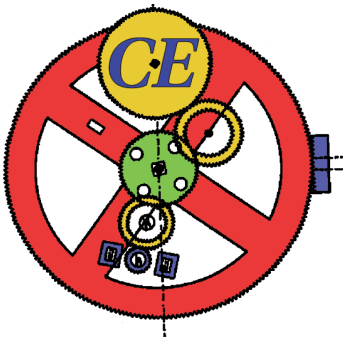


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2016

MSc THESIS

High Performance OpenCL Implementation of Medical Image Processing Algorithms [CP]

Panagiotis Mitsis

CE-MS-2016-13

Abstract

Current X-ray machines use lower radiation doses which introduces noise to the output images. Therefore such systems need to enhance the image and reduce the noise via different algorithms to provide the best possible output. In addition, it is crucial to accelerate these image processing algorithms as the output is intended to be a real time video (fluoroscopy). Such systems are used for example in surgeries for implants or other medical examinations and there is a need to provide constant performance, otherwise they may lead to injuries or fatalities due to latency issues.
Currently, such systems often rely on server PCs to implement the image processing chains. Since PC hardware needs to be replaced regularly during the lifetime of an X-ray machine, this increases the maintenance cost as well as the overall cost of the machine significantly. Therefore, we need to provide a framework that would allow us to develop the algorithm only once and then enable us to port it to a new platform, while the performance is ensured.
In order to do so, a high performance framework solution was investigated. A number of alternative solutions were investigated and the most attractive framework was selected to be the Open Computing Language (OpenCL). OpenCL provides the means to develop the image processing algorithm once and port it to different platforms, changing only the target platform from the OpenCL API.
During this thesis exploration we were able to redevelop a high quality algorithm provided by Philips Healthcare from a Matlab model to OpenCL in an optimal time period, while we investigated portability and performance. We first developed a tool chain that enables transformation from Matlab to OpenCL. Furthermore, 11 image processing kernels which constitute the algorithm were developed in OpenCL performing a speedup of up to 150x in some cases. We were able to run the algorithm on three different hardware platforms using the same OpenCL kernels and achieve a speedup up to of 36x compared to the baseline implementation in C.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

High Performance OpenCL Implementation of Medical Image Processing Algorithms [CP]

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Panagiotis Mitsis
born in Ioannina, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

High Performance OpenCL Implementation of Medical Image Processing Algorithms [CP]

by Panagiotis Mitsis

Abstract

Current X-ray machines use lower radiation doses which introduces noise to the output images. Therefore such systems need to enhance the image and reduce the noise via different algorithms to provide the best possible output. In addition, it is crucial to accelerate these image processing algorithms as the output is intended to be a real time video (fluoroscopy). Such systems are used for example in surgeries for implants or other medical examinations and there is a need to provide constant performance, otherwise they may lead to injuries or fatalities due to latency issues.

Currently, such systems often rely on server PCs to implement the image processing chains. Since PC hardware needs to be replaced regularly during the lifetime of an X-ray machine, this increases the maintenance cost as well as the overall cost of the machine significantly. Therefore, we need to provide a framework that would allow us to develop the algorithm only once and then enable us to port it to a new platform, while the performance is ensured.

In order to do so, a high performance framework solution was investigated. A number of alternative solutions were investigated and the most attractive framework was selected to be the Open Computing Language (OpenCL). OpenCL provides the means to develop the image processing algorithm once and port it to different platforms, changing only the target platform from the OpenCL API.

During this thesis exploration we were able to redevelop a high quality algorithm provided by Philips Healthcare from a Matlab model to OpenCL in an optimal time period, while we investigated portability and performance. We first developed a tool chain that enables transformation from Matlab to OpenCL. Furthermore, 11 image processing kernels which constitute the algorithm were developed in OpenCL performing a speedup of up to 150x in some cases. We were able to run the algorithm on three different hardware platforms using the same OpenCL kernels and achieve a speedup up to of 36x compared to the baseline implementation in C.

Laboratory : Computer Engineering
Codenumber : CE-MS-2016-13

Committee Members :

Advisor:	dr. ir. Zaid Al-Ars, CE, TU Delft
Chairperson:	prof. dr. ir. Koen Bertels, CE, TU Delft
Member:	dr. ir. Martin van Gijzen , DIAM, TU Delft
Member:	ing. Rob J. de Jong, Supervisor, Philips Healtchare

Dedicated to my parents: Christodoulos and Maria, who were always supporting my dreams and my brother Dimitris for believing in me.

Contents

List of Figures **vii**

List of Tables **ix**

List of Acronyms **xii**

Acknowledgements **xiii**

1 Introduction **1**
1.1 Context . . . 1
1.1.1 Philips Healthcare . . . 1
1.1.2 Almarvi . . . 3
1.2 Background . . . 4
1.2.1 Xray system description . . . 4
1.2.2 Image Processing . . . 7
1.3 Problem definition . . . 7
1.4 Thesis Outline . . . 8

2 Background **11**
2.1 OpenCL . . . 11
2.2 Architectures . . . 17
2.2.1 Platform Setup . . . 17
2.2.2 Intel CPU . . . 18
2.2.3 Intel GPU . . . 19
2.2.4 rVex . . . 22

3 Alternative solutions **27**
3.1 Software solutions . . . 27
3.1.1 CUDA . . . 27
3.1.2 SSE/AVX . . . 28
3.2 Hardware solutions . . . 28
3.2.1 rVex on Zynq . . . 29
3.2.2 SDAccel . . . 29
3.2.3 Vivado HLS . . . 31
3.3 Comparison . . . 32

4 Implementation (Summary) **33**
4.1 Quality measurements to translate Matlab model to C . . . 33
4.2 Profiling of the algorithm . . . 37
4.3 OpenCL error handling . . . 37

5 Results (Summary) **39**
5.1 Performance measurements 39
5.2 Design choices result and evaluation 42
5.3 Accuracy tradeoffs 43
5.4 Real-time capabilities 43

6 Discussion **45**
6.1 Quantitative evaluation 45
6.2 Research Question 45

7 Conclusion & Future work **47**

Bibliography **50**

A Appendix **51**
A.1 OpenCL error codes 51

List of Figures

1.1 First medical X-ray by Wilhelm Röntgen [1] . . . 4
1.2 Philips AlluraClarity Intervational System . . . 5
1.3 Brief Introduction of an Xray Machine . . . 5
1.4 Coolidge side-window tube (scheme) . . . 6
1.5 Typical fixed-anode X-ray tube . . . 6
1.6 X-ray system penetrates hand tissue to provide information about the bones [2] . . . 6

2.1 Vendors that support OpenCL . . . 11
2.2 Example of 2D index space [3] . . . 13
2.3 OpenCL memory space [4] . . . 14
2.4 OpenCL system overview [4] . . . 14
2.5 Context overview [4] . . . 15
2.6 Command queue execution [4] . . . 16
2.7 Offline/Online compilation [5] . . . 16
2.8 AOS to SOA transformation [6] . . . 19
2.9 Intel GPU architecture [7] . . . 20
2.10 Group of sub-slices and memory scheme [8] . . . 21
2.11 CPU-GPU relation . . . 22
2.12 rVex schematic [9] . . . 23
2.13 rVex OpenCL overview [10] . . . 24
2.14 rVex PoCL [10] . . . 24

3.1 SDAccel architecture [11] . . . 30
3.2 Vivado HLS [12] . . . 31

4.1 Implementation Process from Matlab to OpenCL . . . 33
4.2 FAST Architecture . . . 35
4.3 Implementation Process from Matlab to OpenCL [13] . . . 35
4.4 FAST GUI demo [13] . . . 36

5.1 Achieved fps for sequential code and different accelerators . . . 40
5.2 Sequential code execution per filter block. . . . 41
5.3 Speedup of all OpenCL accelerators against the sequential version. . . . 42

List of Tables

2.1 Compilation method comparison 17

5.1 Summary of hardware platforms used for experiments 39
5.2 Speedup Comparison between different accelerators and the sequential version 41
5.3 Speedup Comparison between different accelerators and the sequential floating point version 42
5.4 Accelerator comparison between different image sizes and fps 44

A.1 OpenCL Error Codes [14] 56

List of Acronyms

ALMARVI Algorithms, Design Methods, and Many-core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing

API Application Program Interface

ASCII American Standard Code for Information Interchange

AVX Advanced Vector Extensions

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

FAST Fixed-point Analysis & Scaler Tool

FPGA Field-Programmable Gate Array

FPS Frames Per Second

FPU Floating Point Unit

GPU Graphics Processing Unit

GUI Graphical User Interface

HLS High-Level Synthesis

IP Intellectual property

iXR Interventional X-Ray

LLVM Low Level Virtual Machine

OpenCL Open Computing Language

OS Operating System

PC Personal Computer

PCI-E Peripheral Component Interconnect Express

RTL Register-Transfer Level

SIMD Single Instruction Multiple Data

SKMD Single Kernel on Multiple Devices

SM Streaming Multiprocessor

SPIR Standard Portable Intermediate Representation

SSE Streaming SIMD Extensions

Vex VLIW Example

Acknowledgements

This thesis denotes the end of two unbelievable years in TU Delft. On this page I would like to thank everyone who contributed to accomplish this work. First, I would like to thank my supervisor Zaid Al-Ars for his enthusiasm, his insight and our great collaboration. One could not have imagined having a better mentor. Furthermore, I would like to thank my colleague, fellow student but most importantly friend Bas Metman for his companionship during our trip to Best, our insightful conversations, his help and his support. Moreover, I would like to express my gratitude to Rob de Jong from Philips Healthcare for his supervision, his brilliant ideas and guidance and of course our great communication. In addition, I would like to thank Steven van der Vlugt from Topic for his wholehearted support, his assistance during the project and our great conversations during office/lunch hours. Last but not least, I would like to thank Rachana Arunkumar for sharing her ideas and expertise and Bart van Rijnsoever for supporting me actively.

I would like to thank also my friends A. Bougioukos, G. Katsaounis and L. Roussos for their solid and direct support during my thesis and being there everyday for me during our studies. Of course one can not forget: A. Kalatha, A. Kartsiou, G. Sarantis, M. Theodoridis, G. Vranas, G. Petridis for their companionship in the library of TU Delft and late night working. I would like to thank also my friends who brought comfort to me despite the distance between us: P. Kostara, T. Kitsakis, A. Kapodistria and N. Sketopoulos. In addition to my friends, I would also like to thank my amazing girlfriend Angeliki for being enthusiastic and supportive during my thesis and for not complaining for my late night working on some source code or this report.

Moreover, I would like to thank my family. My father Christodoulos for his dearest and honest advice about life, my mother Maria for always seeing the bright side on all issues and the two of them for helping me accomplish my dreams and goals with personal sacrifices. Finally, I would like to thank my brother Dimitrios for believing in me and who convinced me to travel to Delft for this amazing journey.

Panagiotis Mitsis
Delft, The Netherlands
October 13, 2016

1 Introduction

Medical imaging is a very crucial technique used by doctors worldwide in order to create virtual representations of the interior of a body. One well-known medical imaging technique is X-rays imaging. It is a form of electromagnetic radiation which allows the interior representation of a body. To create the image, a heterogeneous beam of X-rays is produced by a generator and is projected toward the object. A certain amount of X-ray is absorbed by the object, which is dependent on the particular density and composition of that object. The X-rays that pass through the object are captured behind the object by a detector which can then provide a superimposed 2D representation of all the object's internal structures. However, through to limitations of the system an amount of noise interferes with the image. To solve this, Philips Healthcare uses a noise reduction algorithm which was invented in the R&D department of the company that specializes on medical imaging. In order to provide high quality though, the algorithm uses several filters, which results in increasing the required computational resources of the processing platform.

This thesis describes the implementation of a medical image processing algorithm in order to allow portability and performance. In other words we investigate the effect of running the algorithm on a couple of processing platforms and investigate the resulting performance we can get. We do that using the OpenCL framework. Our goal is to investigate if there are any accelerators that can fulfill implicitly defined real-time constraints. Those constraints are based on the way that the human eye can process a video and the maximum latency that will allow a physician to operate safely on a patient. This chapter provides an introduction to the problem definition. It starts with the context in which this work originated. Subsequently, the motivation for a real-time solution will be discussed. Then the problem statement that is researched in the thesis is presented. After that, the methodology to analyze the problem statement and the goals of the thesis are discussed. Finally, the outline of the rest of the work is given.

1.1 Context

1.1.1 Philips Healthcare

Koninklijke Philips N.V. (Royal Philips, commonly known as Philips) is a Dutch technology company with primary divisions focused in the areas of electronics, healthcare and lighting. It was founded in 1891 by Gerald Philips and his father Frederik in Eindhoven. One of the most famous subdivision of Philips is Philips Medical Systems or commonly known as Philips Healthcare. Philips Healthcare vision is to improve the life of people with a series of medical products including:

Clinical informatics:

- Cardiology informatics (IntelliSpace Cardiovascular, Xcelera.
- Enterprise Imaging Informatics (IntelliSpace PACS, XIRIS).
- IntelliSpace family of solutions.

Imaging systems:

- Cardio/Vascular X-ray.
- Computed tomography (CT).
- Fluoroscopy.
- Magnetic resonance imaging (MRI).
- Mammography.
- Mobile C-Arms.
- Nuclear medicine.
- PET (Positron emission tomography).
- PET/CT.
- Radiography.
- Radiation oncology Systemsroots.
- Ultrasound.

Diagnostic monitoring:

- Diagnostic ECG.

