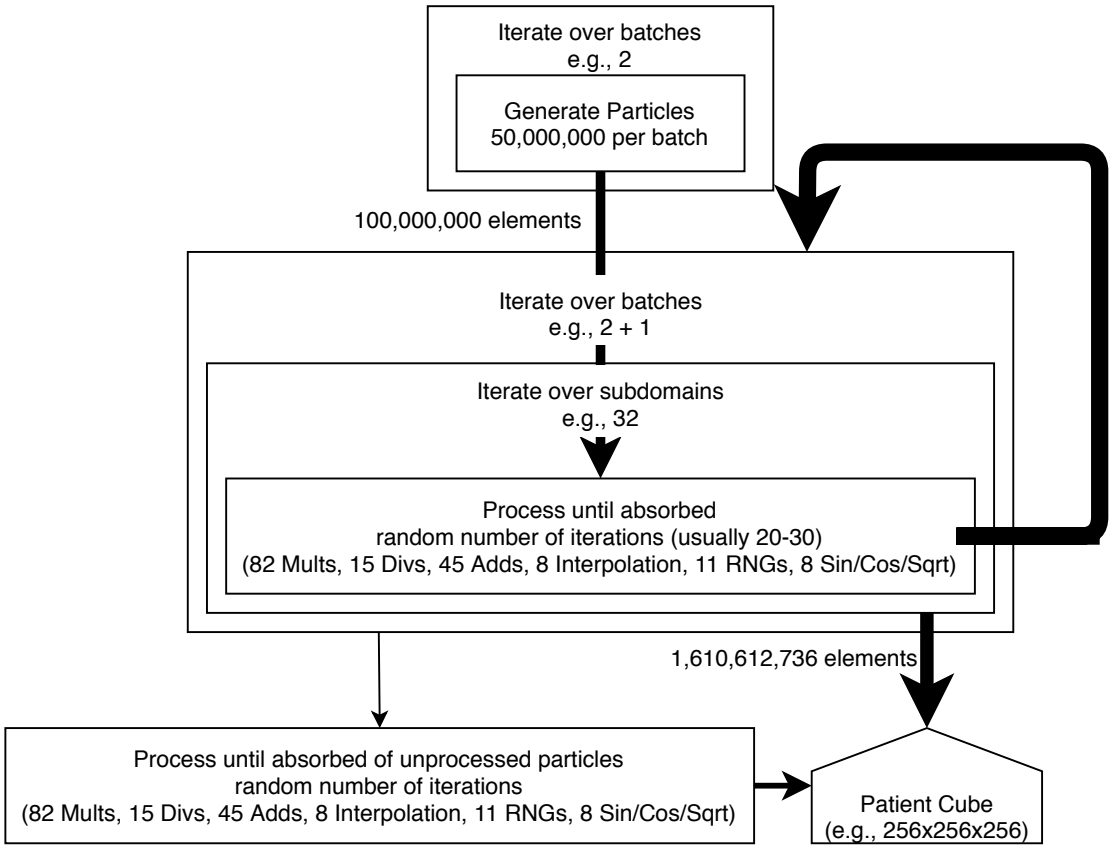

Figure 5.4: Loopflow Graph of the FPGA implementation.

To summarise, the main technical challenges are to support big voxel cubes and the processing of multiple electrons within the same Kernel using the same on-chip dose memory. The first challenge is addressed by splitting the voxel cube into multiple subdomains and adding particle distribution logic to deal with electrons transitioning between subdomains. Additionally, one needs to add logic to improve memory efficiency and maximise the usable memory bandwidth. The second challenge is addressed by adding a unit for resolving potential memory conflicts at the input of the Kernel.

5.3 Performance Model

The architecture described above was developed using an iterative process of performance modelling and refinement based on the analysis of the initial code base (Steps c, d and e). Only the final results are presented in this chapter. The performance model (Part 3) will be used to verify if the proposed implementation meets the expectations in section 5.4.

One of the major challenges in modelling the performance of this application is the extensive use of random number generators. For example, after how many iterations an electron is absorbed, and the number of electrons stored in DDR are both variable. As such, the performance model is built on estimations based on measurements using the CPU code (Step b).

The number of electrons updated by interactions before they are absorbed is denoted as n_{inter}. The percentage of electrons which move between subdomains, and therefore require DDR buffering, is noted as p_{sub}. Finally, the percentage of cases in which there is a memory access conflict in the patient cube buffer of the simulation Kernel is represented by p_{mem}. These factors are also highly dependent on the way in which the external electrons are generated and as a result section 5.4 will discuss these factors in more detail while all equations are kept generic here.

In this section, equations for area usage, the achievable electron processing speed, memory bandwidth and finally PCIe bandwidth requirements are provided.

5.3.1 Area Usage

To forecast the area usage of the implementation the number of operations in the CPU code needs to be counted. The simulation of the electrons does include multiple trigonometric functions and square roots. Some of those are available in MaxCompiler and the remaining ones are implemented as a linear interpolation between values in a ROM lookup table.

Tab. 5.1 shows the operation count and the predicted area usage for one simulation Kernel which processes one electron per cycle. The area usage for each operation is determined using micro benchmarks and then simply multiplying these with the number of operations needed

and calculating the sum over all operations. The area usage will scale linearly with the number of processed electrons per cycle. It should be noted that additional memories and FFs are needed for scheduling of the dataflow graph.

Table 5.1: Overview of operation count and predicted area usage for the simulation of one electron per cycle.

Multiplications	Divisions	Additions	Interpolation	RNG	Sin/Cos/Sqrt	LUT	FF	DSP	BRAM
82	15	45	8	11	8	85,000	120,000	408	54

The simulation Kernel also contains the memory to buffer the patient cube. The size of this memory depends on the dimensions of the subdomain, x_{sub}, y_{sub} and z_{sub} in voxels. Additionally, the depth and width of the physical memories has to be considered, which is called mem_d and mem_w respectively. Eq. 5.8 calculates the number of physical on-chip memories required for a single xy plane. The parameter $accWidth$ represents the number of bits required for the datatype used for the dose accumulation. In total z_{sub} of these memories are needed. However, they might use different memory resources, since MaxCompiler will automatically use either BRAMs or URAMs using the algorithm described in section 4.2.

$$\#mem_{cube} = \left\lceil \frac{accWidth}{mem_w} \right\rceil * \left\lceil \frac{x_{sub} * y_{sub}}{mem_d} \right\rceil \tag{5.8}$$

In addition to the Kernel resources, one also has to consider the ManagerStateMachines and other Manager blocks. The ManagerStateMachines predominantly use LUTs, FFs and on-chip memory. Based on the complexity of the ManagerStateMachines and previous experiences one can make the worst-case estimation, that the number of LUTs and FFs required per ManagerStateMachine is less than 5,000 and 10,000 respectively. The particle distributor does not need any additional memory resources, while the write cache to improve memory efficiency mainly consists of a single buffer. The size of this buffer can be estimated using eq. 5.9 with $elec_w$ representing the width of the electron data structure in bits without padding, 417 bits in the presented case. The depth is determined by the total number of subdomains necessary. d represents the depth of the memory per subdomain, which in this case is 16.

$$\#mem_{cache} = \left\lceil \frac{elec_w}{mem_w} \right\rceil * \left\lceil \frac{\frac{x_{cube}}{x_{sub}} * \frac{y_{cube}}{y_{sub}} * \frac{z_{cube}}{z_{sub}} * d}{mem_d} \right\rceil \tag{5.9}$$

Lastly, the remaining Manager blocks need to be considered. The memory requirements for each FIFO can be estimated using eq. 5.10. Usually, the depth of a FIFO is 512 and since most FIFOs buffer electrons the width is usually either 417 or 512 bits. Each memory controller requires 3 DSPs, roughly 20,000 LUTs and 30,000 FFs and around 50 BRAMs. Per design instance one memory controller is required. Finally, the resource requirements for the PCIe controller can be estimated as 8,000 LUTs, 12,000 FFs and 35 BRAMs. The PCIe controller is shared between all instances of the design.

$$\#mem_{FIFO} = \left\lceil \frac{FIFO_w}{mem_w} \right\rceil * \left\lceil \frac{FIFO_d}{mem_d} \right\rceil \tag{5.10}$$

5.3.2 Electron Processing Speed

To calculate the electron processing speed, the number of electrons that can be processed by the Kernel at a given frequency has to be estimated. Eq. 5.11 shows how to calculate this. n_{elec} represents the number of electrons processed per second, while n_{design} and n_{pipes} represent the parallelism in terms of number of instances of the design and electrons processed in parallel. Finally, f represents the estimated frequency the implementation will be running at.

$$n_{elec} = n_{design} * (p_{mem} + n_{pipes} * (1 - p_{mem})) * f \tag{5.11}$$

Additionally, it has to be considered that for each subdomain the on-chip buffer has to be written back to the host. Normally, this can be overlapped with the compute latency using double buffering. However, if only a very small number of electrons belong to a given subdomain the time required for the computation might not be sufficient to flush the previous buffer. As a result, the design needs to wait for the previous buffer to be fully written back before it can switch to the next subdomain. The number of cycles required for that per subdomain can

be calculated as shown in equation 5.12. In this case, $readout_{width}$ represents the number of voxel values read from the patient cube buffer per cycle. The size of the overlap between the flushing of the patient cube buffer and the electron calculation heavily depends on the electron generation pattern.

$$cycles_{flush} = \frac{x_{sub} * y_{sub} * z_{sub}}{readout_{width}} \tag{5.12}$$

5.3.3 Memory Bandwidth Requirements

The total amount of data that need to be transferred to and from DDR memory, S_{DDR}, is calculated in equation 5.13. Each electron requires 64 bytes and needs to be written and read only once. Additionally, the data volume depends on the number of electrons created by the external particle generator $n_{elec,total}$. The required bandwidth can then be calculated as a function of the execution time t_{total} as shown in equation 5.14, where DDR_{eff} is the average memory efficiency.

$$S_{DDR} = 2 * 64 * n_{elec,total} * n_{inter} * p_{sub} \tag{5.13}$$

$$BW_{DDR} = \frac{S_{DDR}}{t_{total}} * \frac{1}{DDR_{eff}} \tag{5.14}$$

5.3.4 PCIe Bandwidth Requirements

The PCIe bandwidth requirements are determined by two factors. The patient cube has to be streamed back to the host and in addition the electrons, which cannot be processed within the last batch, have to be sent to the CPU as well. Eq. 5.15 estimates the amount of data that has to be transmitted for the patient cube. It can be assumed, that all values sent back from the FPGA are converted to single precision floating-point, to ease usage on the CPU side of the system. As such, the total amount of data is simply the product of the cube dimensions,

the number of batches that are processed and four, the size of single precision floating-point number in bytes.

$$S_{PCIe,PatientCube} = x_{cube} * y_{cube} * z_{cube} * batches * 4 \tag{5.15}$$

Additionally, the amount of data transferred for the electrons that have to be sent back to the CPU is calculated in eq. 5.16 based on the number of electrons sent back $n_{electron,PCIe}$. This factor again depends on the external particle generation.

$$S_{PCIe,Electron} = n_{electron,PCIe} * 64 \tag{5.16}$$

The required bandwidth can be obtained by adding both equations and dividing by the execution time.

5.3.5 Lessons Learned from the Performance Model

As stated above, the performance model and architecture were iteratively developed together, problems identified by the performance model led to changes to the architecture. I tried to highlight during the architecture explanation how some of its properties are a direct result of these design iterations but focus on only the final iteration of this pairing to avoid confusion. It has to be stressed that the detailed performance model was crucial to derive the proposed architecture. In this section a few examples are presented to show the impact the performance model had on the architecture development and how it was used for design space exploration.

It was determined that it would be infeasible to store the complete voxel cube on-chip and it would have to be stored in DDR memory (see eq. 5.4). Afterwards the required memory bandwidth to access the memory regions along the trajectories of the particles were modelled (see eq. 5.3). Due to the random nature of these trajectories, it was quickly discovered that an architecture which fully simulates one particle at a time would be heavily limited by off-chip

memory bandwidth. To mitigate this issue, it was decided to instead focus on sub-regions of the memory cube which are simulated one at a time. This decision leads to most other design decisions in the proposed architecture.

By modelling the memory bandwidth requirements more accurately, it was noticed that the process of writing particles which cross subdomains back into the memory might lead to high bandwidth requirements as well (see eq. 5.7). This is caused by the random-access pattern. To circumvent this issue the write caches were added.

The performance model was also used to determine what could be sent back over PCIe to the host computer at what time. The model showed that the overhead of accumulating the dosage cube on the FPGA would be relatively big and instead it is possible to send each individually processed subdomain to the host where the partial results can be combined. The model suggested that in most cases it would be faster to send particles which cannot be processed in the last batch to the CPU rather than making another iteration.

The hardware builds generated in section 5.4 are a direct result of the design space exploration using the performance model. The evaluation of the equations for area usage enabled a quick determination of the possible sizes for the subdomains stored on the chip. Additionally, it was possible to calculated how much parallelism was feasible while not requiring too much area or too much I/O bandwidth.

