OpenCL acceleration on FPGA vs CUDA on GPU

Quist, R.

Award date:
2019

A joint research by:
The Electronic Systems group, Eindhoven University of Technology
The High-End Computing group, Prodrive Technologies

OpenCL acceleration on FPGA vs CUDA on GPU

Master Thesis

By
Remy Quist

Supervisors:
Prof. dr. K.G.W. Goosens
Prof. dr. ir. C.H. van Berkel
Ir. J.A. Huisken
Ir. R.P.M. van Doormaal

Revision: 5
Tuesday 12th March, 2019

Preface

During this research, many hurdles had to be overcome: inconclusive results, a broken collarbone, and a fire destroying a large part of the company including my entire benchmarking setup. While each of these hurdles postponed my graduation date by several months, I can now finally say that my report is finished!

Without any help, finishing this report would have been impossible. For this reason, I would like to thank Kees Goossens for mentoring and providing invaluable high-level feedback during the entire project. Next, I would like to thank Roy van Doormaal as his continued support and detailed feedback kept the quality standards high. I also want to thank both Mathijs Braakhuis and Bas van Bree for managing the project and ensuring this project included the interests of Prodrive. Next, I thank Frey Franken for stepping in when Roy was on holiday and Enzo Evers for his understanding of the project and his help finalizing the benchmarking setup.

I also thank Prodrive for proving me the environment and tools to perform my research and Xilinx for their help solving issues with their hardware and tooling.

Finally, I would like to thank my partner Jannelize, my family and friends for their continued support and giving me the energy to continue and finish this project.

Remy Quist,
Eindhoven, February 2019

Abstract

The goal of this research is to determine the value that OpenCL FPGA acceleration brings to the CUDA GPU hardware acceleration market. This value is determined by both a literature study and by optimizing and benchmarking the RabbitCT and Demosaic algorithms to a similarly priced Nvidia Quadro P6000 GPU and Xilinx Kintex KCU1500 FPGA. The performance of the selected algorithms is analyzed using six factors: maximum performance, costs per performance, programmability, interconnect options, energy efficiency and product availability.

Here is found that while the maximum FPGA performance does not reach the maximum GPU performance, the costs-per-performance can similar for less memory intensive algorithms. Next, FPGAs offer higher energy efficiency and product availability than GPUs. As for the interconnect, GPUs are limited to the PCIe interface for I/O communication, while FPGAs allow any additional interconnect to be connected to increase the I/O bandwidth. Finally, while FPGAs offer several advantages over GPUs, their programmability is much lower, meaning OpenCL kernels are significantly harder to program for the FPGA due to the long compilation times and manual memory management that is required.

From these results can be concluded that OpenCL FPGA acceleration is most useful in cases where energy efficiency and product maintainability is more important than the maximum performance or having a short development time.

Contents

Contents iv

List of Figures vii

List of Tables viii

Listings x

Abbreviations xi

1 Introduction 2

1.1 Background: hardware acceleration 2

1.2 Problem & project goal 3

1.3 Outline 4

2 Method 5

3 Literature Study 8

3.1 Accelerator availability 8

3.2 Programming methods 9

3.3 API porting 11

3.4 FPGA acceleration research 14

3.4.1 Cloud acceleration 14

3.4.2 Image processing 16

3.4.3 Others examples 17

3.5 Optimization strategies 20

3.5.1 FPGA OpenCL 20

3.5.2 GPU CUDA & OpenCL 24

3.6 Literature study conclusion . . . 27
3.6.1 Accelerator availability . . . 27
3.6.2 Programming methods . . . 27
3.6.3 API Porting . . . 27
3.6.4 FPGA acceleration . . . 27
3.6.5 Conclusion . . . 28

4 Algorithm Analysis **30**
4.1 RabbitCT . . . 30
4.1.1 Theoretical analysis . . . 30
4.1.2 Algorithm implementation . . . 34
4.1.3 Implementation analysis . . . 35
4.2 Demosaic . . . 41
4.2.1 Theoretical analysis . . . 42
4.2.2 Algorithm implementation . . . 44
4.2.3 Implementation analysis . . . 46

5 Hardware Selection **49**
5.1 FPGA selection . . . 49
5.1.1 FPGA requirements and options . . . 49
5.1.2 FPGA selection and specifications . . . 51
5.2 GPU selection . . . 52
5.2.1 GPU requirements and options . . . 52
5.2.2 GPU selection and specifications . . . 53
5.3 Host selection . . . 54
5.3.1 Host requirements . . . 54
5.3.2 Host specifications . . . 54
5.4 Hardware comparison . . . 55
5.4.1 Numerical comparison . . . 55
5.4.2 Interconnect options . . . 57
5.4.3 Roofline comparison . . . 57

6 Algorithm Implementation **60**
6.1 RabbitCT . . . 60

6.1.1 RabbitCT GPU . . . 60
6.1.2 RabbitCT FPGA . . . 68
6.2 Demosaic . . . 75
6.2.1 Demosaic FPGA . . . 75
6.2.2 Demosaic GPU . . . 84

7 Results **88**
7.1 Maximum performance . . . 88
7.2 Costs per performance . . . 89
7.2.1 RabbitCT . . . 89
7.2.2 Demosaic . . . 90
7.2.3 Summary . . . 91
7.3 Programmability . . . 93
7.3.1 Tooling . . . 93
7.3.2 Code . . . 96
7.3.3 Summary . . . 100
7.4 Interconnect options . . . 100
7.5 Energy efficiency . . . 100
7.6 Product availability . . . 101

8 Conclusion **102**

9 Future Work & Expectations **105**
9.1 Future work . . . 105
9.2 Future expectations . . . 106

Bibliography **107**
Programming guides . . . 107
Referenced papers . . . 108

Referenced websites 112

List of Figures

1.1 Hardware hierarchy 2

3.1 Main GPU Programming Languages 10
3.2 Productivity enhancing programming languages 11
3.3 OpenCL and CUDA compilation flow 13
3.4 CUDA OpenCL porting 13
3.5 Cloud-Scale Acceleration Architecture 15
3.6 OpenCL FPGA memory model 21
3.7 OpenCL pipe implementation 21
3.8 SIMD visualization 22
3.9 Partitioning methods 23
3.10 CUDA GPU memory model 25
3.11 Memory access patterns 25

4.1 Heatmap of RabbitCT input image #100 31
4.2 2D backprojection example 31
4.3 Normal vs filtered backprojection 32
4.4 2D to 3D perspective 32
4.5 RabbitCT Dataflow 37
4.6 Pixel access trace 38
4.7 Pixel access heatmap 39
4.8 Resolution 128 input memory usage analysis 40
4.9 From raw input to interpolated output 41
4.10 Bayer filter, colour channels and conversion to RGB 42
4.11 Luminance and chrominance 43

4.12 Malvar-He-Cutler demosaic filter coefficients 43
4.13 Demosaic Dataflow 47

5.1 Kintex UltraScale FPGA KCU 1500 PCB 52
5.2 Nvidia Quadro P6000 54
5.3 Roofline initial implementations 59

6.1 RabbitCT Thread Optimization 64
6.2 The advantage of streams 66
6.3 Simplified Demosaic prefetching 78
6.4 Pixelbuffer example burst-size 4 80

7.2 Costs-per-performance results 92
7.3 NVVP Performance Analysis Reports 95
7.4 Lines of code 98
7.5 FPGA hardware compilation time 99

List of Tables

3.1 Xilinx Ultrascale+ data type comparisons for the addsub unit 23
3.2 Nvidia compute capability 7.5 data type comparisons for add, multiply, and multiply-add commands 26
3.3 Literature study results 29

4.1 RabbitCT output size 38
4.2 RabbitCT input distance between voxels 41

5.1 Intel OpenCL FPGA Accelerators 50
5.2 Xilinx OpenCL FPGA Accelerators 51
5.3 Nvidia Quadro GPUs as of December 2018 53
5.4 Xilinx KCU1500 floating point performance calculation 56

5.5 Hardware comparison 57

6.1 RabbitCT CUDA GPU implementation results 61

6.2 RabbitCT OpenCL GPU implementation results 61

6.3 RabbitCT FPGA implementation final results 69

6.4 RabbitCT FPGA implementation 1 results 69

6.5 RabbitCT FPGA implementation 2 results 70

6.6 RabbitCT FPGA implementation 3 results 71

6.7 RabbitCT FPGA implementation 4 results 72

6.8 RabbitCT FPGA implementation 5 results 73

6.9 RabbitCT FPGA implementation 6 results 74

6.10 Demosaic FPGA implementation final results 76

6.11 Demosaic FPGA implementation 0 results 77

6.12 Demosaic FPGA implementation 1 results 77

6.13 Demosaic FPGA implementation 2 results 79

6.14 Demosaic FPGA implementation 3 results 81

6.15 Demosaic FPGA implementation 4 results 81

6.16 Demosaic FPGA implementation 5 results 83

6.17 Demosaic OpenCL GPU implementation results 84

6.18 Demosaic local work-item performance in FPS 85

6.19 Demosaic GPU tweaking results 87

7.1 RabbitCT costs per performance results 90

7.2 Demosaic costs per performance results 91

7.3 Programmability results 100

8.1 Final results 104

Listings

4.1 RabbitCT-Runner pseudocode 34
4.2 RabbitCT-Algorithm pseudocode 35
4.3 RabbitCT backprojection data 36
4.4 Demosaic host pseudocode 44
4.5 Demosaic algorithm pseudocode 45

Abbreviations

ASIC	Application Specific Integrated Circuit
API	Application Programming Interface
BRAM	Block Random Access Memory
BSP	Board Support Package
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate Memory
DMA	Direct Memory Access
DSP	Digital Signal Processor
FF	Flip Flop
FIFO	First In First Out
Fmax	Maximum frequency
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
Freq.	Frequency
GB	Gigabyte
GbE	Gigabit Ethernet
GBps, GB/s	Gigabyte per second
GDDR	Graphics Double Data Rate Memory
Gen	Generation
GFLOP/s	Giga floating point operations per second
Gop/s	Giga operations per second
GPU	Graphics Processing Unit
HBM2	High Bandwidth Memory 2
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPC	High-Performance Computing
HU	Houndsfield scale Units
HW	Hardware
II	Instruction Interval
Impl.	Implementation
I/O	Input and Output
JTAG	Joint Test Action Group
LTS	Long Term Support
LUT	Look Up Table
MSE	Means Squared Error
MT/s	Mega Transfers per Second
NVVP	NVidia Visual Profiler
OpenCL	Open Computing Language
Opt.	Optimized
OS	Operating System

PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
Perf	Performance
PSNR	Peak Signal to Noise Ratio
QSFP	Quad Small Form Factor Pluggable
RAM	Random Access Memory
RGB	Red Green Blue colour space
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SLI	Scalable Link Interface
SoC	System on Chip
SW	Software
TDP	Thermal Design Power
TIFF	Tagged Image File Format
TFLOP/s	Terra floating point operations per second
WC Latency	Worst Case Latency
XCPP	Xilinx C++ Compiler
XOCP	Xilinx OpenCl Compiler
YUV	Luma (Y) and Chrominance (UV) colour space

Chapter 1

Introduction

1.1 Background: hardware acceleration

Modern data processing workloads often contain large compute-intensive tasks with highly parallelizable sections. These sections consist of a small piece of code that is executed a large number of times. Hardware accelerators are used to increase the throughput and efficiency of these applications. Accelerators differentiate themselves from a general-purpose processor by trading flexibility and functionality for a gain in efficiency. By combining a general-purpose processor, also known as the host, with one or more accelerators, the throughput of an application can be increased significantly. Offloading computationally intensive parts to an accelerator allows these parts to be executed faster because accelerators can perform specific software functionality more efficiently than a general-purpose processor. Additionally, offloading the work to an accelerator allows the processor to process other work while waiting for the accelerator to finish.

When comparing computer hardware on flexibility and efficiency, computer hardware can be divided into four main classes, see Figure 1.1. Here, flexibility means that the device can be used for a wide range of functions and can easily be reconfigured for a new function. Efficiency indicates that this type of device can execute a given job with high performance and power efficiency.

From the four categories, a CPU or a general-purpose processor is the most flexible piece of hardware. It is designed to handle any task given and excels in the execution of complex sequential tasks. Due to the large number of tasks a CPU should be able to handle, its hardware has a large footprint. For this reason, the CPU with the most cores available at the moment of writing contains 32 cores [70], limiting the work that can be performed in parallel. Due to the large number of tasks the CPU can perform, many different hardware elements are present, making it unlikely that all hardware is used in each clock cycle. To improve the hardware usage, hyperthreading or simultaneous multithreading virtually increases the number of cores by allowing two threads to be processed in parallel on a single core. Here, the CPU selects instructions of both threads to make

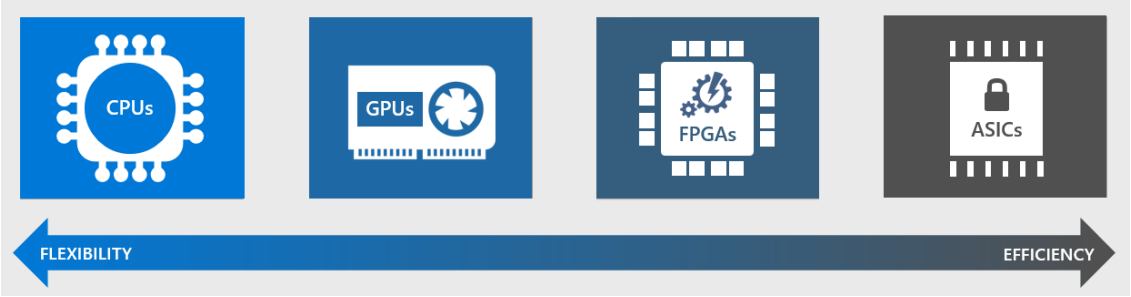


Figure 1.1: Hardware hierarchy, taken from [69]

optimal use of the hardware elements available.

The graphics processing unit, or GPU, is created for parallelism. Parallelism is achieved by limiting the device to simpler processing tasks to allow a large number of processing elements to fit on a chip with the same footprint. As of writing, the GPU with the most processing units contains 5120 cores on a single die [71]. The parallelism of the GPU enables it to efficiently execute simpler tasks that are executed a large number of times. An example of such a task is graphics processing, where the GPU calculates the colour data of each pixel that has to be shown on the screen.

Another type of accelerator is the Field Programmable Gate Array, FPGA for short. An FPGA is more efficient than a GPU or CPU at the cost of programming flexibility. A CPU and GPU both consist of computational cores which can be reconfigured at run-time. Every clock cycle these cores receive instructions that are executed accordingly, which is enabled by the flexible design of these cores where each core contains the hardware for performing all supported instructions every cycle. So, even if most instructions are never executed, the device will always be ready to perform them which reduces the overall efficiency of the device. An FPGA does not contain the programmable core structure of the CPU and GPU; it contains a large number of logic building blocks with a programmable interconnect. By combining these logic elements in certain ways, a data path is formed using only the logic elements required for the execution of a specific function. Forming function-specific data paths makes FPGAs significantly more efficient as only the hardware elements required for a specific task are being used.

A downside to FPGAs however, is that the hardware cannot be reconfigured every cycle, limiting the implemented hardware paths to the configuration assigned at startup. To ease reconfiguration, modern FPGAs add support for partial reconfiguration, allowing certain sections of the FPGA to be reconfigured, while the rest stays operational. However, this reconfiguration operation still takes time, which limits the flexibility of the hardware. Another downside to FPGAs is that they are harder to program. An FPGA is programmed by a binary file generated from Register-Transfer-Level code. RTL code is a low-level programming language, which requires knowledge of the hardware, making it harder to program compared to any high-level language. Programming an FPGA in RTL requires the user to program the data flow of the hardware; for each clock cycle and input, the operation and output destination of each input must be defined. This is not the case when programming for a CPU or GPU, here a compiler reads the program written in a hardware independent high-level programming language and translates this into the hardware specific machine code which the underlying CPU or GPU understands. Compilation for a CPU or GPU is a relatively easy task, as the hardware in a CPU and GPU is constant. The FPGA hardware is not constant, and its paths are user-defined. So, the compilation process must not only map the software to hardware, but it must also synthesize an efficient hardware implementation itself. These additional compilation steps make high-level FPGA compilation a much harder task, requiring a lot of compute resources. For this reason, high-level synthesis has only recently become a viable FPGA programming method.

The final hardware category shown in Figure 1.1 is that of the Application-Specific Integrated Circuit or ASIC. The ASIC fully implements the application in hardware, reducing the flexibility to what it is designed to do, which eliminates any general purpose usage. However, this reduction in flexibility further increases its efficiency as this allows the performance and energy efficiency to be maximized.

1.2 Problem & project goal

The goal of this research is to determine whether OpenCL FPGA acceleration can be a worthy replacement for the currently used CUDA GPU acceleration solutions. The search for a replacement was initiated by customers of Prodrive stating the lifecycle of a GPU accelerator is a problem.

They state that this problem is caused by GPU vendors only fabricating a certain GPU generation for 2-3 years, after which only newer GPU generation are available; forcing any broken GPUs hardware to be upgraded. Upgrading the hardware is by itself not a problem as this normally increases performance or decreases costs. However, architectural changes in the new hardware generation require the software to be updated as well. These changes require a new development and testing process, making the upgrade a costly and time-consuming process, and thus a problem.

In November 2012, Altera (now Intel), released OpenCL compilers for FPGA programming [1], with Xilinx following in early 2015 [2]. These compilers allow the FPGA to be programmed in a similar way to the CUDA acceleration method currently used by Prodrive its customers. Moreover, as FPGAs are available for ten years or more, FPGA acceleration could be an interesting solution to the GPU lifetime problem. However, as this acceleration technique is still in its infancy, detailed information about the performance and cost of this solution not available. So, to find the answers to these unknowns, this research has been initiated.

1.3 Outline

In Chapter 2, the method of this research is explained. Which is followed by a literature study in Chapter 3 to get a view of the current status of the market. The literature study is followed by an algorithm analysis in Chapter 4. With knowledge of both the literature and algorithms, the benchmarking hardware is selected in Chapter 5. Chapter 6 shows the applied optimizations and results for each platform and algorithm. These results are discussed and compared in Chapter 7. Finally, in Chapter 8, a conclusion is made and future work and expectations are shown in Chapter 9.

Chapter 2

Method

The value that OpenCL acceleration on FPGA brings to the GPU hardware acceleration market is determined by a literature study and by the porting and benchmarking of two Prodrive selected algorithms. The literature study is used to gain knowledge on the subject, which helps with porting the selected algorithms to the desired platforms. The literature study consists of a product availability analysis, determining how many years a certain GPU or FPGA can be purchased; an analysis between both the CUDA and OpenCL API, indicating the language differences and similarities and porting methods; and it contains the optimization strategies and results obtained by other FPGA acceleration research papers. This data not only helps to optimize the selected algorithms but also serves as a result comparison, reducing the chance for a bias in the results to one of the platforms.

With the knowledge of the literature study available, two applications are ported from their source code to each of the acceleration platforms. The selected algorithms are the RabbitCT benchmarking suite [22] and the Linear Image Demosaicing algorithm proposed by Malvar-He-Cutler [23]. RabbitCT is a CT-scanner back-projection algorithm used in the medical market; it converts a sum of X-ray images into a 3D voxel representation of the scanned object. The Demosaic algorithm is an image processing algorithm that converts monochrome Bayer images into coloured RGB output images by filtering the image data into their respective colours. Originally, RabbitCT was the only algorithm benchmarked for this project, but as its GPU vs FPGA results were inconclusive, the Demosaic algorithm was added to expand the scope of the project. Both algorithms have been selected for their relevance for Prodrive Technologies, as their customers found these algorithms computationally representative for the algorithms they want to implement. Both algorithms are explained in more detail in Chapter 4.

Six metrics are used to determine what value OpenCL FPGA acceleration brings when compared to the CUDA GPU acceleration market. These metrics are the maximum performance, the cost per performance, the programmability, the interconnect options, the energy efficiency, and the product availability of each platform. The maximum performance shows the maximum capabilities of the hardware and indicates any differences between APIs. Analyzing the cost per performance removes the architectural price differences and shows what a certain performance will cost for each platform. The programmability metric indicates the perceived difficulty of creating an optimized implementation of both algorithms for each platform and is determined subjectively. The interconnect shows the options that are available for the platform to communicate with other platforms. The final two metrics, the energy efficiency and product availability, are used to indicate what longer term cost saving properties a platform might have. If a platform has a higher energy efficiency or is available for longer, then this platform is more cost efficient over time, which might make it a better option even if another platform has a higher performance and programmability. By analyzing the selected applications and platforms using these five metrics, the knowledge Prodrive hopes to gain from this research is covered.

When comparing CUDA GPU acceleration with OpenCL FPGA acceleration, two variables are changed; namely, the programming API and the accelerator hardware. So, to determine if per-

ceived changes are API related or hardware related, an OpenCL GPU benchmark is performed as well. Benchmarking OpenCL on a GPU enables CUDA vs OpenCL GPU testing to show API differences, and it allows the created GPU builds to be compared to the OpenCL FPGA builds to show whether any hardware differences are present.

During the RabbitCT porting process, the code was optimized for CUDA first. Selecting CUDA first enabled this project to start from a similar perspective as Prodrive its customers, as they currently accelerate their applications using CUDA. With the CUDA implementation optimized, an OpenCL GPU port was created starting from the source code of the algorithm. Afterwards, the optimizations used for the CUDA build are implemented, and both the CUDA and OpenCL are updated to ensure that the only code differences present are API related. With both an optimized CUDA and OpenCL GPU RabbitCT implementation, an OpenCL FPGA port is created. This FPGA port is based upon the initial unoptimized OpenCL GPU implementation after which FPGA specific optimizations are iteratively applied.

With CUDA being selected first, there is the danger for a negative bias in either direction. A negative bias could be introduced against CUDA because it is the first implementation, making it seem more difficult due to a lack of porting and optimization experience. And a negative bias might be caused against OpenCL for FPGA because the gained GPU programming experience might not be that useful for programming an FPGA where different optimization strategies are required. The effect of these biases is reduced by optimizing the Demosaic algorithm for OpenCL FPGA first, ensuring that the initial selected platform does not affect the applied optimization strategies. Additionally, by using the information found in the literature study for optimization and a result comparison, the effect of bias on the final results is minimized and a conclusion can be made.

Due to a large fire at Prodrive Technologies on the first of December 2018, the entire benchmarking setup was lost [72, 73]. The loss prevented some last finalization steps from being completed and resulted in the loss of some benchmarking results. As the project was nearly finished and because rebuilding the benchmarking setup might take several weeks, it was decided not to rebuild the benchmarking setup, meaning that not all results could be obtained. For this reason, no Demosaic CUDA implementation is created, and no power measurements could be performed the hardware. Luckily, the RabbitCT CUDA vs OpenCL results provide an insight into the expected CUDA performance, and the power measurement data can be estimated based upon the TDP of the product.

To summarize, Prodrive wants to know the feasibility of using an FPGA as an accelerator compared to a GPU. For this reason, this research was initiated, and it contains the following:

- Literature study
  - Accelerator hardware availability comparison
  - Accelerator programming method comparison
  - CUDA and OpenCL porting methods
  - Related FPGA acceleration research
  - Performance and resource utilization optimization strategies
- Algorithm analysis & implementation
- Benchmarking
  - Platforms
    - GPU with CUDA
    - GPU with OpenCL

    * FPGA with OpenCL
- Metrics
    * Maximum performance
    * Costs per performance
    * Programmability
    * Interconnect options
    * Energy efficiency
    * Product availability

Chapter 3

Literature Study

FPGAs can be used to accelerate or run many different applications, and these applications can be implemented in many different ways. The following sections provide an analysis of several research papers where FPGAs are used for acceleration. These sections explain what the hardware availability is for both the FPGA and GPU, what kind programming methods exist for FPGA and GPU acceleration, what kind of programming API porting tricks can be used, what other literature has achieved when porting applications to an FPGA accelerator, and what optimization strategies they have used to get the reported performance for both FPGA and GPU platforms. The answers to these questions provide a frame of reference and support the development choices for this research.

3.1 Accelerator availability

According to the customers of Prodrive, the main problem with GPU accelerators is the GPU life-cycle. They state that a GPU generation goes end of life quickly, requiring them to replace broken GPUs with a newer generation only several years after launch. Replacing a GPU with a unit of a newer generation requires a new development and testing process to ensure software compatibility, making it a costly process, and thus an issue that should be avoided. To determine whether FPGAs improve upon availability compared to a GPU, this section analyzes the production data and availability of multiple accelerators and searches for the oldest GPU or FPGA generation still in production. As vendors do not openly provide production data, the production data used in this section is provided by third party vendors or information sites stating the availability of a product.

For Nvidia, production information is provided by the GPU database of Techpowerup [74]. This database provides all specs of the GPUs developed by Nvidia, including their release date and production status. As of the 21st of November 2018, the oldest Quadro GPU still in production is the Nvidia Quadro K1200 which was released on the 28th of January 2015. The name of the Quadro K1200 suggests that it is part of the Kepler generation, but it actually contains a first generation Maxwell GPU, which are still in production. As no GPUs of the Kepler generation are being produced anymore, this means that the oldest Nvidia GPU generation still in production is three years old. Similar production statistics can be found for AMD, the other main GPU developer. None of AMD's 2014 FirePro professional GPUs are still in production, and the oldest FirePro card available is the AMD FirePro S9170, released on the 8th of July, 2015. Data from both AMD and Nvidia shows that GPUs are in production for roughly three years, after which the number of remaining units depends on the stock available at vendors.

Intel provides its FPGA availability on the Intel FPGA purchasing website [75]. By performing a search for the oldest FPGA generations in stock listed on their FPGA device overview [76], it shows that the oldest FPGA generation available is the MAX II series, which was introduced in

2004. The Cyclone III and Arria GX series from 2007 are also still available. However, the Stratix III series from 2006 is not on stock and thus likely discontinued. Xilinx FPGAs show an even better availability, with their main reseller Digikey indicating that the Xilinx Spartan-II from 2000 and the Virtex-II Pro FPGAs from 2002 are still being produced and are available for purchase [77]. Additionally, the Xilinx website reports that their 2006 Spartan-6 FPGAs will be supported until at least 2027 [78].

To conclude, the availability of FPGAs is significantly better than GPUs. GPUs are generally available for up to three years, after which their hardware generation is phased out. FPGAs are available for a much longer time, with 11-year-old generations still being available at Intel, and 18-year-old generations being available at Xilinx. Additionally, Xilinx states that they aim to support their Spartan-6 series FPGAs up to 21 years after release, showing their commitment to long-term hardware support.

3.2 Programming methods

FPGAs were initially programmed at register transfer level (RTL) using languages like Verilog and VHDL. These languages allow the FPGA hardware connections to be defined at the register level, which requires in-depth knowledge of the hardware to implement an application. The same used to be true for general-purpose computer programming, where programmers wrote software in assembly language. The introduction of the C language in the late 1970s changed this as it offered more efficient, portable and more readable code than what was possible at a lower abstraction level [24]. Currently, a similar change is happening for programming FPGAs, where according to [25], programming in RTL can take up to six times longer to write and test the code when compared to writing in a high-level programming language like OpenCL.

For this reason, both Intel and Xilinx have been developing high-level synthesis tools that can convert high-level programming languages into the required RTL code for programming an FPGA. Intel has created the Intel HLS tools [79], which generates RTL from C++, and Xilinx has created the Vivado HLS tools [80], which supports RTL generation from the C, C++ and System C languages. The use of these HLS tools allows FPGAs to be programmed using a high-level language. However, these tools are limited to programming the FPGA and require the user itself to develop a communication protocol for using the FPGA as a CPU controlled hardware accelerator. For this reason, both vendors have released an additional tool explicitly created for acceleration. Xilinx has developed the SDx toolset consisting of SDAccel aimed at OpenCL acceleration for PCIe accelerated devices [81], SDSoC for OpenCL acceleration on SoC devices [82], and SDNet for P4 programmed network devices [83]. Similarly, Intel has developed the Intel FPGA SDK for OpenCL toolset, which is used for all Intel related OpenCL acceleration [84].

By adding OpenCL acceleration support, both vendors allow their FPGAs to be used similarly to a GPU accelerator. GPUs are usually programmed in CUDA, OpenCL or OpenACC when used for acceleration. Here CUDA is the most popular technology for GPU programming, created by Nvidia and limited to Nvidia GPUs [26], OpenCL is a royalty-free cross-platform, parallel programming standard similar to CUDA [85], and OpenACC is a directive-based parallel programming language similar to OpenMP [27].

A comparison of the three parallel programming languages is made by joint research from the University of Bristol, University of Warwick and the UK Atomic Weapons Establishment in 2012. Here is reported that the OpenACC version offers about 24% to 34% more performance on a GPU when compared to CUDA or OpenCL [27]. However, they do report that their CUDA and OpenCL implementations are not optimized. The fact that this performance difference is caused by unoptimized code is proven by follow-up research by the University of Warwick in 2015. Here is shown that CUDA and OpenCL implementations do outperform OpenACC implementations

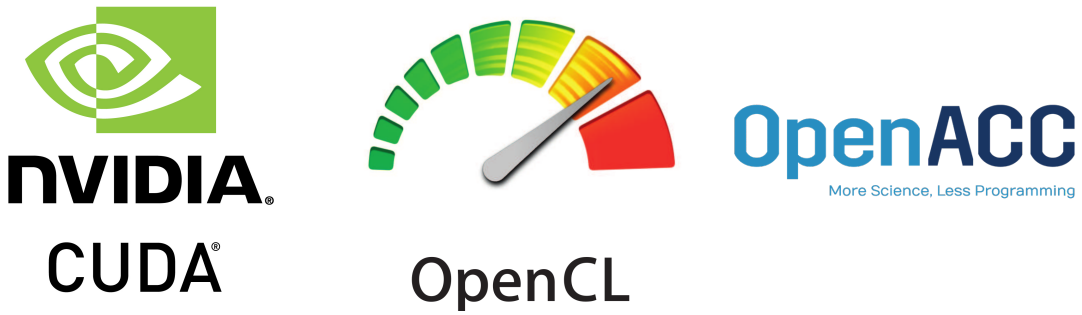


Figure 3.1: Main GPU Programming Languages

by 15% to 20% for CUDA and 10% to 20% for OpenCL [28]. A paper by the Tokyo Institute of technology in 2013 [26] also confirms that CUDA outperforms OpenACC as their fully optimized CUDA implementation outperforms an OpenACC implementation of the UPACS benchmark by 2.7 times. However, the researchers do report that their CUDA implementation contains many more line changes than their OpenACC version, which could indicate that the OpenACC version might not be fully optimized for this implementation.

The main advantage of OpenACC over OpenCL or CUDA is the fact that the standard increases programming productivity and code portability. According to joint research by the University of Bristol, the University of Warwick and the UK Atomic Weapons Establishment in 2012, their OpenACC version had about 10x less code than their CUDA and OpenCL implementations. Namely 1510 words of code for OpenACC, 17930 for OpenCL and 13.6k for CUDA [27]. The follow-up paper by the University of Warwick cites these results and states: "these performances differences are more than acceptable when offset against programmer productivity measure in the number of words of code" [28].

For compiling OpenACC code to an FPGA, a separate compiler framework has been created, namely the Open Accelerator Research Compiler (OpenARC) framework [86]. This framework, currently in Beta, allows for compilation of OpenACC, OpenMP 4, and NVL-C into a shared heterogeneous runtime that can run on CUDA GPUs, GCN GPUs, the Xeon Phi, and FPGAs. For compilation to FPGA, the compilers function to translate the OpenACC code into OpenCL is used. The generated OpenCL code is then used for compiling the FPGA binaries. Early research results by the Oak Ridge National Laboratory shows promising results [29]. Here, comparing a similar OpenACC codebase to both GPUs and FPGAs show the GPUs favouring highly parallel code and the FPGAs favouring highly pipelined code, with the GPUs being 5 to 10 times faster than FPGAs for parallel benchmarks, and the FPGAs being 20 to 30 times faster for the pipelined FFT benchmarks.

Another programming language aiming to increase productivity is Halide [87]. Halide is a programming language developed to ease the writing of high-performance image processing code. Its compiler supports the x86/SSE, ARM/NEON, CUDA and OpenCL target platforms, resulting in a highly portable codebase. As with OpenACC, there is a third-party Halide to FPGA compiler available. However, as with OpenARC, this compiler has only been developed as a research project [30]. Currently, their GitHub wiki states Vivado HLS v2015.4 as a requirement [88], making the project quite outdated as the latest Vivado toolset release is v2018.2.

Where Halide is a programming language aimed at image processing, OpenCV is a programming library aimed at computer vision and machine learning [89]. The library contains more than 2500 optimized algorithms specified for an extensive array of computer vision tasks. It has a C++, Python and Java interface and supports all leading operating systems. Currently, full-featured OpenCL and CUDA interfaces for acceleration are in development. However, research by the

National Tsing-Hua University states: the implementation is portable, but the performance still needs to be tuned for different architecture models [31], showing that the provided implementations are not optimal yet. The Federal University of Sergipe shows the same result in their paper comparing their OpenCL implementations for the VCL library with the OpenCV implementations [32]. For FPGA implementations, an official Xilinx library exists with C++ code optimized for compilation to Xilinx FPGAs [90]. A similar library exists for Intel, which recently has released the Intel Distribution of OpenVINO toolkit [91]. This SDK provides Deep learning tooling, OpenCV and OpenVX optimized code for CPUs, GPUs and their FPGA accelerator platforms.

The final analyzed programming language is SYCL. SYCL is a higher-level programming language than OpenCL, C, and C++, which also allows for the inclusion of lower-level code [92]. SYCL supports the inclusion of OpenCL, C/C++, OpenCV and OpenMP, allowing SYCL code to be used in a wide range of applications and target platforms. Due to the support for lower-level OpenCL, SYCL could be used in combination with OpenCL kernels that are compiled using Xilinx SDAccel and the Intel FPGA SDK for OpenCL. However, a 2017 paper from the University of Campinas reports that the SYCL performance is not equal to an OpenCL or OpenMP implementation. They state that for the used benchmarks, OpenCL is 2.35 to 2.77 times faster than SYCL, and OpenMP is 1.38 to 2.22 times faster than SYCL [33].

Figure 3.2: Productivity enhancing programming languages

Based on the analyzed research papers, it can be concluded that higher-level programming languages like Halide, OpenACC, and SYCL offer the same functionality as other programming methods at a significantly reduced programming effort and a somewhat reduced performance. They are also often still in the experimental phase for FPGA support with beta or research compilers being available. CUDA, OpenCL and OpenMP are mid-level programming languages for parallel programming. They offer the middle ground in programmability and performance and are supported by multiple vendors. Next, programming libraries like OpenCV exist to ease and accelerate the implementation of computer vision and deep learning algorithms in an application. However, the portability of this library hinders its performance making it unsuitable for use in our benchmarking applications.

3.3 API porting

With both OpenCL and CUDA as the selected programming APIs, the chosen applications are written using both languages. This section highlights the hardware-independent differences between both CUDA and OpenCL and analyzes whether automated porting options can help with the porting process.

A paper from the Seoul National University from 2015 reports that even though both CUDA and OpenCL are very similar, some differences remain [34]. Even when a programmer is well known with both CUDA and OpenCL, the porting process is reported to be a cumbersome and error-prone task. For host and device code translation, most code remains untouched. Even though most CUDA or OpenCL API calls look different, they perform the same task and thus can be

translated to the respective API calls from the other programming language. An example of a similar but different looking function is the OpenCL function clCreateBuffer() and the CUDA function cudaMalloc(), which both allocate a certain memory space. Exceptions for one-to-one porting exist as well. For example, the cudaMemcpyToSymbol() and cudaMemcpyFromSymbol() functions are not supported on OpenCL. These functions transfer data between the host and a device using a non-local variable located in device memory, which is not allowed in OpenCL, but is allowed in CUDA.

Other stated differences between CUDA and OpenCL are present in the executable building process. With OpenCL, the host and device code are split into multiple files. Here, the host code loads the device code and compiles this at runtime. With CUDA the device code can be placed in the same file as the host code and is compiled together with the host code before execution. Another difference is found at the kernel execution statement. CUDA kernels are launched using a single function that includes all that is needed to execute a kernel: what kernel, what parameters, the size of the grid, the size of the thread blocks, and the size of the dynamic shared memory per thread block used by the kernel. OpenCL splits the kernel argument declaration into separate lines with the clSetKernelArg(), to set the arguments for the kernel, and clEnqueueNDRangeKernel() to launch the kernel with the given size parameters and event info.

The final differences stated in [34] are language-specific features. CUDA supports the inclusion of C++ features in the kernel, while OpenCL supports this functionality starting from OpenCL version 2.1. As not all hardware supports OpenCL 2.1, support for this functionality is not guaranteed. Another language specific difference is the supported vector formats. OpenCL supports two-, three-, four-, eight-, and sixteen- component vectors, while CUDA supports one-component vectors but lacks the eight-, and sixteen-component vectors that OpenCL supports. CUDA also supports more hardware specific functions as CUDA is limited to Nvidia hardware, enabling Nvidia to implement more specific control into the API. CUDA also supports a unified address space, allowing memory to be addressed by both the host and device. This functionality is present in OpenCL starting from version 2.0.

A paper by the University of Tennessee Knoxville and the University of Manchester indicate similar differences between CUDA and OpenCL in the naming of functions [35]. Additionally, the researchers report that the OpenCL compilation method is similar to CUDA. However, where OpenCL compilation can be performed by multiple compilers, CUDA can only be compiled by the CUDA toolkit as it is closed source. Figure 3.3 shows both compilations flows. It shows that for Nvidia devices to support OpenCL, OpenCL support has to be built into the CUDA toolkit, which converts the kernel code into the PTX intermediate representation, allowing both languages to use the same compilation back-end. Additionally, Figure 3.3 shows that OpenCL can be compiled using multiple compilers, using a compiler specific intermediate code representation to generate the device binaries.

The paper by the University of Tennessee Knoxville and the University of Manchester also shows a runtime breakdown of an OpenCL kernel [35]. For their application, the execution of the kernel took only 6.9% of the total runtime. The rest goes to kernel compilation, kernel creation and kernel data copy functions. Avoiding kernel creation and copy overhead is not possible. However, the latencies can be hidden by creating a streaming kernel that is continuously processing data and copying data in parallel with the kernel execution. The kernel compilation time can and should be avoided to affect the benchmark when benchmarking on a GPU, as this would result in an unfair comparison with CUDA as CUDA applications are compiled prior to execution. This issue can be resolved using offline compilation for OpenCL instead of the standard online compilation. However, this can also be resolved by solely benchmarking the kernel execution time instead of benchmarking the total application execution time.

The report also includes a performance comparison for the GEMM matrix multiplication algorithm where the OpenCL implementation reaches similar performance levels as CUDA when the texture

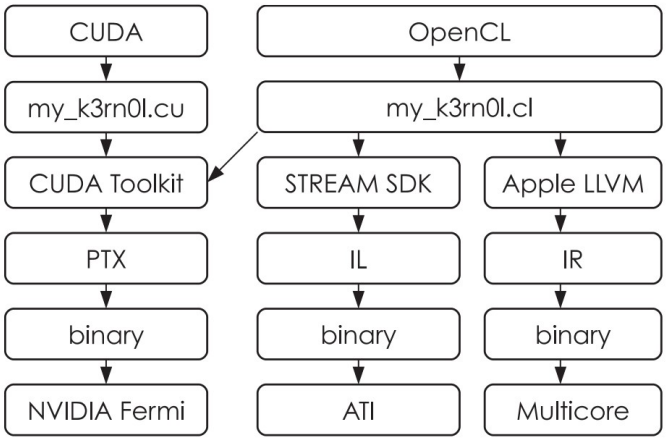
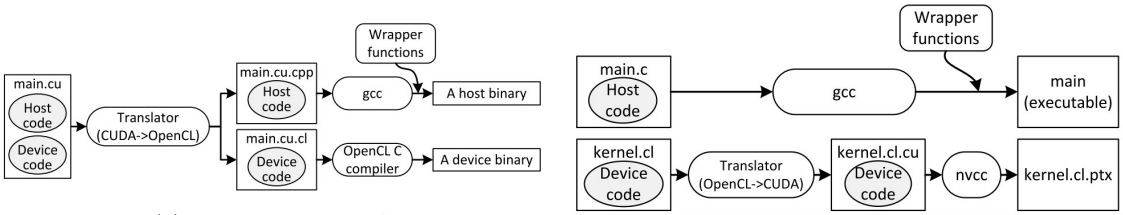


Figure 3.3: OpenCL and CUDA compilation flow, taken from [35]

cache is not used. With CUDA texture caching enabled, the OpenCL implementation performs 5% worse on average, allowing CUDA to perform better due to a reduction in memory fetching overhead.

All in all, several differences exist between CUDA and OpenCL, and porting code between APIs is a time-consuming process. For this reason, the researchers from the Seoul National University have developed a tool for porting code between OpenCL and CUDA [34]. Their tool translates the kernel code directly and uses code wrappers for porting the host code between languages, see Figure 3.4. The porting tool has been benchmarked using the Rodinia benchmarking suite [36]. When porting from OpenCL to CUDA, an average performance difference of about 3% is reported for all 20 benchmarking applications when ran on an Nvidia GTX Titan. When porting from CUDA to OpenCL, seven out of the 21 benchmarks failed due to CUDA specific code. However, the applications that successfully have been ported from CUDA to OpenCL resulted in an average performance difference of 0.3% when compared to the CUDA source code, and an average difference of 0.2% when compared to the OpenCL source files of the same algorithm. This average performance difference indicates that API porting between CUDA and OpenCL on a GPU can be performed with automatically almost no performance loss.



(a) OpenCL to CUDA

(b) CUDA to OpenCL

Figure 3.4: CUDA OpenCL porting, taken from [34]

Swan is a tool that allows programs to execute using both CUDA and OpenCL [37]. Instead of using wrappers like the example from [34], Swan provides an API with similar functions to CUDA and OpenCL. Based upon the selected target API, Swan processes the code and converts the Swan functions to the corresponding OpenCL or CUDA functions in pre-compilation. This conversion allows for support of both APIs without requiring any changes to the source code when the application is written using Swan library functions.
The reported benchmarks show that when using Swan to port CUDA code to OpenCL, the OpenCL code is about 50% slower than the CUDA version on an Nvidia Tesla c1060 GPU. The paper links

this slowdown to the source code initially being CUDA optimized and that the Nvidia OpenCL compiler produces less optimized PTX intermediate representation code. The additional latency Swan introduces to kernels was also measured by launching a thousand single threaded kernels and measuring the launch latency. This measurement shows that Swan decreases the launch latencies by 400 to 500 ns with CUDA and increases the launch latency by 200 to 300 ns for OpenCL. Proving that Swan does not introduce a significant additional latency for converting the Swan code into the respective OpenCL or CUDA code. Another observation made in [37] is that the OpenCL launch latency is up to 9x higher than the CUDA launch latency. This additional latency will hurt the performance of programs with short-duration kernels when using Swan to target OpenCL devices.