During my internship I had the honor to work in the iXR team in Philips Healthcare at Best. This department aim is to improve patient lives by focusing in two areas:

- Maintain the best quality of machines, just like the logo Philips means.
- Improve and pioneer the X-ray machines of the future.

1.1.2 Almarvi

Advanced image and video systems are becoming a crucial and resource consuming part of embedded applications in many sectors. That's why the ALMARVI [15] project was created (Algorithms, Design Methods, and Many-core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing). The partners of the project aim on reducing overall system design cost and time-to-market and enabling low cost solutions for high volume markets in different industrial domains and creating new market opportunities, and supporting SMEs. Such systems can be used for healthcare, security/surveillance/monitoring, and mobile use cases. This particular thesis was pursued by two partners of this project **Philips Healthcare** and **Delft University of Technology**.

The motivation of our project is to develop a high performance real-time solution to process an X-ray image which is distorted by noise during the image sensing process. During the runtime of the implementation, a highly sophisticated image algorithm developed by Philips Healthcare processes the whole image to reduce the noise and sharp the edges. The techniques used have as a result the output to be very clear for a physician. It is very important for the system to be responsive and to provide constant performance that meets the real-time requirements because otherwise it will provide wrong output for the physician which might cause severe or fatal injuries to the patient.

In order to facilitate a solution that meets the requirements of portability and performance, the OpenCL standard was chosen for this project. This standard provides an abstract layer of describing the computational part of an algorithm and then can be transfer to different heterogeneous platforms as it is supported by many different vendors, as well as a soft-core processor made in TU Delft. Making various experiments allowed us to have an overview on many aspects that affect a project like:

- Developing time.
- Portability.
- Scalability.
- Development costs.

These results were very useful for ALMARVI as we showed the means on how to create a medical image processing system which support scalability and portable performance. In our evaluation we conclude as well in which devices are most suitable to develop a high performance system and we proved that performance is constant for the same machine but it varies depending on the accelerator characteristics (better accelerator, better performance and vice versa).

1.2 Background

X-radiation is a form of electromagnetic radiation discovered by Wilhelm Conrad Röntgen in November 1895. After a couple of days of testing he observed that using X-rays, one can look not only into metallic articles but also inside of a human body. His first medical X-ray was of the hand of his wife Anna Bertha Ludwig shown in figure 1.1.

Figure 1.1: First medical X-ray by Wilhelm Röntgen [1]

The industrial use of X-rays began in Germany only two decades after their discovery with the help of radiographic films. Radioscopy with fluorescent screens was developed at the beginning of the 1940s, followed by the introduction of closed cabinets for testing X-rays.

1.2.1 Xray system description

Nowadays X-ray systems are used to help physicians operate on a patient, allowing him to look inside the human body in real-time. For example the Philips AlluraClarity system which is shown in figure 1.2.

Figure 1.2: Philips AlluraClarity Intervational System

But how such a system operates? In brief the process is shown in figure 1.3.

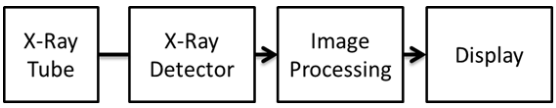


Figure 1.3: Brief Introduction of an Xray Machine

To begin with, as stated X-rays is a type of electromagnetic radiation. Their wavelength is ranging between 0.01 and 10 nanometers and depends on the photon energy from the source. There are two types of X-rays based on their wavelength:

1. Soft X-rays: With wavelength above 0.02nm.

2. Hard X-ray: With wavelength in the range of [0.01, 0.02] nm.

Hard X-rays are the most commonly used with varying applications from crystallography to medical imaging and airport security because they can penetrate the skin. The photon energy of the X-rays can be changed which means that by tweaking this X-ray will change penetration depth. This provides a means to view different layers in the same object like bones, veins, etc.

In order to create such radiation, one needs an X-ray tube. There are different types of X-ray tubes (sources) but the most widely used is the Coolidge tube [16]. The Coolidge tube uses vacuum and high voltage to create X-rays: A tungsten filament is heated. That frees electrons from the metal (but at almost zero energy and thus zero speed) and then the electrons are accelerated using high voltage. As a result, the hotter the filament gets, the greater the emission of electrons. The maximum energy (in electronVolt) of the X-rays is virtually equal to the voltage applied between cathode and anode. Such a water cooled tube scheme is shown in figure 1.4 and a typical tube in figure 1.5.

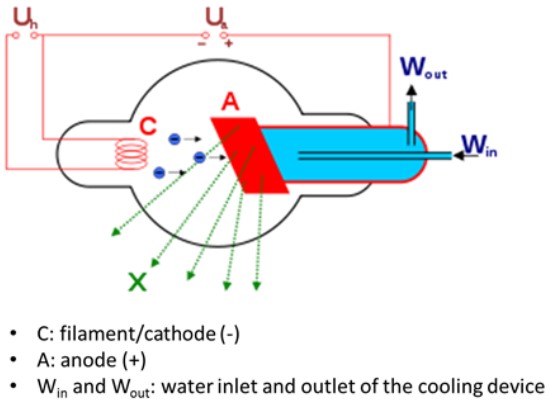


Figure 1.4: Coolidge side-window tube (scheme)

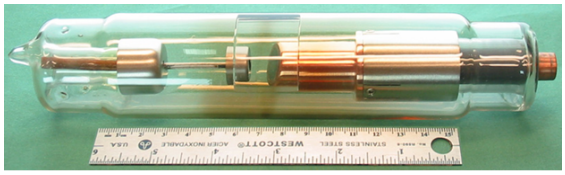


Figure 1.5: Typical fixed-anode X-ray tube

Except of the tube that generates the X-rays, we need also a detector in order to provide medical imaging. That detector facilitate the imaging and the dose measurement. Today, digital detectors such as image plates or flat panel detectors are mostly used. They are placed below the patient as shown in figure 1.6.

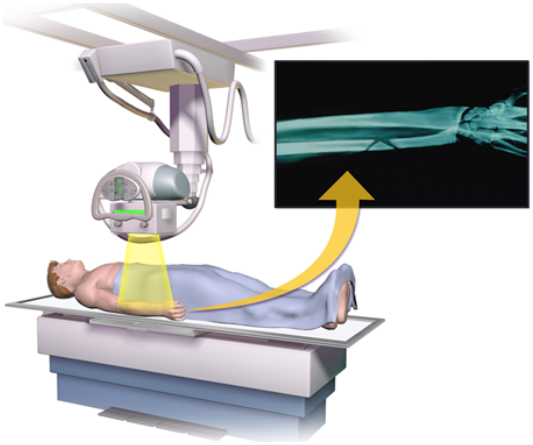


Figure 1.6: X-ray system penetrates hand tissue to provide information about the bones [2]

So far, we have discussed the tube that generate the X-ray, the detector that is placed underneath the object to be scanned (e.g., the patient), which is placed between the tube

and the detector. But the question still remains: how is the virtual representation of the inside of the human body produced? When the X-ray photons come out of the X-ray generator, they will travel freely until they come in touch with matter. When this happens, there are 3 possibilities:

- It penetrates and simply goes through without any path change.
- It can be completely absorbed and give all its energy to the matter (photoelectric interaction).
- It can deposit part of its energy and be scattered (change in trajectory).

In the human body there are several types of tissue, some more dense than others. Dense tissues (for example bones) absorb more X-rays than less dense tissues. That is why it is possible to create an image.

1.2.2 Image Processing

After the image is received to the image detector, it is far from ideal. There are many steps that need to be done before the output can be presented to the final intended user. To begin with, it is a common effect that the captured image contains defect pixels or lines. For that reason, Philips has developed an algorithm that is fitted on a device to clean the image from such defections. After the cleaning algorithm there is a noise reduction algorithm. This algorithm has to remove the noise from the cleaned image and in addition to sharpen the edges to make them as clear as possible for the physician. Finally, when the image is noise reduced, it is represented on a display in real-time and captured as well for future playback by the physician. It is worth noticing that all the processing of one image needs to be completed in a couple of milliseconds, otherwise the physician will not have the necessary information to operate safely on the patient.

1.3 Problem definition

Philips Healthcare has produced a denoising algorithm which supports image resolution of 960x960 pixels. The minimum amount of frames per second that should be processed and shown to the physician is 30fps and ideally 60 fps. As a result of those requirements each frame should be processed in a maximum delay of 33.33 ms or 16.67 ms respectively. Moreover there is a quality constraint that the relative error from the matlab model and our implementation should not be above 0.5%.

That means that there is a need to provide a high performance (as fast as possible and quality reliable) cross-platform solution for the support of a real-time system with the constraints above. The most well-known framework for this is, is the Open Computing Language (OpenCL). Multiple vendors with different hardware architectures support OpenCL such as: Intel, AMD, Nvidia, Xilinx and Altera. Furthermore, there is support for the rVex processor (developed by the TUDelft) to run OpenCL based on an FPGA platform.

The aim of the project is to define if there is a high performance solution which is portable and allows scalability without increasing by far the development time of implementing the algorithms in software and porting them to alternative platforms. In other words the research question of this thesis is:

Are there any means to develop a real-time constrained implementation of a medical image processing algorithm, which is portable among different hardware accelerators by means of performance and scalability, allows low development costs and is easily maintainable?
For the specific algorithms being discussed within the context of this thesis, our goals are to:

1. Implement the X-ray filtering algorithms used by Philips in a portable software language that runs transparently on alternative hardware platforms
2. Use accelerated hardware solutions to achieve an image processing latency as low as 33.33 ms per frame for 960x960 images
3. Use accelerated hardware solutions to achieve a video processing throughput at 30-60 fps for 960x960 resolution videos

In order to answer this several steps need to be done:

1. Detect parallelism both in inner functions but also in the filter model itself (if tasks can be processed in parallel except data parallelism).
2. Convert the Matlab model into C code and get performance measurements in order to detect the time consuming phases of the filter.
3. Convert C code into OpenCL code and detect stalls for each platform.
4. Redesign the code to match all the hardware platforms without affecting their performance (for example the use of fixed point arithmetic instead of floats or doubles).
5. Quality measurements in steps 2-4 are required to ensure that the quality constraint is ensures.

After those steps and after deploying the code on different hardware accelerators a comparison must be done in order to compare the development cost efforts, the scalability and the final performance that someone can get and if it is suitable enough for our test case.

1.4 Thesis Outline

The rest of the thesis is outlined as follows. In chapter 2, the background knowledge regarding the OpenCL framework and the different hardware platforms used is provided. In chapter 3, an evaluation of the alternative solutions that could facilitate the project

exist comparing both hardware and software solutions. Then in chapter 4, a summary of implementing the noise reduction algorithm using OpenCL will be provided. In chapter 5, a summary of the results of various hardware architectures running our OpenCL version is outlined. Moreover, in chapter 6 a discussion on these results based on the research question is provided. Finally, in chapter 7 the conclusions of this research work is given and some interesting points to investigate in the future.

Background 2

This chapter will introduce the concepts and pitfalls of OpenCL. In addition a small introduction for all the hardware architectures that were investigated is presented, alongside their advantages that were taken into consideration for the final implementation. At the end of this chapter a small discussion is taking place for the development setup that was used in order to complete this thesis.

2.1 OpenCL

Open Computing Language (OpenCL) is an independent framework for describing programs for different heterogeneous platforms. Such platforms are central processing units (CPUs), Graphics processing units (GPUs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL is an open standard maintained by a non-profit consortium called Khronos Group [4]. This standard specifies application programming interfaces (APIs) to control the platform and execute programs (kernels) based on the C99 version of the C programming language standard. OpenCL provides a standard interface for parallel computing using task-based and data-based parallelism. A list of all the major vendors that support OpenCL is shown in figure 2.1. It is worth noticing that OpenCL is supported by the softcore produced in TU Delft r-Vex.

Figure 2.1: Vendors that support OpenCL

The OpenCL standard divides the execution between the host (driver) program and the OpenCL kernels that will run on the accelerator. The host program runs on the host CPU and is responsible via the OpenCL API to handle the hardware platform. It is responsible for initializing the accelerator, compile/load a binary with the kernels for the device, and setup the memory buffers. In addition, it is the one that control the memory transfers from/to the device and control the execution of the OpenCL kernels.

An OpenCL kernel code is a C99 code that describes the computational part that the programmer wants to implement via the hardware accelerator. It is the code that actually runs on the different hardware platforms, in order to solve a recurring data-parallel problem. As of April 2016, the new OpenCL standard 2.2 introduce the use of C++ for OpenCL kernel development, but none of the current vendors support it currently. That means that the whole algorithm was described in C99 with some drawbacks (some libraries which were in C++ had to be translated in C).

How does exactly an OpenCL kernel work? Imagine the researchers of a university. Each of the faculties has different research groups. Each individual researcher within a research group has a specific segment to research but the output of his research is combined with the output of his fellow researchers to have a result for the research group. The product of all the research groups represents the research of each faculty. Different research is conducted from different faculties but all of them conduct the academic research of the university. OpenCL follows the same model. Different OpenCL kernels (faculties) are combined to produce a final product (academic research of the university). Inside each kernel different work-groups (research groups) are working individually to provide the output (academic research of a faculty). Finally, each work group has individual work-items (researches) that cooperate together in order to produce a part of the output.