5.4 Evaluation

To evaluate the architecture, it was implemented using Maxeler MaxCompiler version 2019.1 and Vivado 2018.3. The target platforms are the MAX5C DFE and the Xilinx Alveo U250.

The MAX5C based implementation was evaluated with different degrees of parallelism and cube sizes. Up to three MAX5C cards in a 2U server powered by two six-core Intel Xeon E5-2643 v4 CPUs running at 3.4 GHz were used. The U250 design was tested on only one card, which was hosted in a server powered by two 18-core Intel Xeon Gold 6154 CPUs running at 3.0 GHz.

Even though servers are used, building a workstation with a similar configuration is possible.

5.4.1 Area Results

Eight different configurations were implemented, four for the MAX5C and another four for the Alveo U250. In the case of the MAX5C all configurations use three design instances to make optimal use of the three die of the VU9P. For builds 1 and 3, the simulation Kernel processes only one electron per cycle, while builds 2 and 4 process two. In the case of builds 1 and 2, the patient cube size is set to 128 voxels in each dimension and the subdomain size is 64 voxels accordingly. For builds 3 and 4, the resolution is increased and the cube size is set to 256 in each dimension. The subdomain in these cases consists of 128 voxels in x dimension and 64 in y and z.

The four builds targeting the Alveo U250 all consist of four design instances to make optimal use of the four SLRs and DIMMs available. Builds 5 and 7 process one electron per cycle while 6 and 8 process two. Again, the patient cube size is set to 128 voxels in all dimensions and the subdomain size to 64 voxels for builds 5 and 6. Builds 7 and 8 have a patient cube size of 256 in each dimension and the subdomain size is set to 128 in x dimension and 64 in y and z. All designs achieve the same frequency due to the systolic array property of the Maxeler Kernels and their general similarity in terms of architecture when it comes to the memory controller usage and the spread across SLRs. An overview of all design points is provided in tab. 5.2 and the area usage for these designs is depicted in tab. 5.3.

Table 5.2: Design points evaluated for the proposed architecture.

Num	Card	Frequency	Design Count	Kernel Parallelism	Cube Size	Subdomain Size
1	MAX5C	250 MHz	3	1	128	64
2	MAX5C	250 MHz	3	2	128	64
3	MAX5C	250 MHz	3	1	256	128
4	MAX5C	250 MHz	3	2	256	128
5	U250	250 MHz	4	1	128	64
6	U250	250 MHz	4	2	128	64
7	U250	250 MHz	4	1	256	128
8	U250	250 MHz	4	2	256	128

Table 5.3: Area usage for the different design points.

Num	LUT	FF	DSP	BRAM	URAM
1	339,030 (28.68%)	641,249 (27.12%)	1,233 (18.03%)	1,209 (27.99%)	414 (43.13%)
2	547,980 (46.35%)	1,071,404 (45.31%)	2,457 (35.92%)	2,535 (58.68%)	468 (48.75%)
3	346,363 (29.30%)	669,903 (28.33%)	1,233 (18.03%)	2,469 (57.15%)	708 (73.75%)
4	558,875 (47.27%)	1,108,486 (46.88%)	2,457 (35.92%)	3,471 (80,35%)	804 (83.75%)
5	431,572 (24.98%)	830,041 (24.02%)	1,644 (13.38%)	1,482 (27,57%)	548 (42.81%)
6	704,282 (40.76%)	1,395,707 (40.39%)	3,276 (26.66%)	3,198 (59,49%)	596 (46.56%)
7	441,146 (25.53%)	868,371 (25.13%)	1,644 (13.38%)	3,066 (57,03%)	948 (74.06%)
8	719,487 (41.64%)	1,445,042 (41.81%)	3,276 (26.66%)	4,350 (80,92%)	1,052 (82.19%)

The area usage predicted using the equations presented in section 5.3.1 as well as the error compared to the actual usage are shown in tab. 5.4. The prediction for DSPs is highly accurate and has no error. The LUT prediction is also close to the actual usage with errors ranging from an underestimation of 3% to an overestimation of 14.9% compared to the actual usage. In the case of a higher Kernel parallelism the LUT prediction is slightly too high, which can be explained by an overly pessimistic estimation of the area required by the ManagerStateMachines.

Table 5.4: Predicted area usage results for the proposed design points and prediction error.

Num	LUT	FF	DSP	BRAM	URAM
1	338,000 (-3%)	492,000 (-30.4%)	1,233 (0%)	1,888 (36%)	420 (1.4%)
2	623,000 (12%)	912,000 (-17.4%)	2,457 (0%)	2,043 (-24.1%)	455 (-2.9%)
3	338,000 (-2.5%)	492,000 (-36.2%)	1,233 (0%)	3,547 (30.4%)	789 (10.3%)
4	623,000 (10.3%)	912,000 (-21.5%)	2,457 (0%)	3,702 (6,2%)	824 (2.4%)
5	448,000 (3.7%)	652,000 (-27.3%)	1,644 (0%)	2,391 (38%)	570 (3.9%)
6	828,000 (14.9%)	1,212,000 (-15.2%)	3,276 (0%)	2,574 (-24.2%)	614 (2.9%)
7	448,000 (1.5%)	652,000 (-33.2%)	1,644 (0%)	4,506 (32%)	1,073 (11.6%)
8	828,000 (13.1%)	1,212,000 (-19.2%)	3,276 (0%)	4,689 (7.3%)	1,117 (5.8%)

As a result of the memory mapping algorithm developed in chapter 4 predicting the precise memory requirements is more complicated, since URAM and BRAM allocation is handled automatically. In order to provide predictions for the URAM as well as BRAM usage I decided to use the following approach. First, the memory usage using only BRAMs is predicted. Then it is assumed that the same memories can be implemented using URAMs with an efficiency of 50%, since this leads to a balanced mapping between URAMs and BRAMs as targeted by the memory mapping algorithm used. This means that eight BRAM18 can be replaced by one URAM. Using this assumption, one can calculate how many URAMs would be needed to implement all on-chip memories. Based on this translation between URAM and BRAM

memory capacity one can also calculate the percentage of the overall memory which is available as URAMs and BRAMs respectively. The final resource usage is predicted by multiplying these percentages with the worst-case mapping assumptions calculated above.

Applying this method to the design leads to a URAM prediction which is overall close to the actual usage with an error ranging from -2.9% to 11.6%. The error for the BRAM prediction is significantly larger and in the worst case the BRAM usage is overestimated by up to 38% and underestimated by up to 24.2%. In all cases where the BRAMs usage is overestimated by more than 10% the kernel parallelism is one, meaning that the overall design is less filled. We can trace these errors back to pessimistic estimations for the amount of memory required, e.g., in the case of IP cores and large safety margins. The only cases where an underestimation occurs is in the cases where the kernel parallelism is set to two, but the subdomain size is still limited to 64. The reason for this is that the scheduling of the dataflow graph, which requires FIFOs that are often implemented using BRAMs, is not considered. As a result, especially in cases where the degree of parallelism is higher and the dataflow graph is larger, the error is bigger as well. This means that in general the memory footprint of the application is underestimated, which is especially noticeable in the case of these larger designs. However, the URAMs are mostly used by MaxCompiler to implement the voxel cube buffers, which can be implemented with a URAM efficiency of 75%, which somewhat mitigates this underestimation. This can be seen by the fact that in both cases where the kernel parallelism is two and the subdomain size in x dimension is 128 the memory usage is again overestimated, by 6.2% and 7.3%. This means that the efficient memory usage for the larger voxel cube buffers mitigates this issue. As such the performance model tends to overestimate memory usage for design points with less kernel parallelism while underestimating it for design points with a higher degree of parallelism.

It should be noted that the performance model is usually only used for DSE. There is little benefit of accurately predicting the resource usage for smaller design points if a bigger device is targeted anyway. As such the model developed in the last section made some worst-case estimations that would need to be revised to also model smaller designs more accurately. If there would be a desire to target a smaller device this would be recommended. However, the prediction error for the two designs which fill up the chip most (build four and eight) is lower

than for most other designs, especially if memory usage is considered. More importantly, apart from FFs, which are not a limiting resource, the resource estimates overestimate the usage of resource in these two cases and does not underestimate them. While an overestimation might lead to a suboptimal design since resources are left unused and potentially more promising design points are discarded an underestimation might lead to no working design at all. In this case the most limiting resources of DSPs, BRAMs and URAMs are overestimated by only up to 7.3% for these two largest design points and as a result the number of resources wasted is very limited. As a result, the area predictions are sufficiently accurate and enable fast DSE.

5.4.2 Performance Results

The results of processing 100 million externally generated electrons are shown in tab. 5.5. The particles are generated as a single beam, where all electrons enter the voxel cube at the same point with an energy of 6MeV. The offset column indicates whether this voxel is within a subdomain or at the centre of the cube. No offset means that the beam is pointed at the centre of the patient cube. In this case, the cube is entered at the intersection of four subdomains, significantly increasing the number of electrons needing DDR buffering.

FPGA and total runtime are shown separately. The total runtime includes the time required to finish simulation for all electrons sent back to the CPU as well as the time required to combine all partial results into one single patient cube. In the case of the Alveo U250, no particles are sent back to the CPU due to limitations of the PCIe controller and as such only the combination of partial results has to be carried out. To measure the runtime five independent measurements are made and the mean over these five runs is reported. The min and max values as well as the standard deviation are provided for the FPGA runtime. Additionally, the error of the prediction for the compute time compared to both the mean time used only by the FPGA and the whole application is reported.

The FPGA runtime and the time required to perform memory transfers is predicted using the equations in section 5.3. For each combination of offset and build a smaller run on the CPU is simulated to derive the factors determined by random number generators like the number

Table 5.5: Actual and Predicted Runtime.

Run Num	Build Num	Cards	Offset	Mean FPGA Time [ms]	Max FPGA Time [ms]	Min FPGA Time [ms]	Stdev FPGA Time [ms]	Mean Total Time [ms]	Predicted Compute Time [ms]	Predicted DDR Time [ms]	Error Predicted Compute Time to FPGA Time	Error Predicted Compute Time to Total Time
1	1	1	yes	2,869.80	2,881	2,877	1.9	2,968.60	2,667	110	7.60%	11.31%
2	1	2	yes	1,452.60	1,454	1,452	0.9	1,546.60	1,333	55	8.97%	16.02%
3	1	3	yes	980.4	982	978	1.67	1,088	889	37	10.28%	22.38%
4	2	1	yes	2,294.20	2,299	2,291	2.9	2,443	1,901	69	20.68%	28.51%
5	2	2	yes	1,167.80	1,173	1,165	3.1	1,325.20	950	34	22.93%	39.49%
6	2	3	yes	813.2	815	819	2.2	967.8	634	23	28.26%	52.65%
7	3	1	yes	3,956.20	3,962	3,951	4.7	4,572.20	3,333	378	18.70%	37.18%
8	3	2	yes	2,151.20	2,182	2,130	20.2	2,832.40	1,667	189	29.05%	69.91%
9	3	3	yes	1,652.80	1,739	1,597	55.8	2,358	1,111	126	48.77%	112.24%
10	4	1	yes	3,128.20	3,138	3,110	12	4,264.60	2,427	351	28.89%	75.71%
11	4	2	yes	1,702.80	1,738	1,689	20	2,814.40	1,214	176	40.26%	131.83%
12	4	3	yes	1,460.40	1,530	1,396	57.6	2,610.60	810	117	80.30%	222.30%
13	1	1	no	5,431.20	5,575	5,300	103.7	7,075	2,800	1.364	93.97%	152.68%
14	1	2	no	3,539.80	3,609	3,478	60.2	4,585.80	1,400	682	152.84%	227.56%
15	1	3	no	3,134.80	3,552	2,888	252.9	3,575.60	933	455	235.99%	283.24%
16	2	1	no	5,774.00	6,921	4,858	1,119.90	6,386.00	2,011	1.404	187.12%	217.55%
17	2	2	no	3,136.20	3,471	2,635	378.7	3,522.80	1,005	702	212.06%	250.53%
18	2	3	no	2,655	2,807	2,488	128.7	3,079.80	670	468	296.27%	359.67%
19	5	1	yes	2,171.40	2,173	2,167	2.5	2,182.40	2,000	83	8.57%	9.12%
20	6	1	yes	1,735.80	1,751	1,730	9	1,742.40	1,425	52	21.81%	22.27%
22	7	1	yes	3,113.20	3,169	3,084	34.6	3,179.40	3,333	283	-6.59%	-4.61%
23	8	1	yes	2,989.20	3,163	2,878	107.8	3,216.40	1,821	264	64.15%	76.63%
25	5	1	no	1,986.40	2,008	1,963	16	2,010.20	2,100	1.023	-5.41%	-4.28%
26	6	1	no	1,869.40	1,818	1,259	242,78	1,702.00	1,508	1.053	23.97%	12.86%

of iterations for each initial electron and the rate of electrons which require buffering in DDR (Step b).

The target of sub one second runtime for the complete simulation including CPU execution is reached for run 6. This run uses three MAX5C cards simulating a patient cube of 128 voxels in all directions and processing two pixels in parallel in each of the three design instances. It stands to reason that the same target could be reached by two Alveo U250 cards using build 6. However, I did not have access to the hardware required to verify this.