The final CUDA-to-OpenCL porting tool analyzed is one by the Department of Computer Science from Virginia Tech and is called CU2CL [38]. Instead of using wrappers or acting as an intermediate API, this tool converts CUDA source code to OpenCL source code and does not initiate any compilation steps. By analyzing the abstract syntax tree of the original code, a string to string conversion from the CUDA to OpenCL functions is performed. A direct string to string conversion does not always work; sometimes parameters have to be created, moved or converted in order to convert between languages successfully. Examples of this are expressions with unsupported datatypes or expressions where one API requires a pointer as a parameter, and the other does not.
The main difficulty in porting source code from CUDA to OpenCL was reported to be producing maintainable or readable code. Other issues are the rewriting of macros or the use of closed-source libraries, as these macros or libraries still expect the original CUDA data as input. Fixing these issues requires complex rewriting of the macro or the inclusion of a functionally equivalent library in OpenCL, which is not always available. Benchmarks of the original CUDA, a manually created OpenCL implementation, and a converted OpenCL implementation are performed on an Nvidia GTX 280 GPU. Here, the automatic OpenCL converted code has a similar performance to the manually converted OpenCL code. However, the original CUDA implementation was still between 2% and 32% faster depending on the benchmark performed.

3.4 FPGA acceleration research

FPGA acceleration is used for many different computational tasks. These tasks range from small accelerated sections of a larger program to applications where the entire algorithm runs on the FPGA. This section analyzes several research papers and reports their findings on FPGA acceleration. The algorithms used in these papers all fall in the category of high-performance computing and are be divided into three main categories: cloud acceleration, image processing and other HPC examples. For each analyzed implementation, the application itself is explained, and any performance, programmability, energy efficiency, and life cycle results found are noted.

3.4.1 Cloud acceleration

Recently, FPGA accelerators have been introduced to each Microsoft cloud server module to reduce the ever-increasing load on the CPU [39, 40]. The FPGAs are connected via two 8x PCIe connections to the server and via a 40Gbps QSFP connection to the network. These connections allow the FPGA to act as a local accelerator, where the local server accelerates applications; as a network accelerator, where the network data is pre-processed removing load from the CPU; and as a global accelerator, where the FPGA receives commands from other FPGAs on the network to help them process a particular load. Here, global acceleration also helps to prevent issues with failing FPGAs, as this allows tasks to be offloaded to another FPGA if no response is received within a certain interval. Figure 3.5 shows an overview of the system.
The FPGAs used in the cloud server modules are Intel (Altera) Stratix V D5 FPGAs with 4GB of

DDR3-1600 memory. These FPGAs are not the fastest and largest FPGAs available at the time of publishing [93]. However, by combining thousands of mid-range FPGAs in a single network, Microsoft has developed an acceleration network that significantly reduces the server CPU loads while only slightly increasing the server power usage. The introduction of FPGA accelerators into their cloud architecture allows the throughput to increase by 95% with a fixed latency or it can reduce the latency by 29% when aiming for a fixed throughput for the Bing webpage ranking [39]. So, significant improvements have been made despite the FPGAs only increasing the power usage of the server blades by about 35W per FPGA in a worst-case scenario [40].

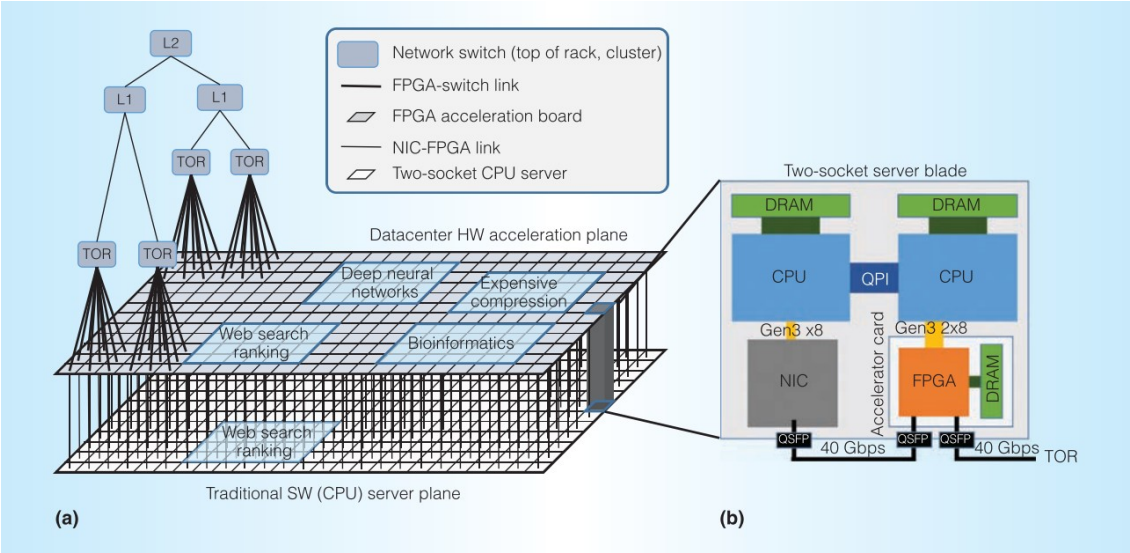


Figure 3.5: Cloud-Scale Acceleration Architecture, taken from [40]

In 2012, the National University of Defence Technology had performed research on the acceleration of another cloud-based architecture [41]. Here, the MapReduce framework for massive data processing is ported to an HDL FPGA based cloud architecture. The framework is split into CPU-based and FPGA-based workers that work together to solve specific tasks. The CPU workers receive the input from a user application and send this task to one of its available CPU or FPGA resources. As soon as the processing on one of these resources has finished, the CPU receives the output data and sends it to its destination. This CPU based dispatching framework makes the acceleration highly scalable as more accelerators allow more work processing in parallel.
Matrix multiplication and page ranking algorithms are used to test the MapReduce framework. The matrix multiplication benchmark is reported to run at least ten times faster than "commodity" hardware when using a single FPGA with a single mapper and reducer implementation. However, what commodity hardware is used as a reference not explained. When implementing multiple pipelines on this FPGA, the performance increases linearly. The same holds for adding a second FPGA to the acceleration framework. With the addition of a third FPGA, an Ethernet bottleneck is reached as the FPGAs cannot receive enough data from the CPU to scale the performance linearly. Final performance reports show that the page ranking benchmark runs 3.94 times faster on three FPGAs when to using unknown commodity hardware.

Similar MapReduce research was performed in 2010 by the Tsinghua National Laboratory for Information Science and Technology [42]. However, instead of writing the framework in HDL, these researches used a high-level synthesis method as well. For synthesizing the C code into RTL, the AutoPilot tool from AutoESL [43] is used. As of February 2011, Xilinx has acquired AutoESL and merged their tools into the Vivado toolset [94]. For benchmarking an Altera Stratix II FPGA is used. The manually written HDL version achieved a speedup of 33.5x for the Rankboost page ranking algorithm when compared to a software implementation on a CPU, whereas the

HLS generated implementation achieved a speedup of 31.8x. These similar performance numbers indicate that High-Level Synthesis can achieve a performance close to manually written code. Together with the increased productivity high-level synthesis offers, this makes HLS programming of FPGAs a worthy replacement to manually writing HDL.

A 2016 survey by the University of Athens summarizes other FPGA cloud acceleration research results [44]. Here, they analyze 15 papers FPGA cloud architecture papers for their results. Next to the MapReduce and Search engine applications that are shown above, this survey also reports on other cloud-based algorithms that are accelerated using an FPGA with either HDL or HLS. Namely, results for database analytics, the Apache Spark framework and the MemCached programming framework are analyzed as well. Of all the 15 papers analyzed, a speedup of 1x up to 31.8x the CPU performance is reported for all but one FPGA acceleration implementation. Namely, one of the four MemCached programming framework implementations reports a slowdown. However, the survey reports that for cases when the speedup is low, the implementation was created aiming for energy efficiency and not for maximum throughput. Reading the paper corresponding to the research with a decrease in performance verifies this reason. Here, the researchers aim to port the MemCached framework, which is generally used with high-end hardware, to a low-power SoC, so a performance decrease was expected [45]. Additionally, all analyzed papers in this survey reporting the efficiency numbers show an energy efficiency increase between 2.6 and 33 times when compared to the CPU energy efficiency.

3.4.2 Image processing

FPGA acceleration is also often used for image processing tasks. For example, in 2017, the Technical University of Dresden has used the AKAZE feature detection algorithm for their comparison of OpenCL performance on different hardware platforms [46]. The AKAZE algorithm detects features in images by filtering the image in several ways, revealing and hiding image features with each filtering step. The detected features can be a particular angle, a line, or a sudden change of colour which are often used for computer vision. These features allow the computer to understand a scene, categorize objects, and or create a 3D structure from the motion present in an image set [47]. For benchmarking, the researchers used a Xilinx Virtex-7 FPGA, and an Nvidia GTX 780 GPU in combination with an Intel Core i7-4770k processor to determine the performance of OpenCL implementations for each platform. The benchmarking results show that the FPGA OpenCL implementation is reported to be 1.47x faster and about 10.6x more power efficient than the GPU OpenCL implementation.

The University of Florida has performed another image processing research in 2016. Here they used a Canney Edge detector, a Sobel filter and a SURF feature extractor for comparing HDL vs OpenCL on FPGAs [25]. All these algorithms take an image as an input and use it to create a specific output data. The SURF feature detection algorithm performs a similar task as the AKAZE algorithm described above. The Canney Edge detector and Sobel filter both filter the image to detect edges in the image and highlights them on the output image which is used to detect objects in the image. Their paper reports that creating the HDL kernel took six months while creating the OpenCL kernels to a similar performance took one month. The final results show that the performance between the OpenCL and HLD implementations is within 10% of each other with no clear winner. The only difference between the generated and handwritten kernel was that the OpenCL implementation uses 70% of the FPGA resources, while the VHDL kernel uses 59% of the resources. So, the researchers conclude that OpenCL FPGA kernel generation offers significantly higher productivity at the cost of additional area usage for achieving similar performance to manually writing the HDL code.

3D image reconstruction is another example of an often-used FPGA accelerated image processing task. A study by the University of California in 2015 has compared the performance of an OpenCL

FPGA application with a CUDA GPU application [48]. They used a 3D reconstruction benchmark that creates a model of an object or environment by using depth information from a camera or depth sensor. Here, two kernels are ported from CUDA to OpenCL for execution on a Stratix V FPGA. The Iterative Closed point kernel is used for object tracking and subsamples the objects. Its original CUDA implementation takes 1.4ms to complete on an Nvidia GTX 760 GPU. A one to one port to FPGA takes 49.9ms, and a fully optimized FPGA implementation takes 3.22ms to complete, showing that a one to one port of the GPU implementation to FPGA will not lead to high performance, and FPGA specific optimization is required to achieve optimal performance.
The Volumetric Integration kernel integrates takes the subsampled objects from the previous kernel and uses the depth information to create a single representation of the volume. This kernel takes 4ms to run on the GPU but takes 100ms when optimized for the FPGA. The reason for this significant slowdown is a memory bottleneck, as each iteration performs operations on 512 MB of data with little reuse to allow for optimizations. So, for each iteration data is transferred back and forth between the FPGA and the host at 15 GB per second, which is the bottleneck of this kernel. Luckily, the available memory bandwidth has significantly improved during the last couple of years, with both Intel and Xilinx releasing a variant of their FPGAs High Bandwidth memory (HBM2) chips on the FPGA substrate [49] [50]. Intel reports that these HBM2 chips can supply a total of 256 GBps bandwidth per chip, increasing the total bandwidth by more than ten times per HBM chip on the die when compared to Stratix V FPGA used in this research.

In 2017, the University of Paris-Saclay also ported a 3D image reconstruction algorithm to OpenCL for FPGA. However, this one is not one for processing depth information, but one for creating a 3D model from a large number of 3D Tomography pictures [51]. Here, a backprojection algorithm similar to the one used in the RabbitCT algorithm is benchmarked on both a GPU and an FPGA. This algorithm takes a set of X-ray images taken from multiple angles around the patient. These images are then used for back-projecting the X-ray image to the light source, which results in a 3D model of the scanned object. For benchmarking, an Nvidia Titan X Pascal and an Intel Cyclone V FPGA with an ARM Cortex A9 processor and 1 GB of DDR3 memory are used. The original CPU single work item benchmark without any optimization took 222.9 seconds to complete, while the best performing FPGA kernel using multiple work items and two compute units took 16.9 seconds.
As the used Cyclone V FPGA is a low power, low clock, and small-sized FPGA from 2011, and the GPU is the best and largest consumer GPU available in 2017 no fair comparison can be made. For this reason, the researchers try to map the gained Cyclone V performance linearly to the expected Intel Arria 10 SX660 performance for a fairer comparison. Here, based on the Cyclone V area usage, they map multiple parallel variants of the Cyclone V implementation to the Arria 10 SX660 and expect the performance to increase linearly with the number of parallel implementations. This linear interpolation leads to an estimated FPGA performance of 991ms, which is 82.6 times slower than the Titan, which only requires 12ms to complete the benchmark. However, by linearly mapping the performance, the researchers ignore FPGA architecture enhancements like DSP units for floating point calculations [52] and a significantly higher clock frequency. For this reason, the conclusion not accurately predicts the performance and efficiency of a real Arria 10 SX660 implementation, making their benchmark comparison useless.

3.4.3 Others examples

A 2016 white paper by Berten Digital Signal Processing analyzed both GPU and FPGA hardware on many aspects, including their performance per euro, performance per watt, and the ease of development [53]. The paper determined their results on facts about the selected hardware meaning that they have not performed any benchmarking themselves. The price per performance was determined by dividing the FPGA price by the total theoretical floating point performance, the performance per watt was determined by dividing the floating point performance by the TDP, and the ease of development was based upon the according to general knowledge stating that

FPGA development is hard and developers are harder to find. Their results show that the FPGA latency is reported to be an order of magnitude faster than the GPU, the FPGA is reportedly 3 to 4x more energy efficient, the FPGA enables the interface of choice to be used, and the reduced power requirement allows an FPGA to fit in a smaller size. Next, the GPU is reported to have a higher floating point processing performance, a higher performance per costs, an easier development process, higher backward compatibility compared to FPGA code in HDL, and higher flexibility as all GPU functionality is available at all times whereas the FPGA needs to be reprogrammed. Additionally, the paper shows that the price per performance is highly dependent on the hardware; thus the results cannot be used for our comparison. However, it can be stated that the price per performance for FPGAs is higher than GPUs according to this research, with the FPGAs being between 3 to 50x more expensive than the tested GPU based upon their floating point performance.

In 2016 the Tokyo Institute of Technology had researched the performance and power efficiency of an implementation of six high-performance-computing benchmarks in OpenMP on CPU, CUDA on GPU and OpenCL on FPGA [54]. All six benchmarks come from the Rodinia benchmarking suite [36] [95]. Here, DNA sequencing, thermal simulation, pathfinding, diffusion calculation, linear equation solving, and a CDF 3D Euler volume solver application are tested on each platform.
For benchmarking, the researchers use a Stratix V A7 FPGA, Nvidia Tesla K20c GPU and an eight-core Xeon E5-2670 CPU. The worst-case result was achieved using the CDF solver benchmark, which computes a 3D Euler volume using single-precision floating point computation. Here, the performance of the FPGA is 11.5 times slower than the GPU and is 0.47x as efficient. The lousy performance and efficiency are reported to be caused by a sub-optimal implementation and the Stratix V FPGA not containing floating point units, hindering the performance of the floating-point heavy implementation. Currently, both Intel and Xilinx have resolved this issue as their latest FPGA generations contain DSP units that accelerate floating point calculations [96] [97].
The next worst performing algorithm is the Pathfinder algorithm which is a kernel that tries to find a path with the smallest accumulated integer weight from the bottom to the top of a 2-D grid. Here, the FPGA performs the algorithm 5.28 times slower than the GPU, but the FPGA is 1.3x more efficient. The main reason for slowdown here is the lack of on-chip memory, limiting the number of exploitable parallelisms.
The best performing benchmark is the Needleman-Wunsch (NW) benchmark that sequences DNA. Here, the FPGA is only 1.48x slower than the GPU but 3.36x more efficient. The reason the NW benchmark performs the best is likely caused by the algorithm only using integer numbers and each computation only requiring data from neighbouring matrix elements. This data usage allows for a more efficient pipeline implementation inside the FPGA than other benchmarks. Another interesting fact this research provides is the performance of code optimized for a GPU running on an FPGA versus FPGA optimized code executing on an FPGA. Here, the researchers achieved a maximum speedup of 133.7 when optimizing an algorithm for FPGA execution when compared to running the original GPU optimized code on the FPGA. The smallest speedup gained by FPGA specific optimization occurs with the CDF 3D Euler volume solver, where only a speedup of 1.28 is reported when compared to the original GPU code running on the FPGA. All other tested algorithms perform between 10.5 to 66 times better when optimized for FPGA execution. More details on code optimization techniques are shown in Section 3.5.

Research by IEEE members in 2017 has also benchmarked several HPC applications using OpenCL for both GPU and FPGA acceleration [55]. Using an Nvidia GTX 960, an Nvidia Quadro K4200 and a Xilinx Virtex-7 690t FPGA, several algorithms are benchmarked. The K-Nearest Neighbour (KNN), the Monte Carlo (MC), and the Bitonic Sorting (BS) algorithm are ported to each platform. Here, KNN is used for pattern recognition in computer vision, and machine learning; MC is used for solving complicated mathematical and physical problems, and BS is one of the fastest known sorting networks. For benchmarking, the researchers initially also planned to benchmark CUDA applications, but after porting the KNN algorithm to CUDA, no significant performance differences were observed between CUDA and OpenCL on a GPU. Due to this similar perform-

ance, they dropped the CUDA comparison and continued with OpenCL applications only.
The benchmarking results show interesting data. Namely, an initial KNN implementation indicated that the FPGA was 1.4 times slower than the GPU. However, with a second implementation where the researchers mapped a part of the off-chip global memory to on-chip BRAM blocks, the FPGA was a factor 2.47 times faster than the initial GPU implementation. This performance difference indicates how significant a bottleneck the off-chip memory can be for the FPGA performance. Using the same optimization technique on a GPU does not offer better performance, as the GPU implementation using on-chip global memory slowed down from 3.04ms to 930ms. Next, four Monte Carlo implementations show a performance increase between two and five times on an FPGA when compared to a GPU, and the Bitonic Sorting algorithm shows that the FPGA is 9.5x slower than a GPU using a non-optimized implementation and that the FPGA is 1.06x times slower with optimized code. Which again shows the importance of FPGA specific optimizations. As for power efficiency, all optimized FPGA implementations were between 1.8 and 80 times more energy efficient than the respective GPU implementations. The researchers conclude that the FPGA performance beats that of a GPU when FPGA specific optimizations are made. Especially optimizations to reduce global memory accesses will increase the FPGA performance significantly. On energy efficiency, FPGAs beat GPUs as FPGAs contain a hardwired control structure, making them much more efficient. Finally, the paper states that FPGA specific optimizations require a comparable amount of effort as optimizing a GPU implementation.

The acceleration of training a recurrent neural network-based language model is another high-performance-computing example. This research was performed in 2012 by the University of Pittsburgh [56]. This recurrent neural network-based model is created for language processing and operates in the time domain as it has to capture dependencies of input data over one or multiple input sentences. The main issue with this network is the time required to train it and the power usage that comes with it. For this reason, acceleration methods were researched to find an optimal platform for training the network. Here, an Intel Xeon E5-2630 was used together with an Nvidia GeForce GTX 580 and a Convey HC-2ex FPGA. The GPU was programmed using an Nvidia CUBLAS (CUDA Basic Linear Algebra Subroutines) implementation based upon research by the Tsinghua University [57]. Here, the FPGA is programmed using System C, which is converted to Verilog by the Convey Hybrid Threading HLS tool. Even though Convey Computers does not exist anymore as Micron has acquired it [98], the result is still valid as both System C and OpenCL code are written in the C language and converted to an RTL specification language. For the neural network training benchmark, the FPGA performed 1.46x slower than the GPU, but at a 6.7x higher efficiency, continuing the higher energy efficiency but somewhat slower performance trend for FPGA acceleration.

A more recent example of neural network acceleration on FPGA is reported in a study of the University of Wisconsin-Madison in 2017 [58]. Based on previous neural network acceleration work on FPGAs they determined the memory bandwidth is the main bottleneck. For this reason, the researchers propose a new kernel that optimally balances the computation, on-chip, and off-chip memory accesses to reduce the memory bandwidth requirements. Their kernel achieved a performance of 866 GFlops/s and 1790 Gops/s. As the researchers have not compared their results to current GPU neural network solutions, no solid performance or efficiency results can be made. However, when compared to the latest graphics architecture from Nvidia this FPGA solution is still at least 16 times slower as the top-of-the-line Quadro RTX 6000 or 8000 GPUs, which both are rated for 16.3 TFlops/s on single precision [99]. However, as Nvidia uses these numbers for marketing, the real-world GPU performance is likely lower than what they report.

A final example comes from the Amazon Web Services (AWS) with their Elastic Compute Cloud (EC2) F1 FPGA cloud acceleration service [59] [100]. This web service offers virtual machines with one or multiple Xilinx Virtex Ultrascale+ FPGAs, allowing customers to take advantage of the speed and efficiency of an FPGA without having to purchase any hardware and having to install the tooling and drivers. Amazon provides the use of the Xilinx SDAccel tool to allow the FPGAs to be programmed using either a high-level programming language like C, C++, OpenCL

or a low-level programming language like Verilog. When an operational Amazon FPGA image is created, it can be executed on their F1 FPGA hardware, and it can be sold on the Amazon Web Service marketplace. So, with the AWS-EC2 F1 service, Amazon provides a development and testing environment together with a store to create an entire FPGA development ecosystem. For this research, the Amazon web services will not be used as Prodrive does not want to be limited to the hardware provided by Amazon.

3.5 Optimization strategies

Multiple papers report their optimization strategies for their target platforms. Here, a small summary of each reported optimization is given together with an explanation of how this optimization helps for executing code on the device.

3.5.1 FPGA OpenCL

The main optimization strategy for optimizing OpenCL for an FPGA is unrolling loops in the code. Most papers reporting their optimization strategies report using this technique [25, 30, 59, 46, 48, 51, 54, 55]. Unrolling a loop exposes parallelism to the compiler enabling it to repeat this short section of code multiple times, resulting in a pipeline of loop sections. This pipeline not only increases the application throughput but it also increases the application latency because all pipeline stages are limited by the slowest stage [1].
Further pipeline improvements are made automatically depending on the present data dependencies. If no data dependencies are present between stages, the compiler can parallelize the loop allowing multiple sections to be updated in parallel, reducing the total number of pipeline stages and reducing the application latency while keeping the pipelined throughput. However, in [55] is mentioned that too many loop unrolls can hamper the performance as the available number of global memory ports is limited, causing data access conflicts resulting in code serialization. So, an optimum between parallelization and pipelining has to be found.

When using OpenCL acceleration on FPGAs, data is stored in two types of memory banks: on-chip memory and off-chip memory. These memory banks are split into multiple software defined sections with different latencies, sizes and access rights, see Figure 3.6. Data stored in the global and constant memory is located in off-chip memory. This memory has a large address space, but as it is located off-chip, also the highest access latency. The local memory sections reside on-chip in the FPGA BRAM memory blocks, and the private memory is located in FPGA shift registers. Both these on-chip memories can be accessed with low latency, but are limited in size.

For processing data, a sliding window approach in combination with shift registers or line buffers will increase data re-use and hide memory latencies. These processing techniques are used in several papers [25, 30, 46, 48, 51, 54]. The line buffers or shift registers are implemented by unrolling a for-loop without data dependencies. When the compiler detects this code structure, it will implement a shift register to move the data along the pipeline. By implementing multiple of these data paths, a sliding window approach can be used to process the data, as each pipeline stage can access the previously loaded data without requiring additional memory accesses.

The next optimization method does not optimize the kernel itself. The use of multiple compute units allows the same kernel to be implemented multiple times in the FPGA hardware [46, 54, 51, 55, 58]. Implementing multiple compute units allows for parallel kernel execution at the cost of more FPGA resources. However, this also increases memory usage, which might cause a lower overall performance due to memory conflicts. So, even though the compute power is available, using compute units might not always lead to better performance due to memory bottlenecks.

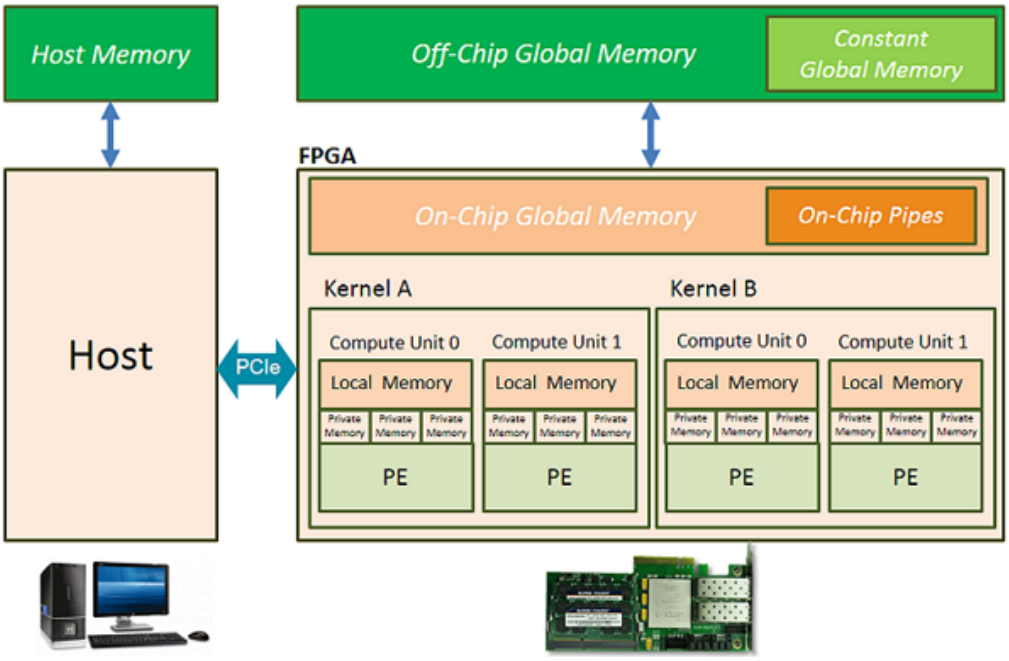


Figure 3.6: OpenCL FPGA memory model, taken from [3]

Another often used method is the use of OpenCL pipes [25, 46, 48, 55]. These pipes are FIFO buffers between kernels that allow kernels to communicate without the host PC managing the communication. Direct communication between kernels significantly reduces the communication overhead and allows multiple kernels to execute as a pipeline, see Figure 3.7. When using OpenCL pipes, the use of vectorization, which is explained below, is prohibited.

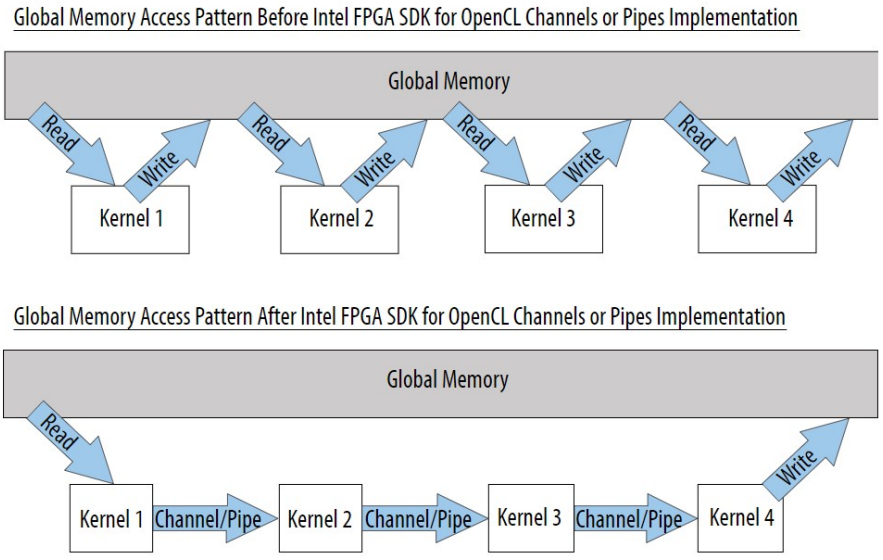


Figure 3.7: OpenCL pipe implementation, taken from [1]

The introduction of vector or SIMD (Single Instruction, Multiple Data) instructions is also an often used optimization step [46, 54]. The use of vector or SIMD instructions results in the data path of a kernel to be implemented multiple times in the hardware. Compared to multiple

compute units, which duplicates the entire kernel, vectorization only duplicates the datapaths and shares the control logic, making vectorization more resource efficient. Having a vectorized kernel allows multiple iterations to be executed in parallel, significantly increasing the computational performance, see Figure 3.8.

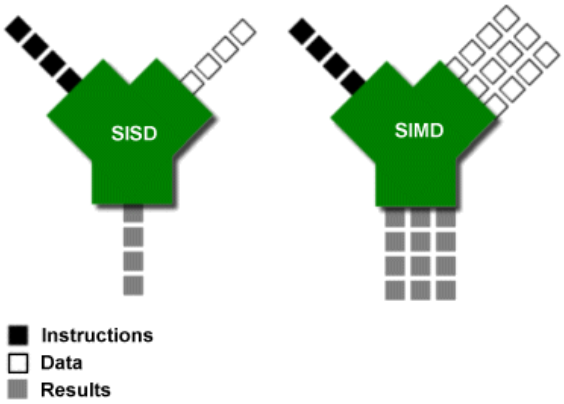


Figure 3.8: SIMD visualization, taken from [101]

Splitting large kernels is another technique reported in [25, 46]. Here, the splitting of the kernel allows these separate kernel tasks to operate in parallel. As an example, a large kernel can be split into a kernel handling the communication with the host PC, and a kernel performing the computations. By splitting up these tasks and combining it with OpenCL pipes, the communication kernel makes sure the computational kernel can continue its work.

The paper by the University of California reported that instead of splitting kernels, merging two data-dependent smaller kernels can increase the performance as well [48]. The merging of these kernels removed a memory bottleneck as this removes the need to transfer data to the DRAM in between kernel executions. The use of OpenCL pipes might also have resolved this issue.

Research by IEEE members and by the Politecnico di Milano mention some other memory management examples not used in other papers [59, 55]. In both research papers, Xilinx FPGAs are used and suggest performing global memory accesses in bursts to reduce the memory access overhead. Next, they mention partitioning the memory by using dedicated memory ports for each global array, reducing the total memory access conflicts. By partitioning the data, the data is divided differently over the on-chip memory, allowing different elements to be accessed in parallel. Figure 3.9 shows the SDAccel partitioning options. Here, block partitioning divides the array into equally sized blocks of consecutive memory elements of the original array, which allows data from multiple blocks to be accessed in parallel. Cyclic partitioning also splits the original array into equally sized blocks, but now interleaves the data over multiple BRAM blocks, which allows serial data to be accessed in parallel. Finally, complete partitioning splits all data into separate memory elements, allowing all data to be accessed in parallel. Finally, the researchers also suggest implementing an on-chip global memory on BRAMs as a buffer for inter-kernel communication to avoid the excessive transfers to the off-chip memory.

Another massive advantage of FPGA acceleration is the more efficient implementation for branching code [55] [102]. If a GPU contains a branch in a parallel section of the code, this section has to execute sequentially due to the parallel execution paths in a compute-unit only supporting a single instruction at a time. For an FPGA branching is less of a problem as all paths are established in hardware and thus does not require serialization. However, all outcomes have to be present in the hardware. So, non-merging branching paths cause the FPGA hardware usage to increase significantly as each branch then requires its separate processing hardware.

The paper by the joint research of the University of Pittsburgh, the Tsinghua University, and

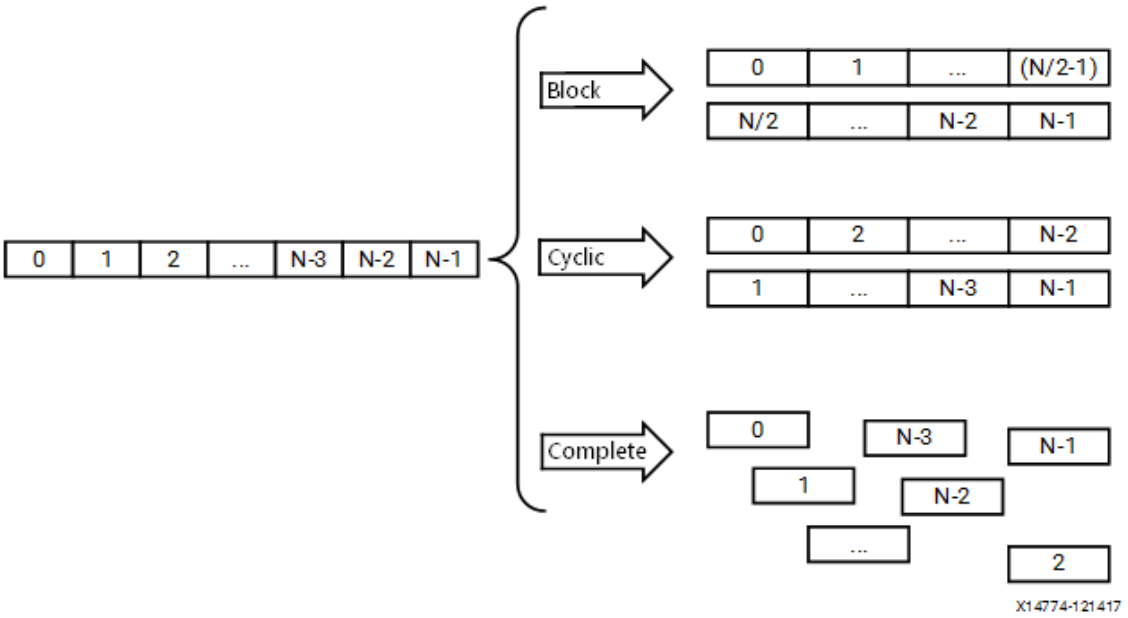


Figure 3.9: Partitioning methods, taken from [4]

the Syracuse University on Neural Network acceleration on FPGA [56] reported implementing a thread manager to manage their memory, as their FPGA had only 26 MB of memory available. Next, they also used the mixed-precision data formats that allow for FPGA area saving when not all bits of a variable are in use.

Data type management is an optimization that is required as floating point operations have a lower performance than fixed-point operations on Xilinx FPGAs [60]. Here Xilinx reports that floating point implementations require more hardware resources which lead to increased power consumption and latency when compared to fixed-point. This statement is proven when comparing the performance and resource utilization tables with the Xilinx Ultrascale+ architecture for fixed-point [103] and floating-point [104] functions, see Table 3.1. The hardware usage is dependent on the required precision, as half-precision reduces the hardware requirements, and double precision increases the requirements when compared to single-precision. Intel reports similar performance changes for their FPGAs [105]. So, for an optimal implementation, calculations should use the least number of bits required to get the desired result, and a fixed-point data format should be used where possible. The Xilinx HLS tool flow also offers arbitrary datatypes, which lets the users define the required bit size for each variable. However, this functionality is not supported for OpenCL kernels as arbitrary sizes are not part of the OpenCL spec [5].

Table 3.1: Xilinx Ultrascale+ data type comparisons for the addsub unit

Datatype	Fmax (MHz)	LUTs	FFs	LUT-FF Pairs	DSP48s	BRAM
32-bit fixed point	833	1	0	0	1	0
16-bit half precision	724	93	227	80	2	0
32-bit single precision	724	208	309	136	2	0
64-bit double precision	680	630	952	526	3	0

A final FPGA performance optimization technique is the addition of RTL code to expand the functionality of the OpenCL kernels [6]. As OpenCL kernels transfer data using standard FPGA interconnect, it allows the kernels to be connected to DMA engines, I/O peripherals, memory controllers, custom interconnects, other OpenCL compute units, and RTL based accelerators via

the high-speed serial connectors [106]. With all these interconnect options, OpenCL kernels can be used for many more tasks than just accelerating data coming from the host PC.

Other optimization examples can be found in the official Intel FPGA SDK for OpenCL Programming Guide [7], Best Practices Guide [1], and the Xilinx Vivado HLS Optimization Methodology Guide [8], SDAccel Environment Programmers Guide [5], and the SDAccel Environment Profiling and Optimization Guide [4].

3.5.2 GPU CUDA & OpenCL

As both CUDA and OpenCL are similar APIs controlling the same GPU device, the optimization techniques are similar as well. For this reason, the optimization techniques for both APIs are combined in this section.

The primary method for porting code from a single-threaded application to a parallel GPU implementation is by computing separate loop iterations in parallel GPU threads. This technique is used by all the analyzed papers with a GPU implementation [26, 27, 31, 46, 57, 61, 62]. The code requires to have no data dependencies between loop iterations to use this optimization. The more data dependencies that are present, the more serialized the code will execute.

A GPU contains two memory banks, off-chip memory and on-chip memory. These memory banks are split into multiple sections creating the memory model shown in Figure 3.10. Here, the local and shared memory is located on-chip, and the global, constant and texture memory off-chip. The main difference between on-chip and off-chip memory is the access latency and storage size. On-chip memory can be accessed very quickly but is limited in size. While off-chip memory has a high access latency, but also has a larger address space, allowing it to store significantly more data than on-chip memory. Additionally, there is the host memory, which has the most storage space, but also the highest access latency of all memory in the system.
Several research papers claim to fill the global GPU memory with as much data as possible to remove the host-GPU latency from affecting the execution speed [26, 27, 57]. In case new data is needed during kernel execution, a separate memory manager is used to handle additional host-GPU memory transfers to make sure all data is available when required.

Other optimization methods can be applied to hide the memory latencies present when copying data from off-chip GPU memory to on-chip GPU memory. Here, research by the Technical University of Dresden [46] used the read-only cache of the on-chip memory as a line buffer for processing images in a sliding window fashion. From this cache, the image data is copied to the thread registers for quick access to the required pixels. Other techniques include kernel and loop fusion, where multiple kernels or loops are merged to reduce the number of global memory accesses present in the code [26, 31]. Shared memory blocking is a technique where threads execute in lockstep which ensures that all threads read and write data at the same time [26]. Using the shared memory reduces the number of global memory accesses, and executing in lockstep prevents thread race conditions where input data from another thread might overwrite the data of the current thread.

Branch hoisting is used to prevent warps from executing sequentially; this occurs when branching occurs in parallel paths [26, 57]. This optimization is required as all threads in a thread block have to execute the same instruction at a time. If a branch occurs where some threads branch differently than others, parallelization is lost as each thread block can only execute a single set of instructions. So, if a branch occurs, two sets of instructions are received, which result in a serialized execution of the code. So, by moving the branches outside of a loop, the branch occurs before the parallel section of the code allowing the loop sections to be executed in parallel no matter the result of the branch.

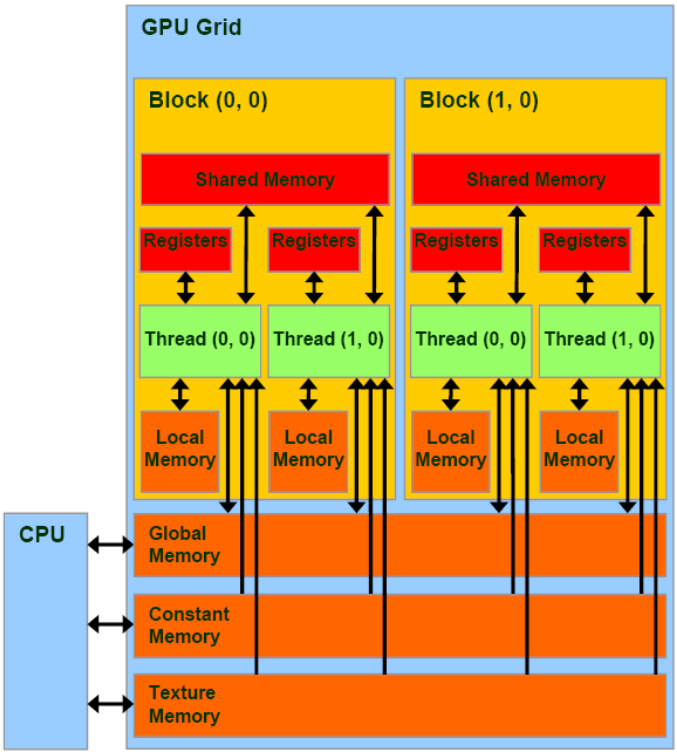


Figure 3.10: CUDA GPU memory model, taken from [107]

Another method to improve the warps on a GPU is warp formatting [31, 57]. Warp formatting requires the code to be adapted to make use of the maximum threads executing in parallel. A CUDA warp or an OpenCL workgroup is a set of GPU threads running concurrently on the same GPU multiprocessor. A multiprocessor uses a single instruction to control multiple CUDA cores, which increases the possible data parallelism in the hardware. By warp formatting, the code is adapted to make use of all the CUDA cores in a streaming processor and thus maximizing the hardware usage. So, if 250 threads are processed on a GPU with 32 threads per SM, but the code only allows up to 25 threads to be processed simultaneously, a total of 10 warps are required. However, by creating thread blocks with 32 threads, only eight warps are required to perform the same computation.

The memory access patterns used in an algorithm also have a significant effect on the performance of an algorithm [63]. Figure 3.11 shows several memory access methods. From these memory accesses, linear coalesced memory accesses will generally result in the best results, as it allows a large array of data to be fetched with a single load. The more displaced or random the memory accesses are, the more individual memory fetches that are required as mostly a single array of neighbouring data can be fetched in a single cycle. Predictability also helps, the more predictable the data access, the easier it is to implement data prefetching, which makes the memory objects available before they are needed and thus hides memory latencies.

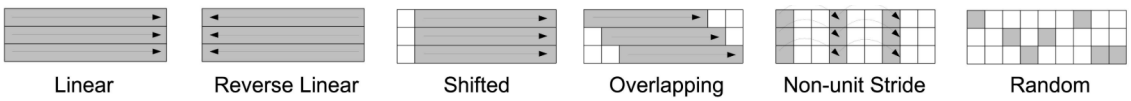


Figure 3.11: Memory access patterns, taken from [63]

Advanced use of the GPU hardware is suggested by researches from IEEE [61] and the Friedrich-

Alexander University [62]. Here, they loaded their image data as 2D textures, which allows the bilinear interpolation circuitry of the GPU to be used, increasing the performance for this specific calculation and offloading the GPU cores for other tasks.

Other improvement methods are the replacement of low throughput operations with multiple higher throughput operations for example, the replacement of a divide function with a multiplication with the inverse. This technique is used in [61], where a divide function is replaced by a fast-square root technique. Here, the bits in a floating-point number are exploited, allowing the inverse to quickly be calculated, resulting in the division being replaced by a multiplication with the inverse of the original number.