Internally this is more complex than the example above. Most of the hardware accelerators have many small processing units which can run in parallel. For this reason the standard splits up the work in multiple index spaces as introduce in the example and can be shown in figure 2.2. Each work-group is assigned to different processing units and run in parallel. The programmer from the host program specifies the global size in both axes (Gy,Gx) and then defines the work-group size in both groups. It is worth mentioning that the work-group size (in total wx*wy) should also be supported from the accelerator (e.g. Nvidia GPUs supports up to 1024 work-items/work-group and on Intel standard GPUs 512 work-items/work-group). The amount of work-size per dimension will define the number of work-items in each dimension.

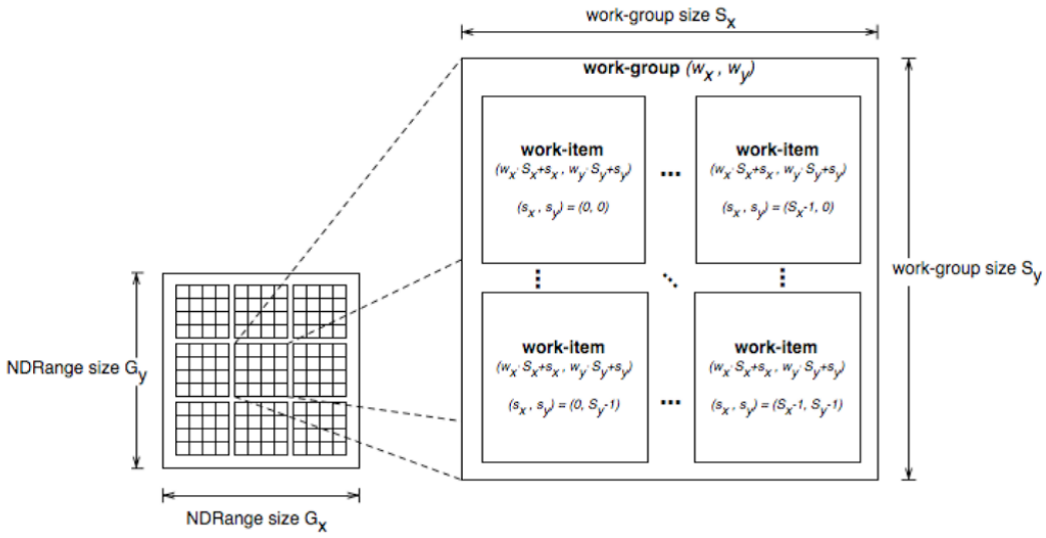


Figure 2.2: Example of 2D index space [3]

A work item is the smallest processing unit which executes the kernel code. Work items are identified with an index (x,y) using the get_global_id(_index_) function with arguments 0/1 depending on the axis that we want to gain information. The amount of work groups per dimension is calculated by:

$$Work\_groups\_per\_dim = \lceil \frac{Gy}{wy} \rceil \tag{2.1}$$

The memory spaces in OpenCL are distinguished in four different types (figure 2.3) ordered by speed:

1. Private memory is space that is work-item oriented, which means that only the work-item is allowed to access that memory and it is non-accessible from other work-items. It usually is implemented using registers.

2. Local memory is a small memory that the work-items of a work-group can access but it is non-accessible from work-items of other work-groups. It usually is implemented as a small cache programmable by the programmer.

3. Constant memory is accessible from all the work-items of work-groups, it is initialized from the host and it is a read-only memory that can't change over the run-time from the work-items. Usually it is implemented as a small area with low latency.

4. Global memory is the one that is used as a storage memory. It provides data from/to host and has the slowest access rate. Data inside global memory can change during the run-time and is accessible from all the work-items of each work-group.

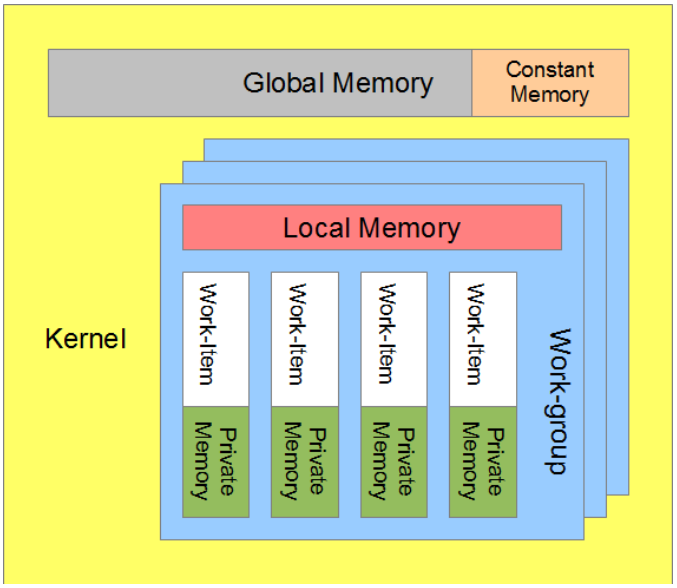


Figure 2.3: OpenCL memory space [4]

After the declaration of how a kernel works lets now focus on how the framework works. A system overview of OpenCL is presented in figure 2.4.

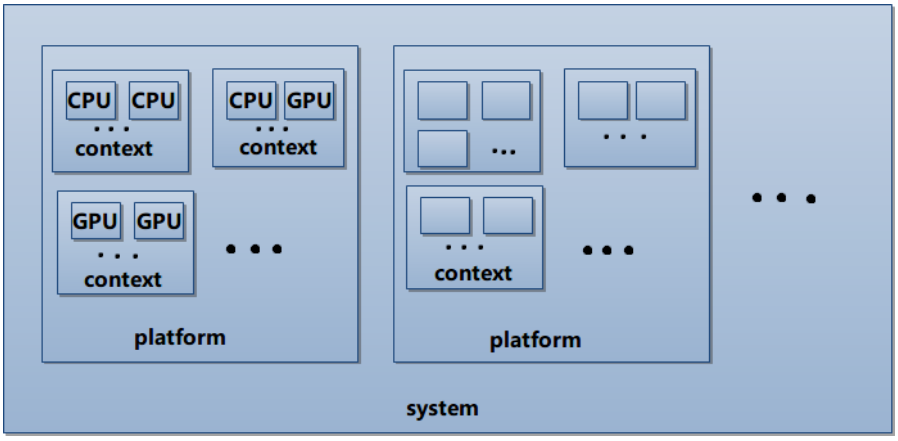


Figure 2.4: OpenCL system overview [4]

Each system can have multiple platforms that support OpenCL. In the initialization state the programmer must clarify from the host code on which platform the code will be executed. Then for each platform the context has to be initialized. The Context includes the available platform devices, the available devices, the command queues and the memory for that platform as shown in figure 2.5. For all the devices that are used in the execution there has to be one command queue describing the tasks that the device has to execute. Yet, in between two devices of the same context, memory objects can be shared and allow an SKMD hybrid implementation.

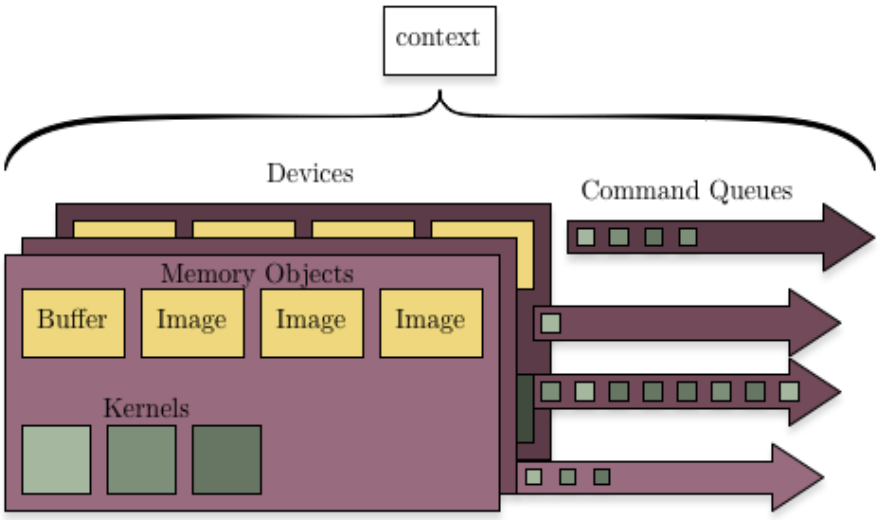


Figure 2.5: Context overview [4]

The command queue is the basic element of the device that determines which task has to run on the accelerator. It is managed from the host code and it can transfer data between the host and the device using the **clEnqueueWriteBuffer/clEnqueueReadBuffer**. Moreover, it executes kernels using the clEnqueueNDRangeKernel with the appropriate arguments. The priority of each queued task is determined by the defined task order when the command queue is initialized. The default order is the in-order queue which means that the execution is first task come first served as shown in figure 2.6. Out-of-order command queues may exist where the synchronization between different kernels is being handled from the OpenCL events objects (clEvent).

In addition the command queue provides timing information when initialized with the flag **CL_QUEUE_PROFILING_ENABLE**. If the flag is enabled, the execution time of each kernel can be retrieved if that kernel has an event object. In general, the event objects provide the following:

- Synchronization among different kernels (when out-of-order queue).
- A time-stamp when the command was queued on the command queue by the host.
- A time-stamp when the command was queued on the command queue by the device.
- A time-stamp when the command started execution.
- A time-stamp when the execution finished.

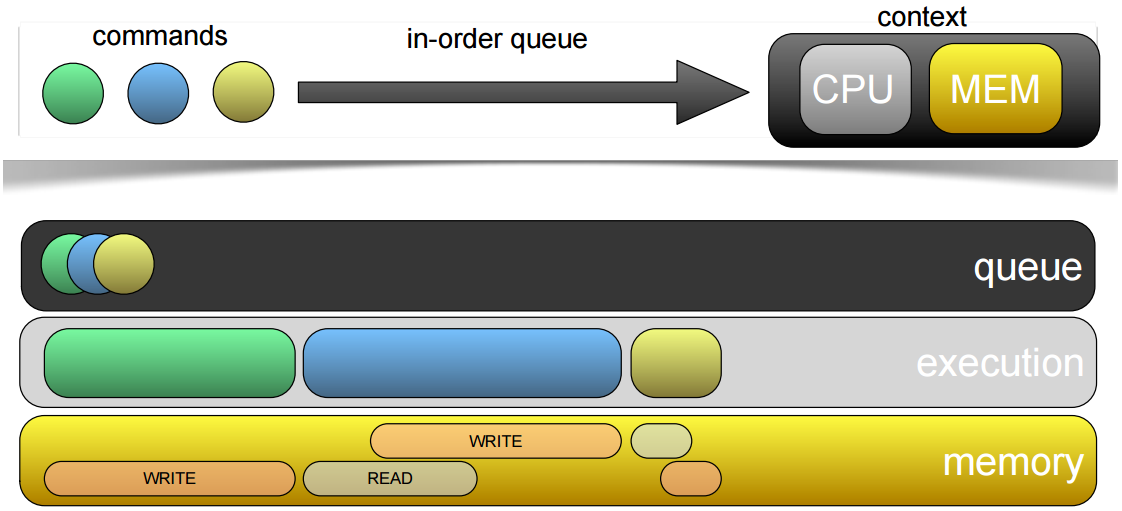


Figure 2.6: Command queue execution [4]

In the end of this small introduction about OpenCL we introduce also the different compilations method of the OpenCL kernels. In OpenCL, a kernel can be compiled either online or offline (figure 2.7).

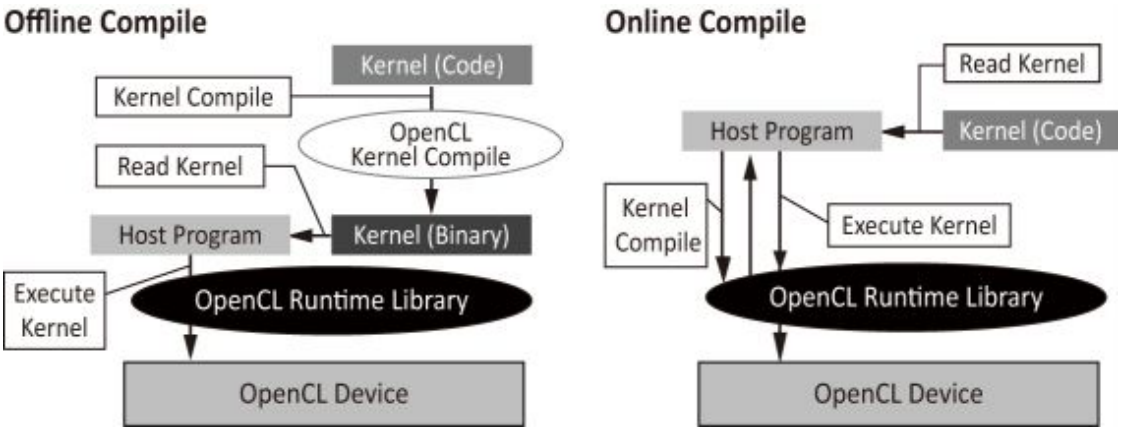


Figure 2.7: Offline/Online compilation [5]

The basic difference between the 2 methods is than during the online compilation the host code has to read the OpenCL kernel source code as a string and during the offline compilation as a binary. In a commercial product it is not allowed to have the source code available, as it is IP of the company. That's why an exploration in the binary compilation was conducted.

There are two types of binaries available the device specific and SPIR(Standard Portable Intermediate Representation). The device specific binary is limited to only the vendor code provided by the offline compiler and the specific device, while the SPIR

binary can be used with all the vendors/devices (that support it). In terms of portability, it is the best option but the portability does not come without a cost. The developer has to sacrifice performance in order to use it. This happens because SPIR contains an Intermediate representation and it is not device optimized. During our tests we noticed that with the same OpenCL kernels and workload when we compared the native binary with SPIR there was an increase from 50ms per frame to 70ms. This means that the design using SPIR became 40% slower on the same device. In table 2.1 we have a comparison table with the advantages and disadvantages of all the compilation methods.