Since the execution time is dependent on the random numbers generated, one can expect a certain amount of deviation between the predicted runtime and the actual runtime. In most cases the standard deviation between executions is low and stays below 10 ms and in some cases even below 1 ms. However, e.g., in the case of run 16, the standard deviation increases significantly and, in this case, reaches more than 1,000 ms.

In general, there are three common patterns for runs with increased standard deviation of

execution times. The first is that multiple electrons are processed in parallel. This can, for example, be observed by comparing runs 1 and 4 or 19 and 20. In these cases the deviation increases slightly for builds which have the same parameters apart from this additional degree of parallelism. The reason for this is that a random chance of conflicts for patient cube memory accesses is introduced. The resolution of these conflicts results in changes to the degree of parallelism possible and as a result impacts the execution time. This deviation can be expected even though the impact is small.

The second factor is the size of the patient cube. If the patient cube is large as in the cases of builds 3, 4, 7 and 8 the discrepancy between runs is also larger. This has two reasons: Firstly, each voxel represents a smaller portion of the actual patient, since the resolution is increased. As a result, particles can travel across more voxels leading to an increase in subdomain changes. Secondly, more data have to be sent back to the CPU leading to stalls if the PCIe bus is currently congested. This congestion can be caused by multiple designs flushing their patient cube at the same time. Overall, the introduced deviation is still limited.

The third factor is the position of entry of the particle beam into the patient cube. If the beam enters in the centre of the cube, which is also the boundary of four subdomains, the standard deviation of execution times increases significantly. By comparing the deviation that occurs in runs 13 to 18 with runs 25 and 26, one can see that only for the first two cases the deviation is high while for the others it is not. The only major difference between both designs, apart from the additional design copy, is that in the case of the U250 based design particles are not sent back to the CPU if they cannot be processed on the card itself. Indeed, closer examination of the run shows that a significant portion of the particles is sent back to the CPU. Since this data transfer is not based on DMA but on active polling, it seems like the CPU execution and the order in which threads get access to the data mostly cause this high deviation.

The prediction accuracy also needs to be evaluated. For runs 1 to 6 the prediction is accurate even though it is a bit too optimistic. In general, the prediction is roughly 100 to 200 ms faster than the real system is. The reason for this prediction error is twofold: Firstly, the system initialisation time is ignored, which can be in the order of hundreds of milliseconds for Maxeler

systems. Secondly, there is a high likelihood that multiple designs are sending their patient cube data back to the CPU at the same time leading to an additional small delay.

Fig. 5.5 shows the activity of the compute Kernel as well as of the output, which is used to send the patient cube back to the CPU for run 7. For these runs (7-12) the difference between predicted and actual runtime starts to grow but is still limited to an error of 80%. There are two interesting observations that can be made from it. First it takes roughly 200 ms until the Kernel becomes active. This verifies that the unexpectedly high initialisation overhead is partly causing the discrepancy between actual and predicted runtime. The reason for the higher-than-expected initialisation time can potentially be traced back to the seed initialisation of the random number generators. It is possible to remove this overhead by sharing the seed between multiple initialisations and not resetting the random number generator in between runs to not generate the same numbers on every run.

The second observation is that the Kernel execution sometimes has to pause for patient cube data to be sent over PCIe back to the CPU. This is caused by subdomains for which no or only very small numbers of calculations have to be performed. It seems like particles do not travel through the complete patient cube but are mostly absorbed close to the impact point. In the performance model it is assumed that it is possible to overlap data transfer and calculation, which is only possible if for each subdomain a sufficiently high number of calculations is performed. The missing overlap between data transfer and compute adds an additional overhead in the order of a few hundred milliseconds.

To mitigate this effect one could, for example, skip subdomains which have nothing to compute or increase the PCIe bandwidth by switching to PCIe Gen. 3, for which driver support is in development. Additionally, it is possible that the addition of Bremsstrahlung will change the distribution of particles across subdomains removing this problem altogether.

The error between the predicted compute time and the runtime of the complete application is also a lot bigger than the error between the prediction and the FPGA runtime. The error for the complete application reaches more than 200% for run 12. This is probably a combination of the need to accumulate significantly more data on the CPU and a need to process more

particles on the CPU as well. The CPU code is not very optimised and cache optimisations to help with the accumulation of these larger patient cubes might mitigate this issue.

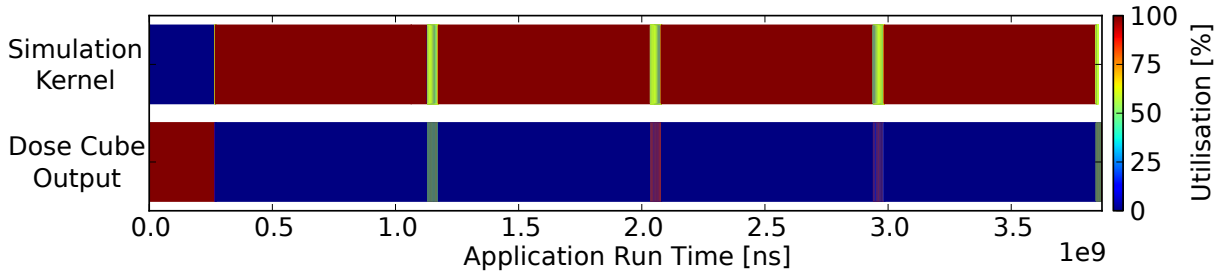

Figure 5.5: Visualisation of the Kernel and PCIe activity for run 7.

For runs 13 to 18 the difference between the actual and predicted runtime is the biggest and reaches up to 300% if only the FPGA execution is considered and 360% for the complete application. Again, by comparing to runs 25 and 26 one can conclude that this is caused by the transmission of particles back to the CPU. While it might be possible that this problem could also be partly mitigated by higher PCIe bandwidth, it seems more promising to perform more batches without the generation of new particles on the FPGA. Additionally, it should be possible to further optimise the CPU code and to avoid this worst case in the actual deployment of the system altogether by shifting the position of the subdomains accordingly.

For the designs using the U250 the same patterns can be seen as for the builds using the MAX5C card as discussed in detail above. To summarise, the model seems to be highly accurate for the most straightforward cases as represented by runs 1 to 6 as well as 19 and 20 but has some significant errors for more complicated cases. If the cube size is increased or the particle beam enters the patient cube at the boundary of subdomains effects that were not included in the model have significant impact on the runtime leading to bigger prediction errors.

There seem to be three major factors leading to these large errors which need to be discussed in more detail in relation to the performance model of the proposed methodology:

1. **Initialisation:** One of the discrepancies between the predicted and actual runtime was shown to be related to the long initialisation time. Following eq. 3.1 this initialisation time is included in the performance model in general. In the case of this application, I estimated the initialisation time to not be of any significance which was an error. The

avoidance of this simplification and more initial testing using a micro benchmark with the random number generators would have shown that the initialisation time cannot be ignored and as a result a more rigours application of the methodology would have led to more accurate results.

2. **PCIe bus congestion:** A second large problem is the congestion on the PCIe bus when multiple subdomains are written back to the host at the same time. Again, this case is handled in the methodology, since it states one should consider data transmissions with a burst behaviour. The error here was to not sufficiently consider that multiple designs might write their buffer back at the same time. I estimated that this would be very unlikely and as such did not model it in detail. However, it would be possible to model it accurately which would lead to a change of the architecture to avoid the issue altogether. This can be achieved by increasing the PCIe bandwidth or including an arbitration to delay selected designs more so that not all designs transmit their data at the same time. For example, the usage of a round-robin arbitration would reduce the likelihood that designs conflict again and ensure that at least one design can start its calculations again as fast as possible.

3. **CPU inefficiencies:** The third and last large problem seems to be related to an inefficient aggregation of subdomains on the CPU and the need to wait for particles to be actively pooled by the CPU. The aggregation of subdomains is less of a problem for the evaluation, since it does not delay the FPGA implementation. However, the impact of the CPU execution order on the FPGA runtime through active pooling is more complicated and currently not addressed within the methodology. Further research would be needed to investigate how these kinds of interactions could be modelled accurately. It should be mentioned that data transfer between host and FPGA not using DMA is fairly uncommon in the context of HPC, so this lack of predictability does not impede the application of the methodology in many HPC use cases.

This Monte Carlo based simulation was selected to stress test the behaviour of the methodology for more dynamic execution patterns. Furthermore, some of the runs used to evaluate the

application further exacerbated this dynamic behaviour. For example, it should be possible to avoid the need to direct the particle beam at the subdomain boundary in all real use cases.

The methodology managed to deal well with the more basic forms of this dynamic execution patterns, however struggled with the more extreme cases. There are two major lessons which can be learned from this. First, for highly dynamic execution patterns the modelling has to be performed with a high level of detail. Two common simplifications that I decided to make caused large errors to the overall runtime later on. As such, especially to predict the execution time more attention to detail is necessary for these classes of applications in the future. Second, it was discovered that close interaction between CPU and FPGA is not modelled accurately enough in the current performance model. While rare for HPC applications further research is needed to further expand the applicability of the methodology.

5.4.3 Comparison to Traditional Systems

The comparison to related work for this application is not easy, since the precise test case is often not reproducible. In [168] the authors report execution times of 10.8 seconds for a patient cube of size 256x256x234 on a two socket Intel Xeon system. Additionally, they report a speed-up of 1.95x compared to the GPU implementation presented in [55], which is based on a single GPU card. A similar test case on the proposed system (Run 12) takes 2.6 seconds including the not fully optimised CPU code. Since the test case presented in this chapter is slightly larger the comparison should be slightly biased towards the CPU and GPU based technologies. On the other side the other two systems might not use the same simplifications as the implementation proposed here. A speed-up of 4.1x compared to the CPU and 8x compared to the GPU implementation was achieved. It is not expected that this speedup will be completely invalidated once the simplifications in the implementation are removed.

The FPGA as well as the CPU and GPU systems can be realised as a single workstation system. For this application the space efficiency of the solution is the most important metric, since the simulation system should be placed close to the treatment machine to reduce latencies and minimise potential network issues. As such the simulation system has to fit into the existing

rooms available for the treatment which are often in dedicated underground buildings due to radiation shielding requirements. The price of the simulation system is negligible compared to the costs of the complete installation.

5.5 Discussion

This chapter shows how the proposed methodology can be applied to a highly complex, real application in the field of radiotherapy and how it provides meaningful insight which aids in the creation of a successful design. Additionally, the methodology was applied to VGG-16 and BQCD in section 3.6.

In this section I discuss how the individual steps of the methodology were applied to the different applications and then draw some conclusions on its application to future designs. Tab. 5.6 shows an overview of the individual parts and steps of the methodology and how they were applied to the different applications.

One of the major lessons of applying the methodology is that all stages are tightly coupled and cannot be viewed in isolation. Instead, they have to be constantly co-developed. For example, a bottleneck discovered in the performance model determines the focus of the analysis. In most cases it is possible to discover early on which property of the algorithm is most problematic and therefore has to be analysed in the most detailed fashion.

On the other side it is possible to save time on parts which have been shown to be less relevant or problematic. For example, if one knows that in the worst case a few megabytes have to be transferred between CPU and FPGA over a longer time span, a detailed analysis of the communication pattern is not required. For the same reason a completely accurate performance model is often not important, since it is possible to quickly make a worst-case calculation, which includes large safety margins. The designer can then focus on modelling the resource usage for only the scarcest resources in greater detail with high accuracy. However, the radiotherapy application also showed that some simplifications can cause significant prediction errors if applied incorrectly. Especially the execution time in dynamic execution environments has to be

Table 5.6: Application of the individual parts of the methodology to the applications.

Step / Part	CNN	BQCD	Radio Therapy
(1) Analysis	Due to the highly regular structure of CNNs and the simplicity of the underlying equations, it was possible to retrieve most relevant information from these equations.	BQCD provides integrated analysis tools which has been used for the initial software split. Further analysis was performed manually on the parts selected for porting.	Elaborate profiling was required to deal with randomness.
(2) Software Model	C++ mainly for debugging purposes later on. Based on caffe.	Reimplementation of FORTRAN code in simple C	Medical research partners provided simplified C++ program. This version was used to build the simplified model.
(3) Forecast System Properties	Main focus on bandwidths, DSP usage and on-chip memory usage	Main focus on bandwidth and on-chip memory	Main focus on bandwidths vs. on-chip memory usage
(4) Architecture Optimisation	Deal with multi-die, use all DSPs make sure that all DSPs can always use data	Minimise on-board memory bandwidth usage	Optimise on-board memory access patterns
(a) Create hardware / software partitioning	Loopflow graph after analysis indicates that split between fully connected and convolutional layers is best	CG provides major contribution to runtime and has to be accelerated	Memory access makes sharing between CPU and FPGA difficult. As a result only final accumulation can be performed on CPU to simplify FPGA design.
(b) measure profiling results	Used caffe profiler and CNN equations	Integrated BQCD profiling	Focus on ratios around random number generators
(c) refine model	Focus on data sizes between layers and operation counts	Detailed measurement of data sizes and communication patterns	Measure random behaviour for different configurations
(d) define what to model <-> (e) determine bottlenecks	Model for fully streaming indicated that architecture is not feasible. As a result change to PE based architecture. Optimise architecture to use arithmetic capabilities to the fullest. Major bottleneck on-board memory bandwidth.	Early on on-board memory bandwidth discovered as major bottleneck. Trade off between on-chip memory usage vs. on-board memory bandwidth. Indicated that card initialisation time has major impact on runtime ->move CG iteration control logic onto FPGA as well.	Indicated that full patient cube could not be stored on-chip. Showed that random access into patient cube not feasible. As a result architecture focused on streamlining on-board memory access by domain decomposition and enabling parallelisation.
(f) verify architecture	Architecture changes not significant enough to require changes to software model.	Tested domain decomposition and architecture changes in software model.	Architecture changes not significant enough to require changes to software model.
(g) start implementation	Final version had no bottlenecks (Maximised bandwidth and DSP usage). Software model was used heavily for debugging and verification of the hardware implementation.	Found a good trade-off between on-chip memory and on-board memory bandwidth usage. Software model was used heavily for debugging and verification of the hardware implementation.	Found architecture which enables parallelisation while limiting on-board memory bandwidth usage. Software model was used for debugging and verification of the hardware implementation.

modelled with great accuracy.