According to the Nvidia developer blog, defining a global memory element as restricted can significantly increase the memory performance [108]. Defining a memory element as restricted tells the compiler that no pointer alias is present. Having no pointer alias means that the memory addresses accessed using a pointer do not change between multiple reads. This enables the compiler to cache the global memory, reducing the number of global memory accesses required and speeding up the algorithm.

Changing the data types used in the code also affects performance. The Nvidia programming guide [9] reports that their latest Turing Quadro GPU generation(compute capability 7.5) offers increased throughput for calculations with a lower precision while reducing the throughput of calculations with a higher precision over their Pascal Quadro GPU generation (compute capability 6.1). As shown in Table 3.2, 32-bit calculations have half the throughput of 16-bit calculations, and 64-bit calculations have only a sixteenth of the throughput of 32-bit calculations. Note that Nvidia's table shows a throughput of 32 for 64-bit double-precision calculations, but it contains a side note that the throughput is only 2 for compute capability 7.5 GPUs, which includes all their Turing consumer and business Quadro GPUs. So, in order to take advantage of the highest throughput, all data should be processed with the minimum precision required for that calculation.

Table 3.2: Nvidia compute capability 7.5 data type comparisons for add, multiply, and multiply-add commands

Datatype	Throughput per cycle per multiprocessor	
	Compute capability 6.1	Compute capability 7.5
32-bit integer	128	64
16-bit half precision	2	128
32-bit single precision	128	64
64-bit double precision	4	2

Another optimization is the introduction of buffers between multiple kernels in a similar fashion to OpenCL channels on FPGA [31]. By using buffers between kernels, multiple kernel tasks can be executed in parallel, allowing for simultaneous computation and communication.

A final optimization is the use of multiple GPUs that share their data using a high-bandwidth interface: the Nvidia Link [109]. Having more GPUs processing the data allows for faster computation. Additionally, the use of the Nvidia link bridge allows the GPUs to communicate and share their memory. However, the Nvidia bridge does not offer additional external bandwidth; it only serves to reduce the bandwidth for inter-GPU communication.

Other CUDA and OpenCL programming techniques can be found in the Nvidia CUDA C Programming Guide [9], the Nvidia OpenCL Best Practices Guide [10], the Nvidia OpenCL Programming Guide [11], and the AMD OpenCL optimization guide [12].

3.6 Literature study conclusion

This section combines the results of the literature study and shows an expectation for the final results of this project based on the analyzed literature.

3.6.1 Accelerator availability

A product availability research comparing the availability of GPUs and FPGAs shows that a certain generation of GPUs is produced up to three years after release, while currently, FPGAs released up to 18 years ago are still available. Additionally, Xilinx states to support their Spartan-6 FPGA up to 21 years after release. So, the availability of FPGAs can be generalized to up to 20 years, whereas GPUs are only produced up to 3 years, making FPGAs a clear winner in availability.

3.6.2 Programming methods

GPUs and FPGAs can be programmed using multiple programming languages. GPUs are most often programmed in C or C++ using CUDA or OpenCL, and FPGAs are most often programmed in RTL using VHDL or Verilog. For programming accelerators, CUDA and OpenCL are most used programming APIs. Higher level languages or libraries like OpenACC, Halide or OpenCV exist as well, each improving code portability and programmability at the cost of performance.

The primary goal of this research is to determine whether FPGAs are a valuable replacement for GPUs used by Prodrive its customers. CUDA is chosen for programming the GPU as Prodrive its customers are currently using it to accelerate their applications. Next, OpenCL is selected for programming the FPGA as it is similar to CUDA and supported by all main FPGA vendors for high-level synthesis. Additionally, OpenCL is reported to increase the productivity up to six times for a similar performance when compared to programming in RTL. So, choosing OpenCL minimizes the FPGA porting effort between both APIs, which helps any customers with porting their applications, and it helps us speed up our research. Another reason to select OpenCL is that Nvidia GPUs also support OpenCL, which allows for an API comparison between CUDA and OpenCL on a GPU.

3.6.3 API Porting

Porting applications between CUDA and OpenCL should be a straightforward task as most functions have a representative in the other API. Only when using certain CUDA specific functions, some additional code has to be written to implement that functionality. Several automated API porting tools are available. Of the analyzed tools, only the CU2CL tool is useful for this project as this tool generates OpenCL source code from CUDA source code, which is required for generating the FPGA kernel. All other analyzed tools either provide a new API or include wrapper functions to their source code, and thus do not provide clean OpenCL source code that can be used to generate the FPGA kernel.

3.6.4 FPGA acceleration

FPGAs can be used to accelerate a wide array of applications. Of the analyzed papers, the accelerated algorithms can be divided into three categories. The largest real-life use case for

FPGA acceleration is cloud acceleration. Here, multiple FPGAs are used to reduce the load on the CPUs from large data centres. These FPGAs can be programmed to accelerate page ranking, to help with the processing of large databases, to perform network pre- and post-processing steps, or it can act as an additional memory cache for big data applications. Image processing is another area in which FPGA acceleration is often used. Multiple pipelines are implemented in the FPGA to execute the required tasks on a set of images. For example, these tasks could include feature extraction, edge detection, and 3D image reconstruction which allows the images to be used for computer vision. The final category consists of all other papers reporting on FPGA acceleration that used algorithms that do not fit into the cloud computing or image processing domains. These HPC applications include algorithms for DNA sequencing, thermal simulation, pathfinding, diffusion calculation, linear equation solving, 3D Euler volume solving and neural network training.

Most of the analyzed FPGA acceleration papers have performed benchmarks of some kind. FPGA acceleration vs CPU execution show results that either decrease the latency by 29%, or a throughput up to 31.8x the original throughput is be achieved. Comparing FPGA HDL with FPGA HLS shows performance numbers within 10% of each other with no clear winner. Next, it is stated that programming FPGAs in HLS is approximately six times faster than programming the FPGA in HDL to achieve similar performance. Additionally, programming in HDL is more resource efficient, with an HDL implementation taking 59% of the FPGA resources, and the HLS implementation taking 70% for a similar performance. Next, papers comparing both CUDA and OpenCL on a GPU report only minor differences between API's, indicating their similarity on the same hardware.
The OpenCL FPGA vs GPU results show mixed results. The FPGA performance is reported to be anything from 20x slower to 2.47x faster for executing a particular algorithm. However, all slower FPGA executions show a common theme. A lack of memory bandwidth or a lack of floating point units on older hardware is reported to be the main issues, making the algorithm execution speed algorithm dependent. These issues should become less of an issue with newer FPGA generations, as each generation improves upon memory bandwidth and modern FPGAs contain DSP units optimized to accelerate floating point processing. The analyzed papers also report that special FPGA optimizations are required to create efficient code, which can gain up to 133.7x the original GPU code performance when executed on the FPGA. These FPGA specific optimizations are reported to require a similar effort as GPU specific optimizations when porting code to run on a GPU. However, all research with an FPGA performance higher than 0.1x the GPU performance reports the FPGA to be more energy efficient than the GPU for their algorithms.
Finally, only a single paper reported on the costs per performance of FPGA acceleration. This paper only used the theoretical hardware performance facts together with the price to determine the costs per GFLOP of the FPGA and the GPU. Here, the FPGA costs reportedly between 3x and 50x more per GFLOP than the GPU. These large variations are caused by different hardware being analyzed, showing that the costs per performance ratio is highly hardware dependent. Additionally, FPGAs are known to perform worse than GPUs on floating point math, especially on the older generation of FPGAs used in this research, making these results inaccurate.

3.6.5 Conclusion

In this report, the value of FPGA and GPU acceleration is compared based on six factors: maximum performance, performance per costs, programmability, interconnect options, energy efficiency and product availability. Based upon the previously performed research, it can be concluded that FPGAs are about ten times more energy efficient than GPU accelerators, and they can be anything from 20x slower to 2.5x faster than a GPU depending on whether memory bottlenecks are present. FPGA specific optimizations are required to achieve a high FPGA performance when programming with OpenCL. These optimizations are reported to require a comparable effort as GPU specific OpenCL optimizations. However, when compared to programming on CUDA, more words of code are required for the same functionality. Next, where GPUs only contain a PCIe

interface and video I/O ports, FPGAs offer many more interconnect options due to the generic I/O ports that can be connected to OpenCL kernels using RTL. The product availability of FPGAs is significantly better than GPUs with FPGAs reportedly being available for up to 20 years, and GPUs being available for three years. Finally, the performance per costs was analyzed by only one paper that did not perform any benchmarks. Here, the FPGAs were reported to cost 3x to 50x more for the same floating point performance than a GPU. However, as this paper has not performed any benchmarks and thus uses theoretical performance numbers, this result is likely inaccurate. A summary of all metrics can be found in Table 3.3.

Table 3.3: Literature study results

Metrics	**GPU**	**FPGA**
Performance	1x	0.05x to 2.5x
Costs per performance	1x	3x to 50x
Programmability	equal	equal
Interconnect options	limited	extensive
Energy efficiency	1x	10x
Product availability	3 years	20 years

Chapter 4

Algorithm Analysis

The value OpenCL acceleration brings to the GPU acceleration market is analyzed using both the RabbitCT and Demosaic algorithms. These algorithms have been selected for their relevance for Prodrive Technologies, as their customers found these algorithms computationally representative for the algorithms they want to implement. This chapter analyzes the selected algorithms to determine how they work and how they are currently implemented. Afterwards, the current implementation is thoroughly analyzed on the data flow, data types, data sizes, and the memory access patterns to determine the main issues that need to be tackled in order to optimize the algorithm for acceleration on a GPU and FPGA.

4.1 RabbitCT

RabbitCT (RCT) is an all-in-one benchmarking application with the goal to set up a competition in developing the most efficient backprojection implementation for a 3-D cone beam reconstruction algorithm [110]. It is based on the backprojection algorithm proposed by FDK [64], which forms the basis for most CT-cone beam image processing algorithms [22]. FDK suggested a method where the raw CT scan input data is first filtered before applying a back-projection algorithm. This backprojection algorithm is the most computationally intensive section of the 3-D cone beam reconstruction [111] and thus requires optimizations for quick execution. RabbitCT offers an environment for users to take a simple backprojection example and optimize the execution as much as possible. It provides pre-processed input data of a CT scanned rabbit, see Figure 4.1, and it provides benchmarking results like the average processing time and the accuracy of execution.

4.1.1 Theoretical analysis

This section analyses the theoretical background of the RabbitCT algorithm. Here, a short explanation is given on the backprojection method and the built-in benchmarking functions.

Backprojection

The RabbitCT backprojection algorithm computes a 3d voxel space based upon a set of X-ray images taken from the subject with a C-arm CT scanner. The provided RabbitCT benchmarking input data consists out of 496 images with a resolution of 1248 by 960 that are taken around a Rabbit along a 200-degree axis [22]. For each of the voxels in the output model, an X-ray beam is back-projected to determine which input pixels correspond with the selected voxel. With backprojection an X-ray beam is traced from the X-ray source, through each voxel of a voxel space representing the object onto an input image received from the detector, see Figure 4.2. At the

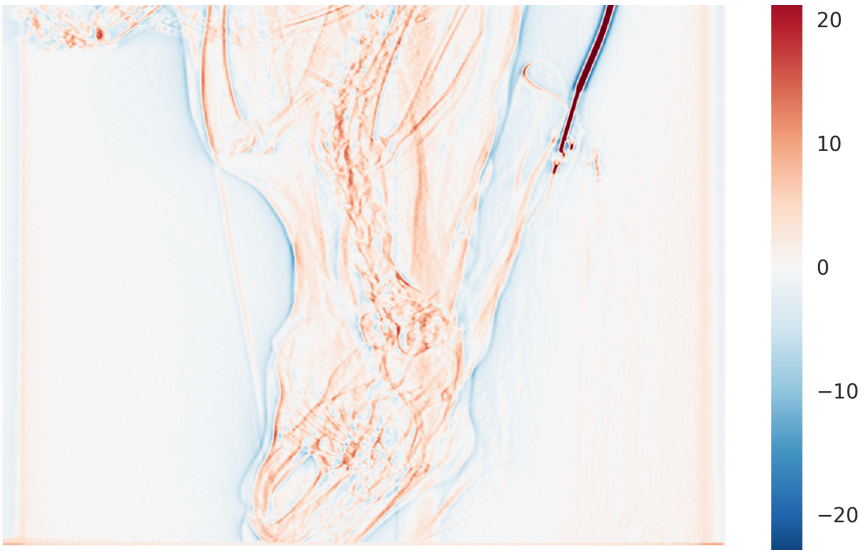


Figure 4.1: Heatmap of RabbitCT input image #100

intersection of the backprojected X-ray with the input image, the neighbouring pixels are read and bilinearly interpolated. Afterwards, this value is normalized and applied to the voxel from which the X-ray was backprojected.

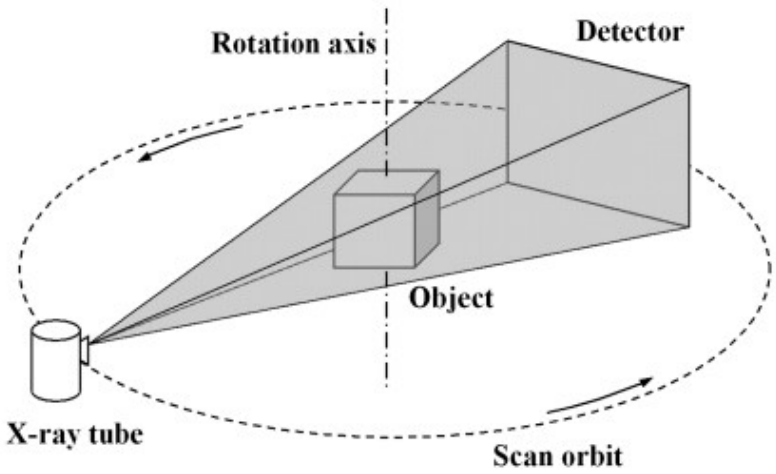


Figure 4.2: 2D backprojection example, taken from [112]

Without pre-filtering the input data, a superposition of all backprojection images leads to the blurred image shown in the top row of Figure 4.3. This blurring is caused by the backprojection input data being a 2D snapshot, that is mapped to a 3D space and summed over all input images results in a smeared out image. This effect is shown in Figure 4.4, where a top or bottom snapshot of a 3D sphere contains a circle, which results in a cylinder when applied to all voxels in the third dimension. Figure 4.3 also shows this effect in the top left image with a 1D snapshot being smeared out in a 2nd dimension. So, when summing over all input images, the object becomes visible, but the smearing leads to blurred edges. By pre-filtering the input data with a high-pass filter, the backprojected images highlight only the edges of the photographed object, resulting in a much sharper output image, see the bottom row of Figure 4.3. The benchmarking data provided by RabbitCT is already pre-filtered, so, for RabbitCT, only the backprojection step has to be implemented and optimized for execution on the desired hardware.

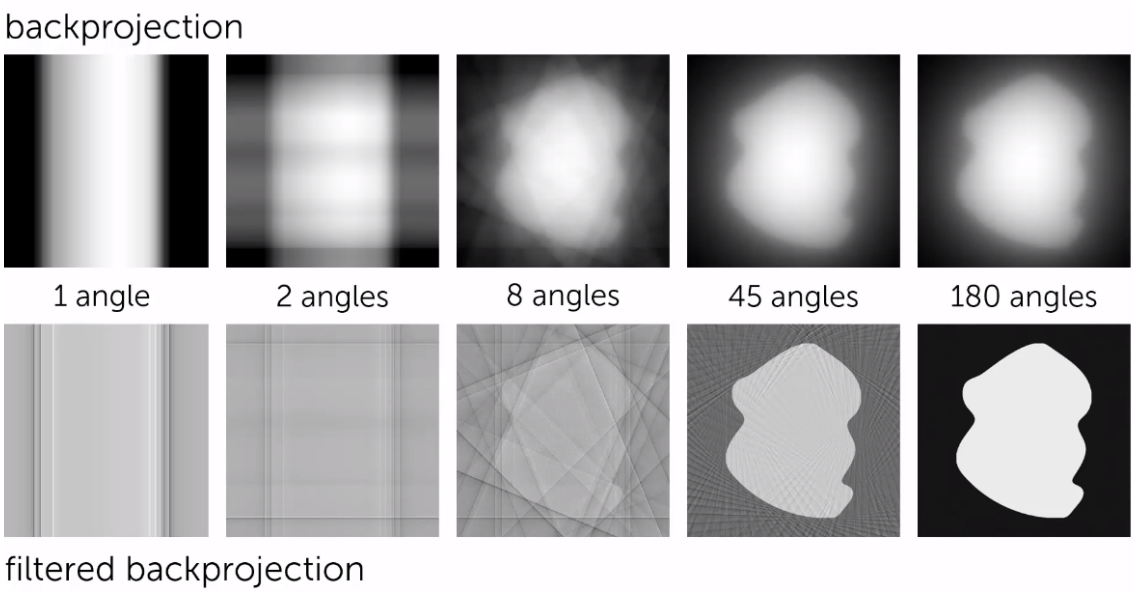


Figure 4.3: Normal vs filtered backprojection, taken from [113]

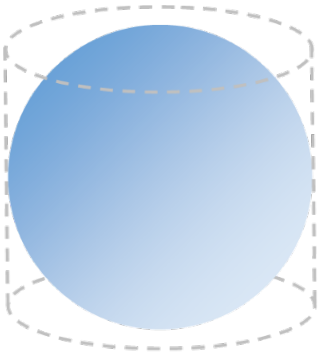


Figure 4.4: 2D to 3D perspective, taken from [114]

The RabbitCT backprojection algorithm is a discrete version of the algorithm proposed by FDK [64]. The algorithm results in a Cartesian 3D voxel space $f(x, y, z)$, and is calculated as shown in Equation (4.1).

$$f(x,y,z) = \sum_{n=1}^{N} \frac{\hat{p}_n(u_n(x,y,z), v_n(x,y,z))}{w_n(x,y,z)^2}, \tag{4.1}$$

where

$$u_n(x,y,z) = \frac{a_0 x + a_3 y + a_6 z + a_9}{w_n(x,y,z)}, \tag{4.2}$$

$$v_n(x,y,z) = \frac{a_1 x + a_4 y + a_7 z + a_{10}}{w_n(x,y,z)}, \tag{4.3}$$

$$w_n(x,y,z) = a_2 x + a_5 y + a_8 z + a_{11}, \tag{4.4}$$

and

$$A_n = \begin{pmatrix} a_0 & a_3 & a_6 & a_9 \\ a_1 & a_4 & a_7 & a_{10} \\ a_2 & a_5 & a_8 & a_{11} \end{pmatrix}. \tag{4.5}$$

.

Here, $f(x, y, z)$ represents a voxel at position x,y,z with respect to the origin. Each voxel is calculated by summing all normalized interpolated pixel values corresponding to the voxel position overall N input images. In this equation, $\hat{p}_n$ represents the interpolated pixel value, calculated using the normalized column and row values u_n and v_n; and w_n represents is the normalization factor. A_n is a pre-calibrated projection matrix given for each projection image. It projects the voxel positions of the 3D object data onto the 2D projection images [22].

The interpolated pixel values $\hat{p}_n(x, y)$ are calculated by performing a bilinear interpolation with a zero-boundary condition on the four pixels surrounding the intersection of the backprojected beam with the input image I_n [22]. The bilinear interpolation is calculated by:

$$\hat{p}_n(x,y) = (1-\alpha)(1-\beta)p_n(i,j)+\alpha(1-\beta)p_n(i+1,j)+(1-\alpha)\beta p_n(i,j+1)+\alpha\beta p_n(i+1,j+1), \tag{4.6}$$

where

$$i = \lfloor x \rfloor, \tag{4.7}$$

$$j = \lfloor y \rfloor, \tag{4.8}$$

$$\alpha = x - \lfloor x \rfloor, \tag{4.9}$$

$$\beta = y - \lfloor y \rfloor, \tag{4.10}$$

and

$$p_n(i,j) = \begin{cases} I_{n,i,j} & if\ i \in \{0, ..., S_x - 1\} \wedge j \in \{0, ..., S_y - 1\} \\ 0 & otherwise. \end{cases} \tag{4.11}$$

$p_n(i, j)$ represents the pixel data of the current input image n from the image matrix I when the requested pixels are within the resolution boundaries; otherwise, it returns 0. For the RabbitCT provided benchmarking data, $S_x = 1248$, and $S_y = 960$, representing the number of pixels in the column and rows of the input image.

Built-in benchmarking

For benchmarking a given backprojection algorithm, RabbitCT provides the total execution time, the average execution time of processing a single CT scan image, and several algorithm quality metrics [110].

The reported program runtime measures only the actual backprojection algorithm. All pre-processing and host data management does not count towards to program runtime, removing any disk access latencies from affecting the performance of the algorithm, leaving only the performance of the actual algorithm on the selected hardware to affect the results.

The quality metrics are determined by comparing the calculated backprojection output with a pre-calculated result of the provided LolaBunny backprojection source code. This reference (f_L^{ref}) is compared with the output of the provided backprojection algorithm (f_L). The mean squared error is calculated according to:

$$q_{mse}(f_L) = \frac{1}{L^3} \sum_{i,j,k} [f_L(i,j,k) - f_L^{ref}(i,j,k)]^2 \tag{4.12}$$

A peak signal-to-noise ratio q_{psnr}, measured in decibels (dB) is also provided. It is calculated with the following equation:

$$q_{psnr}(f_L) = 10 \log_{10} \left(\frac{4095^2}{q_{mse}(f_L)} \right) \tag{4.13}$$

By analyzing the provided image quality metrics, the correctness of the RabbitCT implementation can be determined. Here, the higher the peak signal to noise ratio and the lower the mean squared error, the more similar the current output is to the reference output.

4.1.2 Algorithm implementation

The RabbitCT source application consists of two parts, the RCT-Runner, and the RCT-Algorithm. The RCT-Runner is the main application that provides an environment for the RCT-Algorithm to run in. It loads the benchmarking data into a global data struct, initializes the memory, performs the benchmark, provides the benchmarking results and cleans up. A pseudocode example of RCT-Runner is provided in Listing 4.1 which is based on the pseudocode example provided in the paper by the Friedrich-Alexander University [22].

```
input:
    L;     // Output resolution
    S_x;   // Image width
    S_y;     // Image height
    A_n;   // Projection matrix
    I_n;   // Image buffer
    R_L;   // Voxel size
    O_L;   // Origin

output:
    f_L;      // Voxel reconstruction
    t_total;  // Total execution time
    t_avg;    // Average execution time
    q_mse;    // Mean squared error
    q_psnr;   // Signal to noise ratio

// Define global struct
struct RabbitCtGlobalData{
  unsigned int  L;     // Output resolution
  unsigned int  S_x;   // Image width
  unsigned int  S_y;   // Image height
  double *      A_n;   // Projection matrix
  float *       I_n;   // Image buffer
  float         R_L;   // Voxel size
  float         O_L;   // Origin
  float *       f_L;   // Voxel reconstruction
}

// Initialize data
struct RabbitCtGlobalData RCTdata = initStruct(L, S_x, S_y, A_n, I_n, R_L, O_L);
t_total = 0;

// Initialize algorithm
RCTLoadAlgorithm(RCTdata);

// Run RCT-Algorithm for all input images
for ( int n = 0; n < Nmax; n++) {
```

```
    t = currentTime( );
    RCTAlgorithmBackprojection(RCTdata);
    t_total = t_total + currentTime( ) - t;
}

// Perform final processing steps
RCTFinishAlgorithm(RCTdata);

// Calculate benchmarking stats
t_avg = t_total / N;
q_mse = q_mse(f_L);
q_psnr = q_psnr(f_L);

// Free all allocated data and exit
RCTUnloadAlgorithm();
```

Listing 4.1: RabbitCT-Runner pseudocode

The RCT-Algorithm is the section of the code that performs the backprojection algorithm. Here, the provided LolaBunny algorithm loops over all voxels in three-dimensional Cartesian space and calculates the value of the voxel for each position. Afterwards, the data of the voxel is used to update a variable representing the volume of the object, which is the output of the algorithm. The pseudo code for the RabbitCT-Algorithm can be found in Listing 4.2 and is also based on the pseudocode written in [22].

```
input:
    I_n, // Image buffer
    A_n, // Projection matrix
    L,   // Problem size
    O_L, // Origin
    R_L, // Voxel size
    f_L  // Previous voxel reconstruction
output:
    f_L  // Updated voxel reconstruction

// For each voxel of the output grid
for (int i=0; i < L; i++) {
    for (int j=0; j < L; j++) {
        for (int k=0; k < L; k++) {
            // Calculate voxel coordinates based upon origin and voxel size
            x = O_L + i*R_L;
            y = O_L + j*R_L;
            z = O_L + k*R_L;

            // Update the volume
            f_L(i,j,k) = f_L(i,j,k)
        }
    }
}
```

Listing 4.2: RabbitCT-Algorithm pseudocode

4.1.3 Implementation analysis

A glance at the provided code quickly shows the hotspot of the application. With the provided benchmarks, the RCT-Runner calls the backprojection algorithm 496 times as there are 496 images to be back-projected. For each image, the backprojection algorithm runs three nested for loops for either 128, 256, 512, or 1024 iterations depending on the user-defined output resolution. These iteration counts lead to a total of $532.6e9$ ($496 \cdot 1024^3$) or 532.6 billion iterations of the backprojection algorithm when running the largest benchmark. When using the source code, all

the iterations are executed sequentially on the CPU; which takes a long time to finish. By parallelizing or pipelining the code, multiple iterations are being processed in parallel which, assuming no bottlenecks are introduced, linearly speeds up the algorithm with the number of parallel paths. This section analyzes the backprojection algorithm on its data and memory access patterns, to devise an optimal implementation for both acceleration platforms.

Data flow

Figure 4.5 shows the data flow of the source implementation of the provided LolaBunny backprojection algorithm. RabbitCT-Runner calls the algorithm for every image and provides the necessary image parameters to calculate the output data. This image data is constant and is reused for every voxel calculation of an image, allowing the image and voxel calculations to performed in parallel.

A single iteration of the backprojection algorithm calculates the value for one voxel for a single input image. Each step in the backprojection algorithm requires data from the previous step, preventing the parallel execution of multiple backprojection steps for the same voxel. However, by pipelining these backprojection steps, the throughput can still be increased as different steps of multiple voxel calculations are then executed concurrently. During the backprojection calculation, normalized pixel positions have to be determined. These positions indicate where an X-ray beam from the source through this voxel intersects the input image. The four pixels surrounding the intersection point are loaded from memory for the bilinear interpolation, after which the output voxel value can be calculated, and the next voxel or image calculation is started. As four pixels are loaded per voxel, there is data re-use present. Here, smart caching solutions could mediate memory access latencies. The output voxel value is an array shared between images and voxels. So, for a particular voxel, sequential access to this array is required to prevent race conditions from corrupting the data. A more in-depth memory access analysis is performed in Section 4.1.3.

Data types

As not all data types offer the same performance on all platforms, an analysis of the used datatypes provides an insight into the expected performance of these platforms.

When looking at the per-image provided input data to the backprojection algorithm, the following struct is provided:

```
unsigned int L;   ///< problem size {128, 256, 512, 1024}
unsigned int S_x; ///< projection image width
unsigned int S_y; ///< projection image height (detector rows)
double *     A_n; ///< 3x4 projection matrix
float *      I_n; ///< projection image buffer
float        R_L; ///< isotropic voxel size
float        O_L; ///< position of the 0-index in the world coordinate system
float *      f_L; ///< pointer to where the result volume should be stored
```

Listing 4.3: RabbitCT backprojection data

The input data shows that the algorithm uses unsigned integer, double and floating-point variables. Additionally, all backprojection calculations use double precision except for the output f_L, which is in single precision. So, almost all calculations are performed using 64-bit floating point, which offers a significantly lower throughput on a GPU, and significantly higher resource usage on FPGA, as is indicated in the optimization strategy analysis in Section 3.5. As the final output is a single-precision floating point array, the algorithm can be adapted to perform all calculations in single-precision with minor accuracy loss. Additionally, all variables except the image buffer I_n, matrix A_n, and the output f_L are constant, so they can be removed from the struct to reduce the data usage between input image iterations.

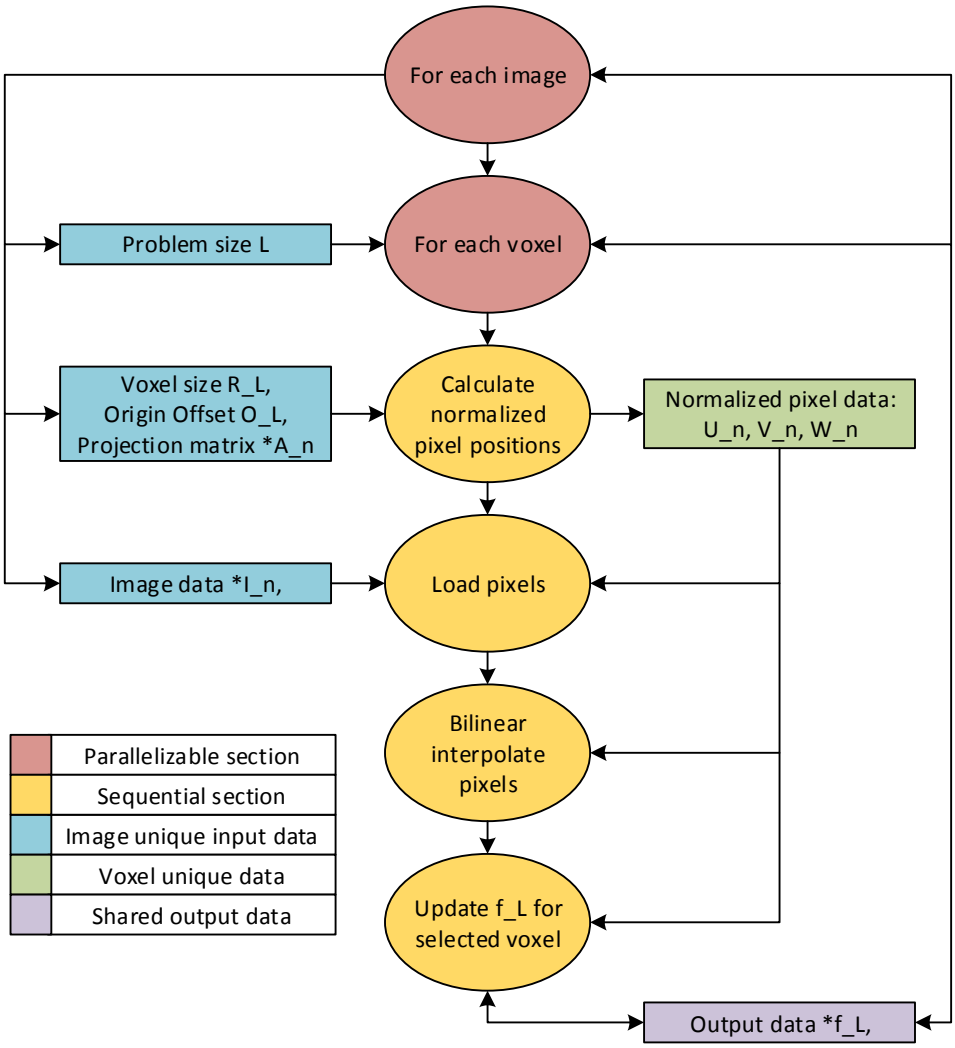


Figure 4.5: RabbitCT Dataflow

Data sizes

An analysis of the data sizes used for the RabbitCT algorithm can show what the memory size and PCIe bandwidth requirements are. An individual input image consists of 1248x960 floating point values. Together with the additional image parameters required, a total of 4.8 MB per image is transferred to the accelerator. Assuming the output variable f_L resides on the global memory of the accelerator and is only copied back when the algorithm is finished, a PCIe 3.0 16x bandwidth of 15.75GB/s allows up to 3281 images to be transferred from the host to the accelerator per second. So, when using a streaming implementation of the algorithm, the algorithm will at least take 0.151 seconds to copy all 496 input images to the device. However, when looking at the RabbitCT ranking [115], the fastest multi-GPU implementation took 0.3 seconds to process all images in 2016. So, this PCIe input bottleneck has not been reached yet. Next, intermediate data created and used in the backprojection calculation consists of five integers, one floating point and ten double-precision variables. Which requires about 104 bytes of data and should easily fit in the on-chip memories of both the GPU and FPGA accelerators as those chips contain multiple megabytes of data.

Finally, the size of the output variable depends on the performed benchmark. As shown in Table 4.1, at the lowest benchmarking resolution only 8.4 MB of output data is generated by the algorithm, while the benchmark with the highest resolution generates 4.3 GB. Because RabbitCT updates all data in the output variable for each image calculation, quick access is required. So, the entire output variable should always be located in the global memory of the FPGA. With these large data sizes, the selected FPGA should contain at least 5GB of global memory data to ensure the entire output array plus the input data can be accessed from device memory. If 5GB RAM is not available, the global array should either remain on the host; with the host performing the final summation of the voxel data, or a separate data handler will have to be written to ensure the correct data resides on the FPGA memory at all times.

Table 4.1: RabbitCT output size

Benchmark resolution	Output size in MB
128	8.389 MB
256	64.11 MB
512	537.8 MB
1024	4.295 GB

Memory access patterns

As explained in Section 3.5, the memory access pattern can have a significant effect on the performance of the algorithm. The more predictable and linear the memory accesses, the easier it is to prevent memory bottlenecks from hampering the performance.

Before starting the backprojection algorithm, the RabbitCT host code loads the benchmark input images from the disk into a buffer on the main memory. This buffer contains all images in order, allowing each host-to-device transfer to copy large chunks of linear memory. In the backprojection algorithm, the image pixel data is selected based upon where the X-ray beams going through a voxel intersect the input image. At this intersection point, the four closest surrounding pixels are loaded from memory for bilinear interpolation. This leads to the semi-random access pattern shown in Figure 4.6. Here, the numbers indicate the order in which the pixel squares are loaded from memory. So, the leftmost square of 1s is loaded first, the square of 2s second, and so on.

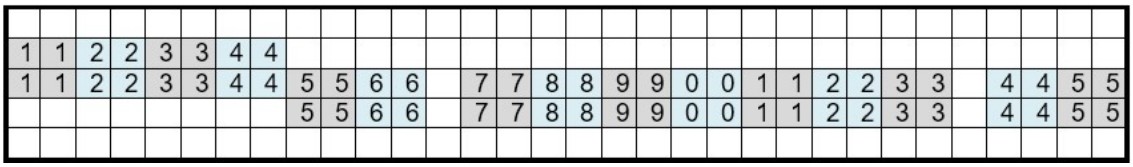
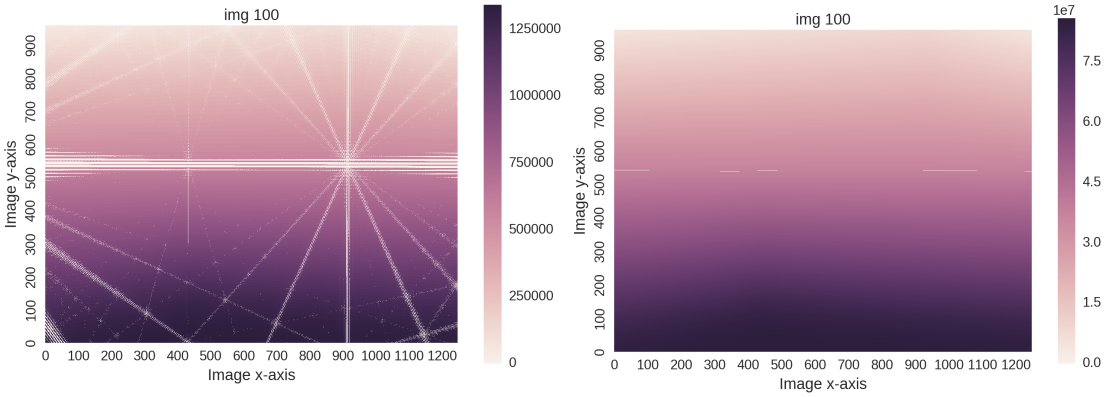


Figure 4.6: Pixel access trace

This pattern is only a small section of the overall access pattern. To see how the access pattern changes between input images and output resolutions, a more in-depth pixel access analysis has been performed. First, a heat map showing the pixel access pattern has been created by adapting the LolaBunny source code to count the total number of pixel accesses, and store this value at the position of the loaded pixel. The adapted LolaBunny script then stores the acquired data to a file which is read by a Python script to create a heat map of the data.

The generated heat maps are shown in Figure 4.7, which show the accumulated pixel access patterns for both the 128 and the 512 resolution. In this image, light coloured pixels are accessed earlier in the program than darker pixels, showing that the input rows are accessed from top to bottom and that each pixel is accessed more often at higher resolutions. Additionally, at lower

resolutions, the heat-map shows a moving star-like access pattern between images. This pixel access pattern is caused by the voxel grid having a slightly different angle for each input image, resulting in the cone-shaped X-ray beams to hit the input image at different positions. With a low resolution, this leads to some pixels not being used to represent the output voxel space which shows up as the star-shaped figure.



(a) Resolution 128

(b) Resolution 512

Figure 4.7: Pixel access heatmap

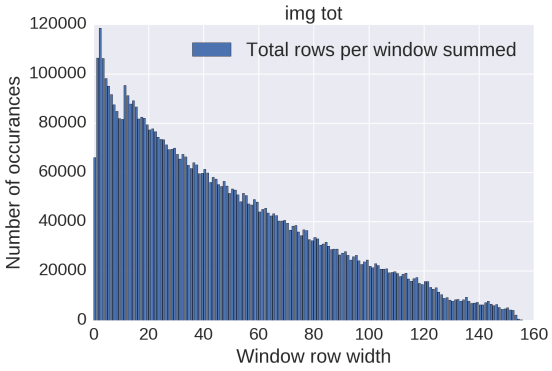
As the output voxel space has a different orientation for each input image, processing it in the XYZ order results in a zigzagging pixel access pattern. Let us define a single zigzag as a window in which pixels are accessed. As the voxel space orientation changes with respect to the input image, so do the required rows and columns of the input image to calculate all voxels corresponding with the window. Figure 4.8a and Figure 4.8b show the total number of times a window requires a certain number of rows or columns. The figures show that most windows require pixels from all 1248 columns of the input image, and only several windows require up to 160 rows of the input image. These numbers are also visible in Figure 4.7, where all columns on a row mostly have the same colour, while the colour between rows changes. As a single window could require a chunk of 160x1248 of input data, a memory bottleneck will be present when all data in the window is processed in parallel, as memory typically allow only a chunk of data to be loaded from a single row per clock cycle. So, to parallelize the RabbitCT algorithm, a caching or prefetching algorithm has to be applied.

The viability of using prefetching depends on whether enough data re-use is present so new data can be loaded in without stalling the calculations, and whether the required prefetch buffer fits on the device hardware. Figure 4.8c shows the first and last time usage of each row. It shows that the number of first used rows grows steadily in a step-like pattern as the algorithm progresses. Each step represents a window that requires data from several new rows, followed by a set of windows re-using this data. As every 200 rows contain between 15 and 25 steps, between 8 and 14 new rows can be required for a new window, which increases the data that must be buffered for it to be available when needed. Assuming about 25% rows of overhead on top of the maximum of 160 rows that can be required, this means that for implementing 200 rows in the FPGA hardware approximately 1MB of on-chip storage is required for prefetching, which easily fits on the FPGA.

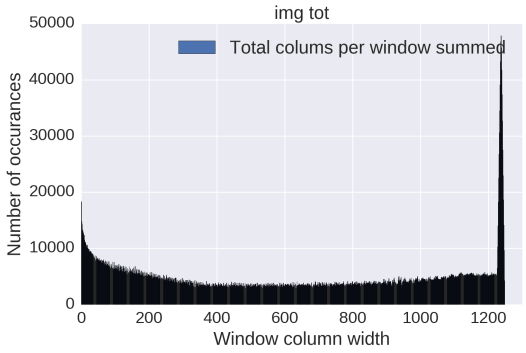
Next, prefetching a single row of input data with the maximum single bank memory bandwidth of 16 floating point values takes 78 cycles (1248 columns divided by 16). Next, each window requires approximately the same number of calculations as the output resolution. So when processing 16 elements in parallel, the data of a row should be re-used by at least ten windows before data from a new row is required to ensure new data is prefetched on time. Comparing the number of windows and rows in Figure 4.8c shows that this is the case. The last row, row 960, is loaded at

window 10500, which results in a single row providing data for approximately 10.9 windows. With a higher resolution, this number only improves as more calculations are performed per window. This is shown in Table 4.2, where the distance between multiple voxels decreases as the resolution increases. For 1024 the distance increases a bit again, but this could be an artifact with how the data is calculated.
Additionally, Figure 4.8c shows several steep drops in the middle of the figure. These drops are an artifact from the data acquisition method where the first and last row usage window number are written to a zero-initialized array. As these values jump to zero, it indicates that these rows are not accessed with the 128 resolution. They are however accessed at higher resolutions. Next, Figure 4.8d shows the number of rows required per window of a single image. As shown in the figure, the first set of windows require data from only a small set of rows, limiting the additional cycles required to initialize the prefetch buffer.

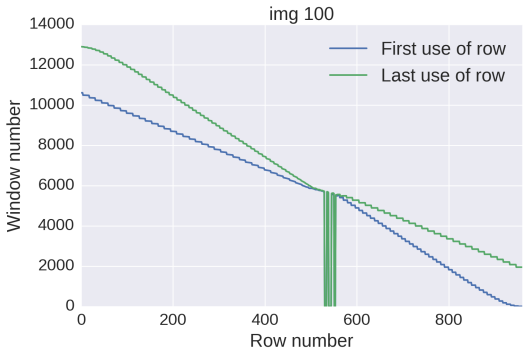
Note that all images in Figure 4.8 are plotted with output resolution 128, which has been chosen for readability. Higher resolutions result in similarly shaped graphs, but with a larger set of windows, and higher total row and column counts.



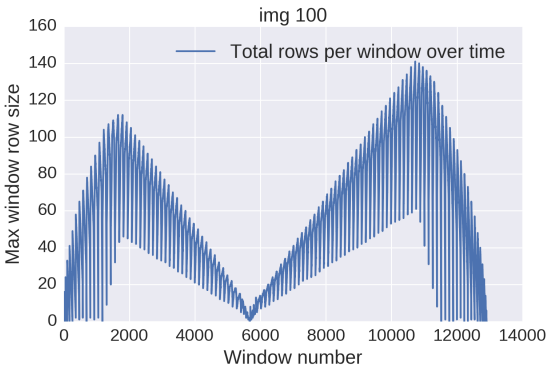
(a) Rows required per window summed



(b) Columns required per window summed



(c) Lifetime of a row



(d) Number of rows required per window

Figure 4.8: Resolution 128 input memory usage analysis

Finally, the output access patterns for the RabbitCT data is a lot simpler than the input access patterns. Here, each element of the big f_L array representing the output voxel space is accessed sequentially. Additionally, no overlapping accesses occur per image calculation, allowing this data to be accessed and adapted in parallel without conflicts for multiple voxel calculations of the same input image.