Compilation Type	Pros	Cons
Online	On the fly will all vendors	How to embed the source code.
Offline- Native	Can be distributed across devices of the same family e.g. CPUs (i7 - Xeon)	• Different binary for each vendor. • It needs the same driver version.
Offline- SPIR	Can be distributed for all vendors/devices.	• Still in early stage. • Not all vendors support it yet. • Performance issues.

Table 2.1: Compilation method comparison

2.2 Architectures

In this section a description about the hardware platforms chosen will be presented. The initial goal of the project was to run the code on the rVex processor and develop it on a device with an Intel CPU, to ease the development effort. However, due to some implications with the PoCL framework provided by the University of Turku the final platform list changed. A discussion though for that processor is taking place because the code is designed to support it in the future as soon as the PoCL framework is available. In the end of the project the implementation was tested on an Intel i7, Intel Xeon and an Intel integrated GPU.

2.2.1 Platform Setup

The development setup for this thesis was an HP Zbook 15 workstation running windows 7. It has three different hardware devices that can run OpenCL:

- Intel i7-4810 MQ
- Intel HD Graphics 4600
- Nvidia Quadro K1100M

The initial tools that someone needs in order to develop the implementation are Microsoft Visual Studio and the Intel OpenCL SDK. For the purpose of our project we used the API specified for OpenCL v1.2. The reason for that is that in the beginning of the project it was decided to support:

- PoCL on rVex (OpenCL v1.2).
- Intel devices (OpenCL v1.2).

Due to the fact that we use some API calls introduced in OpenCL v 1.2 we were unable to run the same code on an Nvidia GPU which only supported OpenCL v1.1 (at the development environment introduced).

2.2.2 Intel CPU

Intel Architecture Processors provide performance acceleration using Single Instruction Multiple Data (SIMD) instruction sets, which include [17]:

- Intel Streaming SIMD Extensions (Intel SSE).
- Intel Advanced Vector Extensions (Intel AVX) instructions.
- Intel Advanced Vector Extensions 2 (Intel AVX2) instructions (Most integer commands to 256 bits and introduces FMA compared to AVX).

Using SIMD instructions the processor can achieve data parallelism. The task of creating the parallel code is upon the compiler with the implicit vectorization module. It uses internally an LLVM intermediate representation in order to create the vectorized code. This module transforms scalar data type operations by adjacent work-items into an equivalent vector operation [17]. When vector operations already exist in the kernel source code, the module scalarizes (breaks them down into component operations) and revectorizes them. This improves performance by transforming the memory access pattern of the kernel into a structure of arrays (SOA), which is often more cache-friendly than an array of structures (AOS) shown in figure 2.8.

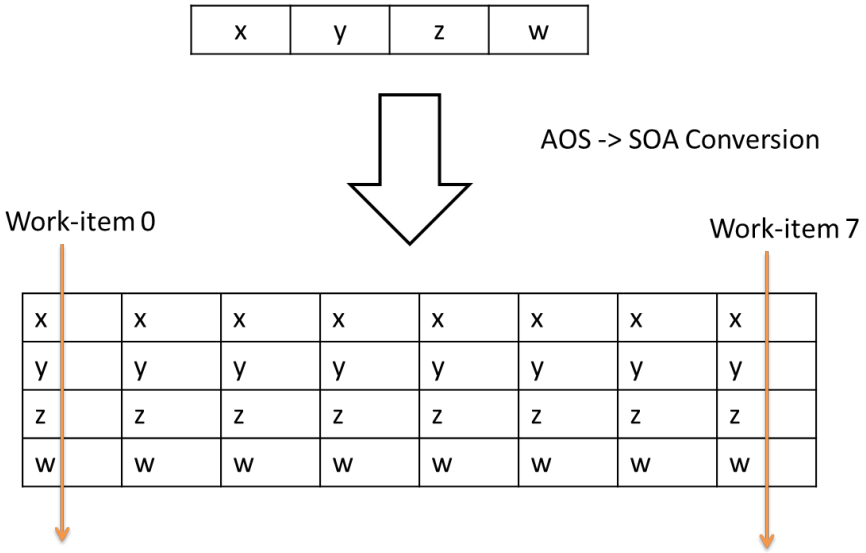


Figure 2.8: AOS to SOA transformation [6]

In order to get best performance from using the vectorization module, the work-group size must be larger or a multiple of 8. To reduce the overhead of maintaining a workgroup, work-groups should be as large as possible, which means 64 and more work-items. One upper bound is the size of the accessed data set, as it is better not to exceed the size of the L1 cache in a single work group. Since work-groups are independent, they can execute concurrently on different hardware threads. So the number of work-groups should be not less than the number of logical cores. A larger number of work-groups results in more flexibility in scheduling, at the cost of task-switching overhead. To recapitulate with, Intel CPU devices allow data parallel processing by executing multiple kernel work-items on different SIMD lanes together, while running multiple work-groups up to the number of the logical cores of the processor concurrently.

2.2.3 Intel GPU

The Intel integrated GPU is equipped with several Execution Units (EUs), each one of them is a multi-threaded SIMD processor. The SIMD width is a heuristic provided to the compiler and if all the threads execute the same instruction the SIMD lanes can be maximally utilized. In case of a branch it could be a stall to the device in order to calculate the different paths, but it also provides a branch prediction unit. An overview of the GPU architecture alongside with the Thread dispatcher is shown in figure 2.9.

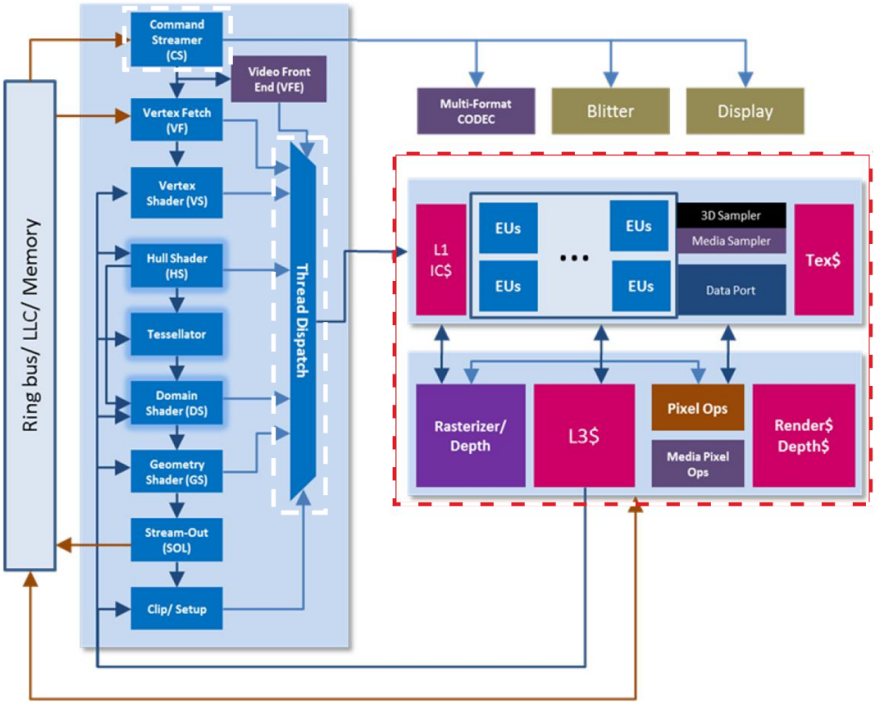


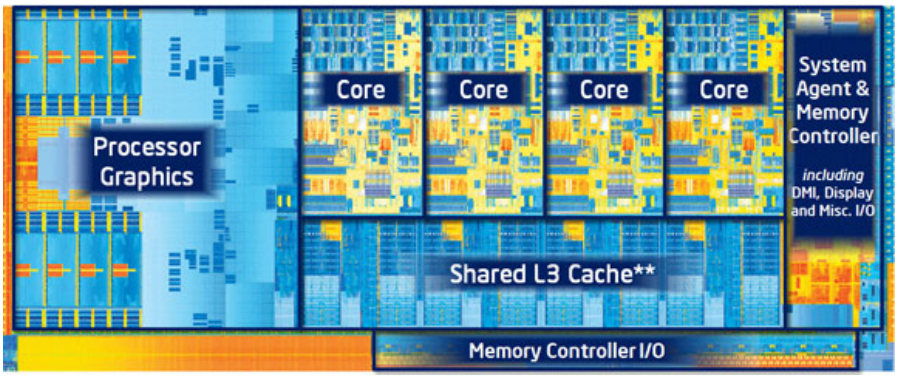
Figure 2.9: Intel GPU architecture [7]

The building block of the architecture is the EUs. Each one is an SMT (Simultaneous Multi-Threading) compute engine that executes SIMD instructions. The highly threaded nature of these blocks ensures continues streams of ready-to-execute instructions which can hide latency of memory operations [18]. A group of such blocks constitute a sub-slice (figure 2.10) which share:

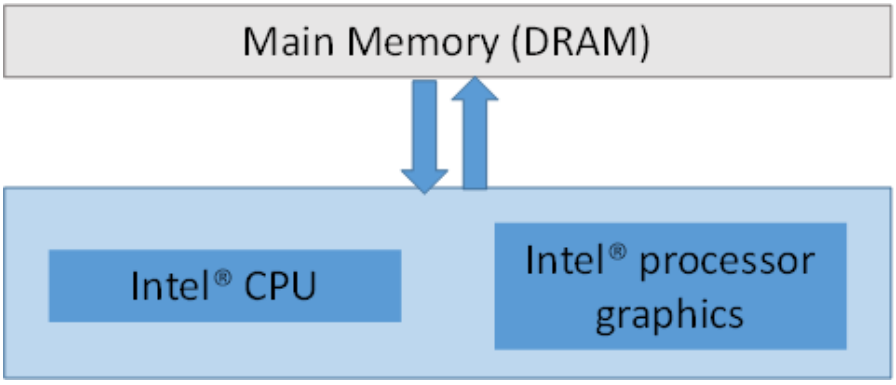
- Texture L1 and L2 caches, which are the path for accessing OpenCL images.
- Data port (L3 cache), which is the path for OpenCL buffers.
- Other hardware blocks like instruction cache.

Figure 2.10: Group of sub-slices and memory scheme [8]

It is worth mentioning that during our implementation we used OpenCL buffers instead of OpenCL Images as they are not supported by PoCL on rVex. Moreover, this GPU architecture has a benefit compared to the PCI-E GPUs. Intel GPUs shares the same uniform memory with the CPUs (figure 2.11), which mean that we can have zero Copy buffers and minimize the transfer time between the host and the device.



(a) Die overview[8]



(b) Schematic of memory between CPU/GPU [8]

Figure 2.11: CPU-GPU relation

Finally, as they are using the same memory for buffers they don't only allow zero copying but also they can work concurrently (SKMD [19]). As there is no need for data transfers someone can use the same OpenCL kernel code and enqueue different workloads both on a CPU and GPU concurrently as they can run under the same Context. In that way concurrent processing can be achieved resulting faster results especially for real-time applications.

2.2.4 rVex

rVex is a VLIW softcore developed by TU Delft with roots to the VEX ISA from HP. In VLIW architecture, a processor executes multiple instructions in parallel. Which instructions are executed in parallel is decided during compilation by the compiler. The group of instructions that get executed in a cycle is called an instruction bundle. In that way a faster execution can be achieved. However not all instructions can be executed in the same cycle due to architectural restrictions or data dependencies.

By default, a VEX cluster has 4 ALU units, 2 multiplier units, 1 branch control unit and 1 memory access unit per cluster as shown in figure 2.12. An example architectural restriction than may prevent instruction on executing in the same cycle, is when there

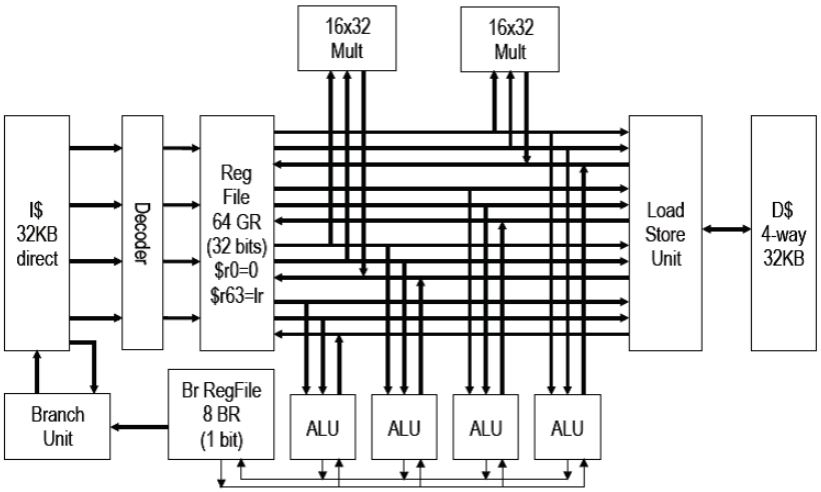


Figure 2.12: rVex schematic [9]

are a lot of instructions that need access to a multiplier unit. This happens because we have only 2 multiplier units and they are not enough to execute more instructions concurrently.