One example of this is the CNN acceleration as described in section 3.6.1 where the focus was fully on multipliers, on-chip memory capacity and bandwidths. In the creation of the radiotherapy simulation application described above the focus was more on accurate prediction of all resources, but even in this case the error for logic is rather high since worst-case assumptions were used and no detailed modelling was performed. Since it was possible to decide early on that logic usage would not be a bottleneck this focus on what is important saves time and helps to concentrate on the properties which could actually be design obstacles.

Additionally, the obtainable accuracy for different hardware resources is different. For example, the usage of DSPs can be usually calculated without any error with low effort, but the usage of memory resources is harder to accurately model due to automated scheduling and tool decisions. However, in order to aid timing closure designs should usually not use more than 80% of each resource. This means that for all resources which are not going to introduce bottlenecks rough calculations and worst case assumptions are sufficient and only for the scarcest resources more attention to detail is required.

In many cases it is also possible to significantly accelerate the analysis and software model steps based on the original application. BQCD contains integrated profiling and analysis code which was successfully used. For the CNN acceleration it was possible to use existing machine learning libraries, in this specific case caffe [56].

In the case of VGG-16 and the Monte Carlo based dose simulation it was possible to verify the changes made to the algorithm without moving them to the software model. This was possible since only the execution order was changed, and it was easy to show that these changes were legal and have no impact on the result from the equations implemented.

This means that in the case of VGG-16 the software model was not strictly required for either the analysis or the verification of the architecture. On top of this, it is well known that neural networks can utilise fixed-point precision, which even removed the need to perform a fixed-point analysis in the software model. However, skipping the software model altogether is

still not a good idea, since it was heavily utilised for debugging and validation of the hardware implementation. To fulfil this target, it was sufficient to create a heavily modified and simplified version of caffe which can be run as a drop-in replacement for the FPGA design enabling detailed comparisons between results.

In all cases the software model was absolutely crucial in the final stages of debugging. While it might seem tempting to skip this step and save development time and effort as a result, this experience shows that it is not beneficial to leave out the software model altogether. Instead, it might be possible to reuse existing code bases and change the time at which the software model is created. However, depending on the application to implement it might be absolutely crucial to use the software model to verify the architecture and assist the code analysis as it was the case for BQCD.

In all applications provided in this thesis the methodology helped to quickly discover bottlenecks and design an architecture which achieved the design goals. This enabled us to only implement the hardware design once and avoid time consuming and risky design iterations on the hardware design. Measuring the advantages of one development process compared to another in great detail is always a difficult undertaking, since developer experience has a great impact. Additionally, the same developer cannot implement the same application using different methodologies, since lessons learned in earlier designs will automatically influence design iterations in subsequent implementations. As such, I cannot measure the time savings using the proposed methodology but need to argue based on my experience. Some of the projects discussed above required significant development effort and, e.g., BQCD was developed by Maxeler over multiple years. Discovering major issues late can result in a complete redesign causing unmanageable financial and operational risks. If one manages to directly implement a hardware design which is not bottlenecked without using the methodology, there is a time saving which can be estimated to roughly 10%. But this also means that if any design iteration in hardware which changes more than 10% of the design is required the methodology will have reduced the overall workload compared to a direct implementation. In all applications presented here the final results were obtained after tens of iterations between the performance model and the architecture. While early iterations were usually impactful, later ones only op-

timised architectural details. As such, the ability to perform design iterations in days instead of weeks, months or even years greatly reduced the risks associated with hardware design.

5.5.1 Thoughts on Automation

Full automation is not aimed at by the methodology proposed in this thesis, it is overall human-centric. This is done intentionally and is not the result of limited time or resources. The main motivation for automation of certain parts of the development methodology would be to shorten design times and to reduce the knowledge and experience required from the engineers implementing the design. While these are both definitely worthwhile targets, I believe they do not fit well into the main goal of the proposed methodology. The methodology aims at complex applications targeting highly demanding industrial and scientific environments. This requires transformation of the unique application properties within tight performance targets. Automated exploration based on heuristics, machine learning and automatically created analytical models in combination with a limited set of possible architectures and optimisations will typically not manage to find the best performing implementation in such a highly complex design space. In many cases certain performance needs to be reached within tight space, time, power and cost limits. For example, weather forecasting for the next day has to be ready, before the start of the next day to be of any practical use. For such systems the overall costs and all additional performance benefits resulting in slightly better results or reduced running costs justify the additional investment into the application engineering. Additionally, knowledge and experience shortcomings can be partly mitigated by better technical documentation as well as relevant examples and libraries.

Current automated solutions often fall short of achieving high performance targets comparable to handcrafted code and at that point the designers often have limited options for improvement. For example, the fully automated development flow presented in [70] implemented single precision General Matrix Multiply (GEMM) on a Maxeler MAX4C DFE and achieved a speedup of 0.1x over a six-core CPU. In comparison, Maxeler demonstrated more than 20x speedup for double precision GEMM targeting the same FPGA hardware [58]. In Maxeler's case only one

core of a roughly 10% slower CPU is used, however double precision arithmetic also uses significantly more space on the FPGA. This nearly two orders of magnitude performance difference shows the potential of the manual approach proposed in this thesis when compared to a fully automated solution even on such a simple use case.

Many problems in the design of complex FPGA applications require human insights and as such the main goal is to expose the most relevant information to the designers performing the task and to facilitate them in making the best decisions. As such the methodology aims at providing relevant insights and does not attempt to reduce the amount of work.

It can be argued that full automation of hardware design will never be able to match the results achievable through human intervention. There are two main arguments in this direction that I want to present here:

1. An automated tool can only support a finite set of optimisation patterns. This prevents the employment of novel custom optimisations tailored to a specific problem. While the requirement for these novel optimisations might be rare in general one can expect that especially for highly complex applications novel approaches to communication and related problems are required;

2. Even for CPUs manual performance optimisations are still required despite the fact that the target architecture is well known and fixed. For example, manual reordering of `for` loops can result in significant performance differences. Similarly, parallelisation of CPU applications is also still a highly manual process especially once multiple nodes are used. These tools have received major funding and research interest over decades, but still have significant limitations. The freedoms inherent to hardware design will only exacerbate these problems.

The above is no formal proof that fully automated hardware design systems are impossible to construct or that HLS systems will always provide inferior performance. However, it stands to assume, that there will always be specific cases in which automated solutions fall short behind an architecture crafted by a human designer. This is especially true for the close future in which

no major jumps in tool quality or computational capabilities, e.g., by quantum computers, are foreseeable.

One has to consider that FPGA based accelerators have strong competition in the form of modern CPUs and GPUs. It is not expected that the programmability of FPGAs can ever become better than these competing technologies for two simple reasons. More precisely, the time required to perform place and route and the limitation to only physical and not virtual resources, which results in many possible designs which simply do not compile due to the limited available area. However, when an automated tool trades too much performance for improved programmability there is simply no incentive to use an FPGA over for example a GPU. As such every programming tool or methodology has to first ensure that the performance advantage of FPGAs is maintained to motivate their usage in the first place.

This however means in no way that automation and tool assistance have no practical use. Quite the contrary, the need for better FPGA programming languages is definitely present and HLS systems have their real use cases, for example, for non-performance critical parts of applications or for applications which for other reasons, e.g., transceiver capabilities, have no alternatives but using FPGAs.

There are many examples for fully automated solutions for certain application domains. Especially in recent years many tools were developed which map machine learning problems onto FPGAs with good performance. However, these domain specific tools are limited to narrow classes of applications, which will always use very specific architectures and optimisations.

Additionally, different parts of the methodology as presented in this thesis can benefit from further automation and tool support. The following list provides some examples for automation and tool support opportunities:

- As discussed in section 3.4.4, the way in which on-board DDR memory is accessed is crucial to maximise the utilisation of the available memory bandwidth. However, writing address generators in hardware description languages or MaxJ is often difficult. Since address generators can run slow in comparison to the remaining design and only occupy

a very small amount of area, they are a good target for synthesis from a higher level, e.g., from C or Java `for` loops. This could also help with detecting errors in the address generation, which are often hard to track down.

- In a similar fashion it might be possible to at least partially synthesize data paths from an available HLS description. The major performance critical blocks which are hard to generate for high level tools are buffers for data reordering and efficient communication schemes between components. If the input and output of a block are well defined an HLS tool could generate the computational datapath in between including loop unrolling based on a parameter defining the number of iterations to unroll.

- The main motivation for a manual creation of a performance model is mainly the insight provided by it. It should be possible to develop a tool which takes a model of the FPGA platform as well as the planned implementation of the application as inputs and automatically creates a performance model to then perform design space exploration using this automatically created analytical model. However, it is crucial to fully expose the performance model and trade-offs as well as discovered bottlenecks to the user to enable changes to the architecture.

- Moving from floating-point to fixed-point arithmetic is often done manually and is time consuming and error prone. Deep integration between a fixed-point simulator, which can be merged into the software model and the FPGA toolchain could help with accelerating this process significantly. For example, a user might automatically explore accuracy based on multiple golden input and output datasets and the chosen types for different parts of the algorithm could be propagated into the FPGA design. It is again crucial that the user maintains full control over precision vs. area trade-offs and knows the used datatypes.

- In the context of multi-die FPGA automated pre-floor planning might be helpful to reduce inter-die crossings and optimally use the resources present on a complete device. As discussed in section 4.2, MaxCompiler designs contain multiple design units, e.g., Kernels and ManagerStateMachines. It is possible to predict the area usage for these components at compile time. Since the resources available on the different SLRs is also

known it should be possible to map the individual design units to specific SLRs. Since the number of design units and possible mappings is limited the complexity compared to general floor planning should be highly reduced. Such a floor plan could drastically reduce the number of SLR crossings, ensure that no single SLR is overused and as a result improve timing characteristics. Additionally, it should improve the efficiency of the memory mapping algorithm described in section 4.2.

For all the opportunities mentioned above it is crucial to integrate these new tools deeply into the proposed methodology and provide full control and insight to the designer in order to enable well informed decision making. There are many further opportunities for automation and for improving FPGA design productivity, however, based on the experience I have built during my research study the tools mentioned above are the most important areas of improvement right now. In general, I believe that the significant amount of research spent on HLS tools might be better focussed on topics like the mentioned above to advance the adoption of FPGAs by improving design productivity while maintaining the performance advantage over competing technologies. In order to reach widespread FPGA adoption, it will be necessary to spend significant resources similar to the investments made by Nvidia to develop CUDA. Due to the flexible architecture of FPGAs one can expect that this effort will be even greater. It will be crucial to set realistic targets for this development and not to try to automatically solve every problem, which brings the risk of losing the performance advantage of FPGAs while still not achieving higher designer productivity compared to CPUs or GPUs. Instead, custom solution for specific application domains and tools to assist manual design should deliver the best results.

5.5.2 Comparison to other Methodologies

To conclude the discussion and evaluation of the methodology a comparison to three other approaches is presented. The selection of those approaches is based on their level of abstraction with one representing classical HDL design, another the usage of current vendor HLS tools and the third a fully automated approach.

HDL based design methodologies

In general, the methodology presented in this thesis is fully compatible with HDLs. In fact, it stands to assume that many parts of the methodology as presented in this thesis are already widely applied to HDL designs in the industrial context but were never formalised or published. However, especially the extensions made in chapter 4 to assist portability will not map well to RTL based hardware design due to the missing abstractions from tool versions and devices on the language level.

In [111] a VHDL based design methodology is presented. It does not mention how an architecture for a hardware design can be developed but assumes that the developer works towards a known architecture. Instead, it fully focuses on the problems arising from the usage of VHDL and how specification, documentation and simulators can facilitate the design process. This shows the general problem with HDL languages, where a programmer can easily be lost in the many details and as a result the focus on the general system architecture or even larger algorithmic optimisations is lost. In contrast, this thesis used MaxJ to boost designer productivity and enable a focus on the system level optimisations as well as quickly enable large changes to the implementation in general. Additionally, it enables device independent programming and fast numeric changes.