Table 4.2: RabbitCT input distance between voxels

Resolution	128		256		512		1024	
	Row	Col	Row	Col	Row	Col	Row	Col
2 voxels	3	17	2	8	2	5	3	10
4 voxels	7	41	4	22	3	12	5	26
8 voxels	13	94	8	48	5	25	9	51
16 voxels	26	199	15	101	8	51	17	124

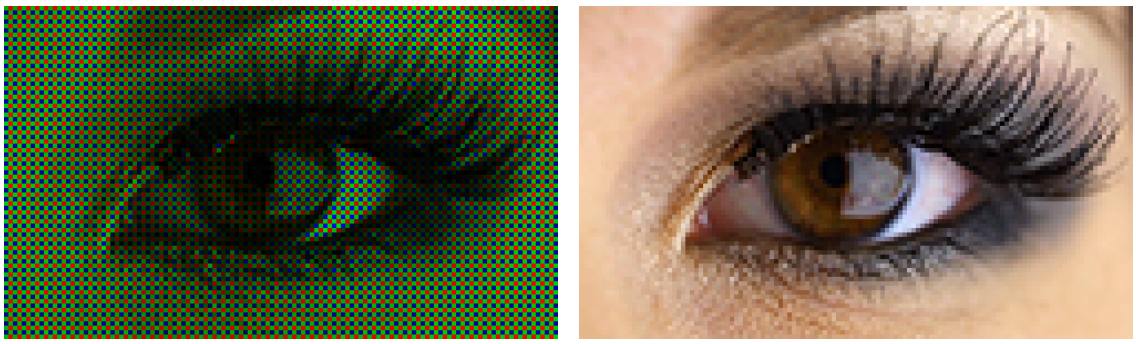
Conclusion

To conclude, the RabbitCT algorithm can easily be optimized for throughput by performing the voxel calculations in parallel. The main difficulty, however, lies in the input memory accesses, where the input pixels are required in a semi-random zigzag pattern. To optimize the processing, all data required for this pattern should be either cached or prefetched to prevent a memory bottleneck from hindering the performance. Luckily, the algorithm at its lowest resolution contains enough data re-use to enable prefetching, and the low number of rows required at initialization will keep the additional prefetch cycles to a minimum. Furthermore, as the total number of rows required to calculate a window is limited, the total buffer size remains limited as well. Finally, as the resolution increases, so does the number of pixel accesses, increasing the data-reuse of the algorithm, which will help with the performance as prefetching and caching will have more effect.

4.2 Demosaic

Demosaicing or debayering is a method to convert monochrome images taken with a colour filter array into an RGB format using interpolation, see Figure 4.9. It is a technique used in most digital cameras to reduce the costs of the hardware [23]. Instead of using three separate light sensors to capture the red, green an blue light at full resolution, a single sensor is used with a colour filter, so each pixel registers a different colour. Multiple colour filters exist for taking pictures using a single sensor. This research focuses solely on the Bayer filter, which is a two by two square with one red, two green and one blue pixel. Using two pixels for green improves the perceived picture quality as the human eye is more sensitive to green colours [116].

The use of a single sensor with a colour filter array also has a downside; it reduces the output colour resolution [116]. So, when the sensor has a monochrome 4k by 4k resolution, the coloured



(a) Raw Bayer pattern (b) Interpolated Bayer pattern

Figure 4.9: From raw input to interpolated output, taken from [117]

output data has a 1k by 1k resolution for the red and blue colours, and a 2k by 2k resolution for green colours. To restore an image to full resolution, the Demosaic algorithm takes the separate colour channels and creates an interpolated pixel based upon the surrounding colour data, as can be seen in Figure 4.10.

A Bayer demosaicing algorithm can be implemented in multiple ways. These options can be divided into the following three categories:

- Linear demosaicing: where each interpolated pixel receives the average colour value of the surrounding pixels
- Matrix demosaicing: where a moving matrix multiplies the surrounding pixels with different coefficients to determine the interpolated pixel value [23].
- Smart demosaicing: where more complex algorithms are used to determine the interpolated pixel values. These algorithms use noise filtering, edge detection or colour wavelet analysis on top of the conventional demosaicing algorithm to reduce demosaicing artifacts and improve the quality of the output image [65, 66].

For this research, the matrix demosaicing option has been selected as the implementation is similar to other popular image processing techniques like edge detection and noise reduction. This similarity allows the same optimized algorithm to be used for other image processing tasks and thus forms a general idea of the image processing performance on both GPU and FPGA hardware.

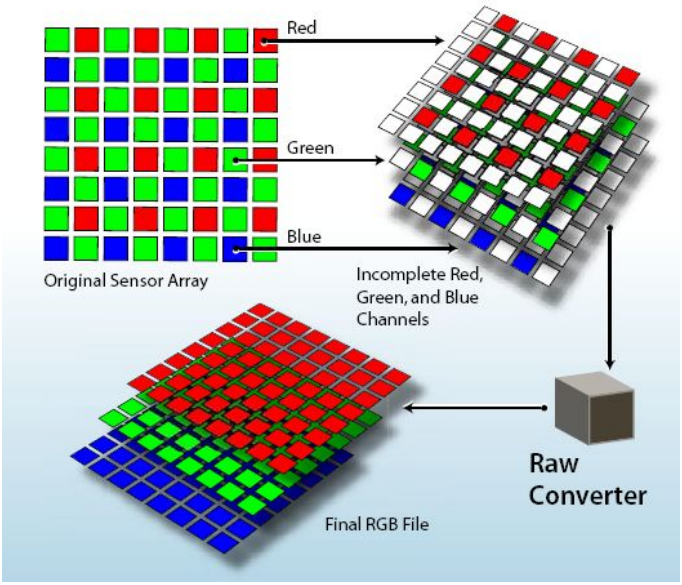


Figure 4.10: Bayer filter, colour channels, and conversion to RGB. Taken from [118]

4.2.1 Theoretical analysis

The demosaicing algorithm used in this research is the one proposed by Malvar-He-Cutler in 2004 [23]. This algorithm uses the luminance and chrominance information of the surrounding pixels to improve the output quality. Luminance and chrominance are part of the YUV colour spectrum, which was developed to enable backwards compatibility of colour television with black and white

television [119]. Here, the Y or luma channel contains the image in black and white, and the U and V channels contain the colour information of the image. Combining both luminance and chrominance information results in the complete image, see Figure 4.11.

Figure 4.11: Luminance and chrominance, taken from [120]

Malvar-He-Cutler states that in the YUV spectrum, edges contain sharp luminance changes [23]. So, if an edge occurs, the interpolated pixel can be colour corrected by using luminance data from the surrounding pixels. The luminance based Demosaic algorithm is performed by filtering the input image with the matrices shown in Figure 4.12. All matrix multiplications sum up to eight, so, to get the final pixel value, the sum of matrix calculations is normalized by eight. By performing the stated matrix calculations and normalizing the final value for all pixels, luminance corrected RGB data remains, which together create the coloured output image.

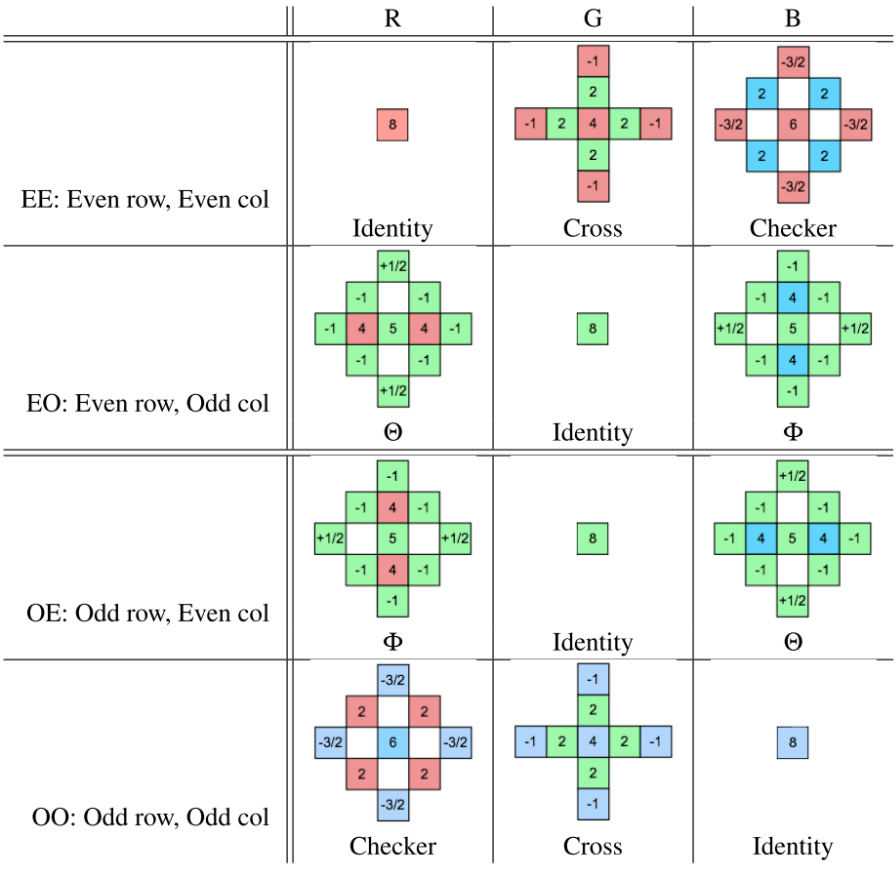


Figure 4.12: Malvar-He-Cutler demosaic filter coefficients, taken from [121]

4.2.2 Algorithm implementation

The initial Demosaic implementation contains two parts, the host code and the Malvar-He-Cutler demosaicing algorithm. The host code reads the input images, initializes the memory, executes the demosaicing algorithm, compares the results with a reference, and cleans up. The input of the algorithm is a set of 5120x3072 monochrome Bayer TIFF images of an American Football match. These images are loaded into memory using libTIFF, a TIFF image reading and writing library [122]. Then they are copied to the accelerator after which the demosaicing algorithm is started and the execution time of the kernel is recorded. When the demosaicing algorithm finishes processing all pixels of an input image, the output is copied back to the host and the data compared with a reference image. Finally, when all colours have the same output value, the output is written to a TIFF output file, and possible errors are reported. A pseudocode implementation of the host algorithm is shown in Listing 4.4.

Only the execution time of the kernel is used for benchmarking. It indicates how many images per second the kernel processes. The measurement does not include any latency introduced by data transfers between host and device and thus gives insight into the maximum accelerator performance.

```
input:
    inputPath, // Path to input images
    outputPath, // Path to store output images
    imgStart,   // Number of first image to process
    imgEnd, // Number of last image to process
output:
    imgOutN, // N output images with N in between imgStart and imgEnd
    fps_max, // Fastest kernel of N processed images
    fps_avg, // Average number of images kernel can process per second
    fps_min, // Slowest kernel of N processed images

// Initialize image containers
tiffData_t* tiffIn = initTiffData( img_parameters );
tiffData_t* tiffOut = initTiffData( img_parameters );
tiffData_t* tiffRef = initTiffData( img_parameters );

// Initialize profiling parameters
ulong t_min = 1e12;
ulong t_max = 0;
ulong t_tot = 0;

// Initialize OpenCL environment
initCL();

for( int imgnr = imgStart; imgnr < imgEnd; imgnr++ ){
    // Read input image imgnr into tiffIn container
    read_tiff(tiffIn, imgnr);

    // Copy input image to device
    clEnqueueWriteBuffer(dev_in, tiffIn->imgData);

    // Start a demosaic kernel for all pixels of input image and measure execution
        time in ns
    t = currentTime( );
    clEnqueueNDRangeKernel(demosaic, img_width*img_height);
    t_exec = currentTime( ) - t;

    // Store image processing data
    t_tot = t_tot + t_exec;
    if(t_exec > t_max)
        t_max = t_exec;
    if(t_exec < t_min)
        t_min = t_exec;

    // Copy output image to host
```

```
    clEnqueueReadBuffer( dev_out, tiffOut->imgData);

    // Read reference image imgnr, compare with imgOut and print errors
    compare_img( tiffRef, tiffOut, imgnr );

    // Write demosaic output data to output image
    write_tiff(tiffOut, imgnr);
}

// Calculate and print profiling data
ulong t_temp = t_tot / (imgEnd - imgStart);
float fps_avg = (1.0f / (t_temp * 1e-9f));
float fps_min = (1.0f / (t_max * 1e-9f));
float fps_max = (1.0f / (t_min * 1e-9f));

printf(fps_max, fps_avg, fps_min);

// Cleanup and exit
teardown(0);
```

Listing 4.4: Demosaic host pseudocode

Pseudocode of the Demosaic algorithm is shown in Listing 4.5. It loops over all pixels and determines whether the row and column of the selected pixel are odd or even. Based on this information, the corresponding matrix multiplication from Figure 4.12 is applied. This value is then normalized, clamped to one byte and stored in the respective colour channel for the output pixel.

```
input:
    uchar* imgIn;
output:
    uchar* imgOut;

uchar matrixMult( imgin, row, col, matrix[25] )
{
    for each row and column of a matrix that falls within image boundaries {
        // Get matrix and pixel value
        float matrixval = matrixvalue;
        uchar pixelval = imgin[pixel pos based on matrix pos];

        // Sum their product and sum all matrix values for edge cases
        colorSum += pixelval * matrixval;
        matrixSum += matrixval;
    }

    // Normalize result and round to nearest integer
    int outputcolor = round(colorSum / matrixSum);

    // Clamp to byte sized output
    uchar returnval = clamp(outputcolor, 0, 255);

    // Return colordata
    return returnval;

}


kernel void demosaic(imgIn, imgOut)
{
    // Define matrices
    float crossMatrix[25];
    float checkerMatrix[25];
    float thetaMatrix[25];
    float phiMatrix[25];

    // Get pixel position
    int col = get_global_id(0);
```

```
    int row = get_global_id(1);

    // Check row/column position and perform matrix multiplication
    if( row % 2 == 0 ){
        if( col % 2 == 0){
            R = imgIn[row * width + col];
            G = matrixMult(imgIn, row, col, crossMatrix);
            B = matrixMult(imgIn, row, col, checkerMatrix);
        } else {
            R = matrixMult(imgIn, row, col, thetaMatrix);
            G = imgIn[row * width + col];
            B = matrixMult(imgIn, row, col, phiMatrix);
        }
    } else {
        if( col % 2 == 0){
            R = matrixMult(imgIn, row, col, phiMatrix);
            G = imgIn[row * width + col];
            B = matrixMult(imgIn, row, col, thetaMatrix);
        } else {
            R = matrixMult(imgIn, row, col, checkerMatrix);
            G = matrixMult(imgIn, row, col, crossMatrix);
            B = imgIn[row * width + col];
        }
    }

    // Write colour data to output image
    imgout[(row * width + col)*3 + 0] = R;
    imgout[(row * width + col)*3 + 1] = G;
    imgout[(row * width + col)*3 + 2] = B;
}
```

Listing 4.5: Demosaic algorithm pseudocode

4.2.3 Implementation analysis

The hotspot of the Demosaic algorithm is in the MatrixMult function, which is executed twice for each pixel. In matrixMult, the 25 surrounding pixels are loaded from memory and multiplied with the values in the matrix. So, for all 5120 by 3072 pixels of an image and a total of 50 iterations of the inner loop of MatrixMult, this leads to 786.4 million iterations of this inner loop per image. When executed on a CPU, the parallelism is limited to the number of cores, resulting in slow processing speeds. However, with the parallel power of a GPU, and pipelined implementations of the FPGA, a huge increase in throughput can be achieved. This section analyzes the Demosaic algorithm on its data and memory access patterns, to devise an optimal implementation for all platforms.

Data flow

Figure 4.13 shows the data flow of the Demosaic implementation. Here, the algorithm loops over all pixels of the input image and determines whether the selected pixel is even or odd for both the row and column. Depending on this result, each colour, red green and blue, is calculated with the matrix data shown in Figure 4.12. This is shown in Figure 4.13 by the branching paths, where two calculations and one memory fetch are performed. Note that for the initial calculation, these branches are executed sequentially and not in parallel. Each of the three branching paths require pixel data to be loaded from memory; the matrix calculations require a five by five grid of pixels, and the identity calculation only requires data of the selected pixel. As all three calculations require data from the same five by five grid of pixels, great memory optimizations are to combine the calculations in a single function to reduce the number of memory accesses, or to cache the

pixel data so that the data is immediately available. When all colour data is calculated, this data is written to the output image in the RGB format.

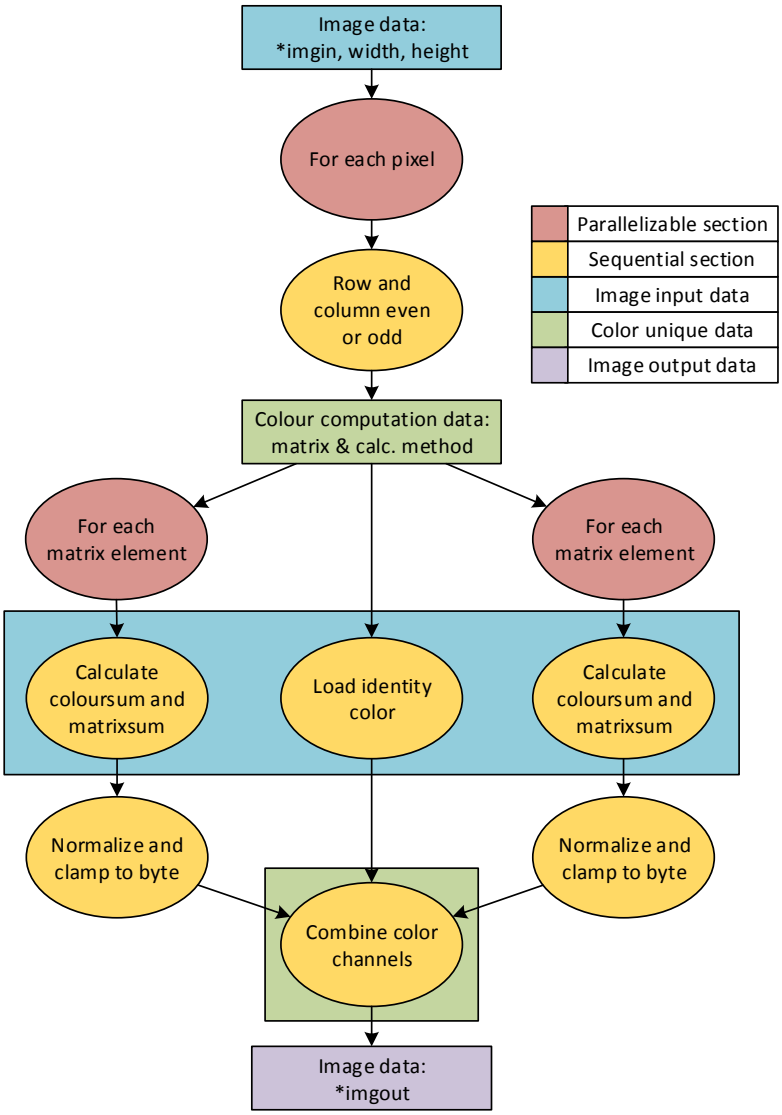


Figure 4.13: Demosaic Dataflow

Data types

All platforms offer a different performance for different data types. An analysis of the used data types can set an expectation for the final performance and show whether changing the data types used in the algorithm could lead to better results.

The input of the Demosaic algorithm is a 5120 by 3072 pixels input image. As the input image consists of a single colour, the input data requires only one byte per pixel and is represented by an unsigned char in the algorithm. Next, the matrices suggested by Malvar-He-Cutler and are shown in Figure 4.12 contain floating point values, which is known to have a lower performance on an FPGA, see Section 3.5. However, the floating point is only required to represent halves

of an integer which are normalized by the factor 8. So, a possible optimization would be to double the matrix values and normalization factor resulting in a char matrix, making the matrix multiplication more optimized for execution on an FPGA. Finally, as each output colour after normalization is clamped to a single byte, the output image is represented by three unsigned chars, which should not hamper the performance of the algorithm.

Data sizes

Knowing the data sizes used in an algorithm can show any external bandwidth issues that might be present. For Demosaic, the input is a monochrome 5120 by 3072 image. Of this image, each pixel is represented by a single byte, resulting in the input image requiring a total of 15MB. Next, the output of the algorithm is the same image but now coloured. As three 8-bit colour channels are used, the output image requires 45 MB. With the demosaicing matrices being hard-coded in the algorithm, a total of 60 MB per image has to be transferred over the PCIe connection. So, with a commonly used PCIe 3.0 16x bandwidth of 15.75 GB/s, a total of 262 images can be processed per second, limiting the maximum throughput. For this reason, the Demosaic algorithm purely benchmarks the kernel, leaving only the kernel performance. Finally, for internal processing, up to 25 pixels are required for processing a single output pixel. As this requires only 25 bytes, all the data required for processing should easily fit on the on-chip memory.

Memory access patterns

Memory access patterns can have a significant effect on the performance of the algorithm, as is explained in Section 3.5. The more predictable a memory access pattern is, the higher the performance of an algorithm can be. For Demosaic only one image is processed at a time. The host loads a TIFF input image as input and stores this in RAM. With the image loaded into RAM, it is copied to the accelerator, and the Demosaic algorithm is started. With sequential execution, the pixels of the input image are processed row by row, enabling the use of burst transfers for the column data. For each calculation, a window of 5 by 5 pixels is loaded from the global memory, and the colour data is computed. Afterwards, the colour data of a pixel in the next column can be computed. This access pattern resembles a sliding window approach that linearly moves across the input image. So, significant memory optimizations are possible as either caching or prefetching will ensure that all data is available and is only once loaded to the device. Finally, for the output, the same linear access pattern as the input image is present. As the input image, the output image is written row by row, which enables burst transfers for the columns of each row.

Conclusion

To conclude, the Demosaic algorithm contains a significant number of optimization possibilities. First, it contains multiple iterations of matrixMult that can be merged to simplify the algorithm. Next, it contains floating point data that can be converted into chars, increasing the processing performance. As for memory, its linear memory access patterns allow for many accesses to be prevented by using burst transfers, prefetching and caching. And finally, the algorithm only requires several bytes of on-chip memory per processed pixel. The only downside and possible performance limiting factor is the 60 MB the algorithm requires per image, which saturates the PCIe 3.0 16x memory bandwidth when processing 262 5k by 3k images per second. To solve this issue, compression could be used. However, as this falls outside the scope of this project, only the kernel performance is benchmarked.

Chapter 5

Hardware Selection

To select the benchmarking platform hardware, the FPGA, GPU, and software requirements must be taken into account. The following sections show what requirements need to be satisfied for each device, and what hardware has been selected that meets these requirements. Based upon these requirements, an FPGA is selected first, after which a similarly priced GPU is chosen to compare the hardware performance at a similar price point.

5.1 FPGA selection

This section considers the selection of suitable FPGAs for OpenCL Acceleration.

5.1.1 FPGA requirements and options

Intel and Xilinx are the two largest FPGA vendors. They produce FPGAs dies with similar specifications at similar price points. Third party companies, including Prodrive, purchase these FPGA dies and integrate these FPGAs into their own hardware. Even though Prodrive does produce FPGA hardware, only FPGA accelerators from either Xilinx or Intel are selected for this project. Buying FPGA accelerators removes the need to create a custom BSP to support OpenCL on Prodrive FPGA hardware, and buying FPGAs from Intel or Xilinx allow these vendors to support us with any hardware or tooling issues directly. Additionally, as the goal is to compare GPU with FPGA acceleration, their communication interface with the host should be similar. So, the selected FPGA should be connected to the host PC via a PCIe connection. The following sections indicate the vendor-specific requirements.

Intel FPGA requirements

According to the Intel FPGA SDK for OpenCL getting started guide [13], Intel only requires that an Intel FPGA accelerator must be used and that this accelerator must be provided by either Intel or a third party Intel partner. According to the custom platform creation guide [14], custom acceleration hardware with any Intel OpenCL enabled FPGAs can be created as well, however, this requires the creation of a PCB and BSP with working PCIe and memory interfaces and falls outside the scope of this project.

For off-the-shelf FPGA OpenCL acceleration, Intel itself provides the development kits shown in Table 5.1 with full BSP support [123]. Here, the FPGA die name is shown together with its number of logic elements, DSPs, and the devkit and FPGA price. As development kits are often sold at a loss to get companies using these products, both the platform and consumer FPGA die prices are shown to show the difference between devkit pricing and consumer pricing. Next, in

2019 Intel will start selling acceleration cards as well [124] with engineering samples being available now. However, these acceleration cards will contain the same FPGA dies as the development kits currently available so should offer the same performance.

Table 5.1: Intel FPGA OpenCL Accelerators, pricing as of February 2019 from digikey.nl [77]

Platform	FPGA	Released in	Logic Elements	DSP units	Memory size & type	Memory bandwidth	Platform price in €	FPGA die price in €
Cyclone V SoC Development Kit [125]	5CSXFC6-D6F31C6N	2015	110k	112	2GB DDR3	6.4GB/s	1556.27	295.07
Arria 10 GX FPGA Development Kit [126]	10AX115-S2F45I1SG	2016	1150k	1518	2GB DDR4	21.3GB/s	3897.17	8897.16
Stratix V GX FPGA Development Kit [127]	5SGXEA7-K2F40C2N	2011	622k	0	1GB DDR3	14.4GB/s	discontinued	7210.88
Stratix 10 GX FPGA Development Kit [128]	1SG280-HU2F50E2VG	2018	2753k	5760	1GB DDR4	21.3GB/s	6936.-	not for sale

Xilinx FPGA requirements

The Xilinx SDAccel Platform Development Guide v2017.4 [15] states the following requirements for creating a custom OpenCL FPGA Acceleration platform:

- A Vivado Design Suite supported FPGA that supports partial reconfiguration
- Global memory in the form of DDR4 SDRAM that is accessible to both the host and user kernels via AXI4 memory mapped connectivity.
- Software compatibility via the hardware abstraction layer driver provided with the SDAccel installation.

Note that Xilinx has not made the v2018.2 release of the Platform Development Guide freely available, with the website stating that those files are only available upon request.

For off-the-shelf OpenCL acceleration, the Xilinx SDAccel Environment User Guide v2018.2 [16] lists that using one of the FPGA development kits in Table 5.2 as a requirement. Here, the exact FPGA, its logic elements and DSP units are shown. Additionally, like Intel, Xilinx recently has released its own FPGA accelerator hardware: the Alveo U200, U250 and U280. These accelerator cards contain a custom FPGA and support SDAccel out of the box. These cards were not available at the start of the project, so, could not be chosen. However, they do give an indication of future FPGA accelerator hardware and pricing. For other accelerator options, Xilinx links to third-party vendors. However, as with Intel; the introduction of another vendor is not preferred and thus not analyzed.

Table 5.2: Xilinx OpenCL FPGA Accelerators, pricing as of February 2019 from digikey.nl [77]

Platform	FPGA	Released in	Logic Elements	DSP units	Memory size & type	Memory bandwidth	Platform price in €	FPGA die price in €
Kintex UltraScale FPGA KCU1500 [129]	XCKU115-2FLVB2104E	2017	1451k	5520	16GB DDR4	76.8GB/s	discontinued	9020.82
Virtex UltraScale+ FPGA VCU1525 [130]	XCVU9P-L2FSGD2104E	2017	2589k	6840	16GB DDR4	76.8GB/s	discontinued	49004.43
Alveo U200 [131]	XCU200	2018	2003k	5867	64GB DDR4	76.8GB/s	9597.67	not for sale
Alveo U250 [132]	XCU250	2018	3007k	11508	64GB DDR4	76.8GB/s	13865.67	not for sale
Alveo U280 [133]	XCU280	2019	2384k	8490	32GB DDR4 8GB HBM2	38GB/s + 460GB/s	unknown	not for sale

5.1.2 FPGA selection and specifications

Prodrive its customers are currently using Nvidia K620 and K4000 GPUs, which launched for €190 and €950 respectively [74]. Optimally, a modern FPGA accelerator within the same price range should be selected. However, this is not possible with the provided options. For this reason, the Kintex UltraScale FPGA KCU1500 was selected as it offered the most FPGA resources and memory bandwidth with the lowest devkit and FPGA price in February 2018. The Alveo FPGAs were not available back then, but would certainly be an interesting option now.

At the time, the Kintex KCU1500 FPGA devkit sold for approximately € 3k and was likely sold at a loss as a separate FPGA die costed € 8k back then. With the Kintex KCU1500 FPGA being discontinued, it has to be compared with the current Alveo offerings to estimate a more accurate accelerator price. The Kintex KCU1500 die sells for € 9k, but this not necessarily represent the current market price due to low selling volumes and discounts companies get for buying them in bulk. Looking at the recently released Alveo U200 FPGA enables a more realistic price estimation. The Alveo U200 FPGA has a newer FPGA architecture, contains about an equal number of DSPs, has about 38% more logic elements, and sells for € 9.5k. Additionally, the Alveo U250 has 33% more logic elements and double the DSPs of the Alveo U200, and sells for € 14k. So, with the KCU1500 having an older less efficient architecture and fewer logic elements than the Alveo U200, a KCU1500 accelerator would likely cost around € 6.5k when sold as an accelerator. To represent a more realistic market value, the approximated € 6.5k KCU1500 price will be used in the rest of this document.

The complete specs of the Xilinx Kintex UltraScale FPGA KCU1500 Development Kit are [129]:

- XCKU115-2FLVB2104E FPGA
- 4 banks with 4GB DDR4 2400MT/s 64-bit memory of which 3 banks support error correction
- 1Gb Dual Quad SPI Flash
- Two 8x PCIe Gen3 connections on a single 16x edge connector
- USB JTAG & JTAG PC4 header & Two 100gBe QSFP+ cages

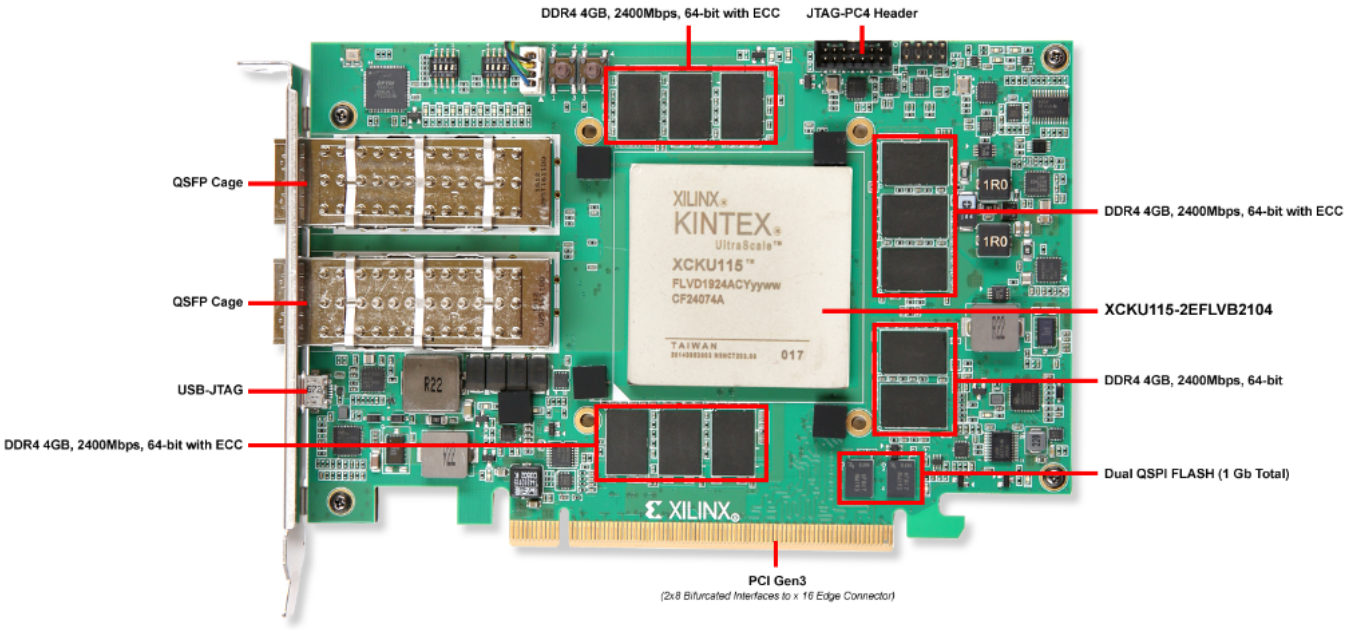


Figure 5.1: Kintex UltraScale FPGA KCU 1500 PCB, taken from [129]

5.2 GPU selection

This section considers the selection of a suitable GPU for OpenCL Acceleration.

5.2.1 GPU requirements and options

There are two requirements for selecting a GPU:

- The GPU should be a successor to the Nvidia Quadro K620 and K4000 GPUs currently used by Prodrive its customers.
- The GPU should be in a similar price range as the selected FPGA for a fair comparison.

At the start of this project, Pascal was the latest generation of Nvidia graphics cards. As of August 2018, Nvidia has released four next-generation Turing Quadro RTX GPUs. An overview of the Pascal and Turing generation of GPUs is listed in Table 5.3.

Table 5.3: Nvidia Quadro GPUs as of February 2019. Data from techpowerup.com [74]

GPU	Architecture	CUDA Cores	Core Clock in MHz	Memory size & type	Memory bandwidth in GB/s	Lowest price in € according to tweakers.net [134]
Quadro K620	Kepler (2012)	384	1058	2GB GDDR3	28.8	184.10
Quadro K4000	Kepler	768	810	3GB GDDR5	134.8	883.-
Quadro P400	Pascal (2016)	256	1228	2GB GDDR5	32.1	136.92
Quadro P600	Pascal	384	1329	2GB GDDR5	64.19	228.40
Quadro P1000	Pascal	640	1266	4GB GDDR5	80.19	369.60
Quadro P2000	Pascal	1024	1076	5GB GDDR5	140.2	462.70
Quadro P4000	Pascal	1792	1202	8GB GDDR5	243.3	797.84
Quadro P5000	Pascal	2560	1607	16GB GDDR5x	288.5	1802.86
Quadro P6000	Pascal	3840	1506	24GB GDDR5x	432.8	4635.12
Quadro GP100	Pascal	3584	1304	16GB HBM2	732.2	5681.43
Quadro GV100	Volta (2018)	5120	1132	32GB HBM2	868.4	11322.-
Quadro RTX 4000	Turing (2018)	2304	1215	8GB GDDR6	416.0	1012.89
Quadro RTX 5000	Turing	3072	1620	16GB GDDR6	448.0	2429.79
Quadro RTX 6000	Turing	4608	1440	24GB GDDR6	672.0	4938.31
Quadro RTX 8000	Turing	4608	1005	48GB GDDR6	672.0	unknown

5.2.2 GPU selection and specifications

To meet the requirements, a GPU had to be selected that was in a similar price range as the selected FPGA, which price was approximated at € 6.5k. As the Pascal generation of GPUs was the latest GPU generation available at the start of this project, no Turing Quadro RTX cards could be chosen. Currently, the Quadro GP100 is the only Pascal GPU in the target price range set by the selected FPGA. However, at the start of this project in February 2018, the average GP100 price was € 8k, and the Quadro P6000 was priced € 5.6k on average [134], which made the P6000 the GPU that was the closest to the price range of the selected FPGA. Next, by looking at the specifications, the Quadro P6000 offers more CUDA cores, a larger memory size, albeit a lower memory bandwidth at a significantly lower price than the GP100. //* Additionally, the selected FPGA has a memory bandwidth that is about five times lower than the Quadro P6000. As there is no need to spend more money to get an even higher memory bandwidth for the GPU, the Quadro P6000 has been selected. If one of the selected algorithms contains a memory bottleneck, the Quadro P6000 will outperform the selected FPGA due to this memory bandwidth advantage.

The complete specs of the Nvidia Quadro P6000 GPU are [74]:

- GPU die: GP102
- CUDA Cores: 3840
- CUDA Core clock: 1506 MHz
- CUDA compute capability: 6.1
- Memory clock: 1127 MHz
- Memory size: 24 GB
- Memory type: GDDR5X
- Memory bus: 384-bit
- Memory bandwidth: 432.8 GB/s
- Displayport video output: 4x
- DVI video output: 1x
- SLI high bandwidth connector: 1x
- DirectX 12.1

- OpenGL 4.6
- OpenCL 1.2
- Max TDP: 250W

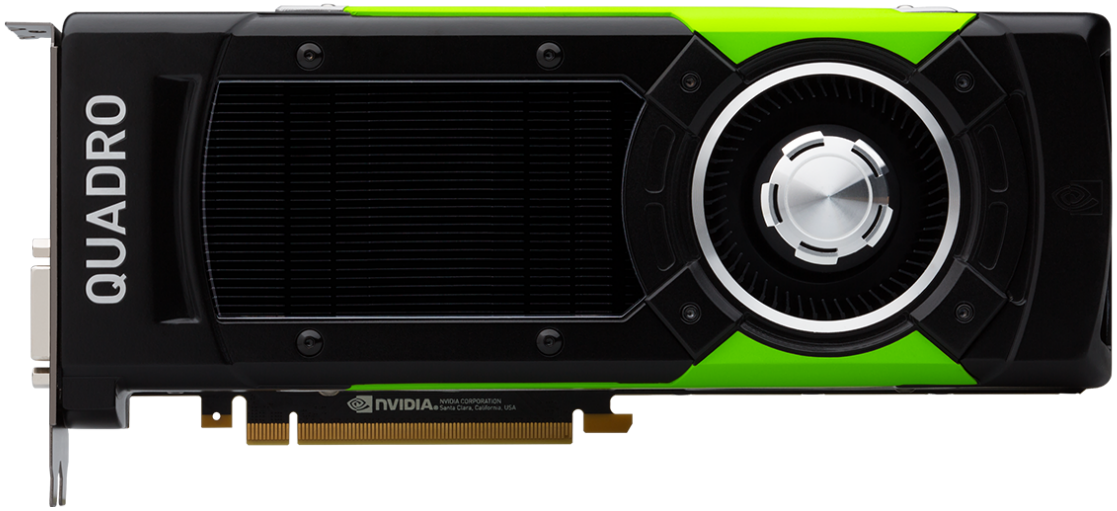


Figure 5.2: Nvidia Quadro P6000, taken from [135]

5.3 Host selection

This section considers the selection of a suitable kernel compilation and benchmarking PC.

5.3.1 Host requirements

As of version 2018.2 of the Xilinx SDAccel Environment Release Notes, Installation and Licensing Guide [16], the following requirements are stated for the use of the SDAccel OpenCL acceleration software:

- Motherboard with a PCIe 3.0 8x slot
- 16GB RAM
- 100GB free disk space
- A Vivado Design Suite 2018.2 installation
- A supported 64-bit Windows or Linux OS [17]

Next, Nvidia does not specify any requirements other than that the GPU should be able to connect to the system, and that an Nvidia driver that supports the card should be installed.

5.3.2 Host specifications

A search for a PC with these requirements resulted in a server (SCH MK2) that was directly available from another project at Prodrive. This server has been used for all compilations and benchmarking.

- 2 * Intel Xeon E5-2640v4 (@2.4Ghz, 6 cores per socket)
- 8 * 32GB DDR4 RAM @2400 MHz

- Motherboard: SuperMicro X10DRH-ILN4 (2x PCI Gen 3.0 x16, 4x PCI Gen 3.0 x8, 1x PCI Gen 3.0 4x)
- 2x Samsung SSD SM863 960 GB

On the MK2 server, the following software has been installed:

- Ubuntu 16.04.5 LTS
- GCC 5.4.0-6
- Nvidia Display Driver version 396.45
- Nvidia CUDA Toolkit 9.2
- Xilinx Vivado v2018.1
- Xilinx SDAccel v2018.2

Ubuntu 16.04.5 is not officially supported according to the Vivado documentation. However, as 16.04.5 only contains security updates over 16.04.3, this should not affect the performance of the FPGA acceleration.

5.4 Hardware comparison

This section shows a comparison between the selected hardware. It includes a numeric comparison showing the theoretical performance of the hardware. With these theoretical number, a Roofline plot is created based upon the initial implementations of the selected algorithms. This Roofline plot visualizes the hardware differences in a single graph and shows where the bottleneck lies for the default algorithm implementations.

5.4.1 Numerical comparison

A comparison between several hardware specifications of the selected benchmarking hardware is shown in Table 5.5. This comparison shows that the GPU has a significantly larger memory bandwidth than the other devices. Due to this bandwidth difference, the Xilinx FPGA will most likely pay a bigger performance penalty than the GPU for memory intensive applications. However, smart FPGA implementations using on-chip memory could alleviate this problem. Here, the on-chip memory can be implemented as a pipeline or prefetch cache, significantly reducing the number of global memory accesses. The next generation Intel and Xilinx FPGAs use HBM2 memory on the die with high-end models, significantly increasing the FPGA memory bandwidth with a bandwidth up to 512GB/s reported [49].

Additionally, a factor six difference in maximum single precision floating point performance is found between FPGA and GPU devices. As only Nvidia reports the single precision performance, the CPU and FPGA floating-point performances are calculated. Here, the Intel Xeon E5-2640v4 Processor floating point performance is based upon the calculations provided at [136], where the clock speed (2.4GHz) is multiplied with the total number of cores (2*6) and the number of Single Precision FLOPs per cycle (32), leading to 921.6 GFLOPs. For the Nvidia Quadro P6000, the performance was reported, but can also be calculated. Here, each CUDA core can execute a floating point multiply and add in a single clock cycle. So, multiplying the number of CUDA cores (3840) with the maximum boost clock frequency (1645 MHz) and the number of operations per cycle (2), leads to the reported 12.634 TFLOPs.
Finally, the Xilinx FPGA single-precision floating point performance is based on the calculations in [67]. By comparing the available resources of the XCKU115 FPGA [137], with the resources required for the least hardware intensive floating point operation [104], the total number of floating point operators that fit in the hardware can be determined. In this case, the DSP multiplication requires the least FPGA hardware. By combining the full and medium DSP usage implementations

of the multiplier, an implementation can be created that achieves the maximum floating point performance possible on the KCU1500 hardware. The total number of multiplier units is then multiplied with the speed of the slowest FPGA (speed grade -1) clock speeds results, which results in a maximum single precision floating point performance of 2116.4 GFLOP/s. All data used for the calculation is shown in Table 5.4.

Table 5.4: Xilinx KCU1500 floating point performance calculation

Multiplier Type	f_{max}	Number of multipliers	LUTs per multiplier	LUTs Total	DSPs per multiplier	DSPs Total	GFLOPs
DSP full	568 MHz	1684	91	153244	2	3368	956.5
DSP med	539 MHz	2152	237	510024	1	2152	1159.9
Total		3022		663268		5520	2116.4
Remaining				92		0	

Note that the calculated performance numbers show the absolute maximum performance numbers that can theoretically be achieved using the hardware. For Intel, it is assumed that all 32 floating point units of all cores will be processing data at the base clock frequency. For Nvidia, it is assumed that all 3.8k CUDA cores are processing computations with a combined floating point multiplication and addition at the maximum boost frequency. Finally, for Xilinx, it is assumed no hardware overhead is present for routing, and that the maximum clock frequency of the FPGA with the lowest speed can be attained for a design that uses all the FPGA hardware. For this reason, Intel states that it does not use FPGA logic to determine the GFLOP/s performance metric and reports such calculations to be false [67].