Data dependencies occur due to the source code of the program and they are not relating to the architecture. Three cases of data dependencies might occur:

- Flow (data) dependence: read-after-write (RAW).
- Anti-dependence: write-after-read (WAR).
- Output dependence: write-after-write (WAW).

An example of such dependency (RAW) that is not resolved is shown in the pseudo code below:

```
A = 3
B = A
C = B
```

Code segment 2.1: Data dependency example

In past years, rVex was evolved in order to support OpenCL. In order to achieve this goal a system was designed as shown in figure 2.13. The selected approach is based on a Linux PC and a connected FPGA on it via the PCI-E bus for data communication.

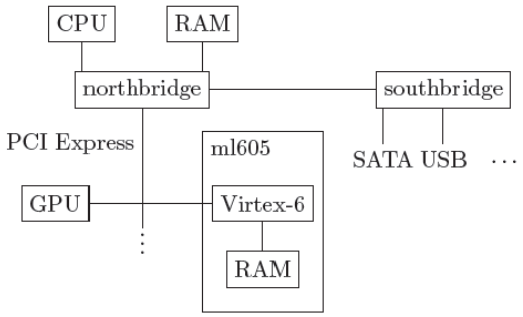


Figure 2.13: rVex OpenCL overview [10]

Then the PoCL framework is taking care of the OpenCL code both for host and device. The PoCL OpenCL implementation is divided into a host layer and a device layer (figure 2.14). The host layer encompasses all the device independent functions required for an OpenCL implementation. This includes an OpenCL C language compiler and generic optimization passes. The device layer contains all device specific functionality, and can be seen as a hardware abstraction layer. Pocl uses the Clang compiler front end to compile the OpenCL C language. This compiler generates LLVM Intermediate Representation (IR) instead of machine code. This IR is architecture-independent, allowing reuse of compiler front ends and optimization passes. Then a back-end translates this IR to VEX specific assembly code.

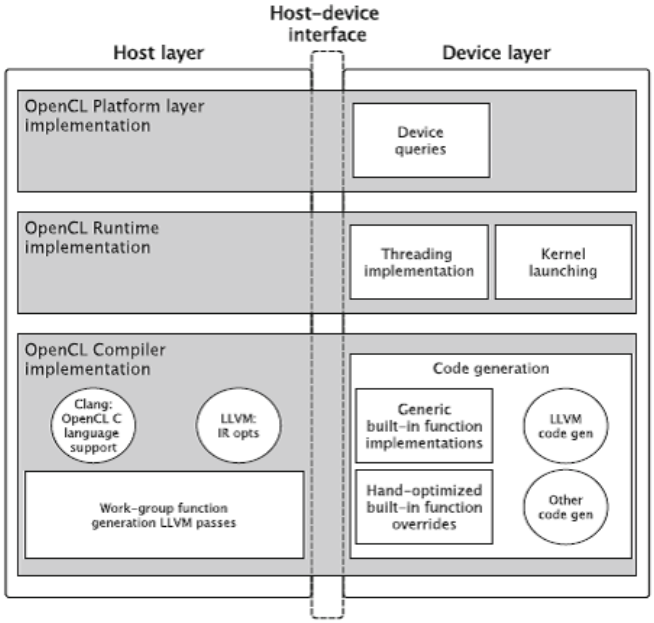


Figure 2.14: rVex PoCL [10]

The device layer contains several components that need to be implemented to add a new device target. To determine how many available devices the OpenCL implementation can access, the layer contains a device query component. The device layer is also responsible for the generation of the machine code to execute on the device. This machine code should be generated from the IR that the host layer delivers, combined with the built-in OpenCL C language functions. Finally, the device layer has to be able to manage the device, implementing data transfer, setup and execution control.

The main reason that rVex was selected for this project is that it can run on an FPGA. Such systems have shown that they can guarantee a lifetime of approximately 15 years [20]. This is very important for X-ray machines that may be in the market for 20 years (time since first release till maintenance is not more guaranteed by the manufacter).

3 Alternative solutions

Before implementing the noise reduction algorithm using the OpenCL framework there was a research on what alternative solutions could be used instead. There were a couple of alternatives available when the project began and also one solution was added which is still in progress. Two types of alternative solutions were investigated: software and hardware solutions. As software solutions we define those that the developer needs to express the algorithm in a high level language, create an executable and run it on an accelerator. On the other hand the algorithm can also be expressed in high level language and create a Register-Transfer Level (RTL) design, which is what a hardware solution means. At the end of this chapter there is a comparison table of the advantages and disadvantages of each solution.

3.1 Software solutions

There are a couple of alternative software solutions available which could have been used. Those are provided by major vendors and cover a CPU solution and a GPU one.

3.1.1 CUDA

CUDA is a parallel programming model invented by NVIDIA. It exploits the power of the GPUs provided by NVIDIA and are connected to a PC via the PCI-E bus. In order to achieve high performance computing CUDA utilizes the Streaming Multiprocessors which are embedded in the GPU. Each SM contains its own L1 cache for faster accesses and all of them have an interconnection with the L2 cache. This level of cache stores all the data from the internal (global) memory of the device. There are a couple of limitations during the programming of such devices that the developer should take care of:

- Occupancy of the device: Limited by shared memory and the amount of registers of each CUDA kernel.
- Memory bandwidth: The bandwidth of reads/writes to and from the global memory is relative to the data-types and the data access patterns that the developer uses.
- PCI-E latency: The developer should try to overlap data-transfers with computations in order to keep the device busy and achieve better performance.

Although that CUDA supports C/C++ and Fortan which makes it relatively easy to develop CUDA code, it is difficult to maintain CUDA code. The reason to that is that the

code is architecture specific. That means that each new architecture model presented onto NVIDIA GPUs creates the need to redevelop the code as it is not performance portable. For example a developer might not utilize the shared memory due to low occupancy from large data allocation into the shared memory, and the next generation may allow increase data-sets into the shared memory. Then he would have to redevelop the code to achieve the desired efficiency.

3.1.2 SSE/AVX

SSE and AVX are extensions to the X86 Instruction set architectures for microprocessors from Intel and AMD. Those extensions allow data parallelism provided by SIMD instructions. There are three methods to implement SIMD programming:

1. Inline Assembly
2. Intrinsic Function
3. Vector Class

The methods are described from the fastest to the slowest. Both three methods are difficult to maintain as they are also architecture specific. The sample code below, perform a multiplication between two groups of 8 floats is provided using AVX. The same principle applies for SSE source code.

```
//define the registers used
__m256 vector0, vector1;
float a[8]={1,2,3,4,5,6,7,8};
float b[8]={2,3,4,5,6,7,8,9};
float c[8];
//load the 8 floats in a into vector0
vector0 = __builtin_ia32_loadups256(a);
//load the 8 floats in b into vector1
vector1 = __builtin_ia32_loadups256(b);

//multiply vector0 and vector1, store the result in vector0
vector0 = __builtin_ia32_mulps256(vector0, vector1);

//copy the 8 floats in vector0 to c
__builtin_ia32_storeups256(c, vector0);
```

Code segment 3.1: AVX Sample code

3.2 Hardware solutions

In this section we analyze different hardware solutions. One of them is still in developing phase (rVex on Zynq) and cover an embedded solution of this problem using OpenCL. In addition, we analyze two different hardware solutions provided from Xilinx. The reason

for that is that the other major vendor Altera follows the same concepts for products and were not chosen by Philips and TU Delft. However, it is worth noticing that Inggs et. al[21] states that the HLS tools from Altera and especially for OpenCL are well-suited to accelerating parallel-friendly algorithms, as parallelism can be made explicit fairly easily. In contrast based on his observations Xilinx's tools are suited more for small functional unit prototyping and will not provide optimal results if used by developers with insufficient hardware design experience.

3.2.1 rVex on Zynq

rVex was analyzed in detail in chapter 2. The new version of PoCL will allow rVex to use OpenCL directly on an FPGA board without the need of the host PC. This can be achieved using the Zynq device provided by Xilinx. Zynq is a feature-rich embedded software and digital development platform. It integrates a dual-core ARM A9 processor with Xilinx FPGA logic. In this case using the ARM processor someone can install an embedded OS and via PoCL to be able to utilize rVex as an accelerator. This is an advantage compared to the current implementation of rVex with PCI-E, as the latency to transfer data from/to the device will be decreased. It will be a performance portable design as the RTL design of rVex is independent of the FPGA (if it fits the rVex softcore). However, this is not ready to be tested yet. Another drawback is that the current version of rVex runs on 100 MHz which means that the processing speed will not be enough for a real-time application of that scale.

3.2.2 SDAccel

SDAccel is a tool provided by Xilinx and allows CPU/GPU like programming for FPGAs. The architecturally Optimizing Compiler for OpenCL, C, and C++ shipped by Xilinx allows:

- Up to 25X better performance/watt compared to CPU/GPU.
- 3X the performance and resource efficiency of other FPGA solutions.
- Optimize applications on FPGA platforms with little to no FPGA experience.
- Easily migrate applications to FPGAs while maintaining and reusing OpenCL, C and C++ code.

The schematic of such a system is shown in figure 3.1.

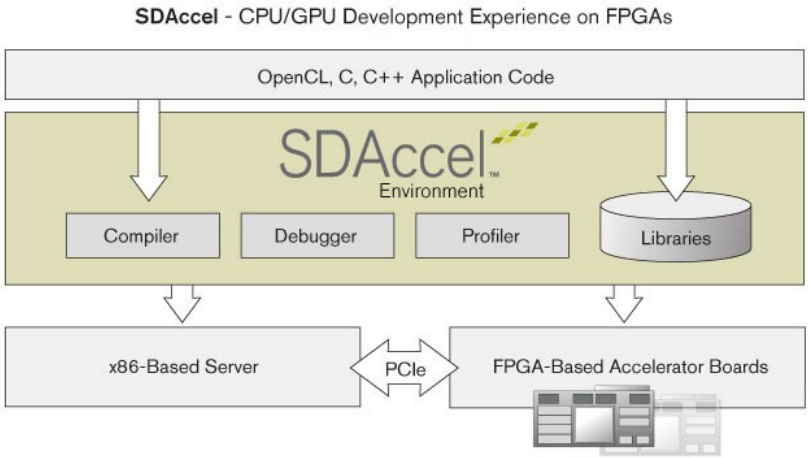


Figure 3.1: SDAccel architecture [11]

Although SDAccel seems very attractive in terms of development time and maintainability it was not chosen as it was not available yet for Zynq FPGAs at the start of this project. Guidi et. al [22] researched the topic and found that the development time to port to FPGA an algorithm was decreased but the performance was not adequate enough. This happens because the compiler provided by SDAccel is still in early phase which means that the optimizations provided are still not optimal comparing to other OpenCL compilers. That implies that it is not suitable for a real-time contrained solution yet. In addition, in the current version they use an X86 host just like the solution provided by TU Delft but this reduces the performance as we have an increase in the latency to transfer data from/to the device.

3.2.3 Vivado HLS

Vivado HLS is a tool provided by Xilinx and focus on automating the workflow of translating a source code described in high level languages to an RTL design. An overview of Vivado is shown in figure 3.2.

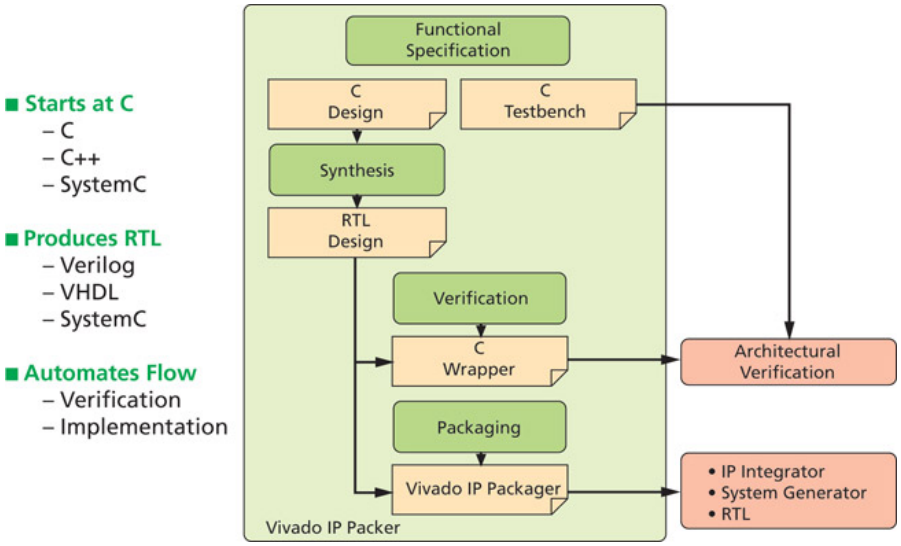


Figure 3.2: Vivado HLS [12]

The Vivado High-Level Synthesis compiler enables C, C++ and SystemC programs to be directly targeted into Xilinx devices without the need to manually create RTL. The focus on that tool is that the FPGAs lifetime is in average 15 years which is over the half lifetime of an X-ray machine (20 years). Concurrently when this thesis was pursued, an investigation regarding the implementation of the same noise reduction algorithm was conducted by a fellow student [20]. Based on his observations Vivado HLS decreased the development time (comparing to a manually created RTL design) however some time and effort had to be invested in order to develop an efficient solution. After an extensive use of large number of diverse directives, the results were that the latency was decreased by 61% and the resource utilization was also decreased by 83%, compared to the initial Vivado HLS design. To summarize this solution, it is still in progress, needs more development time than OpenCL (by far less than describing manually the RTL design) and the final performance of a part of the algorithm was around 15 fps which is the adequate enough based on research conducted by Philips [20].