If no methodology comparable to the proposed one is used to develop a promising system architecture there is a real danger to end up in a prohibitively lengthy iterative design process of incremental improvement, which might require multiple fundamental redesigns. In the context of RTL this is especially problematic, since even small changes might require a change to the pipelining and a manual rebalancing of the complete computational pipeline. This may result in unacceptable design times.

A lot of these problems can be easily mitigated or ignored in the case of designs with trivial sizes, consisting of only a few functional units. Alternatively, huge design teams can be employed where detailed specifications as described in [111] are used and a design is split into many independent blocks. Again, in those cases the first system architecture has to be good enough,

otherwise redesigns of these blocks throughout the design process might be necessary.

HLS based design methodologies

The main idea behind most HLS solutions is to provide a development flow closer to traditional software design. This also influences the development methodologies used. They usually try to emulate the classical software approach where an initial implementation is made and then repeatedly profiled and improved to maximise its performance. To assist in this process the tools provide detailed reports with optimisation advice and profiling results for hardware runs. An example of this is the Intel OpenCL compiler, to which development methodologies are described in [5] and [140].

As explained before, these iterative designs face significant challenges in the context of hardware design. Most notable it might be necessary to fundamentally restructure the implementation to mitigate issues discovered late in the design process. Additionally, hour or even day long place and route times significantly delay the development process. On top of this it is easily possible to get stuck in local minima of the design space and hence never reach the best design.

As such the application of a process as described above might lead to an initial hardware implementation faster than the proposed methodology (assuming that the initial design fits on the target device). However, in most cases it will take significantly longer or might even be impossible to arrive at the same performance point that will be reached if the methodology in this thesis is followed.

While in theory the proposed methodology can be also applied to existing HLS tools, they might be missing the required fine-grained control allowing the mapping of the envisioned architecture to the targeted hardware.

Fully automated design methodologies

An example of a fully automated design methodology is provided in [70]. The main target of fully automated approaches is to provide a single push button solution from an algorithm

description to a complete hardware implementation, including mapping to different devices, architecture selection and design space exploration. Obviously, it is difficult to beat such a solution in terms of designer productivity. However, as discussed in more detail in section 5.5.1 current solutions still deliver designs with inferior performance as compared to more human driven approaches. Additionally, it stands to assume that even in the future such fully automated approaches will often not come close to the performance of applications based on human decision making and original thinking.

5.6 Summary

In this chapter, an FPGA based implementation for real time Monte Carlo dose simulation for application in adaptive radiotherapy was presented. An architecture which decomposes the voxel cube representing the patient into multiple sub cubes to reduce on-chip memory space requirements was proposed. The performance and area usage for this architecture were modelled using a simple analytical model to predict the hardware implementation characteristics. Finally, eight implementations of the architecture were created. The real time goal of simulating 100 million electrons in less than a second using three FPGA cards was fulfilled for a representative voxel cube with a size of 128 in all dimensions.

This chapter showed how the techniques developed in chapter 4 can be used for an application which is newly developed. It was possible to predict the memory usage accurately enough for DSE even though the mapping to URAMs and BRAMs is handled automatically. Additionally, multiple platforms were targeted using a single code base. Finally, a simple performance model was used to make predictions for the overall runtime. For this application with a highly dynamic execution pattern based on random number generators it managed to make area predictions accurate enough for DSE as well as accurate performance prediction for simpler cases while the error increased significantly for the most complex execution patterns. The latter could be traced back to oversimplifications in the design phase and a missing accurate model for FPGA to CPU communication which is not DMA based. The performance model enabled the

identification of the most promising architecture and a quick DSE.

Finally, this chapter discussed the lessons learned from applying the proposed methodology to multiple real-life problems as described in this chapter as well as section 3.6. This led into a discussion on computer aided automation, which argued that the human driven approach of the proposed methodology is able to deliver better performance then fully automatic solutions. Opportunities for automation supporting the methodology and additionally improving designer productivity were presented. Finally, a comparison to HDL and HLS driven development processes as well as fully automated tools was made, showing that the proposed methodology provides a good trade-off between achievable performance and designer productivity.

In summary, the methodology developed in chapters 3 and 4 enabled the rapid design of an FPGA based implementation of the real time Monte Carlo dose simulation application. It was possible to outperform previous work through novel changes to the architecture. These improvements were motivated by concrete bottlenecks highlighted by the performance model. Even though the application is complex and performance prediction is especially difficult due to the random number generation, it was possible to provide an implementation which achieved predicted speeds on the first design iteration for the most common configurations.

Chapter 6

Conclusion

In this thesis three novel contributions towards constructing a methodology for the development of complex applications for reconfigurable systems have been made. This chapter will recap the key research questions as well as the contributions made to answer these questions. Possible future work directions are described, to address current shortcomings of the methodology and to resolve additional challenges. Finally, a personal outlook on the application of FPGAs to the field of HPC is provided.

6.1 Summary of Achievements

In the context of ever-growing computational demands and the end of Moore's law, computer architects are in need of novel approaches to build the next generation of HPC systems. Heterogeneous computing has been identified as a promising approach. This means that the classic CPU based systems are extended with additional computing devices like ASICs, GPUs and FPGAs. Especially FPGAs provide a good fit to emerging problems, since they combine the efficiency of custom hardware circuits with reprogrammability. Many research studies have shown that FPGAs can outperform GPUs in terms of computational performance and energy efficiency, while at the same time being more flexible and cheaper to design compared to ASICs. However, adoption of FPGAs is still quite limited which is often attributed to the non-trivial

challenges inherent to the development process. This thesis aimed at alleviating some of these problems by proposing and refining a development methodology for modern FPGAs.

This research was driven by the following research questions:

Q1. Does a structured approach for accelerating HPC applications with reconfigurable platforms exist?

Q2. Is there an accurate method able to predict reconfigurable systems' performance prior to the creation of a synthesisable implementation of the reconfigurable sub-system?

Q3. Are there techniques to achieve state-of-the-art performance for a given application and reconfigurable platform?

Q4. If such techniques exist, can a single implementation target different reconfigurable platforms while delivering maximum performance on each?

Q5. What are the issues when the techniques mentioned above are applied to multi-die devices?

In order to develop a structured approach for the acceleration of HPC applications using reconfigurable platforms an initial methodology was presented in chapter 3 (Q1). It consists of the following four main steps. First, the existing application is carefully analysed. Second, the portion of the code that is intended for hardware acceleration is reimplemented in a simple software model. Third, using the knowledge gained the performance of the final system is modelled using simple equations. Fourth, the architecture is then iteratively refined until all system bottlenecks are removed. This enables the implementation on the FPGA avoiding time consuming design iterations in hardware. Using the methodology two applications were developed. The first was a CNN implemented by the author of this thesis and the second a quantum chromodynamics simulation developed by Maxeler engineers. The presented performance model based on analytical equations was shown to accurately predict the performance of the reconfigurable system before a synthesisiable implementation of the reconfigurable system existed (Q2). Both applications are based on a static execution pattern and the ability of the

performance model to predict performance accurately for more dynamic execution patterns was discussed later in the thesis. In both cases the initial implementations fulfilled the requirements of the system and achieved state-of-the-art performance (Q3).

To enable portability and performance scalability between different, including modern, FPGA devices the methodology had to be extended and additional tool support had to be provided. Chapter 4 successfully tackled this challenge. First, a novel memory mapping algorithm was developed. This algorithm addresses the problem of heterogeneous memory resources in the context of multi-die FPGAs. A greedy algorithm tries to balance the allocation of different memory resources to allocate them at the same rate. The resulting algorithm managed to successfully place and route all designs, while the second-best performing algorithm failed on one third of the large applications. Second, the methodology was extended by best practices for design portability. To verify these best practices an existing financial pricing application was ported to four additional different FPGA devices and achieved a performance improvement of up to 7.4x compared to the initial implementation on these larger devices. The speedup normalised to the available FPGA capabilities showed that the new devices were used efficiently. This shows that using the proposed methodology the same implementation, using a single code base, was able to target different reconfigurable platforms efficiently (Q4). Furthermore, this chapter identified SLRs as one of the major challenges to the porting effort especially if code is moved between different scales of the same device generation. As such the above memory mapping algorithm was specifically designed to address this challenge and best practices for the usage of SLRs in the context of portability were discussed (Q5).

To verify the extended methodology, it was applied to a medical application in chapter 5. This application was a Monte-Carlo based simulation of the dose accumulation in human tissue during radiotherapy treatment, which, according to medical experts, has to run in less than one second to use it in real time, improving treatment quality and reducing costs. It is of special interest due to its dynamic execution pattern, making the prediction of performance more difficult (Q2). The methodology was applied to implement the simulation without multiple design iterations in hardware. The performance model was shown to predict the allocation of different memory resources, as performed by the modified memory allocation algorithm, and

the overall area usage accurate enough to perform DSE. Additionally, it managed to accurately predict the execution time for the more common configurations of the application having a less dynamic execution pattern. For configurations with a highly dynamic execution pattern the error increased, and the lessons learned from this were discussed. Using the best practices for performance portability it was possible to target two different platforms with minimal code divergence. The resulting implementation managed to achieve the real time target using three MAX5C accelerator cards. Additionally, the lessons learned from the usage of the methodology on all applications presented in this thesis were discussed. A comparison of the proposed approach to other development paradigms, including highly automatic ones, was presented.

To summarise, the answers to my research questions are as follows:

Q1. Yes, I presented a structured process consisting of four individual steps in chapter 3.

Q2. For static execution patterns yes, as shown throughout this thesis, e.g., for the CNN and BQCD implementations in chapter 3 and for the Monte Carlo based dose accumulation simulation presented in chapter 5. For dynamic execution patterns further research is required.

Q3. Yes, as shown for the CNN application presented in chapter 3 which achieved state-of-the-art performance on the first design iteration.

Q4. Yes, as shown for an Asian option pricing application in chapter 4 which was efficiently ported to multiple different devices.

Q5. Memory mapping for multi-die FPGAs was identified as one of the major challenges. A possible solution was presented in chapter 4.

To conclude, fig. 6.1 highlights again the main contributions of this thesis as well as their connections as originally introduced in chapter 1. The development methodology introduced in chapter 3 and extended for multi-die FPGAs and portability in chapter 4 was applied to a real-world medical application in chapter 5 proving the usability of the developed methodology. It helps to understand the compute and communication patterns as well as numeric properties

of even highly complicated applications and assists in the development of a suitable architecture which addresses these properties. As such, it is helpful for application development in the HPC context. While no large-scale experiments were performed in this thesis it stands to assume that especially in the case of very large deployments, even going up to exascale, a methodology like the one proposed in this thesis will be crucial to minimise data movements at all system levels and maximise the number of operations which can be executed within a single chip and per memory access. Both are very critical to limit the system power usage to affordable levels.

Based on the initial experience gained by sharing the methodology with both experienced and novice designers of dataflow-based applications it proves its value in guiding developers quickly to promising architectures and eliminating unnecessary, time wasting design iterations. Maxeler decided to include the extended design methodology in their toolchain documentation and training workshops to educate internal and external users. As such I strongly believe that the proposed development methodology will facilitate the adoption of reconfigurable dataflow technology and help developers with designing highly efficient applications. Additionally, the memory mapping algorithm has been integrated into the commercial MaxCompiler toolchain and is currently used in dozens of ongoing academic and industrial projects. Finally, the radiotherapy simulation could improve cancer treatment significantly, by reducing costs and risk for the patients. While this thesis provides only a proof-of-concept further research based on these initial results has already started.

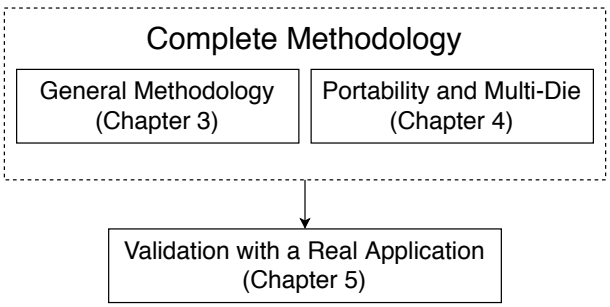

Figure 6.1: Thesis contributions.

6.2 Future Work

In this section current limitations of the contributions are highlighted together with possibilities to further extend on this work.

6.2.1 General Methodology

Chapter 3 introduces a development methodology for the acceleration of complex applications using FPGAs. While it has already proved its usefulness within the industrial and academic environment there exist many possibilities for additional improvement. These include improvements to the accuracy of the prediction of hardware usage as well as further automatisation. The following seven directions seem to promise the biggest improvement:

1. At the moment, accurate prediction of the resources used for the scheduling of the dataflow graph does not exist. Instead, a simple estimate based on experience and the expected size of the dataflow graph has to be made. Only for big, manually created delays an accurate estimation is achievable. Since the scheduling often has a non-trivial contribution to the overall memory usage a more accurate prediction would be very useful. The fundamental problem with such a model is, that one needs to have a very detailed description of the final dataflow graph before making any assumptions about the final graph scheduling. This requirement breaks the fundamental need to provide accurate predictions before the first line of code is written. It should be possible to develop a machine learning based approach to mitigate this issue and provide more accurate predictions than currently available. This method would improve the accuracy of the performance model.