The final single-precision floating point performance numbers show that the selected FPGA generation is about six times slower than the reported Quadro P6000 floating point performance. However, when looking at the next generation Stratix 10 or Virtex Ultrascale+ architectures, FPGAs are closing the floating point performance gap with the DSP performance being reported to achieve up to 7 TFLOP/s for Xilinx [68], and 9.2 TFLOP/s for Intel [138]. However, Nvidia has also improved their architecture, with their next generation Quadro RTX 6000 GPU reporting a 16.3 TFLOP/s performance, making it still theoretically twice as fast in floating point math as the latest Intel and Xilinx FPGAs.

The last two differences between hardware shown in Table 5.5 are the API support and thermal design power (TDP). Nvidia GPUs support the full OpenCL 1.2 API and CUDA with compute capability 6.1. Xilinx FPGAs support OpenCL 1.0 [18] with some additional features from OpenCL 2.0 [4]. Next, the difference in TDP indicates that the GPU is designed to use more power than either the CPU or FPGA as its cooler can dissipate more heat.

Table 5.5: Hardware comparison

Device	Single precision GFLOP/s	Memory size and type	Memory bandwidth	PCIe bandwidth	Supported programming APIs	Max TDP
Host PC (2x Intel Xeon E5-2640v4)	921.6	256GB DDR4	153.6 GB/s	2x 15.75 GB/s, 4x 7.88 GB/s, 1x 3.94 GB/s	OpenCL 1.2, 2.0, and 2.1 experimentally	180 W
Nvidia Quadro P6000 [139]	12.6k	24GB GDDR5x	432.8 GB/s	15.75 GB/s	OpenCL 1.2 with experimental support for some 2.0 functions, CUDA compute capability 6.1	250 W
Xilinx Kintex Ultrascale KCU1500	2.1k	16GB DDR4	76.8 GB/s	2x 7.88 GB/s	OpenCL 1.0 with some 1.2 features	75 W

5.4.2 Interconnect options

The Quadro P6000 GPU has a PCIe 3.0 16x connection, several video output ports, and an optional SLI bridge interface. With the SLI-bridge being for GPU-GPU communication only, the input and output data is limited to the PCIe interface. Next, the KCU1500 FPGA contains two PCIe 3.0 8x connections and two QSFP+ cages that allow for two 100gBe connections. The addition of these QSFP+ cages adds additional I/O bandwidth on top of the available PCIe 3.0 connection. As the QSFP+ moves directly into the FPGA, this data can also be used as a memory interface to increase the available memory bandwidth. However, this does require BSP changes and the addition of a QSFP+ data handler written in RTL to manage the data, which falls outside of the scope of this project.

5.4.3 Roofline comparison

A Roofline graph allows the maximum memory bandwidth and compute performance of a set of hardware to be compared in a single graph. Additionally, the arithmetic intensity of an application can be plotted to indicate whether an application is bottlenecked by either the hardware its memory or its compute performance. The compute and memory performance of the hardware is calculated in Section 5.4.1. Leaving only the numerical intensity of both the RabbitCT and Demosaic algorithm to be determined.

The arithmetic intensity of an application is calculated by dividing the total number of floating point operations by the total number of bytes that need to be loaded from memory for this calculation. This calculation is performed using the following formula:

$$I = \frac{W}{Q}, \tag{5.1}$$

Where W is the work of the application and Q is the number of bytes transferred.

RabbitCT Arithmetic Intensity

For the RabbitCT Lolabunny implementation, the arithmetic intensity is defined as follows:

$$W = Operations\ per\ voxel\ *\ number\ of\ voxels\ *\ number\ of\ images, \tag{5.2}$$

and

$$Q = inbytes \ + \ outbytes, \tag{5.3}$$

with

$$inbytes = pixels\ loaded\ per\ voxel \ * \ bytes\ per\ pixel \ * \ number\ of\ voxels \ * \ number\ of\ images, \tag{5.4}$$

and

$$outbytes = 2 \ * \ bytes\ per\ voxel \ * \ number\ of\ voxels \ * \ number\ of\ images. \tag{5.5}$$

In this case, as the work and memory traffic is defined per voxel per image, the equation can be reduced to:

$$I = \frac{Operations\ per\ voxel}{Pixels\ loaded\ per\ voxel \ * \ bytes\ per\ pixel \ + \ 2 \ * \ bytes\ per\ voxel} \tag{5.6}$$

The source LolaBunny RabbitCT algorithm requires 26 additions, six subtractions, 29 multiplications, three divisions, four compares, and two floors to perform the backprojection algorithm for each voxel and each image. So, a total of 70 operations are executed. Additionally, four pixels are loaded from memory as input; and a single output voxel value is loaded, updated, and written back to the memory, resulting in two transfers. Additionally, both pixel and voxel data are stored as floating point variables, resulting in them requiring four bytes each. As the number of voxels could be removed from the equation, the arithmetic intensity is benchmark independent and results in an arithmetic intensity of 2.92 for each benchmark.

Demosaic Arithmetic Intensity

For the Demosaic initial implementation, the arithmetic intensity is defined as follows:

$$W = Operations\ per\ pixel \ * \ number\ of\ pixels \ * \ number\ of\ images, \tag{5.7}$$

and

$$Q = inbytes \ + \ outbytes, \tag{5.8}$$

with

$$inbytes = pixels\ loaded\ per\ calculation * bytes\ per\ pixel * number\ of\ pixels * number\ of\ images, \tag{5.9}$$

and

$$outbytes = 3 \ * \ bytes\ per\ pixel \ * \ number\ of\ pixels \ * \ number\ of\ images. \tag{5.10}$$

As with the RabbitCT calculation, the calculations are pixel and image independent and can be removed. This reduces the equation to:

$$I = \frac{Operations\ per\ pixel}{(Pixels\ loaded\ per\ calculation \ + 3) \ * \ bytes\ per\ pixel} \tag{5.11}$$

The initial Demosaic implementation requires 95 additions, 16 subtractions, 157 multiplications, two divisions, 86 compares, two modulo, and two rounding functions per calculated pixel. So, a total of 360 operations are executed for each pixel. For the memory transfers, a total of 25-byte sized pixels are loaded from memory for the matrix multiplication, and three-byte sized colour channels are written back to the output. This leads to an arithmetic intensity of 12.86.

Roofline plot

Combining the hardware characteristics and algorithm arithmetic intensities results in the Roofline plot shown in Figure 5.3. It shows that the selected GPU offers significantly more floating point compute performance and memory bandwidth for a device with a similar cost to the selected FPGA. Next, the intersection point of the arithmetic intensity of an application with the Roofline plots of the hardware shows whether the application will be limited by the memory bandwidth, or the computational performance, assuming no other factors limit the execution speed. Applications that intersect the diagonal part of the graph are memory bottlenecked, and applications that intersect the horizontal sections are compute bottlenecked. In this case, a parallel implementation of the RabbitCT source code is always memory bottlenecked, and the Demosaic source algorithm is compute bottlenecked on the host PC, but memory bottlenecked on both the GPU and FPGA.

Finally, it should be noted that this graph only compares the floating point performance between hardware. So, the predicted performance differences are not necessarily true for the integer or double precision calculations present in both the RabbitCT and Demosaic algorithms, reducing the usefulness of this Roofline plot.

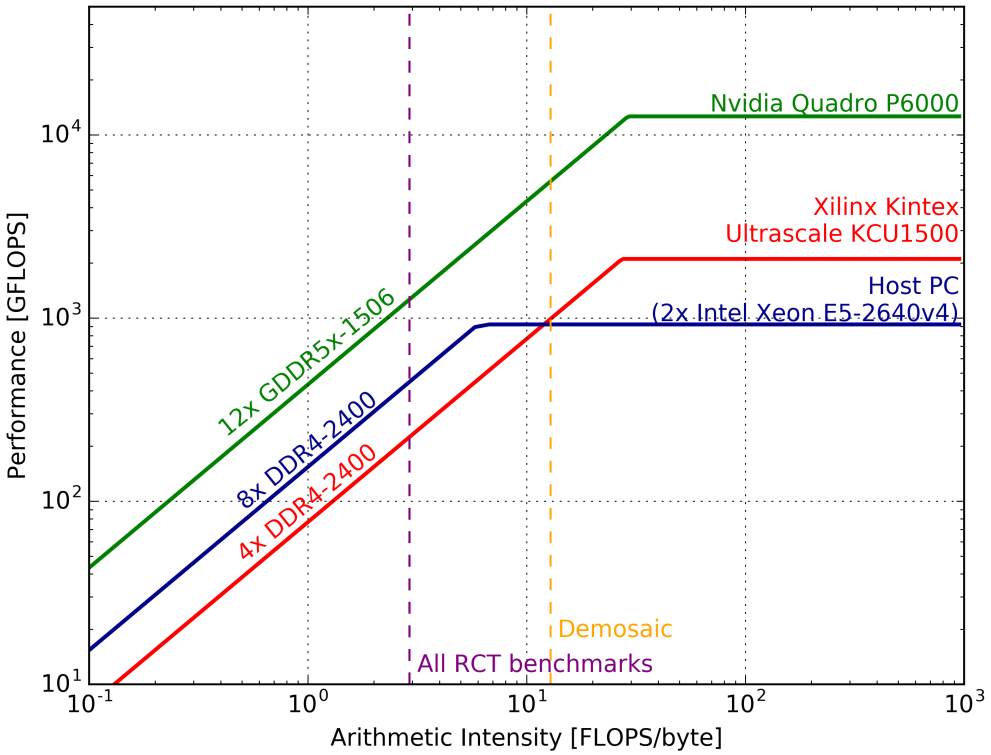


Figure 5.3: Roofline initial implementations

Chapter 6

Algorithm Implementation

The optimization of each algorithm for each platform was done iteratively. This chapter explains the applied optimizations and shows the results that these optimizations achieved. Initially, it was planned to include the power usage of the final implementations of each algorithm. However, due to the loss of the testing setup, these measurements could not be performed.

6.1 RabbitCT

For RabbitCT, the code for each created platform and API started from the LolaBunny source code described in Section 4.1.2. The source code was ported to and iteratively optimized for the respective APIs and platforms to maximize the performance. Initially, a CUDA RabbitCT implementation was created and optimized. Afterwards, the OpenCL GPU implementation was created using similar optimization strategies. For OpenCL, optimizations were applied in a different order due to a better understanding of each optimization technique. To compare both OpenCL and CUDA, the CUDA implementation was recreated from scratch. Only this time the optimization techniques used with OpenCL were implemented in the same order, resulting in both CUDA and OpenCL GPU implementations with a similar code base. Now, the only differences between the CUDA and OpenCL implementations are caused by API differences, allowing a clear CUDA GPU vs OpenCL GPU comparison to be made. This optimization path could not be used for OpenCL on FPGA, as this requires FPGA specific optimizations.

6.1.1 RabbitCT GPU

This section covers the optimization strategies applied to both the CUDA and OpenCL implementations of the RabbitCT algorithm. For each implementation, the state of the algorithm was analyzed to determine the main performance bottleneck. Afterwards, optimizations reported in Section 3.5 and the corresponding API programming guides were used to resolve the found bottlenecks and improve the performance of the algorithm. As both CUDA and OpenCL offer similar optimization techniques, the same optimization techniques were used for both APIs, allowing for easy comparison.

The applied optimizations of each benchmarked implementation are explained in the sections below, and the results are shown in Table 6.1 for CUDA, and Table 6.2 for OpenCL. These results show a selection of the RabbitCT output data for a 512 resolution together with the memory statistics reported by the Nvidia System Management Interface (Nvidia-SMI) [140]. Here, the total run-time represents the total run-time of the entire RabbitCT benchmarking algorithm for a 512 resolution. Next, the Mean Squared Error (MSE) and the Peak Signal to Noise Ratio (PSNR) show the magnitude of calculation errors on the output, and the memory usage shows the total global memory usage of the GPU and gives an indication of the memory requirements of the

algorithm. Finally, some implementation results were not recorded or got lost in the fire. These results are marked by a slash in the table.

Table 6.1: RabbitCT CUDA GPU implementation results

Implementation	Total runtime 512 (s)	MSE (HU^2)	PSNR (dB)	Memory usage (MB)
0 - LolaBunny	42.86293	0	inf	683
1 - Arguments	37.90586	0	inf	681
2 - Memory	20.1344	0	inf	681
3 - Precision	9.29296	7.83e-4	103.307	681
4 - Threading	2.85317	7.87e-4	103.283	681
5 - Multi-image	1.73989	7.79e-4	103.328	683
6 - Streams	1.08831	7.79e-4	103.328	693
7 - Textures	/	/	/	/
8 - Tweaking	0.667255	/	/	/

Table 6.2: RabbitCT OpenCL GPU implementation results

Implementation	Total runtime 512 (s)	MSE (HU^2)	PSNR (dB)	Memory usage (MB)
0 - LolaBunny	/	/	/	/
1 - Arguments	34.2801	0	inf	675
2 - Memory	20.198	0	inf	675
3 - Precision	8.37173	8.96e-4	102.723	675
4 - Threading	2.11372	8.96e-4	102.723	675
5 - Multi-image	1.65845	8.91e-4	102.744	679
6 - Streams	0.996018	8.91e-4	102.744	1433
7 - Textures	0.870298	8.91e-4	102.744	1309
8 - Tweaking	0.655967	8.95e-4	102.727	1329

Implementation 0 - LolaBunny

The goal of the initial GPU implementation was to allow the backprojection algorithm to run in parallel on the GPU. Here, no optimizations are applied other than allowing the code to take advantage of multiple CUDA cores present.

A first GPU implementation was created by changing the `RCTLoadAlgorithm` to allocate a piece of memory on the device for the input image, projection matrix, output volume, and the global data struct. This global struct contains all the settings and data for the current RabbitCT execution. For the first GPU implementation, it is implemented just like the source LolaBunny algorithm; however, this time its pointers point to memory on the accelerator device instead of host memory. After initialization, the global struct is copied to the device and is not changed anymore as all its data is constant.

Next, the `RCTAlgorithmBackprojection` function is adapted to copy the received input image and projection matrix to the device pointers created in the `RCTLoadAlgorithm`. With the data on the device, the backproject function is called using a kernel call function. This function executes the backprojection algorithm with three-dimensional thread blocks that process eight threads per dimension, or 512 threads in total. To enable parallel processing, two main changes are made to

the backproject function. The first change is the replacement of the three for-loops by a thread identifier function representing the iterators of the three loops. The second change is the addition of a CUDA `__global__` or OpenCL `__kernel` statement in front of the function definition. These statements tell the compiler that this function is a GPU kernel that can be called from the host. Similarly, the two helper functions `p_hat_n` and `p_n`, which perform the bilinear interpolation and clamped pixel loading respectively, receive the CUDA `__device__` statement in front of the function definition to indicate that those functions can only be called by another CUDA function and can only be executed on the CUDA accelerated device. This statement is not required for OpenCL as all kernels are located in a different file, from which each function can only be executed by the OpenCL accelerated device.

For CUDA, this direct port of LolaBunny resulted in a total run-time of 42.863 seconds. For OpenCL, the compiler was unable to compile the code due to an unsupported global pointer redefinition in the kernel. This issue is resolved in the next implementation.

Implementation 1 - Arguments

The goal of implementation 1 is to enable OpenCL compilation by replacing the locally defined global pointer with another data sharing method. Lolabunny uses this global pointer to share the global data struct it receives as a kernel parameter with the backprojection helper functions. By parallelizing the execution of the backproject function, each thread will write the same input parameter data to this global address, introducing a large memory bottleneck. Having each thread write the same data to this global variable is a large waste of resources and should be avoided. Luckily, the CUDA compiler detects this inefficiency and resolves it by allowing only a single thread to write to this variable, preventing the memory bottleneck from taking place. However, even without this memory bottleneck, this global pointer still has to be defined and created before the helper functions can read it, and thus still causes a slight performance reduction.

To resolve the issue of defining a global pointer, the global data struct is shared in the function arguments of the helper functions. This way the helper functions still receive a pointer to the global data, but no global pointer redefinition is required. Additionally, the device memory pointers defined in the global struct are moved to function arguments to reduce the double pointer accesses required to load the data. With these small changes, the OpenCL implementation now compiles, and a small performance increase and memory usage reduction are shown for the CUDA implementation.

A result comparison shows that both the CUDA and OpenCL implementations have a similar performance with a slight edge for OpenCL. Additionally, for unknown reasons, OpenCL uses 6 MB less global memory than the CUDA implementation, which is a negligible difference for a 24GB GPU.

Implementation 2 - Memory

The next implementation experimented with further memory optimizations. The first optimization defines variables as constant or read-only when used as such, this did not lead to any performance gains as the compiler already detects this and optimizes for it. Next, unused global RCTData variables are removed, which introduces a minor speedup as it reduces the number of GPU registers that are required per thread. The final memory optimization present in this implementation is the use of constant restricted memory pointers for the global memory input variables. This tells the compiler no pointer aliasing is present, enabling the compiler to cache the global data, which allows the global memory intensive RabbitCT algorithm to speed up significantly. For OpenCL, the constant function argument definition was also tested, but this gave the same performance results as defining the variable as a constant restricted global pointer.

Implementation 3 - Precision

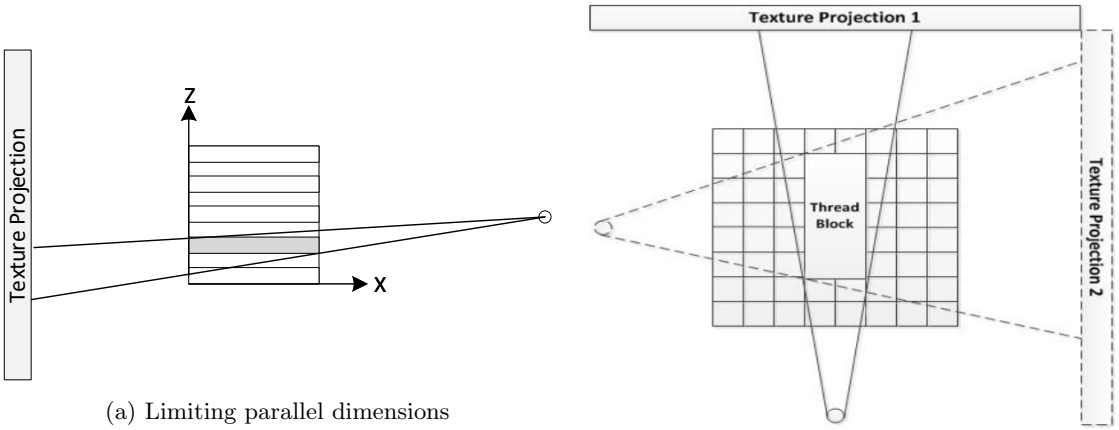
An analysis of the CUDA version of implementation 2 with the Nvidia Visual Profiler [141] showed that the double precision units of the GPU are the main performance bottleneck. As explained in Section 3.5, changing from double precision to either single precision or integer introduces a performance gain at the cost of precision. For RabbitCT, all calculations are performed in double precision, and the output is stored in single precision. So, converting all double precision calculations to single precision leads does not introduce large errors, but does lead to rounding errors showing in the output. The results in Table 6.1 and Table 6.2 confirm this, showing double the performance at the cost of a reasonably small error in the output.

Comparing the performance of CUDA with OpenCL shows that the OpenCL implementation finishes the algorithm about one second faster than CUDA, making OpenCL about 10% faster. However, the move to single precision also resulted in OpenCL having a somewhat lower precision in the final results. With no kernel code differences present other than the API definitions, a likely cause of the performance difference is the OpenCL compiler scheduling the code in a slightly different way than the CUDA code. This different schedule results in a more efficient calculation, but also in a reduction of accuracy.

Implementation 4 - Threading

Profiling the single precision implementation showed that memory latencies became the next performance bottleneck. To optimize the memory usage and improve the cache usage of the algorithm, thread optimizations have been applied. The first optimization is an overhead reduction by having a single thread compute multiple voxels. This change reduces the total number of threads required to calculate the output and thus also the thread initialization overhead. So, instead of having a single thread for each voxel in the X, Y and Z dimension, this implementation contains threads that compute all voxels along the Z-axis, limiting the parallelism to the X and Y dimension only. As shown in Figure 6.1a, limiting the parallelism to the X and Y dimensions leads to a thread block that requires data from a single slice of the input image at a given time, limiting the amount of data that must be loaded for each calculation and thus improving the data locality. This optimization reduced the processing time of the CUDA algorithm from 9.3 seconds to 7.6 seconds, and the OpenCL algorithm from 8.3 seconds to 7.3 seconds.
Next, Figure 6.1b visualizes that the sizing of the thread block also affects the data-locality. So, the thread blocks were iteratively resized to an optimum size of 32 by eight voxels for both APIs, which reduced the run-time to 5.23 seconds for CUDA, and 4.6 seconds for OpenCL.

As the memory latency issue was not fully resolved by optimizing the threading, further memory improvements have been applied as well. Directly sending the RabbitCT struct data to the helper functions instead of providing a pointer to the struct reduced the CUDA run-time to 2.85 seconds and the OpenCL run-time to 2.56 seconds. By providing this data directly, the function access the data from local memory instead of a global memory pointer and thus removes several global memory data accesses from the algorithm.
The final optimization applied for this implementation is the use of loop unrolling. By writing an unroll loop pragma above the Z-axis loop, the compiler places multiple iterations of the loop after each other in machine code. This optimization reduces the number of branches required to check the loop bounds and allows the compiler to perform more extensive code optimizations as more code is present. For CUDA, bounded loop unrolling reduced the performance and an unbounded loop unroll kept the performance at the same level. So, the compiler likely already unrolls this loop to create an optimal loop structure for the GPU. For OpenCL, the best performance was achieved by unrolling the loop four times, which reduced the total run time to 2.11 seconds. With these results, the final OpenCL performance for implementation 4 becomes about 25% faster than CUDA.



(a) Limiting parallel dimensions

(b) Thread block optimization, taken from [61]

Figure 6.1: RabbitCT Thread Optimization

Implementation 5 - Multiple images

In addition to using a single kernel to process the entire Z-axis, this implementation lets threads process the data from multiple input images. Originally, the global output voxel data was updated for each image, requiring a memory write for each thread. By processing multiple images in a single kernel, this output value only needs to be read and written once for each set of images processed, reducing the required memory bandwidth.
A downside to processing multiple images is that this might interfere with the automatic data caching present in the GPU. The additional input images might overwrite previously cached data that is still required for future computations, which then must be loaded from the global memory again. This statement proved this statement as the best run-time improvement was achieved when processing two images with a single kernel.

In addition to multi-image processing, the use of an atomic add function increased the CUDA performance by several milliseconds. An atomic add combines the read, addition and write command in a single instruction, which is executed on a separate on-chip ASIC for Nvidia GPUs [61]. For this reason, using atomics relieves each thread from performing these tasks. The performance gain is minimal because the compiler likely schedules the read earlier in the code, resulting in only the skipped addition and memory write commands from improving the performance. For OpenCL floating point atomic adds are not supported, and thus could not be used.

Comparing the final results shows that the performance gap between OpenCL and CUDA has decreased, with OpenCL being 5% faster. This result shows that the performance gain of OpenCL is likely caused by a more efficient memory usage due to the different schedule the OpenCL compiler provides.

Implementation 6 - Streams

Profiling the multi-image implementation showed two issues. The first being the floating point performance, and the second issue is that there are no simultaneous memory transfers during execution. To tackle this last issue, a multi-stream implementation has been created. A CUDA stream or OpenCL command queue is a FIFO queue listing the tasks the GPU should perform. With a single command queue, the GPU is limited to a single task at a given time, enabling it to either copy data between host and device, or execute a kernel, but not both in parallel. By using

multiple streams, GPU commands can be executed in parallel when the hardware is available. Not only can data be copied between host and device in parallel with a kernel execution, but it also enables for parallel kernel execution when processing elements are available. So, the use of multiple streams does not improve the efficiency of the kernel code itself, but it does improve the efficiency of how this kernel is being executed. Figure 6.2 shows the difference between a single stream and a multi-stream implementation for a kernel bottlenecked application.

A downside to using multiple streams is that both the host and the device must be able to store the data of each command issued in each parallel stream. This data must remain available until all GPU commands using this data have finished executing. To schedule all GPU copy and execution tasks, the host must be able to prepare multiple GPU calls so that the GPU can read data from or write data to the host whenever it is able to. Additionally, to allow the host to process other tasks when the GPU is busy requires the use of pinned in- and output buffers for the GPU. Pinned memory buffers are buffers that are page-locked, this means that this data is always present on the RAM, and cannot be swapped out to the disk. By keeping all data in the host RAM, no CPU is required to load the data, and the GPU can copy this data using its Direct Memory Access (DMA) module enabling significantly faster memory transfers. Without pinning the memory buffers, the GPU requires help from the CPU to load the data, preventing any parallelism between GPU kernels as the host device cannot move ahead to prepare the next GPU kernel iterations.

Each RabbitCT iteration only requires new input data as the output data is reused and remains on the device. So, pinned input buffers for both the images and matrices had to be created. These input buffers store input for 124 iterations of the backprojection algorithm, which is the number of inputs the RabbitCT-Runner host application prefetches at a given time. By copying this input data to the input buffer and queuing up the processing commands to the GPU, the host can fetch the next 124 images from disk while the GPU is processing the input buffer data. Even though fetching the data from the disk is not counted as processing time for the RabbitCT benchmarking data, it still offers an improvement on the overall algorithm efficiency as multiple kernels can execute in parallel. So, by enabling parallel data transfers and kernel execution, a significant performance increase was achieved without implementing any kernel changes. These results show that host code optimizations are just as important as kernel optimizations to enable high-performance acceleration using both APIs.

While the performance gain for both CUDA and OpenCL by using streams with pinned memory is similar, the reported memory usage is not. For CUDA, creating a pinned memory buffer only requires the cudaMallocHost() function call to create the pinned host buffer, and the rest of the code remains as it was with a device buffer initialization and memory copy functions. Here, the API detects that data from a host buffer is copied to the device, which automatically maps the memory spaces and allows the use of a DMA for fast and parallel transfers.
The OpenCL implementation is based on the one shown in the Nvidia OpenCL Best Practices guide [10]. Here, a host and device buffer are created, and the two memory spaces are mapped to each other by the clEnqueueMapBuffer() command. This command returns a pointer to a separate device buffer created for the mapping. So by mapping the host and device buffers, an additional buffer is created, resulting in the high memory usage reported by the OpenCL implementation. A solution to this issue exists by manually mapping and unmapping the buffers for every host-to-device transfer to limit the additional buffer space required. This way, only a second buffer with the size of a single transfer is required instead of a copy of the entire stream buffer. However, after many iterations, no error-free streaming implementation could be created without stalling the host, so the high memory usage remained present in all following OpenCL implementations.

Implementation 7 - Texture memory

The last applied optimization is the use of texture memory. A texture is a 2D or 3D image applied to a 3D polygon model to colour the colourless polygon model. A video game scene

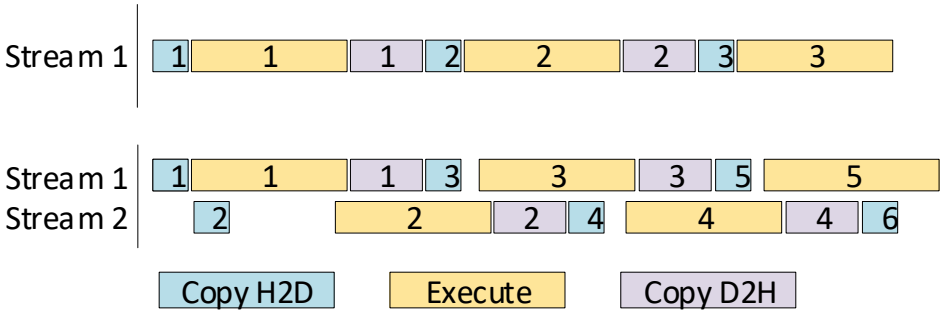


Figure 6.2: The advantage of streams

consists of many textures which can change constantly depending on the scene. To support fast texture changes and to prevent excessive texture loading, textures are stored in the global texture memory and are cached on-chip. Additionally, texture objects have extra parameters defining the image parameters, and support hardware accelerated pixel interpolation and boundary condition checking.

For RabbitCT, the image and matrix inputs are implemented as 3D texture objects in CUDA, and a 2D image array in OpenCL. Here, the first and second dimension of each object represents the size of the input image or matrix, and the third dimension is used for the input of other iterations. Using three-dimensional objects enables a single texture object to contain all the data required for several iterations of the algorithm. Additionally, by setting the texture addressing mode to border in CUDA and clamp in OpenCL, out of bounds accesses to this texture automatically return zero. This setting removes the need for the `p_n` border detection function which removes all branches from the kernel code and thus significantly increases the algorithms throughput.
Additionally, using the hardware accelerated linear interpolation on the textures could also remove the need for the `p_hat_n` bilinear interpolation function. However, the use of the hardware bilinear interpolation did not provide a noticeable performance gain and significantly reduced the accuracy of the algorithm due to it only providing 9-bit precision. So, hardware interpolation was not used for the final GPU implementation.

At the time of the fire which destroyed the benchmarking setup, the CUDA texture memory implementation with code similar to OpenCL was still a work-in-progress and thus not benchmarked. As two CUDA implementations have been created, the results from the first iteration are still available. However, the texture memory optimization was applied much earlier in the optimization process, preventing a CUDA vs OpenCL comparison from being made.

Implementation 8 - Tweaking

The last optimization step is the tweaking of all optimization constants. Here, the input buffer size, multi-image factor, number of streams, and the thread block parameters are changed to find their optimal values.

For OpenCL, all values except the multi-image factor were at their optimum value: the buffer size remained at 124, the threading parameters remained 32 by 8, and the optimum number of streams remained two. Changing the multi-image factor to 4 however, offered a performance increase and reduced the run time to 0.656 seconds. Here, the introduction of texture memory likely improved the caching behaviour of the algorithm, allowing more images to be processed in a single kernel iteration.

For the second CUDA optimization iteration, no tweaking was applied due to the benchmarking setup being lost to the fire. However, the final optimization results from the first CUDA implementations can be used. Here, the same optimizations are applied, but in a different order. Also, this build was not created in tandem with the OpenCL implementation, so code differences are present. This CUDA implementation uses an array of textures instead of a single 3D texture object and uses several arithmetic code changes which speed up calculations. Despite the code differences, this CUDA implementation also uses two streams and a 124 element buffer, but it also uses a 16 by 16 thread block and processes the voxels of eight images with a single kernel. Despite the differences, this CUDA implementation finished the RabbitCT algorithm in a total of 0.667 seconds, which is only 2% slower than the OpenCL implementation.

Ineffective optimizations

Next to the effective GPU optimization described above, several optimizations did not affect the performance of the algorithm. Which is likely caused by the compiler already automatically applying these optimizations with -O3 compilation enabled. These optimizations are:

- Declaring the function arguments and variables as constant if they do not change
- Moving loop independent calculations out of the loop
- Shifting the position of calculations to improve the hardware scheduling of calculations
- Replacing divisions by multiplications with the inverse of the divider
- Replacing division calculations by the fast inverse square root division approximation function

The top two items are still present in the code, as these optimizations improve the code readability. However, the last three optimizations make the code harder to read and thus have not been implemented.

Future improvements

Not many improvements remain untested for the GPU implementations of RabbitCT. Tests could be performed to see the performance impact of using both a read-only input and write-only output volume for the images. Next, shared memory could be used as input for thread blocks which could improve the caching performance somewhat. Hardcoding the loop bounds could also improve the performance as this removes the global data struct and makes it easier for the compiler to optimize the code. However, this does remove the users' ability to change any backprojection setting without adapting and compiling the code themselves. Switching to half-precision would theoretically also increase the performance while also decreasing the accuracy of the algorithm. However, as noted in Section 3.5, the selected Pascal GPU is not optimized for half-precision calculations as it offers only two results per clock cycle per multiprocessor.

Finally, using the next generation of GPU hardware will give another performance boost as the number of CUDA cores has been increased by 20%, and the addition of the tensor cores on Nvidia hardware might speed up the matrix multiplications present in the algorithm.

Implementation conclusion

The final GPU builds for both APIs run the RabbitCT benchmark with a 512 resolution in 0.66 seconds, with a signal to noise ratio between 102.7 and 103.3 dB. Compared to other results on the RabbitCT Ranking website [115], the created GPU implementation achieves a performance between the fourth and fifth position shown on the website, but with significantly higher accuracy. Our achieved accuracy is better than all but the slowest three CPU implementations. Additionally,

this research uses a single GPU, while the top three solutions all use multiple GPUs. So, for a single GPU implementation, this implementation takes the first place based on both accuracy and performance. However, the top solutions in the ranking report that CUDA 5.5 has been used with multiple GPUs. As CUDA 5.5 was released in 2013, this research likely used GPUs that were released in that year as well. So, this implementation taking the first place for its achieved performance and accuracy with a single GPU is probably caused by architectural GPU improvements instead of improvements to the algorithm itself.

6.1.2 RabbitCT FPGA

This section covers the optimization strategies applied to the OpenCL FPGA implementation of the RabbitCT algorithm. For each implementation, the algorithm is analyzed for its bottleneck, and optimizations from Section 3.5 to resolve these issues are applied and tested. As the compilation time for the FPGA kernel takes hours instead of the seconds a GPU compilation takes, functionality is first tested in software. When this software implementation is functional, multiple FPGA builds are compiled for software emulation, which takes several minutes. This compilation method performs the first few steps of FPGA compilation which provides estimations about the actual FPGA compilation and allows for software emulation of the FPGA kernel. With the estimated FPGA build differences known, a select few builds are compiled for the FPGA hardware to determine the actual hardware performance. From each set of builds, the best performing build is selected, which then forms the basis for the next implementation. For the first four implementations, all builds were compiled for the FPGA hardware as no worst-case latency relation was found in the software emulation results. From implementation five and six, software emulation was used to compare builds which significantly increased the implementation speed as build results could be compared within minutes instead of the hours required for compilation and the hardware execution itself.

The applied optimizations and differences between builds for each implementation are explained in the sections below. Here, the first FPGA implementation uses the code of implementation 1 of the OpenCL GPU builds. For this reason, the implementation number starts at 1, after which FPGA specific adaptations are made to optimize the code for FPGA acceleration further. Note that due to the long compilation and testing time, a 128 resolution is used for benchmarking the FPGA. Using this resolution reduces the FPGA workload a factor 4 in each dimension, resulting in a total workload reduction of 64x compared to the 512 resolution used with the GPU build testing. An overview of the FPGA hardware benchmarking results with a 128 resolution are shown in Table 6.3. It allows the most important results of each implementation to be compared and uses a "/" to indicate any results that are lost or cannot be obtained. Additionally, for GPU comparison the estimated 512 resolution run-time is added as well. For this run-time estimation, a linear increase in run-time by factor 64 is expected as increasing the resolution increases the loop bounds of each dimension. So, increasing the resolution by 4x in three dimensions results to 64 more voxels that need to be calculated, and thus 64x more pipeline iterations for the FPGA. However, whether the assumption that total run-time linearly increases with the resolution has never been tested and could be wrong as changing the resolution also affects the memory access patterns.

In addition to the run-time, the most interesting results for each implementation are shown in the respective sections, with the selected data representing the following:

- Burst-size: number of voxels processed in parallel
- Total runtime 128: measured kernel run-time for the benchmark with a 128 resolution
- Estimated runtime 512: run-time estimation for 512 resolution benchmark, equals total runtime 128 multiplied by 64
- MSE: reported mean square error

- PSNR: reported peak signal to noise ratio
- Max WC Latency: maximum reported worst-case latency estimation, this value does not include latencies associated with data transfers to global memory
- FPGA freq: clock frequency at which the FPGA executes the kernel, affected by compilation timing results
- Impl. pipelines: number of pipelines that are implemented
- Max pipeline depth: length of the longest pipeline in stages
- Max II: maximum instruction interval in cycles, shows how many cycles it takes before a new instruction can be issued in a pipeline
- LUT Usage: FPGA resource used the most of all resources, represents the resource limiting factor
- BRAM Usage: FPGA memory block usage
- Compile time: time required to compile the FPGA hardware binaries

Finally, after explaining all implementations, a list of possible future improvements is given, and a conclusion is made.

Table 6.3: RabbitCT FPGA implementation final results

Implementation	Total runtime 128 (s)	Estimated runtime 512 (d:h:m:s)	MSE (HU^2)	PSNR (dB)	FPGA Freq (MHz)	Max II (cycles)	LUT Usage (%)	Compile time (h:m:s)
1 - Work-items	6467.64	04:18:58:49	0	inf	300	no pipeline	7.24	01:52:44
2 - Loop experiments	6107.16	04:12:34:18	0	inf	300	no pipeline	5.66	01:43:29
3 - Hardcoding	3341.28	02:11:24:02	0	inf	300	no pipeline	5.25	01:30:14
4 - Vectorization	592.961	00:10:32:30	0	inf	300	unrecorded	13.02	02:23:10
5 - Memory Opt.	28.8468	00:00:30:46	0	inf	188.3	15	6.65	04:37:20
6 - Prefetching	12.22	00:00:13:03	0	inf	/	5	/	/
X - Fully Opt.	0.153	00:00:00:10	/	/	300	1	/	/

Implementation 1 - Workitems

The first RabbitCT FPGA implementation starts from the first compilable OpenCL GPU code. Here, the FPGA is set as the target device, and the Xilinx SDAccel compiler is used to generate the FPGA binaries. For this implementation, the three-dimensional eight by eight by eight threaded GPU implementation is compared with a single work-item implementation, where only a single thread is launched with three nested loops iterating over the voxels. The single work-item method is recommended for high level synthesis as FPGAs excel in a pipelined execution, and having more work per thread enables the compiler to synthesize a better pipeline. The results of this comparison are shown in Table 6.4 where the single work item not only performs faster, it also uses fewer resources and compiles faster.

Table 6.4: RabbitCT FPGA implementation 1 results

Implementation	Total run-time 128 (s)	Max WC Latency (cycles)	FPGA Freq (MHz)	LUT Usage (%)	Compile time (h:m:s)
NDRange 8,8,8	9039.83	1064560	300	19.02	03:06:55
Single Work-item	6467.65	undefined	300	7.24	01:52:44

Implementation 2 - Loop experimentation

The initial results are not very promising with the FPGA implementation being almost 200x slower for a benchmark that requires 64x less work than the GPU benchmark. An analysis of the compilation logs quickly showed why: the compiler failed to implement a pipeline and thus computes each voxel result sequentially. To improve the performance of the nested loops, implementation 2 experiments with pragmas and compiler optimization techniques to improve the loop compilation. Here, the pipeline and unroll pragmas are used on the main nested loops, and the effect of the inline pragma is tested on the helper functions. The inline pragma tells the compiler to insert the function code at each function call, resulting in the helper functions being implemented directly in the main loops instead of them being implemented as separate FPGA functions. Additionally, a multi-compute-unit approach was tested where the work of the outer loop is split into threads that are executed in parallel over all compute units that can be implemented in the FPGA.

The results in Table 6.5 show that no significant performance improvements were gained. The pipeline builds still did not have a successfully synthesized pipeline as the loop bounds are variable. Next, the default unroll build gave the same results as implementation 1, indicating that the compiler predicatively unrolls a loop by several iterations if no loop bounds are known. Inlining the helper functions helps speeding up the algorithm and shows that having the helper functions directly in the main loops enables the compiler to perform more effective code optimizations.
Getting the compute-unit test to compile took many iterations. Initially, 15 compute units were expected to be the maximum that could fit on the FPGA hardware based upon the single unit hardware usage. As multi-compute-unit compilations could take up to two days before failing, it took a long time before ending up with the first successfully compiled build. The final build had five compute units, which theoretically should improve the processing speed of the original algorithm by a factor 5. However, as the multiple compute units shared the same data and memory bus, a bottleneck was introduced that eliminated the effect of the created parallelism. Having five compute units processing the data of the outer loop in parallel resulted in an implementation with a similar per-cycle performance as the single compute unit implementation. Only this time the maximum performance is reduced due to the lowered FPGA clock frequency caused by timing failures introduced by the increased hardware usage. Due to the long compilation time, compute units were not further analyzed in this research, and all effort was spent on optimizing a single kernel implementation.

Table 6.5: RabbitCT FPGA implementation 2 results

Implementation	Total run-time 128 (s)	Max WC Latency (cycles)	FPGA Freq (MHz)	LUT Usage (%)	Compile time (h:m:s)
Pipeline	8018.93	undefined	300	7.23	01:46:14
Pipeline + inline	6494.58	undefined	300	5.75	01:49:26
Unroll	6467.78	undefined	300	7.24	01:51:41
Unroll + inline	6107.16	undefined	300	5.66	01:43:29
Compute units	8070.08	undefined	246	37.38	04:53:37

Implementation 3 - Hardcoding

The main issue with implementation 2 was that the compiler did not know the loop bounds, preventing it from forming a pipelining or fully optimizing the unrolled code implementation. For this reason, this implementation replaces all variables that remain constant during execution by defines which makes them known during the compilation process. Assertions have also been added to ensure the user input variables are equal to the constant values defined in the code.

These optimizations removed the need for a global data struct, thus leaving only the input image, projection matrix, and output volume as global variables. By hardcoding the input values, all flexibility in code execution is removed and thus requires a recompilation or the loading of different pre-compiled binaries to execute the algorithm using different settings.

The results of these tests are shown in Table 6.6. With the loop bounds now know, the compiler can estimate the worst case latency of the implementation and is able to apply more code optimizations. Even though a pipeline or unroll pragma was used for the main loops without inlining the helper functions, the compiler synthesized the same FPGA implementation for the first three tests as indicated by the similar hardware usage and latency numbers. The unroll implementation was not tested on the hardware because it shows the same compilation results as the default and pipeline compilation results, making it very likely the run-time results will be similar as well.

Next, when using inlined helper functions, both the unroll and pipeline code resulted in a very similarly performing FPGA implementation. Here, the maximum latency has increased by 7% and the run-time is reduced by 30%. The latency increase is caused by the helper functions being included in the main loop, increasing its latency instead of being reported as a separate function with its own latency. The run-time decrease is caused by the compiler being able to optimize the code more efficiently with all code in a single loop instead of also having to manage the communication between two FPGA functions that have been implemented separately.