3.3 Comparison

Platform	Vendor	Advantages	Disadvantages
CUDA	NVIDIA	It is easy to develop source code for this architecture. Moreover it uses the same concept as OpenCL.	Performance is not portable thus it is hard to maintain such solution.
SDAccel	Xilinx	It uses the same concept as OpenCL. Moreover the lifetime is better due to the usage of FPGAs.	Currently needs a PC which means that is will be slow due to data transfers over the PCI-E bus. In addition the compiler is still in early stage so the output is not optimal yet.
SSE/AVX	Intel	Good match to the architecture.	There is a need for specific intrinsic instructions that may alter throughout the lifetime of the algorithm. Thus is hard to develop and maintain such solution.
rVex on Zynq	TU Delft	It utilizes the OpenCL framework and there is no need for a pc which makes the memory transfers faster. Moreover it runs on an FPGA which have better lifetime.	The current version of rVex runs on 100 MHz which means that the performance is not enough for that application.
Vivado HLS	Xilinx	It has better lifetime due to the usage of FPGAs. The final design is also maintainable because the RTL can be transferred to different FPGAs.	The output result is not optimal yet. Thus there is a need for manually optimizing C++ code in order to assist Vivado HLS to a more efficient RTL design for a complex algorithm.

4 Implementation (Summary)

This chapter describes the tools and the decisions that were encountered during the transition from the Matlab model towards the OpenCL implementation. Due to confidentiality there would be only a discussion regarding the tools that were used and developed in order to ensure the correct transition from the Matlab model towards the C and OpenCL implementation. A summary of the implementation process can be shown in figure 4.1.

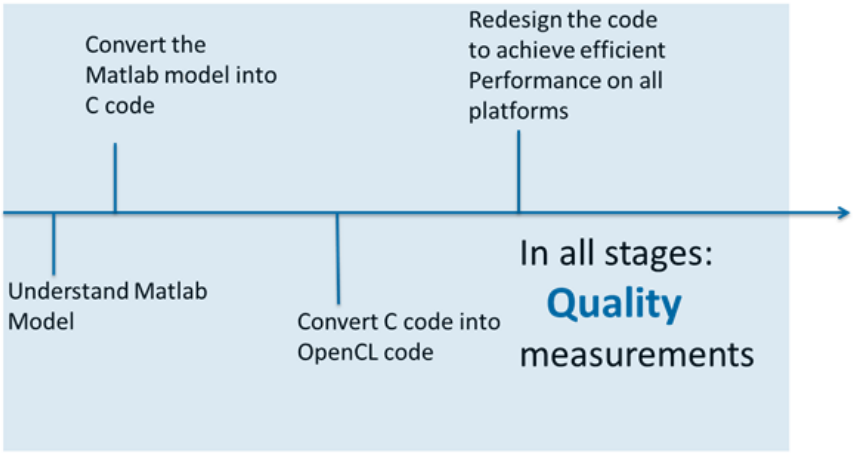


Figure 4.1: Implementation Process from Matlab to OpenCL

4.1 Quality measurements to translate Matlab model to C

When the project started, Philips Healthcare provided a description of the noise reduction algorithm in Matlab. The OpenCL framework as stated previously in chapter 2 uses only the C99 standard; so a transfer from Matlab to C had to take place. Moreover, the original Matlab model uses floating point arithmetic, which is an issue. The issue relies on the fact that the rVex softcore does not have a floating point unit (FPU) and emulates the floating point operations, meaning that each operation using floating point arithmetic has a lot of latency. Those two things generated the following needs:

- **Quality measurements:** Is our quality constraints fulfilled against the Matlab version?
- **Fixed point tool analyzer:** Analyze the amount of bits that each variable needs in order to implement the algorithm using fixed point arithmetic instead of double

precision.

Such a tool was not available, and it was created in the context of the Almarvi project. In order to design such tool some requirements were defined. The following list of requirements emerged:

- The range of the floating point variables need to be determined.
- Simple, portable UI.
- Fast processing to reduce developing time.

Those requirements introduced the Fixed-point Analyzer and Scaler Tool (***FAST***). The FAST was developed in two different parts. The first part the back-end is based on a very simple principal. In order to find the range of a variable someone has to follow the changes during the "life" (scope) of this variable. In order to implement this, we used an overloading macro mechanism. That macro is easy to be embed on the code after each assignment and also can be removed easily after the translation and verification by commending out the overloading phase.

The macro handler is a class that finds (based on the source code name) in which part of the filter we currently are and detects which assignment of that variable takes place. It keeps record of the different variable assigments in order to be able at the UI application to determine the full range of the specific variable.

In order to provide a portable solution, it was decided to create binary output of each variable. This would allow also better performance as reading a binary file of a matrix 960x960 is faster than reading the ASCII representation of that matrix.

After defining the structure of the FAST tool a UI application had also to be created to provide the information that was needed. In order to support all the available Operating systems it was decided to create the FAST tool UI using Java and more specifically the swing framework. That application was the front-end of the FAST tool. It takes as arguments the folder of the outputs provided by the macros and has two functions available:

- Error Comparison
- Fixed-Point Analyzer

The error comparison at the beginning was between the Matlab reference values and the C implementation of the noise reduction algorithm. In this way a verification of the translation into C can be guaranteed. When we reached the final stage of the implementation into C and we verified that the relative error between Matlab and C is relative small (10^{-12}) we established our so called **"Golden standard"**. The reason to that is that Matlab internally use up to 80 bits precision while we were using doubles. After the establishment of the "Golden standard" we could compare it to our fixed point

implementation based on the suggestion of FAST. It is worth mentioning that in order to have a clear view of the amount of bits for each variable, there were used multiple different streams of medical images provided by Philips Healthcare. The way that fast was designed to help the creation of the fixed point implementation from the Golden standard can be shown in figure 4.2.

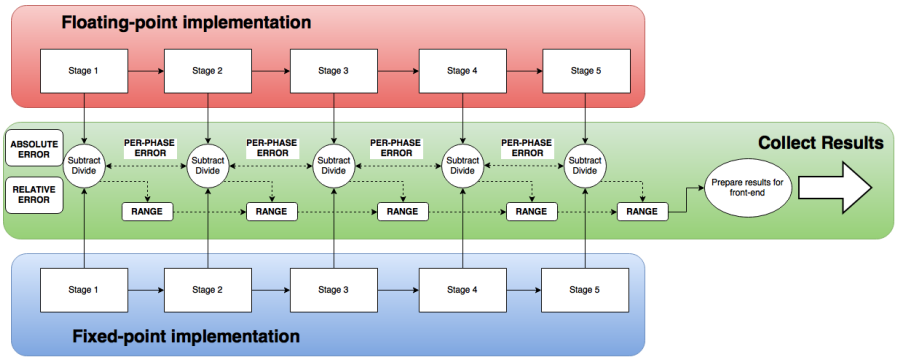


Figure 4.2: FAST Architecture

The summary of the way FAST was used in order to translate the Matlab model to the fixed point OpenCL version can be shown in figure 4.3 on the left side and a sample output of the FAST tool can be shown in figure 4.4.

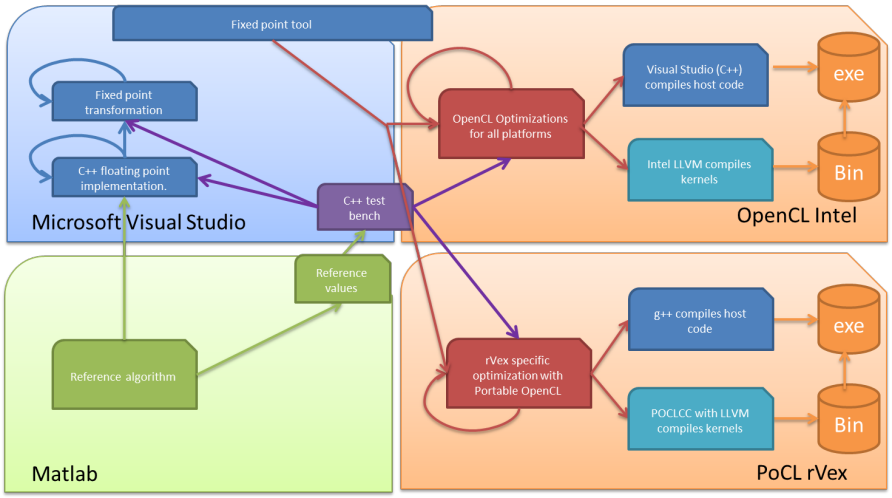


Figure 4.3: Implementation Process from Matlab to OpenCL [13]

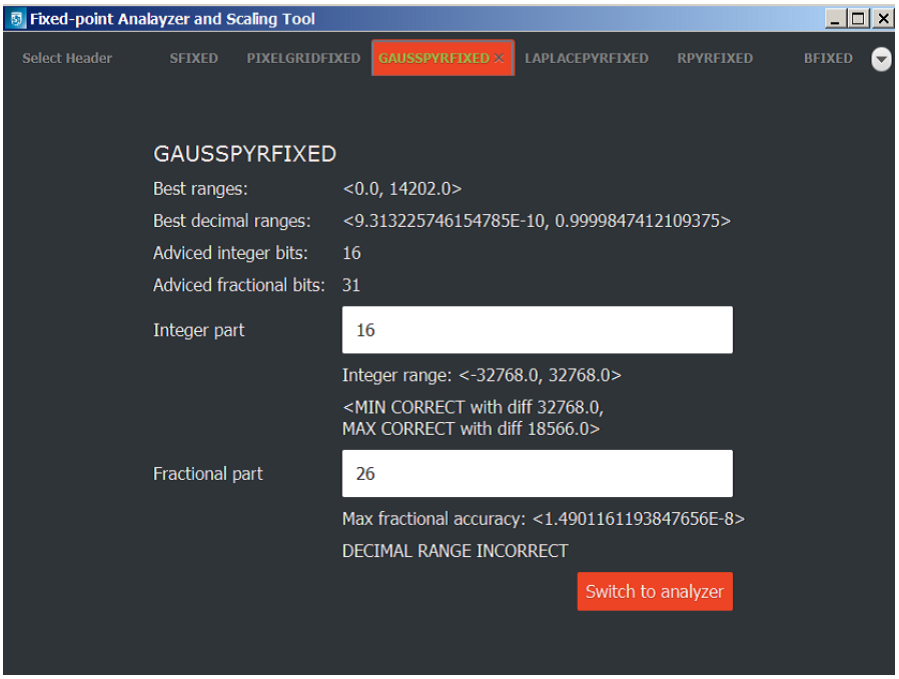


Figure 4.4: FAST GUI demo [13]

After finding the correct fixed point values there was a problem using them in OpenCL. The original library was in C++ and as stated in chapter 2 OpenCL works only with C99. A translation took place using inline functions but based on our measurements the compiler performed better when the fixed point library was implemented using macros instead of function in lining. It was out of the context of this thesis to understand the compiler's behavior on this issue.

4.2 Profiling of the algorithm

One of the most important things for the implementation was to have a comparison mechanism. That mechanism would allow:

- To understand the compute intense parts of the algorithm.
- To have measurements regarding how OpenCL competes with the original sequential code.
- To verify if the optimizations being made improve or decrease the performance.

Such a mechanism is called profiling. In order to achieve that, explicitly defined CPU timers were developed and operated on the sequential code to profile it. The same timers were used in the OpenCL implementation as well, in order to profile the total execution time which consists in three phases:

- Communicate to transfer the input to the device.
- Execute the algorithm on the accelerator.
- Communicate to transfer output.

Furthermore, a list with OpenCL device timers to gather the execution time of each kernel was created. That would allow more insights on how each algorithm step adapts on each device, it would provide the most compute intense parts in parallel and finally, a comparison between different versions of each kernel to choose the one with the best performance.

In order to be able to trust the output of the OpenCL timers a verification that the output that was provided from them was correct was needed. For that reason the OpenCL Builder software bundled with the Intel OpenCL SDK was used. This software relies on extreme accurate timers that Intel has embedded on-chip. Using the OpenCL builder, which provides a graph with the different kernels showed that the timers were correct. These timers helped to interpret the results and identify the compute intensive OpenCL kernels. In order to use the profiler provided by Intel someone needs to install Microsoft Visual Studio and the OpenCL SDK.

4.3 OpenCL error handling

The last mechanism that needed to be created is how someone will handle OpenCL API errors. If someone is programming in C and an error occur then the execution of the program will stop with an error code and a message (e.g. a segmentation fault reading from an erroneous pointer position). However when someone programs an accelerator and handle it from the host system, is mandatory to check after each API call if an error occurred and was reported back. OpenCL by default will report back a status

code which the programmer should check and see if it indicates an error. In order to overcome this and detect the fault origin immediately I created a macro which I use in all my OpenCL API calls. If an error occurs it will stop the execution and report back a string with a meaningful error message. In order to track error codes with the error messages I used the error codes of the OpenCL standard 1.2 as they are described from Khronos group. This means that if in the future someone wants to change the standard that is supported to the version 2.0 for example, that library should be updated with the error codes of that version. A list of the errors with their description is in Appendix A.1.

5 Results (Summary)

Based on the utilization of the timers described in the Implementation chapter, some important results regarding the performance of OpenCL occured. The OpenCL device tested was an Intel i7-4810MQ, an Intel Xeon CPU and the integrated HD Graphics 4600 which is on chip with the i7 processor. A summary of the hardware platforms used can be shown in table 5.1.