2. Currently, the creation of the performance model and the design space exploration based on it are fully manual. A reason for this is that it is crucial to present the bottlenecks of the system to the designer and not hide them within an automated tool. However, it should be possible to develop a tool, which automates this step of the design process and exposes the current design bottlenecks to the user. The tool should be based on two input files.

The first contains a description of the target system, while the second contains equations describing the performance model of the algorithm that is intended for acceleration. The tool can then automatically optimise the different degrees of parallelism to find the best performing design point within the described system characteristics. Additionally, this tool should report which parts of the design significantly contribute to the resource usage and which resources are limiting further acceleration. By exposing the equations used in the calculation and the most limiting characteristics to the designer it would be possible to develop an improved architecture based on the knowledge gained. Such a tool would be helpful to guide the designer into the correct direction and additionally accelerate the design process.

3. The biggest problem in accelerating complex algorithms using hardware is always the selection of datatypes. Since floating-point introduces a significant overhead in terms of area and power usage a fixed-point representation is typically desired. However, finding a fixed-point representation which maintains correct results for all possible input data values is highly problematic. Within this thesis a value profiling library was used, and a fixed-point simulation library was developed. However, both of these still require significant manual labour to derive working datatypes. It should be possible to further automate this and for example automatically suggest possible block floating-point formats, which sometimes can fill the gap between fixed- and floating-point types. Since there is a multitude of existing research on this problem it would first be necessary to gain an overview of the state-of-the-art methods and then integrate a selection into the methodology. Any automation for this datatype transformation would solve one of the most complicated steps in the acceleration process.

4. Especially if fixed-point representations are used it is possible that based on numeric problems wrong results are computed. In most cases it is possible to detect these errors, e.g., overflows, on the hardware level. It is possible to create a second dataflow graph which collects these status bits and checks if any results of the hardware accelerator are based on erroneous partial results. The output of this verification circuit could be used to improve trust in the calculations of an accelerator using a highly reduced fixed-point

data format. If errors occur it would for example be possible to recalculate the required step on the CPU. Providing such a tool would extend the methodology with basic error correction possibilities.

5. It might be possible to further automate the generation of specific hardware structures from a higher abstraction level. For example, data paths are often less performance critical, compared to buffers and data reordering circuits. In those cases, a language closer to HLS can be used if the communication behaviour on the input and output can be easily configured. Additionally, it should be possible to generate address generation units from, e.g., for loops in C. Address generators can usually be very slow, since a single output usually moves large chunks of data which require many cycles to be processed. The automatic generation of such modules would ease the design and especially the verification process significantly.

6. The methodology proposed in this thesis is based around Maxeler's MaxCompiler toolchain and the machine and programming model used by it. It would be of high interest how well the methodology maps to other tools and which changes have to be made. Especially the accurate prediction of the frequency deserves special attention.

7. As discovered using the radiotherapy simulation application the non-DMA based communication between CPU and FPGA is not modelled accurately. This is especially the case if the CPU has to actively load data from multiple data streams. In this case multithreading effects can have a significant impact on the expected performance. It would be beneficial if models could be developed to also predict the runtime behaviour of these communications accurately.

6.2.2 Portability Extension and Support for Modern FPGAs

In chapter 4 the methodology was extended to provide portability between different devices including modern multi-die FPGAs. To achieve this target best practices for the portability and performance scalability between different FPGA devices were presented and a memory map-

ping algorithm for multi-die FPGAs was proposed. Three possible extensions to the proposed memory mapping algorithm are of special interest. More precisely:

1. The current algorithm maps each logical memory only to a single physical memory. However, it would be possible to implement a logical memory using multiple different physical memories. If one for example considers a 72 bit wide and 4,650 deep memory currently the algorithm would map this logical memory either to URAM or BRAM blocks. However, it would be most efficient to map it up to the depth of 4,096 to a single URAM, then use BRAMs for the next 512 elements and map the remaining memory to logic. As such this extension would dramatically reduce the number of allocated but not used memory bits and therefore minimise the area of the implemented circuit.

2. The current algorithm focuses on SLR locality. It might be promising to consider locality within a single SLR to enable design units to reside in a specific subset of the SLR. This would minimise the routing delay and reduce the overall routing congestion. Developing such an algorithm will require more accurate prediction of the resource usage for all FPGA resources and not only a prediction limited to memories. Additionally, an integration with the FPGA vendor placement algorithm might be required.

3. To fully deal with the challenges introduced by multi-die FPGAs automated, device aware, floor planning is required. In general, automatic floor planning is a hard problem to solve. However, in the context of MaxCompiler the difficulty of this problem can be reduced considerably. Each design consists only of a few high-level components. At the moment, the memory mapping algorithm performs its allocation independently for each of these design units. However, it is also possible to predict the resource usage of the entire design and as such perform high level floor planning, mapping the different design units to the separate dies available on the targeted device. If floor planning happens with respect to the required inter-die connections, it would improve the timing characteristics of the design. Additionally, the memory mapping algorithm could operate on die instead of design unit basis, potentially reducing the required area.

6.2.3 Dose Accumulation Simulation for Radiotherapy

An FPGA based accelerator for the simulation of dose accumulation in human tissue in the context of radiotherapy is introduced in chapter 5. While the main objective within this thesis was to use this example to evaluate the extended methodology this implementation can be used as the base for further research leading towards the goal of real time radiotherapy. To accomplish this goal the following shortcomings will need to be addressed.

1. The current simulator only supports water as a material. To extend the simulator with other materials one would need to make two main modifications. A data structure is needed to hold the material type for each voxel of the patient cube. The same splitting into multiple subdomains can be used with on-chip buffers to store the material types for the current subdomain. The complete cube data can be stored in DDR memory and the on-chip buffer can be initialised while the dose cube is streamed out. Additionally, some of the equations used to simulate the particle behaviour have to be adjusted to use different constants based on the specific material that is simulated.

2. Additionally, to the limitations in terms of materials also the support for different high-energy particles is limited. At the moment only electrons are simulated. However, at least photon models have to be also included to achieve a meaningful simulation. For this, the required equations have to be added to the design and the data structures holding particle data have to be upgraded.

3. Since the current implementation of the simulation is mostly a proof of concept even all interactions for electrons were not implemented. For example, the Bremsstrahlung interaction is currently missing. To provide an accurate implementation one would need to add this by introducing the corresponding computations to the dataflow graph.

4. During the evaluation of the implementation, it was noticed that in multiple cases the PCIe speed proves to be the bottleneck. This problem could be partially mitigated by using PCIe Gen3 instead of Gen2. Once the compiler and driver support for this feature is added it should be straight forward to upgrade the design. This could reduce the runtime

of the simulation further. Additionally, one could try to schedule the transmission of patient cube buffers, e.g., using a round robin arbitration to reduce PCIe bus congestion and ensure that individual designs can start executing as fast as possible. This would avoid the current problem where all designs try to transmit their patient cube at the same time and as a result start and finish their transmission roughly at the same time, mostly stalling while waiting for PCIe bandwidth.

5. Another delay to the execution time was caused by the longer than expected initialisation time of the random number generators. It should be possible to optimise this by sharing the seed between runs and therefore reduce the runtime by up to 100ms. If the random number generators are not reset in between runs, they will still produce different numbers for each.

6. At the moment the sub domains are tiled automatically based on the overall cube dimensions. However, as noted during the evaluation, it would be advantageous to ensure that the particle beam enters the patient cube within a subdomain and not at subdomain boundaries. Multiple strategies seem possible to ensure this. For example, it could be possible to change the subdomain size accordingly and add support for subdomains of different sizes. Additionally, it might be possible to change the overall size of the patient cube to move the boundaries accordingly. This change would need to be performed in close collaboration with the medical experts to ensure that no deviation in the final result is introduced. By ensuring that the particle beam does not enter at subdomain boundaries the reliability of the simulation will be greatly improved.

6.3 Outlook

During the years of my research the FPGA landscape has changed significantly. While adoption of FPGAs for HPC is still very limited there is a more concentrated push by the wider industry towards the large-scale deployment of FPGAs. This seems to be mostly driven by the availability of servers with pre-installed FPGAs from major server vendors, the availability

of cloud instances using FPGAs and the acquisition of Altera by Intel. Very recently AMD announced the decision to buy the second big FPGA vendor Xilinx. As a result, the two biggest CPU vendors in the HPC space now also control the two biggest FPGA vendors. In both cases the publicly stated main motivation behind the acquisitions was the employment of FPGAs in the data centre.

However, it seems that all these investments and efforts are not fully paying off. The availability of AWS EC2 F1 instances is still limited to a few regions even though they were announced more than three years ago. There were also no new versions of the instances which could have used a larger FPGA like the VU13P or one containing HBM. Altera was bought more than five years ago from the point of writing and similarly very little new products have surfaced. There was also no significant change in market adoption or growth which Intel seemed to have hoped for at the time of the acquisition. The future will tell if AMD is more successful with the acquisition of Xilinx.

In my personal opinion the main reason for this lack of success is the lack of programmability. When Amazon introduced the F1 instance one of the main tutorials showed how to implement an 8-bit counter in Verilog on the F1 instance. The employment of this completely useless example (especially if one considers that running this counter would cost more than one dollar per hour) shows the complexity required for anything really impressive and useful. To avoid this problem Intel and Xilinx have fully embraced HLS tools. While Intel focuses more on OpenCL, Xilinx initially focused more on Vivado HLS, but seems to have moved more towards OpenCL in recent years as well. Additionally, both companies strive to provide libraries containing completely engineered solutions which can be used through a simple API. However, as discussed in this thesis I believe that OpenCL sacrifices too much performance to justify the inherent complexity in working with FPGAs. On top of this GPUs and CPUs have the advantage of high-volume production which enable them to reduce unit prices compared to FPGAs. As such FPGAs have to be able to provide a significant improvement in terms of performance and energy efficiency to provide a competitive position on the market. While customised libraries for specific functions might enable this on some use cases the integration into the actual application and the required data transfer to achieve this is of major concern. Additionally,

large organisations will require the ability to customise solutions for their specific use case in order to differentiate themselves from competitors. These major challenges seem to not be addressed by the strategies of Xilinx or Intel.

In my personal opinion a refocus of efforts will be required to enable the adoption of FPGAs in the data centre. This thesis attempted to outline one possible approach towards this target. I believe that the usage of a programming model tailored together with a design process tailored to this programming model and FPGAs will significantly reduce the complexity of FPGA programming. Within that framework it is also possible to address other long-term concerns with regards to FPGAs like portability. This reduction of complexity is crucial to make the large-scale adoption of FPGAs worthwhile. This believe has been strengthened by the experience I made with academic partners during my research. Especially in my first years of working with FPGAs I have seen many researchers with varying degrees of experience struggle and fail with their FPGAs focused projects. Over the last few years, I have tried to direct many of them towards the usage of the methodology proposed in this thesis and have made mostly positive experiences. In all cases the final result was a successful and well-designed FPGA application and multiple master thesis and research papers have been written as a result.

In addition to this positive result of my research the initial work on the radiotherapy has led to follow on research which is undertaken by researchers at the Joint Department of Physics at The Institute of Cancer Research and The Royal Marsden NHS Foundation Trust and the Custom Computing Group of Imperial College London. I hope that this research leads to a viable solution for real time radiotherapy which could result in a significant positive impact on cancer treatment quality.

Bibliography

[1] "Catapult C." [Online]. Available: http://calypto.com/en/products/catapult/overview/

[2] "EuroEXA." [Online]. Available: https://euroexa.eu

[3] "OpenSPL." [Online]. Available: http://www.openspl.org

[4] "Altera OpenCL," 2012. [Online]. Available: http://www.altera.com/products/software/opencl/opencl-index.html

[5] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL," in *Proceedings of the International Workshop on OpenCL 2013 & 2014*, ser. IWOCL '14. New York, NY, USA: ACM, 2014, pp. 4:1–4:9. [Online]. Available: http://doi.acm.org/10.1145/2664666.2664670

[6] Altera, *Floating-Point IP Cores User Guide*. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug_altfp_mfug.pdf

[7] ——, *Integer Arithmetic IP Cores User Guide*. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug_altmult_add.pdf

[8] Amazon, *Amazon F1 Instance*. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[9] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA:

Association for Computing Machinery, 1967, pp. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[10] J. Arram, K. H. Tsoi, W. Luk *et al.*, "Hardware acceleration of genetic sequence alignment," in *Reconfigurable Computing: Architectures, Tools and Applications*, P. Brisk, J. G. de Figueiredo Coutinho, and P. C. Diniz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 13–24.