Even with these improvements, the compiler guidance report indicates that for each implementation no loops were pipelined due to too large carried dependencies being present. A carried dependency is an issue where a value acquired in a later pipeline stage is needed for the calculation of the next cycle, which results in the pipeline being stalled until this value is available. An example of a carried dependency is a program that reads a value A and B, adds B to A, and stores this back in A. When executing sequentially, no issues are present as the new value of A is read at the start of the second iteration. However, when pipelining this code, A is read, updated and written every cycle. So, to read the latest value of A, the pipeline has to wait until the previous A is written, stalling the entire pipeline and making the execution of this program sequential.

Table 6.6: RabbitCT FPGA implementation 3 results

Implementation	Total run-time 128 (s)	Max WC Latency (cycles)	FPGA Freq (MHz)	LUT Usage (%)	Compile time (h:m:s)
Default	4789.72	2.252e9	300	6.99	02:06:02
Pipeline	4789.78	2.252e9	300	6.99	02:07:43
Unroll	untested	2.252e9	300	6.99	02:07:49
Pipeline + inline	3341.08	2.408e9	300	5.25	01:30:14
Unroll + inline	3344.44	2.414e9	300	5.22	01:35:29

Implementation 4 - Vectorization

To tackle the issue of the carried dependencies, the code was vectorized to parallelize the execution of the program. Dividing the code into vectorized segments allows the memory loads and stores to be scheduled in bursts. Here multiple memory transactions take place with a single instruction which reduces the busy time of the memory bus and reduces the carried dependencies present. For vectorization, the RabbitCT backprojection kernel was split into three sections. The first section contains backprojected row and column calculation, the second section contains the loading and bilinear interpolation of the input, and the final section contains the normalization and updating of the output volume. The vectorization of each section is implemented by adding for-loops around each section. When pipelining, the compiler unrolls these inner loops a constant number of times defined by a global burst-size variable. So, instead of each pipeline stage working on a single

RabbitCT voxel at a given cycle, each pipeline stage can now work on a burst-size number of voxels in parallel.

The results of this implementation for different burst sizes are shown in Table 6.7. It shows that implementing vectorization increases the performance in all cases due to the burst memory transfers that it enables. Theoretically, a higher burst-size should always lead to a higher performance due to the parallelism it enables. However, using a burst-size of 4 results in the best algorithm performance, as using a burst-size of 8 or higher prevents the compiler from successfully unrolling the vectorized loops for some reason.

Table 6.7: RabbitCT FPGA implementation 4 results

Implementation	Total run-time 128 (s)	Max WC Latency (cycles)	FPGA Freq (MHz)	LUT Usage (%)	Compile time (h:m:s)
Burst-size 2	2503.52	1.934e9	300	7.44	03:25:43
Burst-size 4	592.961	undefined	300	13.02	02:23:10
Burst-size 8	2039.97	1.573e9	286	23.52	07:10:30
Burst-size 16	2701.36	1.513e9	213.3	48.02	34:41:31

Implementation 5 - Memory optimization

The goal of the memory optimization implementation is to further reduce the carried dependencies present in the code to improve the throughput of the pipeline. For this implementation, multiple optimizations were tested using software emulation to determine the effect on the predicted worst-case pipeline latency. The latency is reduced by minimizing the instruction interval (II), which directly corresponds with minimizing the carried dependencies.

Similar to the GPU memory optimizations in Section 6.1.1 implementation 2, each variable was made constant/read-only when used as such, and each global input variable was made restricted to tell the compiler no pointer alias is present. This change had an unknown effect on the latency, as the compilation report of the burst-size 4 build of implementation 4 did not report the worst case latency. However, the resulting latency was 1.444e9 cycles.
Next, the RabbitCT volume was split into a separate read input volume and a write output volume. This split together with the addition of extra vectorization steps removed the carried dependency between the volume reading and writing. For each input image, the host swaps the two pointers so that the output volume of one RabbitCT image calculation becomes the input of the other. These optimizations significantly reduced the carried dependencies present and reduced the latency to 8.914e6 cycles.
Finally, all global variables were split over multiple FPGA memory banks, which increases the memory bandwidth of the application and further reduced the worst-case latency to 8.39e6 cycles.

With these optimizations, another burst-size benchmarking test was performed, resulting in the data shown in Table 6.8. Here, the decrease in latency and stall cycles resulted in less time being available for signal propagation, causing more timing failures. The compiler automatically resolves the timing failures by lowering the clock frequency, which has the undesired effect of reducing the performance. Next, the achieved results nicely show the effect of the instruction interval and the number of pipelines. The burst-size 4 implementation has a carried dependency of 16 because four voxels are calculated in parallel, which each requires four pixels to be loaded from memory and thus result into a 16 cycle pipeline stage or a 15 cycle pipeline stall. By performing four times more work with a burst-size of 16, the stall cycles are increased to 63, negating the performance gain by the additional parallelism and thus resulting in a similarly performing FPGA implementation. The cause for the performance difference between the burst-size 4 and 8 implementation is the lower FPGA clock frequency due to the increased hardware

usage. Finally, the burst-size 8 implementation again proved difficult for the compiler as it failed to unroll an inner loop resulting in a less efficient multi-pipeline implementation.

Table 6.8: RabbitCT FPGA implementation 5 results

Implementation	Total run-time 128 (s)	Max WC Latency (cycles)	FPGA Freq (MHz)	Impl. pipelines (Nr)	Max pipeline depth (stages)	Max II (cycles)	LUT Usage (%)	Compile time (h:m:s)
Burst-size 4	28.8468	8.389e6	188.3	1	542	15	6.65	04:37:20
Burst-size 8	1648.5	undefined	300	2	4	1	30.99	04:25:06
Burst-size 16	33.9638	8.389e6	140.2	1	579	63	12.17	05:23:56

Implementation 6 - Prefetching

With the carried dependencies of the pixel loading being the main bottleneck of the algorithm, the only solution is to prefetch the data so that all data is immediately available on the FPGA. For this reason, a prefetching feasibility analysis is performed and is reported in Section 4.1.3. Here is shown that enough data re-use is present for prefetching to keep up with the calculations and that the required prefetch buffer should easily fit on the FPGA hardware.

After several weeks of analysis and testing prefetching options, the following had been implemented. The main algorithm has received an image buffer array sized 205 rows by 1248 columns to store the prefetched input data. Additionally, the buf_load_col, buf_load_row, and buf_target_row variables are introduced to track the status of the prefetch buffer. The buf_load_row and buf_load_col variables track which column and row have been prefetched into the buffer, and buf_target_row indicates which row should be prefetched. As the algorithm starts at row 960 and moves down to 0, the target row is determined by subtracting a constant reload offset of 15 rows from the lowest row currently in use by the algorithm. The reload offset ensures that new rows are prefetched on time so that the input data is available when needed. Its value of 15 is determined by the maximum number of new rows required per window as calculated in Section 4.1.3.
The maximum required rows for a single window and the reload offset both determine the size of the prefetch buffer. With the reload offset set to 15, at least 160 + 2x the reload offset rows are required to ensure the new prefetched data does not overwrite the data currently in use by the algorithm. However, this setting still resulted in prefetch errors where the data was not available in some cases. Thus the buffer size was increased to 160 + 3x the reload offset or 205 rows, which resolved this issue.

The prefetching itself occurs in bursts of 16 floating point elements with the vload16 statement, saturating the FPGA interconnect bandwidth. The loaded float16 data is then stored in the image buffer array at position (buf_load_row % 205, buf_load_col). Using a modulo equal to the size of the buffer allows the data position in the buffer to be calculated easily, and it ensures the new data overwrites the old data that is not required anymore.

Next to the changes required for prefetching, the `p_n` helper function that loads the pixels and performs the image edge clamping is replaced by a function that loads two neighbouring pixels via a vload2 function when possible to reduce the number of pixel accesses required from four to two. Finally, vload vectorization is applied to the output volume reading and writing as well, were a burst-size of volume input elements are read, updated and written back to the global memory at a given time.

The result of this prefetching implementation with a burst-size of 4 is shown in the no-partition result of Table 6.9. It shows a reduction of the instruction interval from 15 to 9, a run-time reduction with an equal factor, and a 5x increase in BRAM usage. The BRAM increase was

expected, but not such a slight reduction in run-time and instruction interval. The issue lies in the partitioning of data on the FPGA. Without stating how the data should be stored, the compiler assigns the buffer data sequentially over multiple BRAM elements, preventing efficient parallel accesses. The optimal partitioning method to reduce the carried dependencies to zero would be complete partitioning. However, this does not fit on the FPGA hardware for a 205 by 1248 floating point array. Leaving block and cyclic partitioning as the only options. As each voxel calculation always requires data from two columns and rows, optimally this data and the data for the other voxels calculated in this cycle should be accessible in parallel. Table 6.9 shows the software estimation results of several tests with the cyclic partitioning factor, and an intermediate hardware test of the cyclic 64 build showed that implementation was taking 12.22 seconds to finish the resolution 128 benchmark.

Table 6.9: RabbitCT FPGA implementation 6 results

Implementation	Total run-time 128 (s)	Max Worst Case Latency (cycles)	Max pipeline depth (stages)	Max II (cycles)	BRAM Usage (units)
No partition	16.46	4.719e6	542	9	672
Cyclic 16	/	4.719e6	542	9	672
Cyclic (40, 64, 128)	12.22	2.622e6	526	5	824
Cyclic 624	/	4.719e6	542	9	1474

A final RabbitCT FPGA prefetching implementation was never tested due time constraints. With two months of FPGA optimization not resulting in a performance that does not even match the first working GPU optimization, the choice was made to focus on the Demosaic algorithm. Demosaic has a significantly easier memory access pattern. Additionally, there are several similar OpenCL FPGA examples online which should help with getting the most out of the Demosaic OpenCL FPGA implementation.

Future improvements

With the final RabbitCT FPGA build being unfinished, several optimizations remain that could still increase the performance. The first optimization would be to finish the prefetch build with further testing of partitioning options.
If further partitioning does not lead to a reduced instruction interval, a second smaller prefetch buffer could be implemented that completely partitions all data in a 26 by 200 grid. This grid includes all voxels that will be accessed in parallel by a burst-size 16 pipeline as shown in Table 4.2 from the memory analysis in Section 4.1.3. This second prefetch buffer would need to be implemented as some kind of line buffer that efficiently loads the data from the initial prefetch buffer. However, such double prefetched implementations move away from the simplification that programming OpenCL on FPGA should bring.

Another improvement that is not tested for RabbitCT is the switch from double precision calculations to single precision. This change will significantly reduce the resource usage of the algorithm, which likely decreases the compilation time and enables higher FPGA clock frequencies that make the algorithm perform better. Next, increasing the benchmarking resolution might lead to better results as it increases the data re-use in the algorithm. Finally, further parallelization might be possible by using more memory banks. Currently, using a burst-size of 16 saturates the bandwidth for both the input and output memory channels. If the two remaining memory banks would be used for I/O as well, the parallelism can be doubled.

Assuming the instruction interval can be reduced from five to one (5x), the burst size increased from four to 16 (4x), two compute units can be implemented (2x), and the clock frequency can be doubled to the maximum of 300MHz (2x), a maximum 80x speedup can still be achieved with

the RabbitCT implementation. With this speedup, the resolution 128 benchmark will have a total run-time of 0.15 seconds, and the 512 resolution benchmark will have a total run-time of approximately 10 seconds.

Further optimizations require the use of RTL code to expand the available memory bandwidth by the use of the two external QSFP+ ports on the FPGA. Here, an additional image input data bandwidth 25GB/s can be provided when two 100gBe fibre connections are made. However, the use of this additional input requires the host to send data via this connection to the FPGA, and the FPGA BSP must be adapted to support an OpenCL kernels connection to this interface. Adding additional FPGA interfaces requires RTL code to be written and falls outside the scope of this project.

Implementation conclusion

The final FPGA build performed the 128 resolution benchmark within 12.22 seconds. As the 512 resolution benchmark contains 64 times more computational work, the FPGA implementation is expected to take 782 seconds for this implementation. However, whether the benchmarking time increases linearly with the resolution has never been verified. The achieved results place the RabbitCT FPGA implementation on the 18th position in the RabbitCT ranking [115]. Making it about just as fast as a multithreaded CPU implementation. However, with the above described future improvements implemented, the lowest achievable 512 resolution benchmarking run-time is estimated at 10 seconds, finally placing it within the performance range of the GPU implementations. With this estimated future performance, the FPGA would be placed at the 12th position on the RabbitCT ranking. Which proves that currently, an FPGA might not be the best choice for accelerating the RabbitCT algorithm because of the memory bottleneck hampering further performance improvements.

6.2 Demosaic

The Demosaic algorithm has been selected in the hope that its predictable memory patterns would allow a fully optimized FPGA implementation to be created within a short time-frame that would be able to compete with the performance of an optimized GPU implementation. All builds created in this chapter started from the initial single threaded Demosaic implementation introduced in Section 4.2. Due to the fire destroying the benchmarking setup, not all implementations were finished, and no CUDA Demosaic implementation has been created. With the FPGA being the main focus of the Demosaic algorithm, the first implementations were created for the FPGA, with the GPU implementations being created during the long waits for the FPGA compilation to finish.

6.2.1 Demosaic FPGA

This section lists the optimizations applied during the Demosaic FPGA optimization process. The knowledge gained by porting RabbitCT to an FPGA enabled the Demosaic algorithm to be implemented more efficiently. As with the RabbitCT FPGA implementation process, the functionality of each implementation was tested on the CPU first as it only takes seconds to compile for this platform. When all errors are removed from the implementation, a software FPGA emulation build is compiled providing estimated hardware and pipeline statistics within minutes. For each implementation, the worst case latency and pipelining statistics are used to create multiple iterations of the code. Of each set of iterations, the iteration providing the best-estimated results is compiled for the FPGA hardware and tested on the FPGA.

The sections below describe each implementation and show relevant intermediate results, of which the final results are shown in Table 6.10. These results show a selection of the most interesting FPGA data, with the selected data representing the following:

- Burst-size: number of voxels processed in parallel
- Average FPS: kernel FPS averaged over ten kernel executions
- Max WC Latency: maximum reported worst-case latency estimation, the value does not include latencies associated with data transfers to global memory
- FPGA freq: clock frequency at which the FPGA executes the kernel, affected by compilation timing results
- Impl. pipelines: number of pipelines that are implemented
- Max pipeline depth: length of the longest pipeline in stages
- Max II: maximum instruction interval in cycles, shows how many cycles it takes before a new instruction can be issued in a pipeline
- LUT Usage: FPGA resource used the most of all resources, represents the resource limiting factor
- BRAM Usage: FPGA memory block usage
- SW Compile time: time required to compile for software emulation and produce emulation reports
- HW Compile time: time required to compile the FPGA hardware binaries

Finally, at the end of this section, possible remaining future improvements are given, and a conclusion is made.

Table 6.10: Demosaic FPGA implementation final results

Implementation	Burst-size	Average FPS	Max WC Latency (cycles)	FPGA Freq (MHz)	Impl. pipelines (Nr)	Max II (cycles)	LUT Usage (%)	BRAM Usage (%)	SW Compile time (m:s)	HW Compile time (h:m:s)
0 - Initial	1	0.00747	Undefined	300.0	12/22	11	1.74	0.05	/	01:21:41
1 - Loop Opt.	1	0.16068	408.95e6	225.0	1	26	5.99	0.05	/	02:48:32
2 - Prefetching	1	0.88460	15.729e6	188.2	1	1	9.73	0.1	~17m	05:10:00
3 - Parallelization	16	52.2318	984.60e3	262.6	2	1	9.86	2.33	02:53	04:28:00
4 - Split writing	64	270.702	246.59e3	152.4	2	1	42.86	4.95	16:16	11:35:19
5 - Resource Opt.	64	278.195	246.57e3	171.8	2	1	18.37	4.95	17:45	05:57:49
X - Fully Opt.	64	~645	/	300	2	1	/	/	/	/

Implementation 0 - Initial

The initial Demosaic implementation focused on getting the algorithm to work on the FPGA hardware. Here, a single work-item implementation with predefined image parameters was created. Predefining or hardcoding the image parameters in the code makes the loop bounds known to the compiler, enabling it to optimize the kernel loops at the cost of input flexibility of the algorithm.

For implementation 0, an inlining test was performed for the `matrixMult` function. The results of this test are shown in Table 6.11. They show that without inlining, the compiler sees two pipelineable loops, namely, the main Demosaic loop and the matrix iterator loop in the `matrixMult` function. With inlining, eight implementations of `matrixMult` are inlined into the Demosaic function resulting in the compiler detecting 22 possible pipelines of which only 12 short pipelines could be implemented.

Table 6.11: Demosaic FPGA implementation 0 results

Implementation	Average FPS	Max WC Latency (cycles)	FPGA Freq (MHz)	Impl. pipelines (Nr)	Max II (cycles)	LUT Usage (%)	BRAM Usage (%)	SW Compile time (m:s)	HW Compile time (h:m:s)
no inline	0.00684	Undefined	300.0	1/2	11	0.84	0.05	/	01:15:16
inline	0.00747	Undefined	300.0	12/22	11	1.74	0.05	/	01:21:41

Implementation 1 - Loop optimization

The issue in implementation 0 was that the matrix multiplication bounds were variable to accommodate the matrix multiplication at the image edges. Having the loop bounds variable removed the need for boundary checks inside the loop but made it harder for the compiler to optimize for the FPGA. So, the first change in implementation 1 was to make the loop bounds constant and move the boundary check inside the loop. This change enabled the compiler to unroll the matrix multiplications and allowed it to see the entire code as a single pipeline, see Table 6.12. However, implementing this pipeline leads to the compiler trying to parallelize the two matrix calculations that are required to calculate the unknown colour data. As each parallel implementation tries to load a five by five grid of input pixels every cycle, a too large carried dependency is created, resulting in the compiler choosing to skip the pipelining step.

To resolve the carried dependency issue, the two `matrixMult` iterations are merged into one, which allows the pixel data to be shared for both `matrixMult` calculations. Merging the calculations reduced the pipeline carried dependencies to 25 for the input, which together with the three output writes results in an instruction interval of 26 cycles. A summary of the software estimated results are shown in Table 6.12.

Table 6.12: Demosaic FPGA implementation 1 results

Implementation	Average FPS	Max WC Latency (cycles)	Impl. pipelines (Nr)	Max pipeline depth (stages)	Max II (cycles)
Hardcoded loops	/	20494.4e6	0/1	512	250
`matrixMult` merge	0.16068	408.945e6	1	629	26

Implementation 2 - Prefetching

The goal of this prefetching implementation is to optimize memory usage and reduce the instruction interval to a single cycle. The first applied changes were making all the read-only memory elements constant, and making the global memory elements restricted to indicate no pointer alias is present. However these changes did not affect the compilation or performance of the FPGA implementation, see Table 6.13.

The next change was the addition of a shifting prefetch buffer that ensures that all 25 input pixels are available for matrix multiplication each cycle. Here, two buffers are implemented, a pixelBuffer and a loadBuffer. The pixelBuffer is a two-dimensional array containing five rows and columns. It is used to store the pixel data for the matrix multiplications. Next, the loadBuffer is used to store input data loaded from global memory. It contains five rows and two times the memory burst-size in columns and is used to temporarily store new input data before it is copied to the pixel buffer. Before starting the main algorithm, the loadBuffer and pixelBuffer are initialized with the first set of data. Here, three memory fetches fill the first part of the loadBuffer, of which the first columns are copied over to the pixelBuffer. During the main algorithm, the next set of data is fetched into

the other burst-sized memory element of the loadBuffer if required. Afterwards, the pixelBuffer data is shifted one column to the left, and new data for the last column is copied over from the loadBuffer. With the pixelBuffer ready, the Demosaic processing loop starts with the only change that data is now fetched from the pixelBuffer instead of global memory. After processing the first pixel, the next iteration of the main algorithm starts where the pixelBuffer is updated, and the loadBuffer fetches a new row if required. A simplified version of this cycle is shown in Figure 6.3. Here, a 3x3 pixelBuffer is used with a memory burst-size of 4 instead of the 5x5 pixelBuffer and burst-size 8 original implementation. Note that for the Demosaic calculations of the first row, no image data is loaded as this falls outside the image boundaries. The data in these rows have been indicated with a slash in Figure 6.3.

The emulation results of this implementation in Table 6.13 show a significantly improved worst-case latency and an instruction interval. However, the instruction interval is not at its optimum value of 1 yet as the writing of the output data forms the next bottleneck.

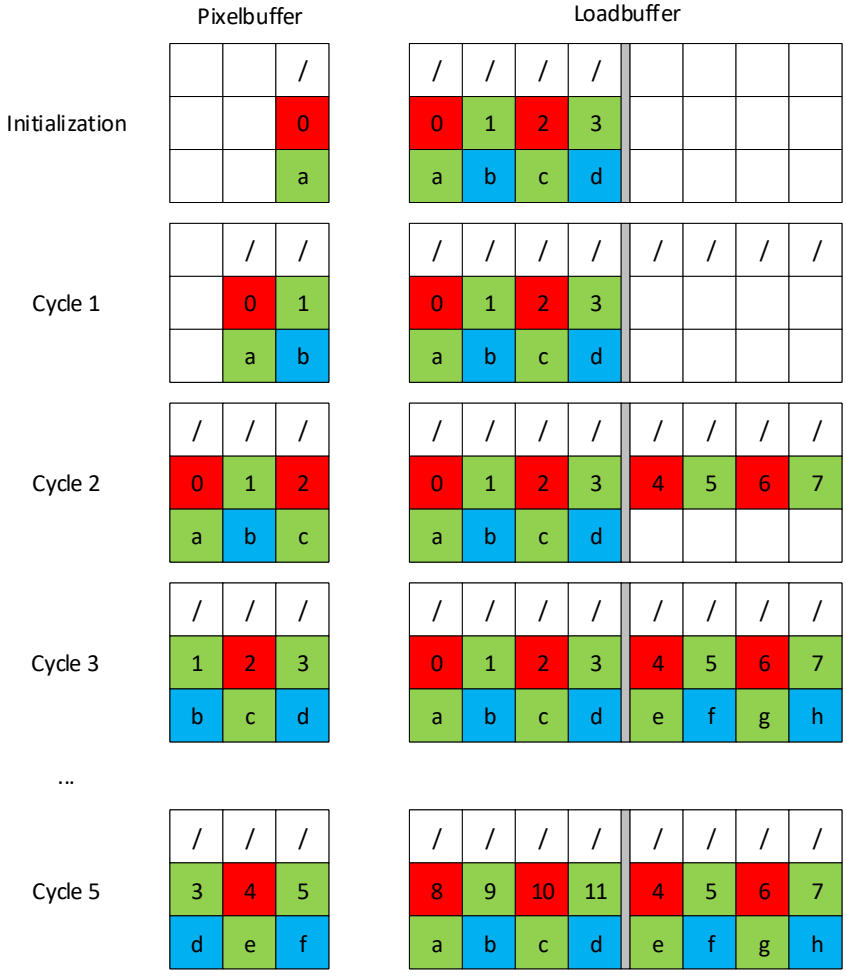


Figure 6.3: Simplified Demosaic prefetching

The final change for this implementation is the vectorization of the output writing to resolve the carried dependency it contains. As a single 8-bit input pixel results in three 8-bit output pixels, the total output data is 24 bits or 3 bytes. Until now, each implementation wrote those three bytes back to separate addresses, which results in three memory writes. Changing the format to a vectorized format requires the uchar3 format to be used, allowing the compiler to write these three bytes in a single command. However, this leads to a distorted output image as each 3-component

vector always maps its data to 4 component memory segment [19], leaving gaps in the data. So, the output data now contains four bytes for every three colour channels. To work around this issue, the output image format was changed to a 4-byte per pixel image with the additional byte representing the alpha or transparency channel. This alpha byte is set to 255 which indicates no transparency is present, resulting in an output image visually similar to the original. As shown in Table 6.13, vectorizing the output writing resolved all memory dependencies in the pipeline and finally reduced the instruction interval to 1, which allowed this build to perform the Demosaic algorithm five times faster than implementation 1.

Table 6.13: Demosaic FPGA implementation 2 results

Implementation	Average FPS	Max WC Latency (cycles)	Impl. pipelines (Nr)	Max pipeline depth (stages)	Max II (cycles)
Constant & restrict	/	408.945e6	1	629	25
Shifting prefetch	/	47.1867e6	1	625	3
Vectorized writing	0.86548	15.7294e6	1	630	1

Implementation 3 - Parallelization

With the memory dependencies resolved, the next step is the parallelization of the pipeline to increase the throughput. However, the parallelization of implementation 2 shows the limitations of its prefetching implementation. Namely, the loadBuffer requires five cycles to load new data, which limits the parallelism to the used burst-size divided by 5. So, with a maximum memory bandwidth of 64 bytes per cycle and a four-byte output variable, out of the 16 parallel computations based on memory bandwidth, only three parallel computations can be performed due to the loadBuffer bottleneck.

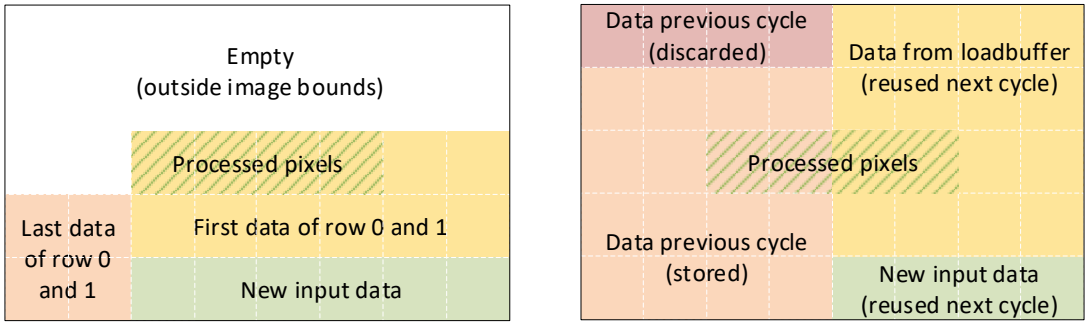
To resolve this issue, a new prefetching approach is developed similar to the Xilinx Edge Detection or Sobel filter examples available on their GitHub [20]. This buffered prefetching implementation retains the same working principle but uses significantly larger buffer sizes to reduce the memory fetches required. For this new implementation, the pixelBuffer contains five rows and 2x the burst-size columns. Two times the burst-size columns are required for processing a burst-size number of elements as the 5x5 matrix calculation requires two additional columns left and right of the current processing window. The loadBuffer now contains four rows of data with the image-width minus the pixelBuffer-width number of columns, enabling it to buffer all the input data and retain this data until it is reused in the next row. The pixelBuffer width is subtracted from the loadBuffer to save some FPGA memory, as data in the pixelBuffer does not need to be stored in the loadBuffer as well.

Similar to implementation 2, during the initialization phase, both buffers are filled with an initial set of data. However, this time two entire rows of the input image are loaded into the loadBuffer instead of five burst-sized fetches. Due to the many memory fetches that are required, the initialization loop is implemented as an additional pipeline that must finish before executing the main pipeline. Next, the 2x burst-size memory savings on the loadBuffer require the pixelBuffer to initialize with data from the last columns of row one and two as well. This data is moved into the loadBuffer when the pixelBuffer is updated. Figure 6.4a shows the initial pixelBuffer data for a burst-size of 4.

To process a burst-size number of pixels each cycle, a total of burst-size + 4 input pixels are required. As four pixels of the previous cycle are reused, the only time more than a burst-size number of pixels have to be loaded is during the initialization. This additional memory fetch requires to load only two pixels as two out of the four extra pixels are outside of the image bounds for the first calculation. For loading new input data, an input-buffer is added which contains the

data of two burst-sized memory transfers. It enables all memory transfers to use addresses located at a burst-sized multiple and allows the pixelBuffer to read the required data from the two bursts window it requires.

The main processing loop is as follows. First, the data in the input buffer is shifted by a burst-size to the right to enable new data to be loaded in the second burst-sized memory segment. Afterwards, the pixel and loadBuffers are updated, and finally the `matrixMult` computation takes place for a burst-size number of pixels. The updating of the pixel and loadBuffer occurs via a loop that iterates over the first four rows of both the pixel and loadBuffer. For each row, all data of the pixelBuffer is shifted by the burst-size to the left, and the freed memory segment receives the next data from the loadBuffer. This copy frees a loadBuffer memory segment that is overwritten with the first burst-sized pixelBuffer data from the next row; ensuring all pixelBuffer data that will be reused is stored. After iterating the previous three steps for the first four rows, the fifth row of the pixelBuffer is updated by shifting the columns and adding new data from the input buffer in the freed memory segment. Figure 6.4b shows an overview of the data division of the pixelBuffer for a burst-size of 4.



(a) Pixelbuffer initial data

(b) Pixelbuffer data division

Figure 6.4: Pixelbuffer example burst-size 4

The described buffered prefetching approach successfully removed the parallelization bottleneck caused by the original prefetching implementation. Next, it increased the maximum parallelism to 16 pixels per cycle, which is now limited by the output data. However, compiling the buffered prefetching code with a burst-size of 16 for FPGA hardware resulted in a compilation failure due to not enough FPGA resources being available. So, additional optimizations were applied to reduce resource usage and allow this build to compile.

To save resources, three main optimizations have been applied. The first optimization is the doubling of all values in the by Malvar-He-Cutler defined matrices [23]. The original matrices contain multiplications by halves, which requires floating-point hardware. By merely doubling the matrices and normalization factor, the matrix values can be stored as chars, hugely simplifying the calculations performed in the algorithm. The second optimization is the combination of multiple `matrixMult` iterations into a single function call. Combining the even and odd `matrixMult` calls reduce the total number of parallel paths that have to be implemented while only slightly increasing the work of a single path as the multiplications now occur on a five by six grid instead of a five by five. Additionally, this optimization simplifies the main algorithm as no even or odd checks have to be performed anymore. The last optimization is adding the row branching to the `matrixMult` function as well. This creates a single `matrixMult` function that can handle all cases, and further simplifies the main algorithm as now only a single `matrixMult` call is required.

Table 6.14 shows the results from the above-described optimizations. It shows that removing floating point math from the algorithm significantly reduces the LUT resource usage from a number

above 100% to 11.6% of the total resources, which enables this build to compile. Additionally, combining the four `matrixMult` variants resulted in small resource savings. These resource savings allowed the compiler to create a more efficient implementation increasing the FPGA clock frequency and the FPS.

Table 6.14: Demosaic FPGA implementation 3 results

Implementation	Burst-size	Average FPS	Max WC Latency (cycles)	FPGA Freq (MHz)	Max pipeline depth (stages)	LUT Usage (%)	BRAM Usage (%)	SW Compile time (m:s)	HW Compile time (h:m:s)
Buffered Prefetching	16	/	984.77e3	/	490	>100	/	~42m	/
Char matrix	16	50.669	984.60e3	208.9	321	11.62	2.33	17:43	05:35:11
Char + comb col	16	50.423	984.60e3	240.9	321	11.54	2.33	05:13	04:09:35
Char + comb row & col	16	52.232	984.60e3	262.6	320	9.86	2.33	02:53	04:28:00

Implementation 4 - Split writing

Output memory bandwidth is the main performance limiting factor of implementation 3. The cause of this bottleneck is the fact that the algorithm generates four output bytes for each input byte. To resolve this, the global output buffer is split into three separate colour channels. Additionally, the alpha channel is removed from the output, reverting the change from implementation 2. So, instead of a single RGBA output buffer being created, there are now three output buffers containing the individual data of the R, G, and B, colour channels. The merging of this data is done on the host just before the data comparison. Here, the read_and_merge function reads the data from the three buffers and combines this into the RGB format required for a TIFF image. Finally, to write data with the full memory bandwidth, each output channel has been assigned to a different memory bank.

Table 6.15 shows the results of both the single bank and multi-bank performance of the split writing implementation. It shows that when using a single memory bank, the instruction interval is equal to three as three write cycles are required to write the three 64 byte buffers to a single memory bank. When using multiple memory banks, the II reduces to 1, enabling a considerable performance increase. Another notable difference can be found in the LUT usage, where the single-bank implementation requires 244k LUTs, and the multi-bank implementation requires 430k LUTs. This LUT usage increase is caused by the multiple memory controllers required for writing to the different memory banks. So, using multiple memory banks does offer significantly more bandwidth, but also comes at a resource usage penalty which can decrease the operating FPGA frequency.

Table 6.15: Demosaic FPGA implementation 4 results

Implementation	Burst-size	Average FPS	Max WC Latency (cycles)	FPGA Freq (MHz)	Max pipeline depth (stages)	Max II (cycles)	LUT Usage (Nr)	SW Compile time (m:s)	HW Compile time (h:m:s)
Single bank	64	72.6	738.12e3	/	323	3	244519	~16.5m	~5h
Multi bank	64	270.7	246.60e3	152.4	321	1	430022	16:16	11:35:19

Implementation 5 - Resource Reduction

The use of multiple memory banks resulted in a large FPGA resource usage increase. This resource usage makes it harder for the compiler to meet the timing requirements when implementing the kernel, resulting in an FPGA clock frequency reduction to meet the timing constraints. Implementation 4 resulted in a kernel operating at 152.4 MHz out of the maximum 300MHz. So, if the clock frequency can be restored to the maximum, the achieved performance can be doubled. For this reason, the main goal for this implementation is to reduce the hardware usage of the kernel, which can increase the FPGA clock speed and enables smaller FPGAs to be used to decreasing the accelerator costs.

The critical path of implementation 4 is the single to floating-point conversion required for the matrix normalization. To optimize this critical path, several methods for normalization of the matrix multiplication sum have been tested. The first tests replaced the division with either an OpenCL half_divide, native_divide, or a standard divide with half-precision parameters. These changes reduce the precision of the division which could lead to a faster computation. Here, half_divide has an accuracy of at least 10 bits, the half-precision division has an accuracy of exactly 10 bits, and the native_divide has an accuracy that is defined by the hardware [19]. The results are shown in Table 6.16 and indicate that the native_divide and half_divide functions barely have any effect. Both implementations, reduce the LUT usage by approximately 1%, while the pipeline gains four stages. The half-precision implementation did have an effect. Here, the LUT usage is reduced by approximately 19%, and the pipeline depth is reduced by 16 stages when compared to implementation 4. However, for all three implementations, the single to floating-point conversion is still the critical path.

The OpenCL round function only works on floating point numbers, so to get rid of the floating point conversion [19], an integer division with rounding is required. Here, an integer division with a shift-based rounding method is used. With the shift rounding, half the divisor is added to the dividend before the division as shown in Equation (6.1). As the integer division always rounds down, adding half the divisor ensures that fractions larger than half the divisor are rounded up, and fractions smaller than the divisor are rounded down. With this change, the LUT usage has decreased by 43% and the floating point conversion critical path has been resolved, resulting in a slight FPS and FPGA clock frequency increase.

$$outputcolor = \frac{colorSum + (matrixSum >> 1)}{matrixSum} \tag{6.1}$$

With the critical path resolved, other small optimizations were tested in order to reduce the resource usage even further. From all tested tweaks, only a change in the clamping function resulted in a notable improvement. Here, the initially used branched clamping is replaced by the min-max clamping reported in the OpenCL spec, which returns min(max(x, minval), maxval) [19]. When using this OpenCL clamp function, the SDAccel compiler reports that this function is unknown. For this reason, clamping has been implemented by a function definition using the min-max clamping method. This implementation enables the compiler to produce a more efficient FPGA implementation, reducing the LUT usage, and increasing the FPGA clock frequency and throughput.

Finally, three smaller tweaks have been tested as well. However, these tweaks did not result in any improvements in the estimated FPGA compilation results. The first change was replacing the multiplications and divisions with a constant multiple of two by shifts; the second change was to replace the modulo function used to determine whether a row is even or odd by a binary check on the first bit; and the third change was the addition of an extra branch to `matrixMult` to skip the multiplications by zero that are present in the constant multiplication matrices. Theoretically, all three changes simplify the required calculations significantly and should lead to a resource

reduction. However, this was not the case, making it likely that the compiler already implements these changes automatically.

Table 6.16: Demosaic FPGA implementation 5 results

Implementation	Burst-size	Average FPS	Max WC Latency (cycles)	FPGA Freq (MHz)	Max pipeline depth (stages)	LUT Usage (Nr)	SW Compile time (m:s)	HW Compile time (h:m:s)
Half & native divide	64	/	246.60e3	/	325	425790	~18m	/
Half precision divide	64	/	246.58e3	/	305	350009	~17m	/
Int-divide	64	272.45	246.57e3	158.2	295	199992	17:16	04:44:50
Int-divide, minmax-clamp	64	278.19	246.57e3	171.8	296	198209	17:45	05:57:49

Future improvements

As the kernel uses the maximum memory bandwidth for both its input and output data, not many performance optimizations remain as the algorithm is limited by memory bandwidth that is available every clock cycle. However, FPGA clock speed optimizations can still improve the FPS of the Demosaic algorithm. Currently, the final implementation has a clock frequency of 171.8 MHz out of the maximum of 300 MHz. If all critical paths affecting this clock frequency reduction are removed, a maximum 74.6% speedup can be achieved.

Currently, the critical path is at an unsigned remainder function. It is used when accessing the separate loadBuffer elements. Here, a mux is implemented in hardware to fetch data from the FPGA memory elements corresponding with the requested address. As all loadBuffer fetches access a burst-size number of sequential memory addresses, this can still be optimized by using the `xcl_array_reshape` command instead of the `xcl_array_partition` command that is currently used. Where array partitioning divides the parallel accessed data over multiple elements, array reshaping restructures the data into a single memory element containing all this data. Reshaping the array enables all loadBuffer accesses to use a single memory element instead of branching out to 64 different memory elements. Accessing a single memory element with a single but 64x larger bus would significantly optimize the FPGA resource usage and likely resolves the critical path as the mux accessing the memory elements is simplified.

Another improvement would be the addition of an additional kernel which uses the remaining memory bank. This kernel can only operate at 33% the throughput of the current kernel due to the memory write bandwidth bottleneck, but it would increase the throughput by an additional 33% at the cost of additional FPGA resources.

Assuming the critical paths can be resolved and that a second kernel is implemented using the remaining memory bank, a maximum performance increase of 2.32x could still be achieved with the current hardware. This speedup will increase the performance of the kernel to approximately 645 FPS at the cost of additional resource usage. However, this 2.32x speedup is the absolute maximum and might not be realistic as these suggested changes could introduce new critical paths, that might make it hard to reach the 300 MHz clock frequency with the selected hardware.

As with RabbitCT, the use of the two available QSFP+ cages would allow an additional 25GB/s memory bandwidth to be added on top of the available bandwidth as the data from these connections moves directly into the FPGA. However, as support for these connections requires BSP changes in RTL, it falls outside the scope of this project.

Finally, upgrading to the latest FPGA hardware generation will also improve the throughput of the algorithm. These FPGAs have a more advanced architecture, enabling higher kernel clock

speeds without making any changes. Additionally, the FPGAs containing HBM2 memory have a more than 10x higher per-bank memory bandwidth which will remove the performance hampering memory bottleneck.

Implementation conclusion

The final Demosaic build achieved a performance of 278.195 FPS, with a theoretical maximum fully optimized performance of 645 FPS. As the final Demosaic implementation has reduced the instruction interval to 1 and since the Demosaic pipeline is parallelized as far as the memory bandwidth allowed; it can be stated that the Demosaic algorithm is well optimized for FPGA execution. Theoretically, the performance and resource usage can still be improved by using the fourth memory bank and by optimizing the prefetching algorithm.

However, if only the PCIe interface is used for data management, any further improvements will not lead to a better performance when a streaming implementation is created. As mentioned in Section 4.2.3, the maximum number of images with a 5k by 3k resolution that can be processed based upon the PCIe 3.0 16x bandwidth is 262 images per second. So, the current implementation will already be bottlenecked by the PCIe bandwidth, making any further performance optimizations useless unless additional processing steps are added or a different I/O method is used for host-accelerator communication.

6.2.2 Demosaic GPU

This section lists the optimizations applied when optimizing the Demosaic algorithm for the GPU. For Demosaic, only an OpenCL build is created due to time limitations and a loss of the benchmarking setup. Each implementation is described below, and their results are shown in Table 6.17. These results show the average Demosaic FPS over a total of 10 input images, and show the memory usage as reported by the Nvidia System Management Interface (Nvidia-SMI) [140]. Finally, possible future improvements are listed, and a conclusion is given that analyzes the achieved GPU Demosaic performance.

Table 6.17: Demosaic OpenCL GPU implementation results

Implementation	Average FPS	Memory usage (MB)
0 - Initial	1.11326	219
1 - Loop Optimization	24.84783	217
2 - Thread Optimization	542.08508	217
3 - Vectorization	545.60291	231
4 - Kernel simplification	923.2375	231
5 - Tweaking	988	/
X - Reduced precision	1109	/

Implementation 0 - Initial

The goal of the initial GPU implementation was to get the algorithm to run on the GPU. Where the FPGA implementation used a single work-item, the GPU implementation creates one work-item per pixel so they can execute in parallel. The results show that without any optimization other than specifying the number of work items, the GPU can run the algorithm at 1.16 FPS.

Implementation 1 - Loop Optimization

Just as implementation 1 of the FPGA, the second GPU implementation focuses on optimizing the matrix multiplication code as the initial implementation with variable loop bounds is not efficient. Originally, variable loop bounds were used to prevent matrix multiplications outside of the image boundaries. By hardcoding the loop boundaries and moving the boundary check inside the loop, the performance of the previous implementation was increased to 22.5 FPS. Indicating that branch hoisting only works when the branches do not affect the loop boundaries.

Next, `matrixMult` was expanded to include the calculations for all three colour channels. So, instead of `matrixMult` computing the data for a single colour channel, it now computes the data for all three colour channels with a single function call, increasing the frame-rate to 24.8 FPS. Additionally, making all variables constant or read-only when used as such, transforming the Malvar-He-Cutler matrices into 2D instead of 1D to improve the readability of the code, and hardcoding matrix parameters by defines all did not affect the frame-rate.

Implementation 2 - Thread Optimization

With the matrix multiplication optimized, the main issue is the data usage of the GPU. As a global work item is assigned for every pixel of the input image and the local work items are set to one, the GPU will schedule its tasks randomly, resulting in inefficient data usage. By increasing the number of local work-items, the GPU executes all global work-items in batches of the sized accordingly to the specified local work-item dimensions. So, by increasing the local work-items in multiple dimensions, multiple sets of neighbouring threads are started in parallel, improving the data locality and the performance of the algorithm.

Experimentation results for multiple local work-items are shown in Table 6.18. The maximum local work-items supported on the Quadro P6000 is 1024, so any local work-item containing more work-items result in a compilation failure. The results show that executing multiple neighbouring work-items offers a massive increase in performance. Additionally, it is shown that parallelizing in the x-dimension offers a larger performance increase than in the y-dimension. The memory likely causes this difference as fetching on the x-axis enables the use of burst memory transfers that fetch multiple elements within a single cycle, while y-axis transfers require data that is not stored sequentially and thus requires an additional memory fetch for each transfer.