Platform Name	Intel i7-4810MQ	Intel Xeon	Intel HD Graphics 4600
#cores	4	8	20 (EU)
#threads	8	16	7(per EU)=140
#frequency	3.8 GHz	3.4GHz	1200 MHz
Cache size	6MB	20MB	(shared with i7)

Table 5.1: Summary of hardware platforms used for experiments

5.1 Performance measurements

After the completion of the Matlab translation and the result verification, the usage of the optimizations in Microsoft Visual studio was activated. The /Ox flag was chosen which allows the compiler to produce code that favors execution speed over smaller size. With that option the following is allowed:

- Function inlining: Expansion of functions marked as inline, __inline, or __forceinline, and any other function that the compiler chooses.
- Global Optimizations: Provides local and global optimizations, automatic-register allocation, and loop optimization.
- Generate intrinsic functions.

The performance of the sequential code was measured both using single and double precision floating point arithmetic. Intuitively, we were expecting floats to be faster than doubles and the result was that it was faster by 30 ms which is 4.3%. But the performance of the sequential code was not even close to be effective as it could produce only 1.4 fps. The Intel GPU did not support double precision arithmetic. For that reason, we started the measurements for the OpenCL version of the algorithm only when the code was translated from double precision to single precision floating point arithmetic. Immediately we spotted a decrease on the amount of time that the algorithm needed to operate and of course an increase on the fps that was processed. The total amount of fps achieved can be shown in figure 5.1.

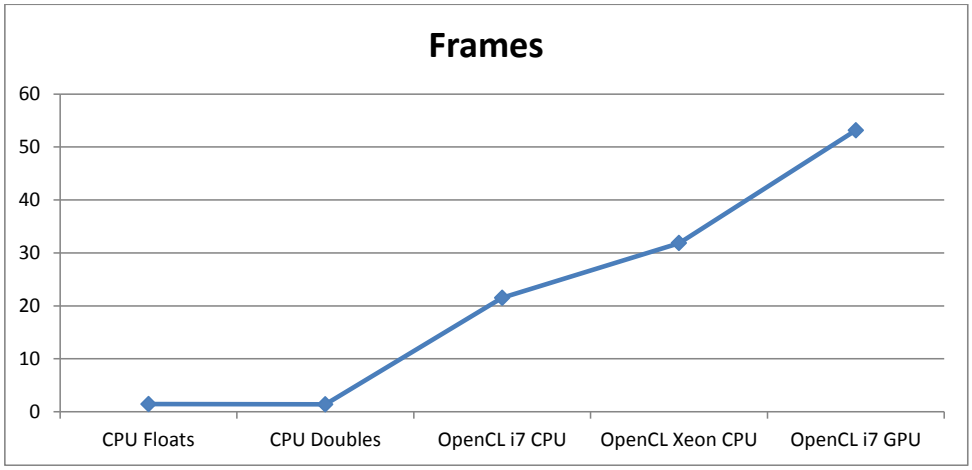


Figure 5.1: Achieved fps for sequential code and different accelerators

From this figure one can understand the potential that the OpenCL implementation has. First of all after applying the OpenCL code the same CPU (i7) achieved higher performance than in the sequential one although in the later we enabled SIMD intrinsics (vectorized code). In addition, OpenCL code exploits also the fact that i7 is a many-core processor which can run different processor engines in parallel. Moreover it can be shown that the Xeon CPU which has more processor engines than the i7 can achieve better performance which implies more fps. This shows that the code is performance portable because we transferred it from one machine to another without changing anything and it could perform better. It was able to exploit the full power of the Xeon CPU without changing neither the kernel code or the work-group slicing. The OpenCL runtime did all the configuration to the device automatically (find the PEs, manage buffers etc.). Finally, the integrated GPU from Intel performed the best result as expected. This happened due to three reasons:

- The GPU has more processor engines than the CPUs.
- It fits more to image processing.
- It does not need a lot of time to transfer data from/to host like an external GPU because the communication over PCI-express is more expensive even with pinned memory.

In terms of runtime, the contribution of this thesis can be shown in figure 5.2. The figure shows the computational time needed by the different blocks (or kernels) that the algorithm has to execute in comparison with the computational time of the OpenCL optimized versions of these kernels. The sequential implementation shows higher execution time for all optimized kernels in the algorithm. The most computationally expensive kernel is Block 9, which was reduced from a sequential 600ms down to 4ms for the GPU, achieving a speedup of 112x. The kernel Block 5 was reduced from a sequential 89ms down to 0.59, achieving a speedup of 151x.

Table 5.2 lists the speedup comparison on all platforms, and for each of the kernel blocks. The table shows that all kernels are able to achieve increased performance on all platforms, except for Block 11 on the i7, which decreases performance to 0.86x.

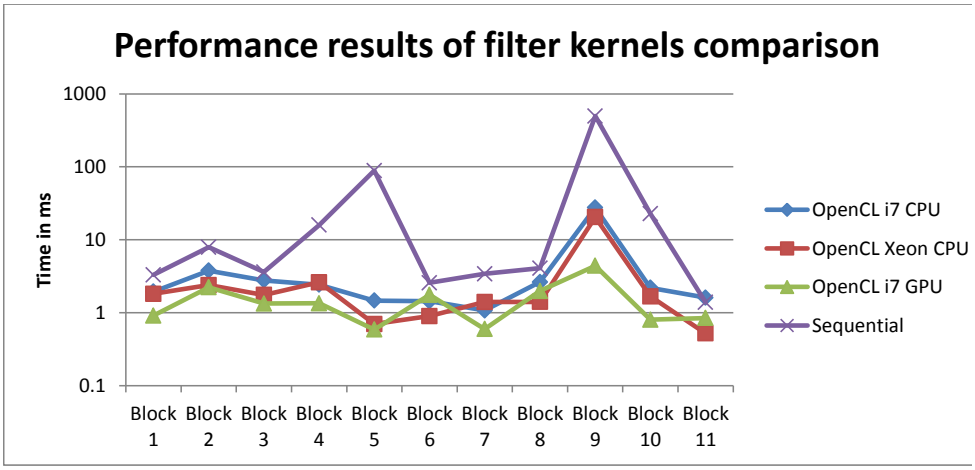


Figure 5.2: Sequential code execution per filter block.

Block	i7 CPU speedup	Xeon speedup	HD Graphics speedup
Block 1	1.70	2.88	3.63
Block 2	2.09	4.10	3.54
Block 3	1.30	2.59	2.70
Block 4	6.58	8.84	11.86
Block 5	60.48	130.26	150.89
Block 6	1.79	2.49	1.46
Block 7	3.18	4.62	5.71
Block 8	1.55	2.99	2.05
Block 9	18.00	23.15	112.98
Block 10	10.42	13.57	28.48
Block 11	0.86	4.44	1.64

Table 5.2: Speedup Comparison between different accelerators and the sequential version

A comparison between the total speedup that was gained against the sequential version is presented in figure 5.3. This represents the weighted average speedup achieved at the algorithm level by running the combined kernels of the algorithm.

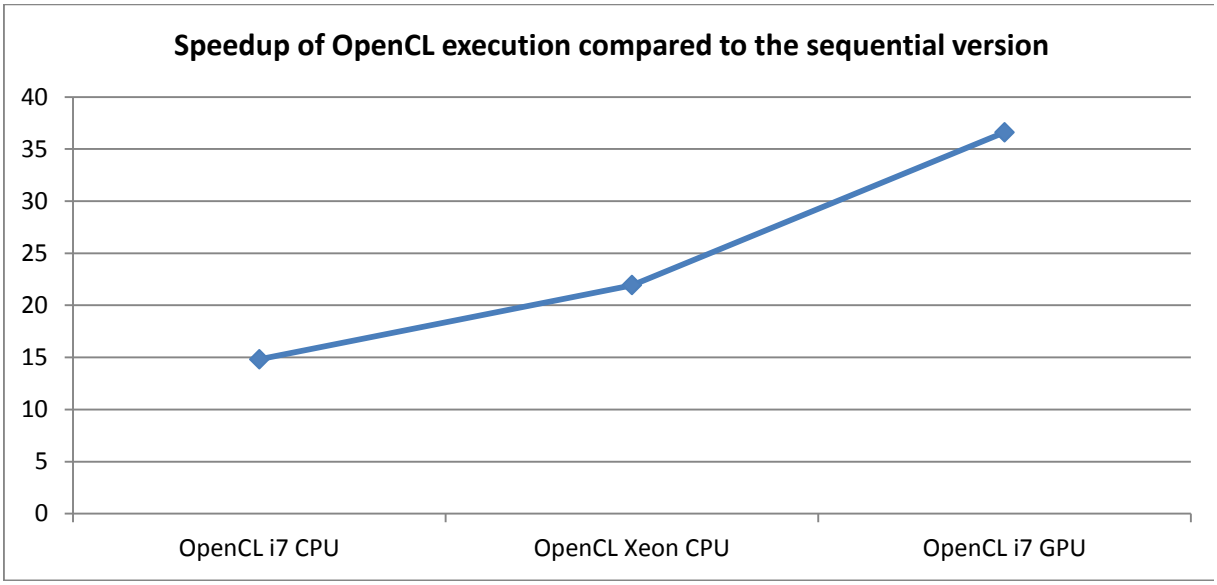


Figure 5.3: Speedup of all OpenCL accelerators against the sequential version.

The figure shows that the achieved speedup is in the range between 13-33 times faster than the sequential version. The platform specific detailed speedup numbers are listed in table 5.3.

Accelarator	Total Speedup
OpenCL i7 CPU	14.81
OpenCL Xeon CPU	21.93
OpenCL i7 HD 4600 GPU	36.60

Table 5.3: Speedup Comparison between different accelerators and the sequential floating point version

5.2 Design choices result and evaluation

One of the most important choices that was made was to allow autovectorization. This allowed to the code to generate and fit the CPU as better as possible. It is shown that if a better CPU is used the code adapts to its possibilities. Moreover, the change from double precision to floating not only allowed us to have a smaller error and double the CPU execution time but in addition, they allowed the execution on the integrated GPU. The later one showed almost doubled speedup than the CPU.

Fixed point arithmetic was developed in order to support FPGAs as well and more specifically to run the code on rVex (or directly using SDAccel). Although it is the only way to ensure that every accelerator used will produce the same output (if any vendor does not comply with the IEEE-754 standard), it is very costly in terms of developing

time. It took a lot of time to investigate the variable ranges in order to support the arithmetic on OpenCL.

The algorithmic based optimizations that were made had as a result a faster execution time but they have a drawback. During runtime there is no way to change the filter attributes, the variables responsible for the visual outcome. Such change can only be made when restarting the application in the initialization state.

5.3 Accuracy tradeoffs

In every step of the transition of the algorithm from C/C++ to OpenCL I had to make some tradeoffs. In order to be able to measure the impact of my tradeoffs I used FAST to measure the relative error produced. My observations of that were very interesting.

In the beginning we developed the filter code using double precision floating point arithmetic, but the developing GPU wasn't supporting double precision arithmetic so I had to use floats instead of doubles. As a result of that, the round-off error increased but there was no visual artifact as the difference was too low. Moreover, I noticed that although the Intel platform utilize the arithmetic functions it was faster to not use the power function (pow) but instead use normal multiplication for this calculation. That added a bit of round-off error but again the resulted error was tolerable.

Finally, we noticed that the relative error produced from the CPU and the GPU with floating point arithmetic was slightly different, while when using fixed point arithmetic the error is the same. My intuition about that is that the GPU handles differently floating point arithmetic than the CPU, for example the amount of guard and round bits that it use. That's why the error is bigger in the GPU. Nevertheless, when using fixed point arithmetic it remains the same as they have the same architecture to handle integers. In the following results can be observer:

- Relative error for different accelerators.
- Relative error with relaxed math.

As the use of relaxed math doesn't provide a significant increase in speed, it is not going to be used.

5.4 Real-time capabilities

Based on our performance results there are a number of accelerators that are already able to perform sufficiently for the minimum real-time constraints that we wanted to achieve. We consider a result successful if it can process at least 30fps this is our threshold. As none of our setups reached 60fps (or above) which was the desired value, we will only use 30fps for the comparison. In Table 5.4 there is a list with accelerators and setups and if they were fast enough to be considered as real-time capable.

Accelerator Name	**960x960 @ 30fps**	**1024x1024 @30fps**
Intel i7-4810MQ CPU	NO	NO
Intel Xeon	YES	YES
Intel i7 HD 4600 GPU	YES	YES

Table 5.4: Accelerator comparison between different image sizes and fps

6 Discussion

The focus of this chapter is to discuss the results of the previous chapter, evaluate them and then verify if the original research question has been answered. In the first section a quantitative evaluation of the results is going to take place and then the verification of the research question is going to get addressed.

6.1 Quantitative evaluation

The results shown in the previous chapter indicates that we created a sufficient solution for the baseline 30fps requirement. Although the i7 CPU doesn't meet this requirement, there is a study pursued by Philips Healthcare that provides information that 15fps is adequate enough for the human eye. That means that all the hardware platforms tested can be used to solve the image processing problem for the X-ray machines.

Moreover, the solution of the GPU has not only better performance but also keeps the CPU free from calculations, which means that the CPU can be used concurrently for other purposes. In addition, it is important that we can dedicate the GPU to computations only, because in the native implementation some specific changes had to be done on the OS in order to not interfere during the calculations. That was because the OS is responsible for the resources of the CPU and could allocate them for different tasks which would imply that the maximum performance could not be achieved. One of the most important aspects in this project is to guarantee constant performance and this was achieved with our implementation.