[11] Arvind and D. E. Culler, "Dataflow architectures," *Annual Review of Computer Science*, vol. 1, no. 1, pp. 225–253, 1986. [Online]. Available: https://doi.org/10.1146/annurev.cs.01.060186.001301

[12] J. Auerbach, D. F. Bacon, P. Cheng *et al.*, "Lime: A java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 89–108. [Online]. Available: https://doi.org/10.1145/1869459.1869469

[13] U. Aydonat, S. O'Connell, D. Capalija *et al.*, "An OpenCL™ deep learning accelerator on Arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021738

[14] J. Bachrach, H. Vo, B. Richards *et al.*, "Chisel: Constructing hardware in a Scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[15] I. G. Y. Bengio and A. Courville, "Deep learning," 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[16] D. Boland and G. A. Constantinides, "Word-length optimization beyond straight line code," in *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013*, 2013, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435285

[17] J. Bosch, X. Tan, A. Filgueras *et al.*, "Application acceleration on FPGAs with OmpSs@FPGA," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 70–77.

[18] L. A. Buckley, I. Kawrakow, and D. W. O. Rogers, "CSnrc: correlated sampling Monte Carlo calculations using EGSnrc." *Medical physics*, vol. 31 12, pp. 3425–3435, 2004.

[19] D. Burger, "Keynote: Will programmable hardware reach scale," in *30th International Conference on Field Programmable Logic and Applications (FPL)*, 2020.

[20] D. Burger and B. Pelton, "Personal communication (16.09.2020)."

[21] M. A. Cantin, Y. Blaguiere, Y. Sarvaria *et al.*, "Analysis of quantization effects in a digital hardware implementation of a fuzzy ART neural network algorithm," in *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353)*, vol. 3, 2000, pp. 141–144 vol.3.

[22] M. A. Cantin, Y. Savaria, and P. Lavoie, "A comparison of automatic word length optimization procedures," in *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353)*, vol. 2, 2002, pp. II–612–II–615 vol.2.

[23] L. C. Carrington, M. Laurenzano, A. Snavely *et al.*, "How well can simple metrics represent the performance of HPC applications?" in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 48–48.

[24] J. Cassidy, A. Nouri, V. Betz *et al.*, "High-performance, robustly verified Monte Carlo simulation with FullMonte," *Journal of Biomedical Optics*, vol. 23, no. 8, pp. 1 – 11, 2018. [Online]. Available: https://doi.org/10.1117/1.JBO.23.8.085001

[25] R. V. Cherabuddi and M. A. Bayoumi, "Automated system partitioning for synthesis of multi-chip modules," in *Proceedings of 4th Great Lakes Symposium on VLSI*, March 1994, pp. 15–20.

[26] I. J. Chetty, P. M. Charland, N. Tyagi *et al.*, "Photon beam relative dose validation of the DPM Monte Carlo code in lung-equivalent media." *Medical physics*, vol. 30 4, pp. 563–73, 2003.

[27] G. C. T. Chow, A. H. T. Tse, Q. Jin *et al.*, "A mixed precision Monte Carlo methodology for reconfigurable accelerator systems," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 57–66. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145705

[28] R. Cmar, L. Rijnders, P. Schaumont *et al.*, "A methodology and design environment for DSP ASIC fixed point refinement," in *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, 1999, pp. 271–276.

[29] M. Curran, "Valuing Asian and portfolio options by conditioning on the geometric mean price," *Manage. Sci.*, vol. 40, no. 12, pp. 1705–1711, Dec. 1994. [Online]. Available: http://dx.doi.org/10.1287/mnsc.40.12.1705

[30] B. da Silva, A. Braeken, E. D'Hollander *et al.*, "Performance and resource modeling for FPGAs using high-level synthesis tools," in *Advances in Parallel Computing*, M. Bader, A. Bode, H.-J. Bungartz *et al.*, Eds., vol. 25. IOS Press, 2014, pp. 523–531. [Online]. Available: http://dx.doi.org/10.3233/978-1-61499-381-0-523

[31] B. da Silva, A. Braeken, E. H. D'Hollander *et al.*, "Performance modeling for FPGAs: Extending the roofline model with high-level synthesis tools," *Int. J. Reconfig. Comput.*, vol. 2013, pp. 7:7–7:7, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1155/2013/428078

[32] S. Dai, Y. Zhou, H. Zhang *et al.*, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[33] A. Davis and R. Keller, "Data flow program graphs," *Computer*, vol. 15, no. 02, pp. 26–41, feb 1982.

[34] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, B. Robinet, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 362–376.

[35] ——, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, Nov. 1980.

[36] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2nd Annual Symposium on Computer Architecture*, ser. ISCA '75. New York, NY, USA: Association for Computing Machinery, 1974, pp. 126–132. [Online]. Available: https://doi.org/10.1145/642089.642111

[37] J. Dongarra and W. Heins, "Professor Jack Dongarra announces new supercomputer benchmark," *The University of Tennessee Knoxville*, Jul 2013. [Online]. Available: https://news.utk.edu/2013/07/10/professor-jack-dongarra-announces-supercomputer-benchmark/

[38] A. Emmen, "Benchmarking Fugaku and Summit: a revealing process," *Primeur Magazin*, Jun 2020. [Online]. Available: http://primeurmagazine.com/flash/LV-PL-06-20-7.html

[39] V. Fanti, R. Marzeddu, C. Pili *et al.*, "Dose calculation for radiotherapy treatment planning using Monte Carlo methods on FPGA based hardware," in *2009 16th IEEE-NPSS Real Time Conference*, May 2009, pp. 415–419.

[40] L. Gan, H. Fu, W. Luk *et al.*, "Accelerating solvers for global atmospheric equations through mixed-precision data flow engine," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–6.

[41] P. Giannozzi, O. Andreussi, T. Brumme *et al.*, "Advanced capabilities for materials modelling with QUANTUM ESPRESSO," *Journal of Physics: Condensed Matter*, vol. 29, no. 46, p. 465901, 2017. [Online]. Available: http://stacks.iop.org/0953-8984/29/i=46/a=465901

[42] V. Gokhale, J. Jin, A. Dundar *et al.*, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2014.

[43] A. D. Gonzalez, A. Filgueras, D. Caballero *et al.*, "Nanos++ runtime library with Maxeler support," https://github.com/legato-project/ompss-maxeler, 2020.

[44] D. Goodman and M. Luján, "Scientific GPU programming with data-flow languages," 01 2011.

[45] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: ACM, 1982, pp. 120–126. [Online]. Available: http://doi.acm.org/10.1145/800230.806987

[46] C. Guo, H. Fu, and W. Luk, "A fully-pipelined expectation-maximization engine for gaussian mixture models," in *2012 International Conference on Field-Programmable Technology*, Dec 2012, pp. 182–189.

[47] K. He, X. Zhang, S. Ren *et al.*, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[48] M. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, pp. 409–436, December 1952.

[49] W. K. C. Ho and S. J. E. Wilton, "Logical-to-Physical memory mapping for FPGAs with dual-port embedded arrays," in *Field Programmable Logic and Applications*, P. Lysaght, J. Irvine, and R. Hartenstein, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 111–123.

[50] B. Holland, K. Nagarajan, C. Conger *et al.*, "RAT: A methodology for predicting performance in application design migration to FPGAs," in *Proceedings of the 1st International Workshop on High-performance Reconfigurable Computing Technology and Applications: Held in Conjunction with SC07*, ser. HPRCTA '07. New York, NY, USA: ACM, 2007, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1328554.1328560

[51] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser.

ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 280–289. [Online]. Available: https://doi.org/10.1145/1815961.1815998

[52] IBM, "Liquid Metal." [Online]. Available: https://researcher.watson.ibm.com/researcher/view_group.php?id=122

[53] Intel, *Intel Advisor Profiler*. [Online]. Available: https://software.intel.com/en-us/advisor

[54] X. Jia, X. George Xu, and C. G. Orton, "GPU technology is the hope for near real-time Monte Carlo dose calculations," *Medical Physics*, vol. 42, no. 4, pp. 1474–1476, 2015. [Online]. Available: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.4903901

[55] X. Jia, X. Gu, Y. J. Graves *et al.*, "GPU-based fast Monte Carlo simulation for radiotherapy dose calculation," *Physics in Medicine and Biology*, vol. 56, no. 22, pp. 7017–7031, oct 2011. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F56%2F22%2F002

[56] Y. Jia, E. Shelhamer, J. Donahue *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[57] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, Mar. 2004. [Online]. Available: https://doi.org/10.1145/1013208.1013209

[58] C. Jones, "Maxeler dense matrix multiply," https://github.com/maxeler/Dense-Matrix-Multiplication, 2015.

[59] N. P. Jouppi, C. Young, N. Patil *et al.*, "In-Datacenter performance analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. ACM, 2017, pp. 1–12.

[60] G. Kahn, "The semantics of a simple language for parallel programming," in *IFIP Congress*, 1974.

[61] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinancy, termination, queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966. [Online]. Available: http://www.jstor.org/stable/2946247

[62] I. Kawrakow, "VMC++, electron and photon Monte Carlo calculations optimized for radiation treatment planning," in *Advanced Monte Carlo for Radiation Physics, Particle Transport Simulation and Applications*, A. Kling, F. J. C. Baräo, M. Nakagawa *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 229–236.

[63] I. Kawrakow and M. Fippel, "Investigation of variance reduction techniques for Monte Carlo photon dose calculation using XVMC," *Physics in Medicine and Biology*, vol. 45, no. 8, pp. 2163–2183, jul 2000. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F45%2F8%2F308

[64] H. Keding, M. Willems, M. Coors *et al.*, "FRIDGE: a fixed-point design and simulation environment," in *Proceedings Design, Automation and Test in Europe*, Feb 1998, pp. 429–435.

[65] S. Kim, K.-I. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," in *VLSI Signal Processing, VIII*, Oct 1995, pp. 197–206.

[66] ——, "Fixed-point optimization utility for C and C++ based digital signal processing programs," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 45, no. 11, pp. 1455–1464, Nov 1998.

[67] P. J. Kinsman and N. Nicolici, "NoC-based FPGA acceleration for Monte Carlo simulations with applications to SPECT imaging," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 524–535, March 2013.

[68] R. Kirchgessner, A. D. George, and H. Lam, "Reconfigurable computing middleware for application portability and productivity," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, June 2013, pp. 211–218.

[69] R. Kirchgessner, G. Stitt, A. George *et al.*, "VirtualRC: A virtual FPGA platform for applications and tools portability," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 205–208. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145728

[70] D. Koeplinger, R. Prabhakar, Y. Zhang *et al.*, "Automatic generation of efficient accelerators for reconfigurable hardware," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 115–127.

[71] K. Koliogeorgi, N. Voss, S. Fytraki *et al.*, "Dataflow acceleration of Smith-Waterman with traceback for high throughput next generation sequencing," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 74–80.

[72] D. Komatitsch, J.-P. Vilotte, J. Tromp *et al.*, *SPECFEM3D Cartesian v2.0.2 [software]. Available: https://geodynamics.org/cig/software/specfem3d/*, Computational Infrastructure for Geodynamics.

[73] P. R. Kosinski, "A data flow language for operating systems programming," in *Proceeding of ACM SIGPLAN - SIGOPS Interface Meeting on Programming Languages - Operating Systems*. New York, NY, USA: Association for Computing Machinery, 1973, pp. 89–94. [Online]. Available: https://doi.org/10.1145/800021.808289

[74] A. Kouris, S. I. Venieris, and C. Bouganis, "A throughput-latency co-optimised cascade of convolutional neural network classifiers," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1656–1661.

[75] W. Kramer, "Top500 versus sustained performance - the top problems with the TOP500 list - and what to do about them," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 223–230.

[76] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982.

[77] H. Kung, C. Leiserson, C.-M. U. P. P. D. of COMPUTER SCIENCE. *et al.*, *Systolic Arrays for (VLSI)*, ser. CMU-CS. Carnegie-Mellon University, Department of Computer Science, 1978. [Online]. Available: https://books.google.nl/books?id=pAKfHAAACAAJ

[78] J. J. W. Lagendijk, B. W. Raaymakers, A. J. E. Raaijmakers *et al.*, "MRI/linac integration," *Radiotherapy and Oncology*, vol. 86, no. 1, pp. 25–29, 2019/04/10 2008. [Online]. Available: https://doi.org/10.1016/j.radonc.2007.10.034

[79] Y.-H. Lai, Y. Chi, Y. Hu *et al.*, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 242–251. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293910

[80] J. Lant, J. Navaridas, A. Attwood *et al.*, "Enabling standalone FPGA computing," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2019, pp. 23–26.

[81] J. Lant, J. Navaridas, M. Luján *et al.*, "Toward FPGA-based HPC: Advancing interconnect technologies," *IEEE Micro*, vol. 40, no. 1, pp. 25–34, 2020.

[82] V. W. Lee, C. Kim, J. Chhugani *et al.*, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[83] Y. Lin and L. Shao, "Super Vessel: The open cloud service for OpenPOWER," *White Paper, IBM corporation*, 2015.

[84] B. Lindsey, M. Leslie, and W. Luk, "A domain specific language for accelerated multilevel Monte Carlo simulations," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2016, pp. 99–106.