Table 6.18: Demosaic local work-item performance in FPS

		Work-items Y					
		8	16	32	64	128	256
Work-items X	8	266.8	267.7	270.2	290.1	287.2	invalid
	16	274.1	276.1	292.5	298.6	invalid	/
	32	495.4	508.4	542.6	invalid	/	/
	64	506.7	542.2	invalid	/	/	/
	128	538.5	invalid	/	/	/	/
	256	invalid	/	/	/	/	/

Implementation 3 - Vectorization

For the third implementation, vectorization has been applied to the output of the algorithm. Instead of separately writing three 8-bit values to the output, now a single 32-bit uchar4 variable is written. This 32-bit variable contains all three colour channels plus an additional alpha channel

to make use of the 32-bit memory format. As explained in the second Demosaic FPGA implementation, having an alpha channel enables the host to write the image data into a TIFF image without the pre-processing that would be required for the empty byte caused by a 3-component vector. Vectorization of the output only increased the frame rate by 3 FPS, indicating that the compiler likely already performed some optimizations on output data handling.

Implementation 4 - Kernel simplification

Implementation 4 simplified the kernel by removing floating point calculations, which were the bottleneck of implementation 3. By doubling all the values in the Malvar-He-Cutler matrices, the original floating point matrix data is transformed into chars, hugely simplifying the calculations required which almost doubles the achieved frame rate.

Implementation 5 - Tweaking

The final GPU build was still a work-in-progress at the time the benchmarking setup was destroyed. However, several tests have been performed, and their results are shown in Table 6.19.

The first test was the use of the OpenCL image format, as this improved the performance of the RabbitCT GPU algorithm. However, for Demosaic it reduced the performance, likely caused by the straightforward memory usage of the Demosaic algorithm. When processing the Demosaic kernel with 32 by 32 local work-items, the cache usage is already optimized for as much data re-use as possible. So, the introduction of the image format or textures might reduce the caching efficiency as it also loads neighbouring data, whereas for RabbitCT the efficiency was improved due to the semi-random memory accesses.
Next, the effect of several small tweaks was tested. Here, the use of the native_divide for a reduced precision floating point division offers a 25 FPS boost over implementation 4. Using the OpenCL clamping function instead of the branch based clamping improves the frame rate by an additional 36 FPS, and as with the FPGA build, using bit masking to determine whether a row is even or odd instead of a modulo function has no effect on the performance.
The integer division with shift rounding that improved the FPGA performance was tested as well. However, this time it reduced the performance, indicating that a GPU offers better performance with a single native-divide operation when compared to the multi-operation integer division and rounding.

Finally, two tests were performed where bit-shifting is used to replace the matrix normalization division. One test applied the shift normalization to all pixels, and the other test used the shifting division for all results except those that are affected by the image boundaries. Both tests increased the performance of the algorithm at a loss of output precision. When taking the edges into account, errors are minimized to single-bit rounding errors caused by the bit-shifting always rounding down. When applying the bit-shift to all pixels normalization errors are present at the last two rows and columns of the output image in addition to the single-bit rounding errors occurring over the entire image. By not taking the edges into account the normalization factor is too high resulting in darker pixels in the output data.

Future Improvements

Even though GPU testing was not finished, most optimizations have been implemented. Currently, only three optimizations remain untested. The first being half-precision floating point divisions, which might provide better performance results than the native_divide function with single precision floating point data currently used. The second untested optimization is the merging of

Table 6.19: Demosaic GPU tweaking results

Implementation	Average FPS
Image format	776.8
Native divide	952
Native divide + OpenCL clamp	988
Native divide + OpenCL clamp + Bitmasking	988
Integer division + shift rounding + OpenCL clamp	849
Shift division except edges	1052
Shift division everywhere	1109

multiple `matrixMult` calculations. Merging multiple column calculations like in Demosaic FPGA implementation 3 halves the total number of threads required to calculate all pixels, which might reduce the performance impact of thread handling as each thread then needs to perform more calculations. The third optimization is creating a streaming Demosaic implementation. Using multiple CUDA streams or OpenCL command queues could improve the average kernel execution time as this enables multiple kernels to process in parallel when hardware units are available.

Finally, as with all other tests, testing the algorithm on the next generation of GPUs will improve the performance as the Nvidia Turing generation offers 20% more CUDA cores and dedicated matrix multiplication tensor cores that might help accelerate the application even further.

Implementation conclusion

The maximum achieved error-free Demosaic kernel performance on an Nvidia Quadro P6000 is 988 FPS when using images with a 5120 by 3072 resolution. However, as determined in the Demosaic algorithm analysis in Section 4.2.3, the maximum number of images that can be processed based upon the PCIe 3.0 16x bandwidth is 262 images per second. As the current GPU implementation adds an alpha channel to the output, an additional 15 MB of data transferred over the PCIe reducing the maximum number of images that can be processed per second to 215. This limitation is not recorded in the output, as only the kernel processing time is used for the frame-rate calculation. So, the PCIe connection will severely bottleneck the P6000 performance as it requires at 5x the PCIe bandwidth based upon its kernel performance, which indicates that the final Demosaic implementation is well optimized for the GPU.

Chapter 7

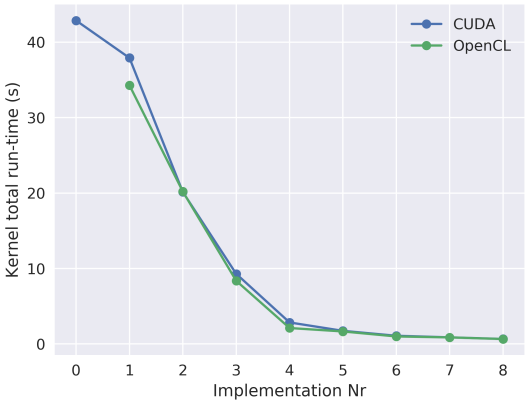
Results

This chapter analyzes the results achieved in Chapter 6. Here the implementation results are compared, and the cost per performance and programmability of both APIs and hardware platforms are determined. Additionally, the selected hardware is analyzed together with the literature study to determine the available interconnect options, energy efficiency and product availability.

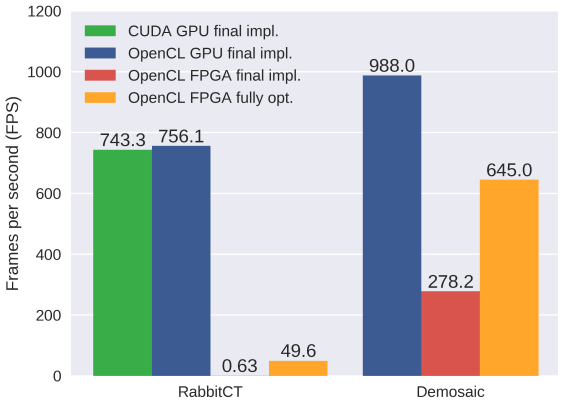
7.1 Maximum performance

In this section, a comparison is made between the achieved and expected fully optimized performance of all created implementations. Figure 7.1a compares the performance of the CUDA and OpenCL API builds for each implementation, and Figure 7.1b compares the final implementation results. The figures show that both the CUDA and OpenCL RabbitCT GPU implementations offer very similar performance for each implementation, indicating that no API has a significant performance advantage over the other on the selected GPU hardware. Memory access latencies limit the final GPU build of the RabbitCT implementation due to its semi-random memory access pattern, and the Demosaic GPU implementation is limited by the number of CUDA cores available.

The FPGA implementations show lower maximum performance than the achieved GPU performances. This difference is caused by a lack of optimization for the final FPGA implementations, and a lack of memory bandwidth for the expected fully optimized FPGA implementations. The memory bandwidth limitation has a reduced effect on the Demosaic algorithm as it requires less I/O data than RabbitCT and has a predictable memory access pattern. Additionally, the Demosaic algorithm does not require any floating point calculations allowing it to perform better than RabbitCT on both the GPU and FPGA hardware platforms. However, when a streaming Demosaic implementation is created, both the GPU and FPGA performances will be limited by the available PCIe memory bandwidth. However, the interconnect freedom of an FPGA allows the two QSFP+ cages to be used as two 100gBe connections, which would allow the FPGA to surpass the GPU performance because of the increased I/O bandwidth.



(a) CUDA vs OpenCL GPU performance



(b) Final implementation performance

7.2 Costs per performance

In this section, the costs per performance for the GPU and FPGA are determined using the results of both the RabbitCT and Demosaic algorithms. While both GPU implementations fully utilize the available hardware, the created FPGA implementations do not. As the hardware price increases with the available resources, smaller/cheaper hardware is selected to improve the costs per performance ratio based upon the hardware utilization. Here, the hardware is selected to optimize for both the performance and the costs. The following sections show the costs-per-performance for both the created RabbitCT and Demosaic implementations.

7.2.1 RabbitCT

The RabbitCT performance is expressed as the total run-time of the benchmark. As costs per total run-time of the benchmark is a vague metric, the performance results have been converted to FPS by dividing the total images in the benchmark (496) by the achieved total run-time. Table 7.1 shows the results for the current hardware and the prices reported in Chapter 5. The results show that due to the lack of an optimal FPGA pipeline, the performance and costs per FPS results are significantly worse than the GPU results. However, with the algorithm not being fully optimized and the algorithm not using all the FPGA hardware, the cost per performance ratio can be significantly improved.

To show the best achievable costs-per-performance ratio, both the performance of a fully optimized algorithm and costs of the optimal hardware should be used. The fully optimized performance is already reported in Section 6.1.1, which leaves only the optimal hardware to be selected. The optimal hardware should be selected from the same FPGA architecture to prevent any architecture changes from affecting the results. Additionally, the final hardware usage of the FPGA must be taken into account to determine the optimal size of this FPGA. As the final RabbitCT hardware usage result is lost in the fire, the hardware usage of this implementation has to be estimated. Between implementation five and six, the BRAM usage increases 5x. So, assuming the LUT usage increases with a similar number, about 33% of the total available LUTs are used. As Xilinx reports FPGA size in logic cells which is directly related to the available LUTs, using 33% of the available 1451k logic cells of the KCU1500 results in an FPGA requiring at least 480k logic cells to run the algorithm. The smallest Kintex Ultrascale replacement FPGA is the XCKU040, as it contains 530k logic cells. However, according to the Xilinx external memory capacity utility [142], this FPGA only supports two DDR4 64-bit memory banks. While this does not affect the final implementation as it uses only two banks, it will result in only half the performance

of the fully optimized implementation. As FPGAs supporting four 64-bit DDR4 banks are only available in a much higher price range, both a cost-optimized and a performance-optimized FPGA is selected. Here, the performance optimized FPGA uses the fully optimized four memory bank implementation results from Section 6.1.1, and the costs-optimized implementation uses the fully optimized implementation with only two memory banks and thus half the performance.

According to digikey.nl [77], the cheapest XCKU040 die supporting two DDR4 memory banks currently costs about € 1430, and the cheapest FPGA supporting four DDR4 memory banks, the XCKU095, currently costs about 4500 €. However, these prices are for the FPGA die only, and thus still require a PCB with DDR4 elements to be added. Let us assume a PCB with two DDR4 banks cost € 250, and a PCB with four DDR4 banks adds € 500 to the total price, making the complete FPGA accelerators cost around 1750 and 5000 euros when rounding to the nearest 250 euro price point.

The final results in Section 6.1.1 show that the fully optimized implementation can significantly reduce the costs per performance on the current FPGA, making it cost 21x more per frame than the GPU. Next, applying this algorithm to either a 2-bank or 4-bank DDR4 FPGA, the costs-per-performance can be further reduced to 16.3x the GPU when maximizing performance, or 11.4x the GPU costs-per-performance when optimizing for costs.

Table 7.1: RabbitCT costs per performance results

Implementation	Hardware	Total run-time (s)	FPS	Hw costs (€)	Costs per FPS (€ / FPS)
CUDA GPU Final	Current	0.667255	743.34	4635.12	6.24
OpenCL GPU Final	Current	0.655967	756.14	4635.12	6.13
OpenCL FPGA Final	Current	782.08	0.63	~6500	10317
OpenCL FPGA Perf Opt.	Current	~10	49.6	~6500	131
OpenCL FPGA Perf Opt.	Optimal 4-bank	~10	49.6	~5000	101
OpenCL FPGA Cost Opt.	Optimal 2-bank	~5	24.8	~1750	70.6

7.2.2 Demosaic

The final performance results for the Demosaic algorithm are shown in Table 7.2. It shows that for processing 5k by 3k input images with the selected hardware, the costs for a given frame-rate is 4.69 euro per FPS for the GPU implementation and 23.36 euro per FPS for FPGA implementation. However, the FPGA implementation only uses 18.37% of the available hardware, and thus only requires 267k of the 1451k logic elements that are available in the KCU1500 [143]. So, smallest available Xilinx Kintex FPGA, the XCKU025, should still have enough hardware to run the algorithm with its 318k logic cells. However, as with RabbitCT, the XCKU025 supports only two out of the four initially available DDR4 memory banks. As the final implementation uses three memory banks and the fully optimized implementation four, both implementation will see a performance drop due to the lower memory bandwidth, so a larger FPGA has to be selected.

Four bank 64-bit DDR4 support is only available with high-end FPGAs of the Kintex Ultrascale architecture. So, to show the best costs-per-performance results, both a 3-bank cost-optimized and a 4-bank performance-optimized FPGA are selected. Here, the XCKU035 has been selected for the final implementation as it is the smallest FPGA supporting up to three DDR4 64-bit memory interfaces, and the XCKU095 has been selected as it is the cheapest FPGA supporting up to four DDR4 64-bit interfaces. According to digikey.nl [77], the cheapest XCKU035 supporting three memory banks currently costs about € 1250, and the cheapest XCKU095 that supports four memory banks costs about 4500 €. Assuming that a PCB with three DDR4 banks costs € 375,

and a PCB with four DDR4 banks adds € 500 to the total price, these FPGA accelerators cost € 1625 and € 5000 respectively when rounding to the nearest 250 euro price point. Running a clock optimized implementation of the Demosaic algorithm, these smaller and more cost-optimal FPGAs drop the costs-per-performance from € 10 per FPS to 7.75 for the four memory bank FPGA, and 3.34 for the FPGA with three memory banks. However, with both the optimal FPGA cases, not all hardware resources are used as the memory bandwidth is the main limitation. If the computational intensity of the algorithm is increased by adding another processing step for the same data, the costs-per-performance ratio will improve as the FPGA still has computational resources to spare.

The above results have one issue in common; they use the kernel execution time to calculate the result and ignore the PCIe bandwidth limit. When demosaicing input images with a 5k by 3k resolution, the PCIe interface becomes the bottleneck as this limits the frame-rate to 215 FPS for the 4-byte output GPU implementation, and 262 FPS for the 3-byte output FPGA implementation. So, no matter how fast and optimized the kernel performance is, the PCIe bandwidth will limit the performance of both platforms. While FPGAs allow the PCIe bandwidth issue to be resolved by using their high-speed serial transceivers with other interconnect, it cannot be resolved for the GPU. For this reason, a smaller/cheaper GPU will also be selected to lower its performance to the PCIe bottleneck.

Lowering the FPS of the GPU can be done by either adding additional processing steps to each pixel to perform more work on a given set of input data or by selecting hardware with less processing cores to reduce the available parallelism and thus increasing the time required to process the data. If only the Demosaic algorithm has to be performed, a GPU can be selected which contains only 22% of the current 3840 CUDA cores. Assuming a linear performance decrease, this will limit the maximum GPU performance to 217 FPS, minimizing the effect of the PCIe bandwidth limitation. From the available Pascal GPUs, the Quadro P2000 seems the best option to minimize the effect of the PCIe bottleneck. According to TechPowerup [74], it contains 26% of the Quadro P6000 CUDA cores and has 32% of the P6000 memory bandwidth. On top of that, the Quadro P2000 costs € 462.70 according to the Tweakers pricewatch [134], reducing the costs per performance to € 2.15 per FPS.

Table 7.2: Demosaic costs per performance results

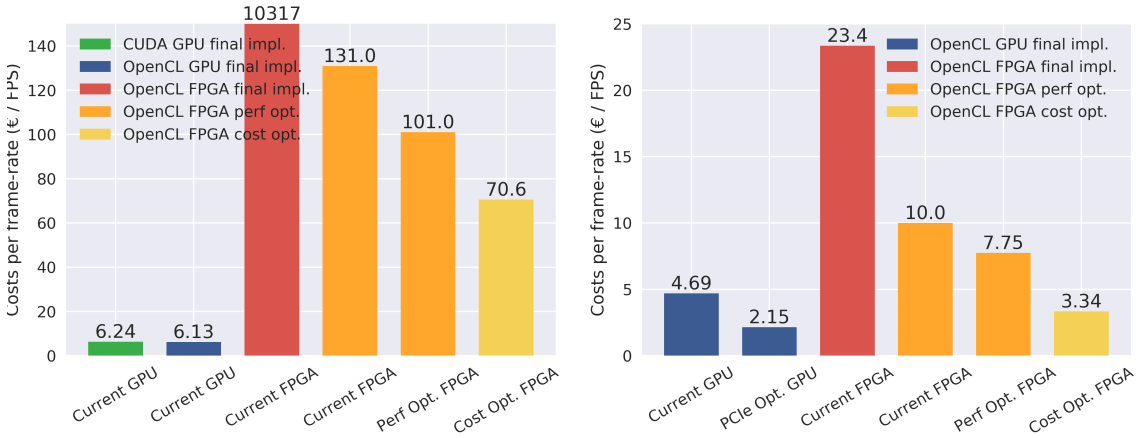
Implementation	Hardware	FPS	Hw costs (€)	Costs per FPS (€ / FPS)
OpenCL GPU	Current	988	4635.12	4.69
OpenCL FPGA	Current	278.195	~6500	23.36
OpenCL FPGA Perf Opt.	Current	~645	~6500	10
OpenCL FPGA Perf Opt.	Optimal 4-bank	~645	~5000	7.75
OpenCL FPGA Cost Opt.	Optimal 3-bank	~486	~1625	3.34
OpenCL GPU PCIe-limit	Optimal PCIe	215	462.70	2.15

7.2.3 Summary

The RabbitCT results show that both the CUDA and OpenCL GPU implementations result in a very similar performance. With both algorithms using the same GPU, this results in both algorithms also sharing a similar price per FPS ratio, see Figure 7.2a. Additionally, the unoptimized final state of the RabbitCT FPGA implementation causes it to have a bad costs-per-performance ratio. However, when using the expected performance of the fully optimized implementation with optimal FPGA hardware, the costs-per-performance ratio decreases significantly, but the FPGA implementation remains more than 11x as expensive per performance than the GPU.

For Demosaic, the FPGA implementation was well optimized. When using the optimal FPGA hardware, the costs-per-performance ratio of the FPGA and the GPU becomes closer, see Figure 7.2b. Additionally, when taking the expected fully optimized FPGA performance on the optimal hardware into account, the FPGA costs-per-performance is reduced up to 3x when using only three memory banks. With these final results, the FPGA costs-per-performance ratio can become 0.71x the GPU ratio, making the FPGA costs per performance cheaper than the current GPU implementation. However, when taking the PCIe bandwidth bottleneck into account and selecting a smaller and cheaper GPU for this bottleneck, the performance stays the same while the costs-per-performance ratio is reduced. Using a PCIe bandwidth optimized GPU, the costs-per-performance drops to 2.15, which makes the FPGA implementation cost 1.55x the GPU costs-per-performance.

The achieved 11x higher price per frame-rate for the optimal RabbitCT FPGA implementation compared to the GPU implementation neatly falls in the 3x to 50x price per performance ratio determined in the literature study. However, the 1.55 ratio when using the optimal GPU and FPGA implementations for Demosaic show significantly better results for the FPGA. As the RabbitCT algorithm mainly uses floating point mathematics, it falls in line with the price-per-GFLOPs ratio the literature study analyzed, whereas the integer mathematics of the Demosaic algorithm do not. These different results for different calculation precisions show that costs-per-performance ratio of the literature study was inconclusive as it only covered the costs-per-performance ratio of single-precision floating point algorithms.



(a) RabbitCT

(b) Demosaic

Figure 7.2: Costs-per-performance results

The costs-per-performance can be further reduced if the FPGA high-speed serial transceivers are used to provide additional memory bandwidth. This change would either improve the overall performance or allow cheaper FPGAs to be selected when this additional bandwidth is used to replace certain memory banks. However, the use of these transceivers requires BSP changes and a host-FPGA interface that uses this connection, making the kernel optimization a more difficult task. Another option is the use of an FPGA with HBM2 memory, which would significantly increase the performance as it alleviates the memory bottleneck. However, the use of HBM2 memory also increases the costs of FPGA. As the price of such FPGA accelerators is unknown at the moment, no such costs-per-performance ratio can be calculated.

7.3 Programmability

The programmability metric analyzes the experienced ease of using a particular API or hardware platform for accelerating the selected algorithms. To make the programmability metric as objective as possible, the subjective experiences are explained using two main categories, the tooling and code, each containing several subcategories. For each of the subcategories, a ranking is made between the tools. Finally, the summary section combines the given ranking allowing a conclusion to be made.

7.3.1 Tooling

The use of the right tools can make or break the programming experience. The correct tooling will accelerate the user to its goal, while poorly designed tooling only hampers the progress. The tooling of CUDA, OpenCL GPU and OpenCL FPGA are analyzed on the installation, ease of usage, and code profiling abilities in the subsections below.

Installation

For CUDA, all code is compiled using Nvidia CUDA Toolkit version 9.2. With a single installation, the toolkit provides everything needed to compile and run CUDA code on the Nvidia GPU. It provides the NVCC compiler, the NVVP code profiler, the Nvidia display driver, a large set of tools and libraries, and CUDA code examples. For more information, see the CUDA Installation Guide [21].

For OpenCL on Nvidia GPUs, an all-in-one toolkit is not available. The host code of our created OpenCL GPU implementations is compiled using GCC 5.4, which comes pre-installed on the Ubuntu OS. The host code loads the OpenCL kernel source files or binaries during execution and compiles them using the Nvidia-OpenCL-Runtime, which is installed together with the Nvidia display driver. So, for both OpenCL and CUDA compilation, only the Nvidia CUDA Toolkit had to be installed. However, whereas this installation offers all required tools for CUDA, it offers only bare-bones support for OpenCL.

Finally, for SDAccel, Xilinx offers an installer for the tooling which installs all Xilinx components. It installs the SDAccel compilers XCPP and XOCC for compiling the host and kernel code respectively. It installs the default supported board support packages (BSP) which contain the FPGA design and drivers to support OpenCL acceleration on the supported accelerators, and it installs the Vivado toolset to compile the XOCC generated RTL code into an FPGA binary. In addition to the installation, an extensive list of packages have to be installed in order to offer full support on the Ubuntu OS, and a license has to be downloaded from the Xilinx website for the SDAccel tooling to work. For more information, see the SDAccel Release Notes, Installation, and Licensing Guide [16].

As no installation difficulties were observed with each programming API, no ranking of each API is made.

Ease of usage

The use of both the CUDA and OpenCL GPU compilers is very similar. Here, the CMake find_package(API) command is used to search for compilers that support the selected API. With the supported compilers found, the find_package(API) command sets the required compilation settings and makes several commands or flags available to ease the compilation. For CUDA, the

cuda_add_library command becomes available, allowing CMake to compile the linked files using the NVCC compiler. For OpenCL, an OpenCL target flag becomes available, linking the attached code to the OpenCL library shipped with the Nvidia-OpenCL-Runtime.

Creating a CMake environment to compile OpenCL using SDAccel is more difficult as no default CMake find_package() command is available. Instead of creating a find_package CMake script for SDAccel, the make file compilation method provided with the Xilinx SDAccel GitHub examples has been used for compiling the FPGA binaries [20]. The use of this Makefile enabled compilation with minor changes. A successful compilation using the provided Makefile results in both a host executable and a kernel FPGA binary file. These two files are copied by the original CMake scripts to the required location, allowing the code to be tested using the same test scripts as the GPU builds.

Another advantage the Xilinx GitHub compilation environment provides is a set of helper functions for setting up the OpenCL environment. These helper functions combine a large number of OpenCL initialization commands into a single function call. An example is the xcl_input_source() command that allocates memory, loads the source file, creates an OpenCL program using the device settings and source file, compiles the program, and checks for success for each step. Because these helper-functions hugely simplify the host code, they have also been added to the OpenCL GPU builds to simplify the host code.

After the compilation setup, both the CUDA and OpenCL GPU compilers worked without any issues. However, this was not the case with SDAccel for which two mayor issues were present. The first issue was that Vivado failed linking when compiling for hardware emulation, and the second issue was that any use of the printf function crashed the FPGA kernel, resulting in an immediate halt of the program. The linking issue was resolved by installing a missing ubuntu package that was not on the requirements list [144]. The second issue was resolved by replacing the objcopy executable SDAccel uses with the Ubuntu variant. This fix prevented printf from crashing the kernel but resulted in it being unable to print variables [145]. The printf variable printing issue was never resolved, preventing in-depth debugging from being used on the FPGA hardware. Luckily, the portability of OpenCL code allows the FPGA code to run on the CPU or GPU as well, resulting in these platforms being used for code debugging.

The tools of each platform were easy to use with either a CMake script or the provided Makefiles. For CUDA and OpenCL everything worked without any issues. However, this was not the case with SDAccel as some bugs were present. The missing package bug can be forgiven, but the non-operational printf function cannot as it significantly complicates the debugging of the kernels. For this reason, SDAccel is placed last, with CUDA and OpenCL sharing the first position.

Profiling

Profiling the created CUDA kernels is made easy with the Nvidia Visual Profiler (NVVP) [146]. It provides a CPU and GPU timeline giving a clear overview of what happens in the kernel. Additionally, it provides performance analysis statistics and gives hints to optimize the kernel code further, see Figure 7.3. An example is shown in implementation 3 of the RabbitCT GPU build in Section 6.1.1. Here, NVVP reported that the available double precision units significantly bottlenecked the code. Replacing the double precision calculations with single-precision doubled the performance of the algorithm after which the bottleneck was shifted from a compute problem to a memory problem.

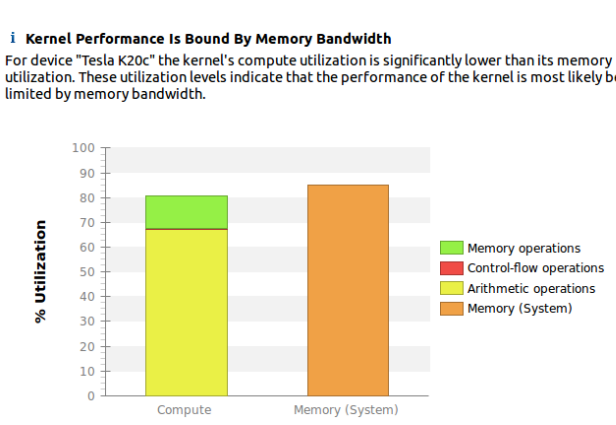
The Nvidia Nsight toolset supports OpenCL profiling on Nvidia GPUs. Nvidia Nsight extends the functionality of both the Eclipse and Visual Studio editors with debugging and profiling options for their GPUs. However, OpenCL profiling is only supported in the Visual Studio editor, limiting this functionality to the Windows operating system [147]. Other OpenCL profilers are available,

but limit their performance analysis functionality to a specific hardware vendor. For example, Intel provides OpenCL SDK containing debug tools for their platforms [148] and AMD provides the CodeXL profiler for AMD GPUs and CPUs [149]. With the CUDA profiler giving in-depth information that also applies to the OpenCL implementation, no separate OpenCL profiler has been used. So, the RabbitCT build was optimized based on CUDA profiling results, and the Demosaic build was optimized based upon optimization experience as no CUDA build has been created.

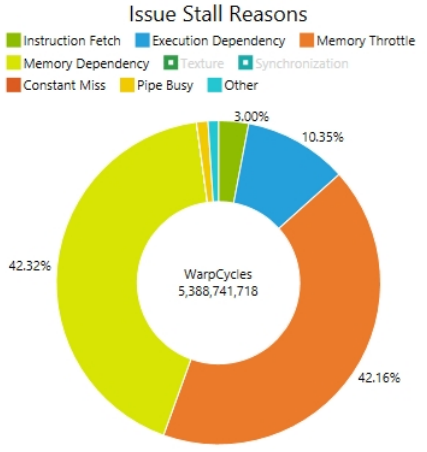
SDAccel also provides several profiling tools and reporting options. It offers design guidance, estimation reports, an in-depth profile summary, an application timeline, and a waveform viewer. Of these options, the estimation report is used the most for creating the RabbitCT and Demosaic kernels as it offers insight in pipelining issues recognized by the tool and shows where the critical path lies. Additionally, design guidance is used to further optimize the code for FPGA. However, this guidance is basic and only checks whether a pipeline is present, whether the full memory bandwidth is used, and whether all memory banks are used. Next, both the in-depth profile summary and waveform viewer were not used as they offer too much detail which makes it hard to find the exact issue. The application timeline tool was also not used as it is aimed for kernel scheduling, which is not required when executing only a single kernel.

One main difference between the Nvidia Visual Profiler and the SDAccel profiling tools is that NVVP is significantly easier to use. NVVP is one toolset that contains all the required data within its interface which requires only one or several runs of the algorithm to fill with data. Next,m the profiling tools of SDAccel are all provided separately and thus are not incorporated into a single program. Each of them activates by adding specific commands to the compilation process. After compilation, several reports are generated which are split over multiple directories, making the data hard to find. Additionally, profiling with either the application timeline or waveform viewer results in a file that first needs conversion into a readable format, further reducing the accessibility of the data.

For OpenCL on Nvidia GPUs, no profiler is available that supports performance analysis on Linux, resulting in it being placed last. Next, both SDAccel and CUDA provide profiling and performance analysis tools. However, where CUDA provides an easy to use all-in-one toolset, SDAccel provides many tools which require different settings to be enabled and generate data scattered over multiple directories. For this reason, SDAccel takes the second place, and CUDA is placed first.



(a) Utilization report, taken from [150]



(b) Stall reason report, taken from [151]

Figure 7.3: NVVP Performance Analysis Reports

7.3.2 Code

Even when using the best tools for acceleration, the code must still be written. In this section, the perceived programmability of porting both RabbitCT and Demosaic to the selected acceleration platforms is analyzed. Here, a comparison is made between the ease of programming, lines of code, compilation time and code portability.

Ease of programming

The main difference between both the CUDA and OpenCL API is present in the host code. With CUDA both the host and kernel code is compiled by NVCC, allowing the compiler to analyze the code for both platforms and use this knowledge to optimize the code. Compilation results in a single executable file containing the compiled binaries for both the host code and the kernel code. With OpenCL, the host code is compiled by a standard C or C++ compiler that links to the OpenCL libraries. Its kernel is either compiled before execution and loaded as a binary file, or during execution and loaded as source files. This compilation freedom results in OpenCL host code requiring additional functions for loading and preparing the separate kernel files for execution. As the host code does not know anything about the OpenCL kernel before execution, each kernel parameter has to be defined, whereas the CUDA compiler can determine these settings automatically. Because of these required additional initialization steps, programming the OpenCL host code is perceived as more difficult.

The increased difficulty in writing the host code also hampered the streaming implementations of RabbitCT. Where CUDA automatically detects a pinned memory buffer allowing for fast host-device transfers, OpenCL requires manual mapping and unmapping for each transfer. As no successful mapping and unmapping for each transfer implementation could be created in the available time, the final build maps the entire input buffer in a single mapping command. The result is a final RabbitCT OpenCL GPU streaming implementation where the memory footprint of the input data is doubled.

The increased difficulty of writing in OpenCL for a GPU is limited to the host code. The differences between the kernels of both APIs remains limited to the syntax of each API, resulting in functionally the same code being written. The same cannot be said when writing kernels for FPGA acceleration. While the OpenCL GPU host code can be reused, the writing of an OpenCL FPGA kernel is perceived as significantly more difficult than its GPU counterpart. The programming API does not cause this increase in difficulty; the hardware causes it. GPUs are created for parallelism. They have a large number of processing cores, a large number of memory banks, and use automated caching and memory access algorithms that hide most memory management related difficulties. FPGAs, on the other hand, are created for pipelining data where the parallelism is limited. A pipeline contains an input and output stage for every input or output variable. Here, data flows from the input or calculation stage along the pipeline towards the output or calculation stage where this data is last needed. In a fully optimized pipeline, each stage is active every cycle, resulting in the memory bus of each variable being busy each cycle. As all required data is available every cycle and moves along with the pipeline, no data is never offloaded to make room for other data, and thus no caching algorithms or additional memory fetches should be present. So, when both previously loaded data and new data is required from the same input buffer, it is up to the programmer to implement a buffering strategy that prevents the need for multiple global memory accesses. Accessing the same global memory variable multiple times in a cycle is an issue because the bus is already in use at the input stage. So, if multiple fetches are required, a pipeline stall must be scheduled to fetch the data.

The main reason FPGA kernel creation is perceived as more difficult is because a buffering strategy must be created. For Demosaic, creating such a buffer strategy was doable as the memory access

patterns are predictable and because Xilinx SDAccel GitHub contains good buffering examples for image processing algorithms. The semi-random memory access pattern used by the RabbitCT algorithm make it significantly harder for a buffering strategy to be created and optimized. The increased difficulty resulted in there not being enough time available to fully optimize the buffering for RabbitCT, causing the bad results for RabbitCT on an FPGA.

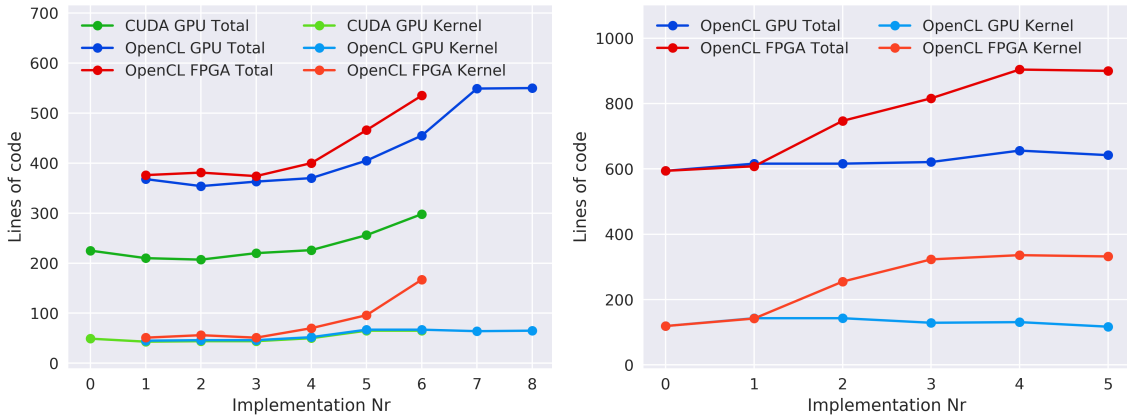
As programming in CUDA requires the least lines of code in both the host and kernel code, it takes first place on ease of programming. OpenCL is placed second because of the additional difficulty in writing the host code. SDAccel is placed last as it adds additional kernel programming difficulties on top of the already more difficult host code of OpenCL.

Lines of code

A great way to visualize and objectify the above-described differences in difficulty is by showing the lines of code that are required to create each implementation. Figure 7.4 shows the total implemented lines of code and the lines of code required for the kernel for both the RabbitCT and Demosaic implementations. Here, the total lines of code graph represents the code in the header, host, and kernel code combined, and the kernel lines of code graph shows only the code written for execution on the accelerator. Both line-counts only show the code required for the execution of the algorithm, and thus do not include debug and preprocessor code. Additionally, the Xilinx SDAccel GitHub provided helper functions are used to initialize the OpenCL environment for all OpenCL implementations. The use of these helper functions makes the host-code several hundred lines shorter than required for a complete implementation.

Comparing the RabbitCT CUDA and OpenCL GPU lines of code show that OpenCL has a significantly higher total line-count than CUDA for each implementation. However, the kernel line-counts show almost no difference between both APIs. This result proves that most differences between OpenCL and CUDA are present in the host code. Additionally, comparing OpenCL GPU with OpenCL FPGA shows that while the first implementations share the same kernel code, the FPGA kernel quickly requires more lines of code to implement the memory buffering optimizations. Additionally, the FPGA total line-count increase is similar to the kernel line-count increase, indicating that the header and host code do not require many changes for an optimized execution on an FPGA accelerator.

For the lines-of-code required, CUDA is placed first, then OpenCL GPU and finally OpenCL for FPGA because of the additional lines required for the host code with OpenCL and the kernel code for OpenCL FPGA.



(a) RabbitCT (b) Demosaic

Figure 7.4: Lines of code

Compilation time

The next major factor affecting the programmability is the compilation time. The shorter the compilation time, the faster iterations can be made upon the code and the faster the code will be optimized. The compilation time for both the CUDA and OpenCL GPU implementations was not recorded as it was so fast that it never was an issue. All compilations for the GPU finished in seconds, after which the code could be tested immediately.

FPGA compilation is something else entirely. Here, a successful compilation for the hardware can take anything from one hour and twenty minutes to 34 hours depending on the created implementation. On average, the time required for compiling the main RabbitCT implementations took 2.5 hours and the main Demosaic implementations took 5.25 hours to compile, see Figure 7.5. Generally, the higher the FPGA hardware usage, the longer the compilation time. The compiler causes this increase in compilation time mainly during the routing of the compiled hardware paths. When one or more of these paths do not meet the timing constraints, the compiler tries to swap around implemented modules in the hope to meet the timing requirements. The more FPGA resources required, the more modules that are available for re-routing and the longer the paths between modules become.

To work around the long compilation time, SDAccel produces estimation reports of the final kernel early in the compilation process. During these early compilation steps, the compiler uses the OpenCL software to create and schedule a pipeline that can be implemented in the hardware. By writing the pipelining results in the estimation report, early pipelining issues can be found, and the expected worst-case latency and initiation interval can be analyzed. However, as shown in the Demosaic FPGA results, this software compilation step can still take between two and twenty minutes, with outliers of an hour also being observed. These long compilation steps make the FPGA kernel creation and optimization a time-consuming process.

As both OpenCL and CUDA GPU kernels compile within seconds, they share the first place. SDAccel is placed last due to the software compilation taking up to twenty minutes, and final hardware compilation taking multiple hours.

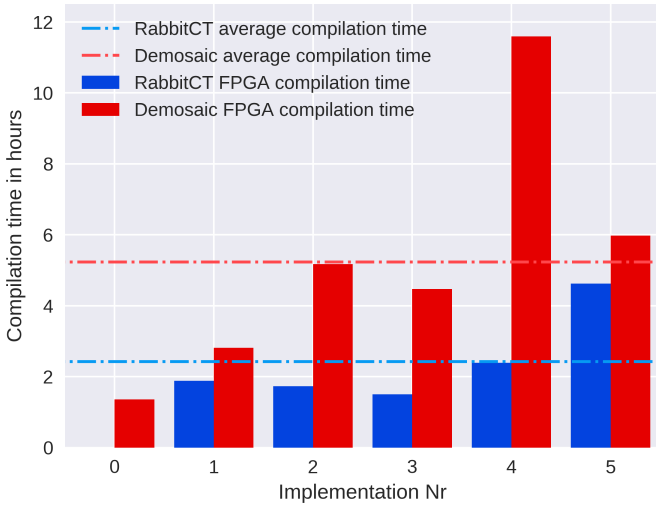


Figure 7.5: FPGA hardware compilation time

Portability

The last code programmability metric is the portability of the code. Higher portability means that the code can be executed on hardware other than the target platform. Because CUDA is a closed source standard, no other compilers are available for CUDA than the one made by Nvidia, limiting its portability to Nvidia GPUs. Next, OpenCL is an open-source standard, which allows its code to be executed on all devices for which an OpenCL compiler has been made available, giving it a high portability.

Having a high portability does not mean that the code needs to be optimized for the other platform; it just needs to be able to execute and produce the same calculation results. An example of this can be seen in the first RabbitCT FPGA results where the first OpenCL GPU implementation is tested on both the FPGA and GPU. This FPGA implementation took 2.5 hours to produce the same results as the GPU can produce in 34 seconds.

The portability of a given API can be reduced when compiler specific functionality is used. For example, SDAccel introduces a set of pragmas to optimize the code for the FPGA, limiting code that uses these pragmas to the SDAccel compiler which reduces the portability. This limitation was circumvented by using preprocessor commands to detect whether the Xilinx SDAccel compiler is used and only enabling FPGA specific optimizations when compiling with SDAccel. This change enabled the FPGA code to be compiled for the GPU and CPU as well, allowing the functionality of the kernel to be tested without needing to wait on the lengthy FPGA compilation process. Additionally, testing on the CPU or GPU bypassed the broken printf functionality of the SDAccel compiler, allowing for significantly faster debugging of the kernel and showing the advantages of portable code.

With OpenCL GPU code being compatible with all OpenCL compilers, it is ranked first. OpenCL FPGA code is also compatible with all OpenCL compilers unless FPGA specific pragmas are used, ranking it just behind OpenCL GPU code. Finally, as CUDA code is limited to Nvidia GPUs only, the CUDA portability score is the lowest of the analyzed acceleration methods.

7.3.3 Summary

Table 7.3 summarizes the results of each acceleration method on the above-stated categories. For each acceleration method, a score between one and three stars is given. Three stars indicate that this acceleration method scores the best, and one star indicates that this acceleration method scores the lowest in comparison to the other acceleration methods. Analyzing the results shows that CUDA offers the best programmability results in all categories but portability. Next, OpenCL for GPU offers high portability at the cost of a more difficult host code and no profiling support for Nvidia GPUs on Linux. Finally, OpenCL for FPGA scores the lowest of the three tested acceleration methods due to its additional programming difficulty, large compilation times, and the printf issues with the SDAccel compiler.

Table 7.3: Programmability results

Metrics		CUDA GPU	OpenCL GPU	OpenCL FPGA
Tooling	Installation	☆☆☆	☆☆☆	☆☆☆
	Ease of usage	☆☆☆	☆☆☆	☆
	Profiling	☆☆☆	☆	☆☆
Code	Ease of programming	☆☆☆	☆☆	☆
	Lines of code	☆☆☆	☆☆	☆
	Compilation time	☆☆☆	☆☆☆	☆
	Portability	☆	☆☆☆	☆☆

7.4 Interconnect options

While GPUs only contain a PCIe interface, video output ports, and an optional SLI or NVLink bridge for GPU-GPU communication, FPGAs have generic high-speed serial connections that allow any serial interconnect to be added. These generic high-speed serial transceivers can be used for additional I/O bandwidth and additional memory bandwidth when used in combination with internal FPGA memory. While using these generic interconnect with OpenCL kernels was not within the scope of this project, it still adds value to OpenCL FPGA acceleration as it allows the OpenCL programmed FPGA to be used for a broader range of applications than the GPU.

Comparing the interconnect options for each programming API shows that CUDA is PCIe limited as it only supports acceleration with Nvidia GPUs. Next, the interconnect options of the OpenCL API are hardware dependent as it can execute on any hardware for which a compiler has is available. So, if an OpenCL kernel is created for a GPU, it is limited to PCIe I/O, and if an OpenCL kernel is created for a CPU or FPGA, it is free to use with any available interconnect the infrastructure provides.

7.5 Energy efficiency

Initially, a hardware power measurement was planned for the final implementations of each algorithm. These power measurements would have shown the average power usage difference between both the GPU and FPGA. However, due to the loss of the benchmarking setup, these measurements could not be performed. For this reason, the final energy efficiency results are an estimation based on the known information of the selected hardware.

As the power usage of the hardware varies with the load, the hardware manufacturers only provide the Thermal Design Power (TDP) of a product. The TDP of a product represents the maximum power output for which the cooler is designed, which is often close to the maximum power output of the hardware. As reported in Section 5.4.1, the selected Quadro P6000 GPU has a TDP of 250W, and the Kintex KCU1500 FPGA has a TDP of 75W. From these numbers, it can be concluded that the FPGA is 3.3x more energy efficient than the GPU at full power. However, this number does not represent the actual power usage.

To determine the actual power usage, hardware usage and clock speeds must be analyzed. Here, a significant difference is present between the FPGA and the GPU. The GPU runs at its maximum clock and uses 100% of its processing units for the created final implementations, resulting in the power usage that likely is close to the maximum TDP. Next, the created Demosaic FPGA implementation uses only 20% of the KCU1500 hardware resources and runs at 57% of the maximum clock speed due to path timing violations. The final results for RabbitCT are lost, but similar hardware and clock usage as Demosaic are expected. So, with only ~20% of the hardware in use and the clock running at ~60% of the maximum value, the KCU1500 power usage should be significantly lower than the 75W maximum.

With the literature study results indicating that the FPGA is approximately 10x more energy efficient than a GPU, and most FPGAs from the literature study requiring between 20 and 30W of power, it is likely the achieved final FPGA results also fall within this power envelope. Taking the estimated fully optimized FPGA performances of 0.07x to 0.65x the GPU performance into account, the FPGA becomes 0.7x as efficient as the GPU for RabbitCT and 6.5x more efficient for Demosaic. These results indicate that the efficiency of FPGAs can be anywhere from slightly less efficient to significantly more efficient than the GPU for executing a specific task.

7.6 Product availability

The product availability is not something that can be tested. For this reason, the results from the literature study in Section 3.1 are used. Here, the GPU database of Techpowerup [74], the Intel FPGA catalogue [76], and the Xilinx FPGA catalogue on Digikey.nl [77] were analyzed on the availability of the hardware. The product availability results showed that both Nvidia and AMD GPUs are generally available for two to three years after release, that the oldest Intel FPGA available is 15 years old, and that Xilinx offers to support their FPGAs up to 20 years after release.

Chapter 8

Conclusion

The goal of this research is to determine the value that OpenCL FPGA acceleration brings to the CUDA GPU hardware acceleration market. This value is determined by both a literature study and by optimizing and benchmarking the RabbitCT and Demosaic algorithms to a similarly priced Nvidia Quadro P6000 GPU and Xilinx Kintex KCU1500 FPGA. The performance of the selected algorithms is analyzed using six factors: maximum performance, costs per performance, programmability, interconnect options, energy efficiency and product availability. By comparing these factors for the CUDA GPU, OpenCL GPU, and OpenCL FPGA acceleration methods the differences between the CUDA and OpenCL APIs are revealed, and the value OpenCL FPGA acceleration brings to the market is determined. Below, the results of each analyzed factor are shown, and a recommendation is made.

Maximum performance

Based upon the maximum performance results can be concluded that both the CUDA and OpenCL GPU APIs offer similar performance. The FPGA implementations of both RabbitCT and Demosaic were not fully optimized due to time being limited and FPGA optimization being a time-consuming task. For this reason, an estimated fully optimized FPGA performance is used to compare the FPGA with the fully optimized GPU implementations to make a fair comparison. However, even with the most optimal performance estimation, the FPGA performance never came close to the GPU performance due to to a memory bandwidth bottleneck. Here, the fully optimized RabbitCT FPGA implementation performs at 0.07x the GPU performance, and the Demosaic implementation performs at 0.65x the maximum GPU performance. However, where the final GPU implementations use 100% of the GPU resources, the FPGA implementations use only a fraction of the available FPGA hardware. This hardware usage difference shows that the selected FPGA has room for further parallelization of the implemented pipeline or additional processing tasks which might give the FPGA a higher performance. So, for performance can be concluded that FPGAs cannot reach the same performance as GPUs for memory intensive applications. However, for compute-intensive applications, the FPGA performance will likely come close or even surpass the GPU performance.

Costs per performance

The cost per performance ratio is determined by dividing the costs of the accelerator by the achieved performance. With the final FPGA builds using only a fraction of the available FPGA resources, a smaller FPGA has been selected to reduce the costs for the created FPGA solution. Additionally, as the final FPGA builds were not fully optimized, the remaining optimizations were used to estimate the fully optimized FPGA costs per performance ratio on the optimal hardware. With this in mind, the RabbitCT FPGA implementation costs about 11x more than the GPU for a given performance, and the Demosaic FPGA implementation costs 0.71x as much as the

GPU implementation for a given frame-rate when ignoring PCIe bandwidth limitations. The results show that the floating-point heavy RabbitCT algorithm is more cost-effective on the GPU, whereas the Demosaic algorithm containing only integer calculations is more cost effective on the FPGA. However, when taking the PCIe bottleneck into account and selecting the cheapest GPU PCIe bandwidth limited GPU, the costs-per-performance ratio changes in favour of the GPU, making FPGAs 1.55x more expensive than the GPU for a given performance. However, as the final FPGA implementation on optimal hardware does not use all the FPGA resources, the FPGA costs-per-performance ratio can be further increased if additional computational work is added to the algorithm. Here, the GPU will require more cycles to calculate the result which lowers the throughput, while the FPGA creates a longer pipeline which increases the latency but leaves the throughput unaffected.

Programmability

To make the programmability analysis as objective as possible, the analysis has been divided into a tooling and code analysis on multiple categories for each acceleration method. The tooling was analyzed on the installation, ease of usage, and profiling capabilities; and the code was analyzed on the ease of programming, lines of code, compilation time and portability.

Ranking the programmability of CUDA, OpenCL GPU and OpenCL FPGA showed that CUDA takes the first place by it scoring the highest on all categories except portability. OpenCL on a GPU takes the second place, where its high portability reduces the ease of programming as many extra lines-of-code are required for the host code. Additionally, no fully supported profiling tool is available for OpenCL on Nvidia hardware using the Linux operating system, making it harder to analyze the kernel performance. Finally, OpenCL on FPGA scores the lowest on programmability due to its multi-hour compilation time and the significantly higher complexity for writing efficient kernels as manual memory management is required. The combination of these two factors results in a one week GPU kernel optimization taking a month or longer on the FPGA.

Interconnect options

GPU accelerators only use the PCIe connector for I/O communication, while FPGA accelerators have the freedom to connect any interconnect to the hardware as long as it is supported in the BSP. Adding BSP support for custom interconnect might make the FPGA programmability somewhat harder due to the FPGA knowledge that is required, but it also enables the PCIe bandwidth limitation to be bypassed as the data from this interconnect can be directly routed into the OpenCL kernels.

Energy efficiency

The energy efficiency could not be measured due to the loss of the benchmarking setup in the final weeks of the project. For this reason, an estimation is made which resulted in the FPGA being reportedly up to 10x more energy efficient than the GPU. However, when including the estimated maximum FPGA performances, the FPGA energy efficiency can be anywhere from 0.7x as efficient as the GPU for RabbitCT, to 6.5x more efficient for Demosaic.

Note that these numbers are estimations and do not represent the actual energy efficiency in any way. However, it can be stated that the FPGA is more energy efficient than the GPU when it achieves 10% of the maximum GPU performance or more.

Product availability

The product availability is not something that can be tested, so, only literature study results are used. These results showed that GPUs are produced up to three years after the initial release, while FPGAs are produced up to 20 years after release. Having an extended availability removes the need for a new development process to ensure compatibility with the next generation of hardware when the original hardware is no longer available. So, the extended availability of FPGAs gives them a considerable advantage when selecting hardware for maintainability of a particular acceleration platform.

Summary

Table 8.1 shows all results in a single table. From these results can be concluded that OpenCL FPGA acceleration is most useful in cases where energy efficiency and product maintainability is more important than the maximum performance or having a short development time. A recommended use case for FPGA acceleration is an implementation in accelerators that process data around the clock and that have to be maintained for longer than three years or in machines where custom interconnect is required. GPU, on the other hand, excel in performance due to their larger memory bandwidth, have a short kernel development time, but have a short product availability and high energy usage. Their optimal use-case is a kernel for which high performance is required and which is constantly updated to perform each task as quickly as possible.

Table 8.1: Final results

Metrics	**GPU**		**OpenCL FPGA**	
	CUDA	OpenCL	RabbitCT	Demosaic
Maximum performance	1x	1x	0.07x	0.65x
Costs per performance	1x	1x	11x	0.71x
Programmability	easy	moderate	hard	
Interconnect options	PCIe	HW dependent	Free of choice	
Energy efficiency	1x	1x	10x	
Product availability	3 years	3 years	20 years	

Chapter 9

Future Work & Expectations

9.1 Future work

While many aspects of GPU and FPGA acceleration are analyzed, several factors were not included. The most important factor is the use of custom interconnect. Theoretically, the data from the custom interconnect moves directly into the FPGA, allowing this data to be used to expand the total I/O or memory bandwidth. These changes could increase the FPGA performance as it becomes less dependent on the available PCIe or DDR4 memory bandwidth. The only issue with this custom interconnect is that it requires the FPGA BSP to be adapted to support it and it requires a source for the data which sends the data in the required order over this interconnect. An additional advantage of custom interconnect is that it might reduce access latencies as data is moved directly into the FPGA, removing the need to fetch the data from the global memory first. FPGA and GPU latencies have not been analyzed in this research and could provide another advantage for FPGA acceleration.

Another way to reduce the effect of the memory or PCIe bandwidth limitations is by using compression [152]. Compression can be used to either lower the image quality to reduce the file sizes or to store the image in a more efficient format that first needs to be decompressed to be used. When the image quality is lowered, compression results in smaller I/O data sizes, it reduces both the global memory and cache usage, and it reduces the required PCIe bandwidth. So, the accelerator will be able to perform the algorithm at a higher performance at the cost of introducing compression errors in the output. The more efficient file format compression option requires the compressed image first to be decompressed before being used as accelerator input. This compression method saves some PCIe bandwidth and reduces global memory usage, but does not improve the caching behaviour as both the compressed and decompressed data is present on the accelerator. Future work could analyze the effect of both compression methods on both the FPGA and GPU to see whether it helps to overcome bandwidth limitations and increase the overall performance.

As both the selected RabbitCT and Demosaic algorithms are memory bandwidth bottlenecked on the FPGA, additional research should also analyze a compute-intensive application. Benchmarking a compute-intensive application might show the actual performance capabilities of an FPGA when the memory bandwidth is not an issue. Additionally, it might allow all FPGA hardware elements to be used resulting in the FPGA being 'too small' for the algorithm. In this case, the FPGA might have to be reprogrammed during execution, after which the reprogramming time can be analyzed. If this FPGA reprogramming time is short enough, reconfiguring the FPGA to execute new kernels might not have a significant performance impact, which would allow even the smallest FPGAs to perform large and complex tasks by switching its configuration.

Finally, the energy efficiency of GPU and FPGA acceleration should still be analyzed. Here, performing an average energy usage measurement over several benchmarks for both the GPU and FPGA will allow both platforms to be compared. When this energy usage is then to determine

the performance per watt, the actual energy usage of both platforms for a particular benchmark can be visualized.

9.2 Future expectations

The future of FPGA acceleration looks bright as both Intel and Xilinx recognize the value OpenCL acceleration on FPGAs can bring to the market. Both vendors have recently released a new generation of FPGA accelerators that use their newest architectures. These architectures are further optimized for acceleration allowing higher clock speeds/performance than the current FPGA generation, and they are more energy efficient [153, 154]. Next, from both vendors, FPGA accelerators using HBM2 memory are available at third party companies [155, 156]. The use of HBM2 removes the memory bottleneck that currently limits the FPGA performance and allows them to compete or surpass the performance of the latest GPUs.
Additionally, these FPGA accelerators have four QSFP network cages, supporting up to 100 GbE per port. These network cages add a maximum of 50 GB/s bandwidth on top of the already available PCIe 3.0 15.8 GB/s bandwidth, eliminating the host-accelerator bandwidth issue. The only downside to these HBM2 FPGAs is that they currently are only used in the flagship models of both Intel and Xilinx. These flagship models use the largest FPGA dies available and are produced in limited quantities, which makes them very expensive. So, HBM2 FPGAs are not available yet on the lower end of the market, limiting those FPGAs to the currently available DDR4 memory solutions.

The future of GPU hardware lies in the addition of specialized hardware to the GPU die for Nvidia. Nvidia has added specialized ray tracing and tensor cores to their Turing GPU hardware, which are used to accelerate real-time ray tracing and floating point matrix computation tasks for deep-learning. So, instead of focusing on performance increases of the available hardware, Nvidia focuses on increasing the performance of their GPUs in two specific areas. Next, AMD is hast just released a set of Vega 20 GPUs on the 7nm node and have their next-generation GPU architecture Navi coming up [157]. This Vega 20 GPU series uses PCIe 4.0, which doubles the available PCIe bandwidth over PCIe 3.0. However, currently no CPUs or motherboards PCIe 4.0 are available yet, but they will be in mid-2019 [158]. Finally, Intel also aims to launch a dedicated GPU in 2020 to compete with both Nvidia and AMD, but no other information is known [159].

While the maximum performance of both acceleration platforms will keep increasing, the programmability is not expected to change. GPUs will remain a quicker and easier to program than FPGAs because of the required manual FPGA kernel memory management and the long FPGA hardware compilation times. The kernel memory management is not expected to become easier as this requires to OpenCL compiler to detect an optimal memory management strategy for all divisible FPGA implementations, which requires a significant investment in the OpenCL tooling and is unlikely to happen soon. Next, the FPGA hardware will always need to be compiled, and thus will remain a bottleneck for testing on the hardware. Software analysis tools help to postpone the need for actual FPGA compilation, but analyzing these results still takes time, resulting in FPGA kernel development always taking more time than GPU development.

Bibliography

Programming guides

[1] Intel. Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide, 2018. URL https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html.

[2] Xilinx. SDAccel Development Environment User Guide v2015.1. Technical report, 2015. URL https://china.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug1023-sdaccel-user-guide.pdf.

[3] Xilinx. SDAccel Environment Profiling and Optimization Guide 2017.4, 2017. URL https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/ehb1504034292718.html.

[4] Xilinx. SDAccel Environment Profiling and Optimization Guide 2018.2, 2018. URL https://www.xilinx.com/html_docs/xilinx2018_2/sdaccel_doc/zgr1534452172723.html.

[5] Xilinx. SDAccel Environment Programmers Guide 2018.2, 2018. URL https://www.xilinx.com/html_docs/xilinx2018_2/sdaccel_doc/vno1533881025717.html.

[6] Xilinx. SDAccel Optimization Guide v2016.4, 2017. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1207-sdaccel-optimization-guide.pdf.

[7] Intel. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide, 2018. URL https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html.

[8] Xilinx. Vivado HLS Optimization Methodology Guide 2018.1, 2018. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf.

[9] Nvidia. CUDA C Programming Guide, 2018. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[10] Nvidia. OpenCL Best Practices Guide, 2011. URL https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf.

[11] Nvidia. OpenCL Programming Guide for the CUDA Architecture, Version 4.2. *CUDA SDK*, 2012. URL https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Programming_Guide.pdf.

[12] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide. Technical report, 2013. URL http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf.

[13] Intel. Intel FPGA SDK for OpenCL Pro Edition: Getting Started Guide, 2018. URL https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807309901.html.

[14] Intel. Intel FPGA SDK for OpenCL Pro Edition: Custom Platform Toolkit User Guide, 2018. URL https://www.intel.com/content/www/us/en/programmable/documentation/ewa1402666946838.html.

[15] Xilinx. SDAccel Environment Platform Development Guide v2017.4. 2018. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1164-sdaccel-platform-development.pdf.

[16] Xilinx. SDAccel Environment Release Notes, Installation, and Licensing Guide v2018.2, 2018. URL https://www.xilinx.com/html_docs/xilinx2018_2/sdaccel_doc/yrc1534452173645.html.

[17] Xilinx. Vivado Design Suite User Guide: Release Notes, Installation, and Licensing v2018.2. Technical report, 2018. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug973-vivado-release-notes-install-license.pdf.

[18] Xilinx. SDAccel Environment User Guide 2018.2, 2018. URL https://www.xilinx.com/html_docs/xilinx2018_2/sdaccel_doc/itd1534452174535.html.

[19] Khronos Group. OpenCL 1.2 Reference Pages. URL https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/.

[20] Xilinx. Xilinx SDAccel Examples GitHub, 2018. URL https://github.com/Xilinx/SDAccel_Examples.

[21] Nvidia. CUDA Installation Guide for Microsoft Windows, 2019. URL http://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html.

Referenced papers

[22] C. Rohkohl, B. Keck, H. G. Hofmann, and J. Hornegger. Technical Note: RabbitCT-an open platform for benchmarking 3D cone-beam reconstruction algorithmsa). *Medical Physics*, 36(9 Part 1):3940–3944, 2009. ISSN 00942405. doi: 10.1118/1.3180956. URL http://doi.wiley.com/10.1118/1.3180956.

[23] H.S. Malvar, Li-wei He, and R. Cutler. High-quality linear interpolation for demosaicing of Bayer-patterned color images. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 485–8. IEEE, 2004. ISBN 0-7803-8484-9. doi: 10.1109/ICASSP.2004.1326587. URL http://ieeexplore.ieee.org/document/1326587/.

[24] Brian W. Kernighan. A programming language called C. *IEEE Potentials*, 2(December): 26–30, 12 1983. ISSN 0278-6648. doi: 10.1109/MP.1983.6499601. URL http://ieeexplore.ieee.org/document/6499601/.

[25] Kenneth Hill, Stefan Craciun, Alan George, and Herman Lam. Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, volume September, pages 189–193. IEEE, 7 2015. ISBN 9781479919246. doi: 10.1109/ASAP.2015.7245733. URL http://ieeexplore.ieee.org/document/7245733/.

[26] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. CUDA vs OpenACC: Performance case studies with Kernel benchmarks and a memory-bound CFD application. In *Proceedings - 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013*, pages 136–143. IEEE, 5 2013. ISBN 978-0-7695-4996-5. doi: 10.1109/CCGrid.2013.12. URL http://ieeexplore.ieee.org/document/6546071/.

[27] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, pages 465–471. IEEE, 11 2012. ISBN 9780769549569. doi: 10.1109/SC.Companion.2012.66. URL http://ieeexplore.ieee.org/document/6495848/.

[28] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Achieving portability and performance through OpenACC. In *Proceedings of WACCPD 2014: 1st Workshop on Accelerator Programming Using Directives - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 19–26. IEEE, 11 2015. ISBN 9781479970230. doi: 10.1109/WACCPD.2014.10. URL http://ieeexplore.ieee.org/document/7081674/.

[29] Seyong Lee, Jungwon Kim, and Jeffrey S. Vetter. OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. In *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pages 544–554. IEEE, 5 2016. ISBN 9781509021406. doi: 10.1109/IPDPS.2016.28. URL http://ieeexplore.ieee.org/document/7516051/.

[30] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming Heterogeneous Systems from an Image Processing DSL. 10 2016. ISSN 15443566. doi: 10.1145/3107953. URL http://arxiv.org/abs/1610.09405.

[31] Hsiang-Wei Sung, Yuan-Ming Chang, Shao-Chung Wang, and Jenq-Kuen Lee. OpenCV Optimization on Heterogeneous Multi-core Systems for Gesture Recognition Applications. *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 59–65, 8 2016. doi: 10.1109/ICPPW.2016.24. URL http://ieeexplore.ieee.org/document/7576453/.

[32] Davy Oliveira Barros Sousa Daniel Oliveira Dantas, Helton Danilo Passos Leal. Fast 2D and 3D image processing with OpenCL. *IEEE International Conference onImage Processing (ICIP)*, pages 4858–4862, 2015. ISSN 15224880. doi: 10.1109/ICIP.2015.7351730.

[33] Hercules Cardoso Da Silva, Flavia Pisani, and Edson Borin. A comparative study of SYCL, OpenCL, and OpenMP. In *Proceedings - 28th IEEE International Symposium on Computer Architecture and High Performance Computing Workshops, SBAC-PADW 2016*, pages 61–66. IEEE, 10 2017. ISBN 9781509048441. doi: 10.1109/SBAC-PADW.2016.19. URL http://ieeexplore.ieee.org/document/7803697/.

[34] Junghyun Kim, Thanh Tuan Dao, Jaehoon Jung, Jinyoung Joo, and Jaejin Lee. Bridging OpenCL and CUDA: A Comparative Analysis and Translation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*, pages 1–12, 2015. ISBN 9781450337236. doi: 10.1145/2807591.2807621. URL http://dl.acm.org/citation.cfm?doid=2807591.2807621.

[35] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 8 2012. ISSN 01678191. doi: 10.1016/j.parco.2011.10.002. URL https://www.sciencedirect.com/science/article/pii/S0167819111001335.

[36] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009*, pages 44–54. IEEE, 10 2009. ISBN 9781424451562. doi: 10.1109/IISWC.2009.5306797. URL http://ieeexplore.ieee.org/document/5306797/.

[37] M. J. Harvey and G. De Fabritiis. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 182(4):1093–1099, 4 2011. ISSN 00104655. doi: 10.1016/j.cpc.2010.12.052. URL https://www.sciencedirect.com/science/article/pii/S0010465511000117.

[38] Gabriel Martinez, Mark Gardner, and Wu Chun Feng. CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pages 300–307. IEEE, 12 2011. ISBN 9780769545769. doi: 10.1109/ICPADS.2011.48. URL http://ieeexplore.ieee.org/document/6121291/.

[39] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro*, 35(3):10–22, 6 2015. ISSN 02721732. doi: 10.1109/MM.2015.42. URL http://ieeexplore.ieee.org/document/6853195/.

[40] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Daniel Firestone, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. Configurable Clouds. *IEEE Micro*, 37(3):52–61, 2017. ISSN 02721732. doi: 10.1109/MM.2017.51. URL http://ieeexplore.ieee.org/document/7948672/.

[41] Dong Yin, Ge Li, and Ke Di Huang. Scalable MapReduce framework on FPGA accelerated commodity hardware. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7469 LNCS, pages 280–294, 2012. ISBN 9783642326851. doi: 10.1007/978-3-642-32686-8{_}26.

[42] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPMR: MapReduce Framework on FPGA A Case Study of RankBoost Acceleration. *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '10*, page 93, 2010. doi: 10.1145/1723112.1723129. URL http://portal.acm.org/citation.cfm?doid=1723112.1723129.

[43] Berkeley Design Technology. The AutoESL AutoPilot High-Level Synthesis Tool. *Design*, 2010. URL https://www.bdti.com/InsideDSP/2010/02/16/Autoesl.

[44] Christoforos Kachris and Dimitrios Soudris. A survey on reconfigurable accelerators for cloud computing. In *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016. ISBN 9782839918442. doi: 10.1109/FPL.2016.7577381. URL http://ieeexplore.ieee.org/document/7577381/.

[45] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes. In *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, pages 36–47, 2013. ISBN 9781450320795. doi: 10.1145/2485922.2485926. URL http://dl.acm.org/citation.cfm?doid=2485922.2485926.

[46] Lester Kalms and Diana Gohringer. Exploration of OpenCL for FPGAs using SDAccel and comparison to GPUs and multicore CPUs. In *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*, pages 1–4. IEEE, 9 2017. ISBN 9789090304281. doi: 10.23919/FPL.2017.8056847. URL http://ieeexplore.ieee.org/document/8056847/.

[47] Pablo Fernndez Alcantarilla, Adrien Bartoli, and Andrew J. Davison. KAZE features. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7577 LNCS, pages 214–227, 2012. ISBN 9783642337826. doi: 10.1007/978-3-642-33783-3{_}16.

[48] Quentin Gautier, Alexandria Shearer, Janarbek Matai, Dustin Richmond, Pingfan Meng, and Ryan Kastner. Real-time 3D reconstruction for FPGAs: A case study for evaluating the performance, area, and programmability trade-offs of the Altera OpenCL SDK. In *Proceedings of the 2014 International Conference on Field-Programmable Technology, FPT 2014*, pages 326–329. IEEE, 12 2015. ISBN 9781479962457. doi: 10.1109/FPT.2014.7082810. URL http://ieeexplore.ieee.org/document/7082810/.

[49] Deo Manish, Schulz Jeffrey, and Brown Lance. Intel Stratix 10 MX Devices Solve the Memory Bandwidth Challenge, 2017. URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01264-stratix10mx-devices-solve-memory-bandwidth-challenge.pdf.

[50] Xilinx. Xilinx Memory Solutions. URL https://www.xilinx.com/products/technology/memory.html.

[51] Maxime Martelli, Nicolas Gag, Alain Merigot, and Cyrille Enderli. 3D tomography back-projection parallelization on FPGAs using opencl. In *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–6. IEEE, 9 2017. ISBN 978-1-5386-3534-6. doi: 10.1109/DASIP.2017.8122119. URL http://ieeexplore.ieee.org/document/8122119/.

[52] Amulya Vishwanath. Enabling High-Performance Floating-Point Designs. *Intel Whitepaper*, 2016. URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01267-fpgas-enable-high-performance-floating-point.pdf.

[53] BertenDSP. GPU vs FPGA Performance Comparison White Paper 2. Technical report, 2016. URL www.bertendsp.com.

[54] Hamid Reza Zohouri, Naoya Maruyamay, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, pages 409–420. IEEE, 11 2017. ISBN 9781467388153. doi: 10.1109/SC.2016.34. URL http://ieeexplore.ieee.org/document/7877113/.

[55] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. *IEEE Access*, 5:2747–2762, 2017. ISSN 2169-3536. doi: 10.1109/ACCESS.2017.2671881. URL http://ieeexplore.ieee.org/document/7859319/.

[56] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. FPGA Acceleration of Recurrent Neural Network Based Language Model. *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 111–118, 2015. doi: 10.1109/FCCM.2015.50. URL http://ieeexplore.ieee.org/document/7160054/.

[57] Li Boxun, Zhou Erjin, Huang Bo, Duan Jiayi, Wang Yu, Xu Ningyi, Zhang Jiaxing, and Yang Huazhong. Large scale recurrent neural network on GPU. *Neural Networks (IJCNN), 2014 International Joint Conference on*, pages 4062–4069, 7 2014. doi: 10.1109/IJCNN.2014.6889433. URL http://ieeexplore.ieee.org/document/6889433/.

[58] Jialiang Zhang and Jing 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 - FPGA '17*, pages 25–34, 2017. ISBN 9781450343541. doi: 10.1145/3020078.3021698. URL http://dl.acm.org/citation.cfm?doid=3020078.3021698.

[59] Lorenzo Di Tucci, Marco Rabozzi, Luca Stornaiuolo, and M.D. Santambrogio. The role of CAD frameworks in heterogeneous FPGA-based cloud systems. In *Proceedings - 35th IEEE International Conference on Computer Design, ICCD 2017*, pages 423–426. IEEE, 11 2017. ISBN 9781538622544. doi: 10.1109/ICCD.2017.74. URL http://ieeexplore.ieee.org/document/8119247/.

[60] Ambrose Finnerty and Herv Ratigner. Reduce Power and Cost by Converting from Floating Point to Fixed Point. volume 491. 2017. URL https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf.

[61] Eric Papenhausen and Klaus Mueller. Rapid rabbit: Highly optimized GPU accelerated cone-beam CT reconstruction. In *IEEE Nuclear Science Symposium Conference Record*, 2013. ISBN 9781479905348. doi: 10.1109/NSSMIC.2013.6829126. URL http://ieeexplore.ieee.org/document/6829126/.

[62] Christian Siegl, H G Hofmann, B Keck, M Prümmer, and J Hornegger. OpenCL: a viable solution for high-performance medical image reconstruction? *SPIE Medical Imaging: Physics of Medical Imaging*, 7961:79612Q, 2011. ISSN 0277-786X. doi: 10.1117/12.878058. URL https://pdfs.semanticscholar.org/32b6/45ca9aeef6724c7368f20919faf0a0cd1e1a.pdf.

[63] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):105–118, 1 2011. ISSN 10459219. doi: 10.1109/TPDS.2010.107. URL http://ieeexplore.ieee.org/document/5473222/.

[64] L. A. Feldkamp, L. C. Davis, and J. W. Kress. Practical cone-beam algorithm. *Journal of the Optical Society of America A*, 1(6):612, 6 1984. ISSN 1084-7529. doi: 10.1364/JOSAA.1.000612. URL https://www.osapublishing.org/abstract.cfm?URI=josaa-1-6-612.

[65] Hen-Wai Tsao and Da-Cheng Sung. Demosaicing using subband-based classifiers. *Electronics Letters*, 51(3):228–230, 2015. ISSN 0013-5194. doi: 10.1049/el.2014.1557. URL http://digital-library.theiet.org/content/journals/10.1049/el.2014.1557.

[66] Marcel Nawrath and Jens Jäkel. Deriving color images from noisy Bayer data using local demosaicking and non-local denoising. In *Proceedings - 4th International Congress on Image and Signal Processing, CISP 2011*, volume 2, pages 668–672, 2011. ISBN 9781424493067. doi: 10.1109/CISP.2011.6100289.

[67] Michael Parker. Understanding Peak Floating-Point Performance Claims. Technical report, 2017. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf.

[68] Xilinx. Pushing Performance and Integration with the UltraScale+ Portfolio. volume 471, pages 1–10. 2015. URL http://www.xilinx.com/support/documentation/white_papers/wp471-ultrascale-plus-perf.pdf.

Referenced websites

[69] Jamal Robinson. FPGAs, Deep Learning, Software Defined Networks and the Cloud: A Love Story Part 1, 2017. URL https://medium.com/@jamal.robinson/fpgas-deep-learning-software-defined-networks-and-the-cloud-a-love-story-part-1-c685dc6b657

[70] Bill Thomas. AMD Ryzen Threadripper 2nd Generation release date, news and features, 10 2018. URL https://www.techradar.com/news/amd-ryzen-threadripper-2nd-generation.

[71] Ryan Smith. Big Volta Comes to Quadro: NVIDIA Announces Quadro GV100, 2018. URL https://www.anandtech.com/show/12579/big-volta-comes-to-quadro-nvidia-announces-quadro-gv100.

[72] Prodrive Technologies. Prodrive Press Release Fire, 2018. URL https://prodrive-technologies.com/press-release-prodrive-technologies/.

[73] Omroepbrabant. Prodrive Fire Pictures, 2018. URL https://www.omroepbrabant.nl/nieuws/2894606/Foto-s-leggen-enorme-brand-bij-Prodrive-op-industrieterrein-in-Son-vast.

[74] Techpowerup. Techpowerup GPU Database, 2018. URL https://www.techpowerup.com/gpu-specs/?mfgr=NVIDIA&generation=Quadro&sort=name.

[75] Intel. Intel FPGA Store, 2018. URL https://buyfpga.intel.com/Search?kw=maxII&stock=True.

[76] Intel. Intel FPGA Devices, 2018. URL https://www.intel.com/content/www/us/en/products/programmable/fpga.html.

[77] Digikey. Digikey Store, 2018. URL https://www.digikey.nl/.

[78] Xilinx. Xilinx Spartan-6, 2018. URL https://www.xilinx.com/products/silicon-devices/fpga/spartan-6.html.

[79] Intel. Intel HLS Compiler - Overview, . URL https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html.

[80] Xilinx. Vivado High-Level Synthesis, . URL https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[81] Xilinx. SDAccel Development Environment, . URL https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[82] Xilinx. SDSoC Development Environment, . URL https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.

[83] Xilinx. SDNet Development Environment, . URL https://www.xilinx.com/products/design-tools/software-zone/sdnet.html.

[84] Intel. Intel FPGA SDK for OpenCL - Overview, . URL https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[85] Khronos Group. OpenCL Overview, . URL https://www.khronos.org/opencl/.

[86] Future Technologies Group. OpenARC: Open Accelerator Research Compiler. URL https://ft.ornl.gov/research/openarc.

[87] Halide website. URL http://halide-lang.org/.

[88] Jingpu. Halide HLS. URL https://github.com/jingpu/Halide-HLS.

[89] OpenCV. OpenCV Library. URL https://opencv.org/about.html.

[90] Xilinx. Xilinx xfOpenCV, . URL https://github.com/Xilinx/xfopencv.

[91] Intel. Intel® Distribution of OpenVINO toolkit, 2018. URL https://software.intel.com/en-us/openvino-toolkit.

[92] Khronos Group. SYCL Overview, . URL https://www.khronos.org/sycl.

[93] Intel. Intel Stratix V - Product Table, . URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-v-product-table.pdf.

[94] Clive Maxfield. AutoESL acquisition a great move for Xilinx, 1 2011. URL https://www.eetimes.com/author.asp?section_id=36&doc_id=1284904.

[95] Rodinia. Rodinia: Accelerating Compute-Intensive Applications with Accelerators. URL https://rodinia.cs.virginia.edu/doku.php.

[96] Intel. Intel Arria 10 Architecture, 2016. URL https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10/features.html.

[97] Xilinx. Xilinx IP: Floating-Point Operator, . URL https://www.xilinx.com/products/intellectual-property/floating_pt.html.

[98] Micron. About Convey Computer Accelerator Products, 2015. URL https://www.micron.com/about/about-convey-computer-accelerator-products.

[99] Nvidia. Quadro in desktop workstations, 2018. URL https://www.nvidia.com/en-us/design-visualization/quadro-desktop-gpus/.

[100] Amazon. Amazon EC2 F1 Instances. URL https://aws.amazon.com/ec2/instance-types/f1/.

[101] Jon "Hannibal" Stokes. SIMD Architectures, 2004. URL http://archive.arstechnica.com/cpu/1q00/simd/m-simd-1.html.

[102] Acceleware. OpenCL on FPGAs for GPU Programmers. Technical report, 2014. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-201406-acceleware-opencl-on-fpgas-for-gpu-programmers.pdfhttp://design.altera.com/Acceleware_OpenCL_FPGA_WP.

[103] Xilinx. Performance and Resource Utilization for Adder/Subtracter v12.0, 2018. URL https://www.xilinx.com/support/documentation/ip_documentation/ru/c-addsub.html.

[104] Xilinx. Performance and Resource Utilization for Floating Point v7.1, 2018. URL https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html.

[105] Intel. Floating-Point IP: Resource Utilization and Performance, 2016. URL https://www.intel.com/content/altera-www/global/en_us/index/documentation/eis1410764818924.html.

[106] Xilinx. High Speed Serial. . URL https://www.xilinx.com/products/technology/high-speed-serial.html.

[107] Jeremiah van Oosten. 3D Game Engine Programming, 2011. URL https://www.3dgep.com/cuda-memory-model/.

[108] Jeremy Appleyard. CUDA Pro Tip: Optimize for Pointer Aliasing, 2014. URL https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/.

[109] Nvidia. NVLink, . URL https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/.

[110] Informatik department of the Friedrich-Alexander-Universität. RabbitCT Website. URL https://www5.cs.fau.de/research/former-projects/rabbitct/.

[111] Saoni Mukherjeet, Nicholas Moore, James Brock, and Miriam Leeser. CUDA and OpenCL implementations of 3D CT reconstruction for biomedical imaging. *2012 IEEE Conference on High Performance Extreme Computing, HPEC 2012*, 2012. doi: 10.1109/HPEC.2012.6408674. URL http://ieee-hpec.org/2012/index_htm_files/mukherjee.pdf.

[112] Stanislav Maslan. Cone-beam backprojection tool, 2018. URL http://elektronika.kvalitne.cz/SW/graphics/cone_beam_backprojection/cone_beam_backprojection_eng.html.

[113] ASTRA Toolbox. Filtered Backprojection (FBP) Youtube, 2015. URL https://www.youtube.com/watch?v=pZ7JlXagT0w.

[114] Datagenetics. Sphere in Cylinders. URL http://datagenetics.com/blog/july32014/index.html.

[115] Informatik department of the Friedrich-Alexander-Universität. RabbitCT Ranking, 2016. URL https://www5.cs.fau.de/research/former-projects/rabbitct/ranking/.

[116] CambridgeInColour. Digital camera sensors. URL https://www.cambridgeincolour.com/tutorials/camera-sensors.htm.

[117] RED. The Bayer Sensor Strategy, . URL https://www.red.com/red-101/bayer-sensor-strategy.

[118] Yves Roumazeilles. Camera Raw 5.6 is here, 2009. URL https://www.ylovephoto.com/en/2009/11/19/camera-raw-5-6-is-here/.

[119] Joe Maller. FXScript Reference: RGB and YUV Color. URL http://joemaller.com/fcp/fxscript_yuv_color.shtml.

[120] RED. Chroma Subsampling Techniques, . URL https://www.red.com/red-101/video-chroma-subsampling.

[121] Morgan McGuire. Efficient, High-Quality Bayer Demosaic Filtering on GPUs. *Journal of Graphics Tools*, 13(4):1–16, 2008. ISSN 1086-7651. doi: 10.1080/2151237X.2008.10129267. URL https://pdfs.semanticscholar.org/088a/2f47b7ab99c78d41623bdfaf4acdb02358fb.pdfhttps://www.tandfonline.com/doi/full/10.1080/2151237X.2008.10129267.

[122] Even Rouault, Bob Friesenhahn, Frank Warmerdam, Andrey Kiselev, Joris van Damme, and Lee Howard. LibTIFF, 2017. URL http://www.simplesystems.org/libtiff/.

[123] Intel. Intel FPGA SDK for OpenCL overview, 2018. URL https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html.

[124] Intel. Intel FPGA Acceleration Hub, 2018. URL https://www.intel.com/content/www/us/en/programmable/solutions/acceleration-hub/platforms.html.

[125] Intel. Cyclone V SoC Development Kit and Intel SoC FPGA Embedded Development Suite, . URL https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-soc.html.

[126] Intel. Arria 10 GX FPGA Development Kit, . URL https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-a10-gx-fpga.html.

[127] Intel. Stratix V GX FPGA Development Kit, . URL https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-sv-gx-host.html.

[128] Intel. Stratix 10 GX FPGA Development Kit, . URL https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-fpga.html.

[129] Xilinx. Xilinx Kintex UltraScale FPGA KCU1500 Acceleration Development Kit, . URL https://www.xilinx.com/products/boards-and-kits/dk-u1-kcu1500-g.html.

[130] Xilinx. Xilinx Virtex Ultrascale+ FPGA VCU1525 Acceleration Development Kit, . URL https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html.

[131] Xilinx. Alveo U200 Data Center Accelerator Card, 2018. URL https://www.xilinx.com/products/boards-and-kits/alveo/u200.html.

[132] Xilinx. Alveo U250 Data Center Accelerator Card, 2018. URL https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

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

[134] Tweakers. Tweakers Pricewatch, 2018. URL https://tweakers.net/pricewatch/.

[135] Elsa-jp. Quadro P6000, 2018. URL http://www.elsa-jp.co.jp/products/products-top/graphicsboard_pro/quadro/ultra_high_end_2/quadro_p6000/.

[136] Georg Zitzlsberger. Intel Architecture for HPC Developers, 2015. URL https://indico.cern.ch/event/403113/contributions/1847268/attachments/1123555/1603259/01_Intel_Architecture_for_HPC_Developers.pdf.

[137] Xilinx. Ultrascale FPGA Product Table, 2016. URL https://www.xilinx.com/support/documentation/selection-guides/ultrascale-fpga-product-selection-guide.pdf.

[138] Intel FPGA. Intel Stratix 10 Product Table. pages 5–6, 2018. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf.

[139] Techpowerup. Techpowerup Quadro P6000, 2016. URL https://www.techpowerup.com/gpu-specs/quadro-p6000.c2865.

[140] Nvidia. Nvidia System Management Interface, 2018. URL https://developer.nvidia.com/nvidia-system-management-interface.

[141] Nvida. Nvidia Visual Profiler, 2018. URL https://developer.nvidia.com/nvidia-visual-profiler.

[142] Xilinx. Xilinx Technology: FPGA Memory, . URL https://www.xilinx.com/products/technology/memory.html.

[143] Xilinx. Kintex Ultrascale Product Table, . URL https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale.html.

[144] Xilinx. Xilinx Forums Linking Error, 2018. URL https://forums.xilinx.com/t5/SDAccel/Vivado-failed-to-link-libdpi-so/m-p/870521.

[145] Xilinx. Xilinx Forums Printing error, 2018. URL https://forums.xilinx.com/t5/SDAccel/My-SDx-IDE-can-t-run-the-printf-example-code/m-p/870574.

[146] Nvidia. Nvidia Visual Profiler, 2018. URL https://developer.nvidia.com/nvidia-visual-profiler.

[147] Nvidia. Nvidia Nsight Visual Studio Edition, . URL https://developer.nvidia.com/nsight-visual-studio-edition.

[148] Intel. Intel SDK for OpenCL, . URL https://software.intel.com/en-us/intel-opencl.

[149] AMD. Code XL. URL https://gpuopen.com/compute-product/codexl/.

[150] Mark Harris. CUDA Pro Tip: nvprof is Your Handy Universal GPU Profiler, 2013. URL https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/.

[151] User3829636. New issue stall reasons in NVIDIA Nsight, 2015. URL https://stackoverflow.com/questions/25067518/new-issue-stall-reasons-in-nvidia-nsight-visual-studio-edition-4-1-rc1.

[152] Nathan Reed. Understanding BCn Texture Compression Formats, 2012. URL http://www.reedbeta.com/blog/understanding-bcn-texture-compression-formats/.

[153] Xilinx. FPGA Power Efficiency, . URL https://www.xilinx.com/products/technology/power.html.

[154] Intel. Intel Stratix 10 FPGAs, . URL https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html.

[155] BittWare. Bittware 520N-MX - Intel Stratix 10, . URL https://www.bittware.com/fpga/520n-mx/.

[156] BittWare. Bittware XUPVVH - Xilinx Virtex Ultrascale+, . URL https://www.bittware.com/fpga/xupvvh/.

[157] Zhiye Liu. AMD Claims First 7nm GPUs With Radeon Instinct MI60, MI50, 2018. URL https://www.tomshardware.com/news/amd-radeon-instinct-mi60-mi50-7nm-gpus,38031.html.

[158] Ian Cutress. AMD Ryzen 3rd Gen 'Matisse' Coming Mid 2019: Eight Core Zen 2 with PCIe 4.0 on Desktop, 2019. URL https://www.anandtech.com/show/13829/amd-ryzen-3rd-generation-zen-2-pcie-4-eight-core.

[159] Mark Campbell. Intel Invites Gamers to join a Global Graphics Odyssey, 2019. URL https://www.overclock3d.net/news/gpu_displays/intel_invites_gamers_to_join_a_global_graphics_odyssey/1.