[85] O. Lindtjorn, R. Clapp, O. Pell *et al.*, "Beyond traditional microprocessors for geoscience high-performance computing applications," *IEEE Micro*, vol. 31, no. 2, pp. 41–49, March 2011.

[86] M. T. Ltd., "MaxGenFD white paper," *2012*. [Online]. Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxGenFD.pdf

[87] L. Lu, Y. Liang, Q. Xiao *et al.*, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 101–108.

[88] C.-M. Ma, J. S. Li, T. Pawlicki *et al.*, "A Monte Carlo dose calculation tool for radiotherapy treatment planning," *Physics in Medicine and Biology*, vol. 47, no. 10, pp. 1671–1689, may 2002. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F47%2F10%2F305

[89] Y. Ma, Y. Cao, S. Vrudhula *et al.*, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021736

[90] G. Madec, *NEMO Ocean engine. Note du Pôle de modélisation.* Institut Pierre-Simon Laplace (IPSL), 2008.

[91] M. Makni, M. Baklouti, S. Niar *et al.*, "Hardware resource estimation for heterogeneous FPGA-based SoCs," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: ACM, 2017, pp. 1481–1487. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019683

[92] F. Mao, W. Zhang, B. Feng *et al.*, "Modular placement for interposer based multi-FPGA systems," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, May 2016, pp. 93–98.

[93] *Multiscale Dataflow Programming*, Maxeler Technologies, 2018.

[94] *Dataflow Acceleration Process and Optimisation Book*, Maxeler Technologies, 2020.

[95] O. Mencer, "ASC: a stream compiler for computing with FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1603–1617, 2006.

[96] O. Mencer, M. Morf, and M. J. Flynn, "Hardware software tri-design of encryption for mobile communication units," in *in Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, pp. 3045–3048.

[97] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014. [Online]. Available: http://science.sciencemag.org/content/345/6197/668

[98] *Data sheet: 2GB, 4GB, 8GB (x72, ECC, DR) 204-pin DDR3L SODIMM*, Micron. [Online]. Available: https://www.micron.com/parts/modules/ddr3-sdram/mt18ksf1g72hz-1g6

[99] Microsoft, *Inside the Microsoft FPGA-based configurable cloud*. [Online]. Available: https://azure.microsoft.com/en-gb/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/

[100] ——, *Project Catapult*. [Online]. Available: https://www.microsoft.com/en-us/research/project/project-catapult/

[101] E. Monmasson and M. N. Cirstea, "FPGA design methodology for industrial control systems - A review," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 4, pp. 1824–1842, Aug 2007.

[102] J.-M. Muller, N. Brisebarre, F. de Dinechin *et al.*, *Handbook of Floating-Point Arithmetic*, 1st ed. Birkhäuser Basel, 2009.

[103] S. W. Nabi and W. Vanderbauwhede, "FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis," *Journal of Parallel and Distributed Computing*, vol. 133, pp. 407 – 419, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373151730165X

[104] Y. Nakamura and H. Stüben, "BQCD - Berlin quantum chromodynamics program," vol. abs/1011.0199, 2014. [Online]. Available: https://arxiv.org/abs/1011.0199

[105] A. Negoi and J. Zimmermann, "Monte Carlo hardware simulator for electron dynamics in semiconductors," in *1996 International Semiconductor Conference. 19th Edition. CAS'96 Proceedings*, vol. 2, Oct 1996, pp. 557–560 vol.2.

[106] A. M. Nestorov, E. Reggiani, H. Palikareva *et al.*, "A scalable dataflow implementation of Curran's approximation algorithm," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 150–157.

[107] Z. Obradovic, L. Verdoscia, and R. Giorgi, "A data-flow soft-core processor for accelerating scientific calculation on FPGAs," *Mathematical Problems in Engineering*, vol. 2016, p. 3190234, 2016. [Online]. Available: https://doi.org/10.1155/2016/3190234

[108] O. Pell, J. Bower, R. Dimond *et al.*, "Finite-Difference wave propagation modeling on special-purpose dataflow machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 906–915, May 2013.

[109] A. Rajagopal, D. A. Vink, S. I. Venieris *et al.*, "Multi-precision policy enforced training (MuPPET): A precision-switching strategy for quantised fixed-point training of CNNs," 2020.

[110] R. Rashid, J. G. Steffan, and V. Betz, "Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 20–27.

[111] T. Riesgo, Y. Torroja, and E. de la Torre, "Design methodologies based on hardware description languages," *IEEE Transactions on Industrial Electronics*, vol. 46, no. 1, pp. 3–12, 1999.

[112] B. Rountree, D. K. Lowenthal, M. Schulz *et al.*, "Practical performance prediction under dynamic voltage frequency scaling," in *2011 International Green Computing Conference and Workshops*, July 2011, pp. 1–8.

[113] K. Roy and C. Sechen, "A timing driven N-way chip and multi-chip partitioner," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, Nov 1993, pp. 240–247.

[114] C. d. Schryver, I. Shcherbakov, F. Kienle *et al.*, "An energy efficient FPGA accelerator for Monte Carlo option pricing with the Heston model," in *2011 International Conference on Reconfigurable Computing and FPGAs*, Nov 2011, pp. 468–474.

[115] J. Sempau, S. J. Wilderman, and A. F. Bielajew, "DPM, a fast, accurate Monte Carlo code optimized for photon and electron radiotherapy treatment planning dose calculations," *Physics in Medicine and Biology*, vol. 45, no. 8, pp. 2263–2291, jul 2000. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F45%2F8%2F315

[116] Y. S. Shao, B. Reagen, G. Y. Wei *et al.*, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 97–108.

[117] C. Shi and R. W. Brodersen, "Automated fixed-point data-type optimization tool for signal processing and communication systems," in *Proceedings. 41st Design Automation Conference, 2004.*, July 2004, pp. 478–483.

[118] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[119] D. P. Singh, T. S. Czajkowski, and A. Ling, "Harnessing the power of FPGAs using Altera's OpenCL compiler," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 5–6. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435268

[120] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.

[121] R. Tessier, V. Betz, D. Neto *et al.*, "Power-efficient RAM mapping algorithms for FPGA embedded memory blocks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 278–290, Feb 2007.

[122] X. Tian and C. Bouganis, "A run-time adaptive FPGA architecture for Monte Carlo simulations," in *2011 21st International Conference on Field Programmable Logic and Applications*, 2011, pp. 116–122.

[123] M. M. Tikir, L. Carrington, E. Strohmaier *et al.*, "A genetic algorithms approach to modeling the performance of memory-bound computations," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–12.

[124] A. Tisan and J. Chin, "An end-user platform for FPGA-based design and rapid prototyping of feedforward artificial neural networks with on-chip backpropagation learning," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 3, pp. 1124–1133, June 2016.

[125] R. Townson, X. Jia, S. Zavgorodni *et al.*, "SU-E-T-476: GPU-based Monte Carlo radiotherapy dose calculation using phase-space sources," *Medical Physics*, vol. 39, no. 6 Part 17, pp. 3814–3814, 2012. [Online]. Available: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.4735565

[126] N. Tyagi, A. Bose, and I. J. Chetty, "Implementation of the DPM Monte Carlo code on a parallel architecture for treatment planning applications," *Medical Physics*, vol. 31, no. 9, pp. 2721–2725, 2004. [Online]. Available: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.1786691

[127] Y. Umuroglu, N. J. Fraser, G. Gambardella *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *The 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[128] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 40–47.

[129] L. Vermond, "Accelerating basecalling with dataflow computing," Master's thesis, TU Delft, October 2020.

[130] J. Villarreal, A. Park, W. Najjar *et al.*, "Designing modular hardware accelerators in C with ROCCC 2.0," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 127–134.

[131] N. Voss, M. Bacis, O. Mencer *et al.*, "Convolutional neural networks on dataflow engines," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 435–438.

[132] N. Voss, P. Quintana, O. Mencer *et al.*, "Memory mapping for multi-die FPGAs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 78–86.

[133] N. Voss, P. Ziegenhein, L. Vermond *et al.*, "Towards real time radiotherapy simulation," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, July 2019, pp. 173–180.

[134] N. Voss, T. Becker, O. Mencer *et al.*, *Rapid Development of Gzip with MaxJ*. Cham: Springer International Publishing, 2017, pp. 60–71. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-56258-2_6

[135] N. Voss, S. Girdlestone, T. Becker *et al.*, "Low area overhead custom buffering for FFT," in *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, D. Andrews, R. Cumplido, C. Feregrino *et al.*, Eds. IEEE, 2019, pp. 1–8. [Online]. Available: https://doi.org/10.1109/ReConFig48160.2019.8994775

[136] N. Voss, S. Girdlestone, O. Mencer *et al.*, "Automated dataflow graph merging," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 219–226.

[137] N. Voss, B. Kwaadgras, O. Mencer *et al.*, "On predictable reconfigurable system design," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 2, Feb. 2021. [Online]. Available: https://doi.org/10.1145/3436995

[138] N. Voss, P. Ziegenhein, L. Vermond *et al.*, "Towards real time radiotherapy simulation," *Journal of Signal Processing Systems*, vol. 92, no. 9, pp. 949–963, 2020. [Online]. Available: https://doi.org/10.1007/s11265-020-01548-9

[139] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *FPGA Accelerator Design Using OpenCL*. Cham: Springer International Publishing, 2018, pp. 29–43. [Online]. Available: https://doi.org/10.1007/978-3-319-68161-0_3

[140] Z. Wang, B. He, W. Zhang *et al.*, "A performance analysis framework for optimizing OpenCL applications on FPGAs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 114–125.

[141] Wave Computing, "Wave Computing." [Online]. Available: https://wavecomp.ai

[142] X. Wei, C. H. Yu, P. Zhang *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 29.

[143] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[144] K. G. Wilson, "Confinement of quarks," *Physical Review D*, vol. 10, pp. 2445–2459, Oct 1974. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevD.10.2445

[145] S. Winograd, *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611970364

[146] J. Wulff, K. Zink, and I. Kawrakow, "Efficiency improvements for ion chamber calculations in high energy photon beams." *Medical physics*, vol. 35 4, pp. 1328–1336, 2008.

[147] Xilinx, *Baidu Deploys Xilinx FPGAs in New Public Cloud Acceleration Services.* [Online]. Available: https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html

[148] ——, *Large FPGA Methodology Guide.* [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/ug872_largefpga.pdf

[149] ——, *Performance and Resource Utilization for Adder/Subtracter v12.0.* [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/c-addsub.html

[150] ——, *Performance and Resource Utilization for Divider Generator v5.1.* [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/div-gen.html

[151] ——, *Performance and Resource Utilization for Floating-point v7.1.* [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html

[152] ——, *Performance and Resource Utilization for Multiply Adder v3.0.* [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/xbip-multadd.html

[153] ——, *UltraScale Architecture and Product Data Sheet: Overview.* [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[154] ——, *UltraScale Architecture Configurable Logic Block. User Guide.* [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[155] ——, *UltraScale Architecture Memory Resources. User Guide.* [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

[156] ——, *UltraScale+ FPGAs. Product Tables and Product Selection Guide.* [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf

[157] ——, *Vivado HLS: How can I infer UltraRAM in HLS?* [Online]. Available: https://www.xilinx.com/support/answers/71259.html

[158] ——, *Xilinx ALVEO Adaptable Accelerator Cards for Data Center Workloads.* [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo.html

[159] ——, "The Xilinx SDAccel development environment." [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf

[160] *Vivado Design Suite. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html*, Xilinx Inc., 2019.

[161] Y. Yamaguchi, R. Azuma, A. Konagaya *et al.*, "An approach for the high speed Monte Carlo simulation with FPGA - toward a whole cell simulation," in *2003 46th Midwest Symposium on Circuits and Systems*, vol. 1, Dec 2003, pp. 364–367 Vol. 1.

[162] ZDNet, *Intel FPGAs picked up by Dell EMC and Fujitsu.* [Online]. Available: https://www.zdnet.com/article/intel-fpgas-picked-up-by-dell-emc-and-fujitsu/

[163] M. D. Zeiler, M. Ranzato, R. Monga *et al.*, "On rectified linear units for speech processing," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 3517–3521.

[164] C. Zhang, Z. Fang, P. Zhou *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–8.

[165] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *Proceedings of the 2017 ACM/SIGDA International*

Symposium on Field-Programmable Gate Arrays, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 25–34. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021698

[166] J. Zhao, L. Feng, S. Sinha *et al.*, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 430–437.

[167] P. Ziegenhein, I. N. Kozin, C. P. Kamerling *et al.*, "Towards real-time photon Monte Carlo dose calculation in the cloud," *Physics in Medicine and Biology*, vol. 62, no. 11, pp. 4375–4389, may 2017. [Online]. Available: https://doi.org/10.1088%2F1361-6560%2Faa5d4e

[168] P. Ziegenhein, S. Pirner, C. P. Kamerling *et al.*, "Fast CPU-based Monte Carlo simulation for radiotherapy dose calculation," *Physics in Medicine and Biology*, vol. 60, no. 15, pp. 6097–6111, jul 2015. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F60%2F15%2F6097