6.2 Research Question

The research question of this thesis was if there are any means to develop a real-time constrained implementation of a medical image processing algorithm which would:

1. Be portable among different hardware accelerators by means of performance and scalability.
2. Allow low development costs.
3. Be easily maintainable.

In order to address this question we used the OpenCL framework. This portable language allowed us to develop the full algorithm in the duration of 3 months after the translation of the original Matlab model, which means that the development costs were decreased due to smaller development time. The same results are shown by Ferreira et. al [23].

Moreover, the OpenCL implementation is easily maintainable. This is because the OpenCL kernels were described once in OpenCL and there is no need to redevelop them in the future. Throughout the life of the algorithm though some API calls in the host (driver) program might become deprecated. This can be easily solved as the standard evolves, because they will mention which API call replaces the deprecated ones.

And last but not least, we have proven that the code is performance portable and scalable. The implementation was tested against three different platforms. It is shown in the previous chapter that when we tested it on the Xeon CPU which was better than the i7 not only we didn't have to redevelop the source code but it could be used plug and play. As the performance of the Xeon is better than the i7 CPU we also notice the performance improvement using our code. In addition, we noticed that using the same OpenCL kernels we were able to run the algorithm on an i7 integrated GPU, which outperform the CPUs due the compute overhead of a filter, which means that our solution is scalable.

Based on our findings and the reasons mentioned above we can conclude that this thesis research question was answered successfully.

7 Conclusion & Future work

Creating a performance portable and maintainable source code is one of the most complex tasks a computer engineer will encounter. The problem to that is that each vendor provides different solutions to achieve better performance (e.g. Nvidia with the CUDA framework). That's why the OpenCL standard developed and maintain by Khronos group has so many vendors to support it. Although many vendors support it, there are still different architectural concepts to keep in mind while developing such a solution.

This thesis provided a roadmap and the toolflow to achieve such a solution. Using that toolflow and based on specific engineering decisions we were able to provide such a solution. That comes with a drawback though; we can't really achieve the best performance possible on all platforms due to the different architectural concept of each platform. While developing the code we had in mind such problems: the fixed point implementation or the usage of auto-vectorized code instead of implicit declaration of vectorized code. Those decisions were covered alongside with the tools needed in this thesis. However, the results were more than fascinating.

One reason for not achieving the best performance in each platform is that the compilers for OpenCL are still in an initial phase. However, with the rapid advances on the compiler side in the following years, they will provide better performance and the code will still remain the same in the OpenCL kernel side. That proves the power of the OpenCL framework.

For the specific project though there are some areas that should be addressed in the future. There should be careful measurements and architectural improvements on the rVex softcore in order to be able to provide an embedded solution of our OpenCL source code. In addition, a research that could focus on using the SDAccel tool should be explored. Last but not least, measurements with the integrated GPUs of the high-end CPUs provided by Intel and other Vendors should be explored in order to investigate if there are other platforms that can fit better, which is a simple task to do as someone needs only the platforms and some changes on the API already provided by this thesis to run it.

Bibliography

[1] Wilhelm rontgen museum. [Online]. Available: http://www.roentgenmuseum.de/

[2] D. Mery, *Computer Vision for X-Ray Testing*, S. I. Publishing, Ed. Springer International Publishing, 2015.

[3] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0320-y

[4] Khronos group. [Online]. Available: https://www.khronos.org/opencl/

[5] R. Tsuchiyama, *The OpenCL Programming Book*, 1st ed. Fixstars Corporation;, April 13, 2010. [Online]. Available: https://www.fixstars.com/en/opencl/book/

[6] *Writing Optimal OpenCL Code with Intel ® OpenCL SDK*, Intel Corporation, 2011.

[7] *OpenCL Device - Intel Processor Graphics*, Intel Corporation, 2014.

[8] *OpenCL Optimization Guide Intel ®*, Intel Corporation, 2014.

[9] T. Van As, "r-vex: A reconfigurable and extensible vliw processor," Master's thesis, Delft University of Technology, 2008. [Online]. Available: http://repository.tudelft.nl/

[10] H. Van Der Wijst, "An accelerator based on the ρ-vex processor: an exploration using opencl," Master's thesis, Delft University of Technology, 2015. [Online]. Available: http://repository.tudelft.nl/

[11] Sdaccel information. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[12] Vivado information. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[13] Philips, "D4.3 design space exploration [public]," Almarvi, techreport, 2016. [Online]. Available: http://www.almarvi.eu/assets/almarvi_d4.3_final_v10.pdf

[14] S. Computing. Opencl error codes. [Online]. Available: https://streamcomputing.eu/blog/2013-04-28/opencl-error-codes/

[15] Almarvi consortium homepage. [Online]. Available: http://www.almarvi.eu/

[16] W. Coolidge, "X-ray tube." Patent, Jan. 2, 1917, uS Patent 1,211,092. [Online]. Available: https://www.google.com/patents/US1211092

[17] *A Guide to Vectorization with Intel C++ Compilers*, Intel Corporation. [Online]. Available: https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf

[18] *The Compute Architecture of Intel ® Processor Graphics Gen8*, Intel Corporation, 2014.

[19] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, Sept 2013, pp. 245–255.

[20] S. Metman, "Software to hardware: Alternatives for reducing design time of optimized fpga implementations in medical devices [cp]," Master's thesis, Delft Univesity of Technology, 2016. [Online]. Available: http://repository.tudelft.nl/

[21] G. Inggs, S. Fleming, D. Thomas, and W. Luk, "Is high level synthesis ready for business? a computational finance case study," in *Field-Programmable Technology (FPT), 2014 International Conference on*, Dec 2014, pp. 12–19.

[22] G. Guidi, E. Reggiani, L. D. Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, "On how to improve fpga-based systems design productivity via sdaccel," *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 247–252, 2016. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7529874

[23] J. R. Ferreira, M. C. Oliveira, and A. L. Freitas, "Performance evaluation of medical image similarity analysis in a heterogeneous architecture," *2014 IEEE 27th International Symposium on Computer-Based Medical Systems*, pp. 159–164, 2014. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84907395053{&}partnerID=tZOtx3y1

Appendix A

A.1 OpenCL error codes

Code	OpenCL Error Flag	Function(s)	Description
0	CL_SUCCESS		The sweet spot.
-1	CL_DEVICE_NOT_FOUND	clGetDeviceIDs	if no OpenCL devices that matched device_type were found.
-2	CL_DEVICE_NOT_AVAILABLE	clCreateContext	if a device in devices is currently not available even though the device was returned by clGetDeviceIDs.
-3	CL_COMPILER_NOT _AVAILABLE	clBuildProgram	if program is created with clCreateProgramWithSource and a compiler is not available i.e. CL_DEVICE_COMPILER_AVAILABLE specified in the table of OpenCL Device Queries for clGetDeviceInfo is set to CL_FALSE.
-4	CL_MEM_OBJECT _ALLOCATION_FAILURE		if there is a failure to allocate memory for buffer object.
-5	CL_OUT_OF_RESOURCES		if there is a failure to allocate resources required by the OpenCL implementation on the device.
-6	CL_OUT_OF_HOST_MEMORY		if there is a failure to allocate resources required by the OpenCL implementation on the host.
-7	CL_PROFILING_INFO_NOT _AVAILABLE	clGetEventProfiling Info	if the CL_QUEUE_PROFILING_ENABLE flag is not set for the command-queue, if the execution status of the command identified by event is not CL_COMPLETE or if event is a user event object.
-8	CL_MEM_COPY_OVERLAP	clEnqueueCopy-Buffer, clEnqueueCopyBuffer-Rect, clEnqueue-CopyImage	if src_buffer and dst_buffer are the same buffer or subbuffer object and the source and destination regions overlap or if src_buffer and dst_buffer are different sub-buffers of the same associated buffer object and they overlap. The regions overlap if src_offset to dst_offset to src_offset + size 1, or if dst_offset to src_offset to dst_offset + size 1.
-9	CL_IMAGE_FORMAT _MISMATCH	clEnqueueCopy-Image	if src_image and dst_image do not use the same image format.

-10	CL_IMAGE_FORMAT_NOT _SUPPORTED	clCreateImage	if the image_format is not supported.
-11	CL_BUILD_PROGRAM _FAILURE	clBuildProgram	if there is a failure to build the program executable. This error will be returned if clBuildProgram does not return until the build has completed.
-12	CL_MAP_FAILURE	clEnqueueMap-Buffer, clEn-queueMapImage	if there is a failure to map the requested region into the host address space. This error cannot occur for image objects created with CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR.
-13	CL_MISALIGNED_SUB _BUFFER_OFFSET		if a sub-buffer object is specified as the value for an argument that is a buffer object and the offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue.
-14	CL_EXEC_STATUS_ERROR_ FOR_EVENTS_IN_WAIT_LIST		if the execution status of any of the events in event_list is a negative integer value.
-15	CL_COMPILE_PROGRAM _FAILURE	clCompileProgram	if there is a failure to compile the program source. This error will be returned if clCompileProgram does not return until the compile has completed.
-16	CL_LINKER_NOT_AVAILABLE	clLinkProgram	if a linker is not available i.e. CL_DEVICE_LINKER_AVAILABLE specified in the table of allowed values for param_name for clGetDeviceInfo is set to CL_FALSE.
-17	CL_LINK_PROGRAM_FAILURE	clLinkProgram	if there is a failure to link the compiled binaries and/or libraries.
-18	CL_DEVICE_PARTITION _FAILED	clCreateSubDevices	if the partition name is supported by the implementation but in_device could not be further partitioned.
-19	CL_KERNEL_ARG_INFO _NOT_AVAILABLE	clGetKernelArgInfo	if the argument information is not available for kernel.
-30	CL_INVALID_VALUE	clGetDeviceIDs, clCreateContext	This depends on the function: two or more coupled parameters had errors.
-31	CL_INVALID_DEVICE_TYPE	clGetDeviceIDs	if an invalid device_type is given
-32	CL_INVALID_PLATFORM	clGetDeviceIDs	if an invalid platform was given
-33	CL_INVALID_DEVICE	clCreateContext, clBuildProgram	if devices contains an invalid device or are not associated with the specified platform.
-34	CL_INVALID_CONTEXT		if context is not a valid context.
-35	CL_INVALID_QUEUE_ PROP-ERTIES	clCreateCommand-Queue	if specified command-queue-properties are valid but are not supported by the device.
-36	CL_INVALID_COMMAND_ QUEUE		if command_queue is not a valid command-queue.

-37	CL_INVALID_HOST_PTR	clCreateImage, clCreateBuffer	This flag is valid only if host_ptr is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by host_ptr.CL_MEM_COPY_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.CL_MEM_COPY_HOST_PTR can be used with CL_MEM_ALLOC_HOST_PTR to initialize the contents of the cl_mem object allocated using host-accessible (e.g. PCIe) memory.
-38	CL_INVALID_MEM_OBJECT		if memobj is not a valid OpenCL memory object.
-39	CL_INVALID_IMAGE_FORMAT_DESCRIPTOR		if the OpenGL/DirectX texture internal format does not map to a supported OpenCL image format.
-40	CL_INVALID_IMAGE_SIZE		if an image object is specified as an argument value and the image dimensions (image width, height, specified or compute row and/or slice pitch) are not supported by device associated with queue.
-41	CL_INVALID_SAMPLER	clGetSamplerInfo, clReleaseSampler, clRetainSampler, clSetKernelArg	if sampler is not a valid sampler object.
-42	CL_INVALID_BINARY	clCreateProgramWithBinary, clBuildProgram	The provided binary is unfit for the selected device.
			if program is created with clCreateProgramWithBinary and devices listed in device_list do not have a valid program binary loaded.
-43	CL_INVALID_BUILD_OPTIONS	clBuildProgram	if the build options specified by options are invalid.
-44	CL_INVALID_PROGRAM		if program is a not a valid program object.
-45	CL_INVALID_PROGRAM_EXECUTABLE		if there is no successfully built program executable available for device associated with command_queue.
-46	CL_INVALID_KERNEL_NAME	clCreateKernel	if kernel_name is not found in program.
-47	CL_INVALID_KERNEL_DEFINITION	clCreateKernel	if the function definition for __kernel function given by kernel_name such as the number of arguments, the argument types are not the same for all devices for which the program executable has been built.
-48	CL_INVALID_KERNEL		if kernel is not a valid kernel object.

-49	CL_INVALID_ARG_INDEX	clSetKernelArg, clGetKer-nelArgInfo	if arg_index is not a valid argument index.
-50	CL_INVALID_ARG_VALUE	clSetKernelArg, clGetKer-nelArgInfo	if arg_value specified is not a valid value.
-51	CL_INVALID_ARG_SIZE	clSetKernelArg	if arg_size does not match the size of the data type for an argument that is not a memory object or if the argument is a memory object and arg_size != sizeof(cl_mem) or if arg_size is zero and the argument is declared with the __local qualifier or if the argument is a sampler and arg_size != sizeof(cl_sampler).
-52	CL_INVALID_KERNEL_ARGS		if the kernel argument values have not been specified.
-53	CL_INVALID_WORK _DIMENSION		if work_dim is not a valid value (i.e. a value between 1 and 3).
-54	CL_INVALID_WORK_GROUP _SIZE		if local_work_size is specified and number of work-items specified by global_work_size is not evenly divisable by size of work-group given by local_work_size or does not match the work-group size specified for kernel using the __attribute__ ((reqd_work_group_size(X, Y, Z))) qualifier in program source.if local_work_size is specified and the total number of work-items in the work-group computed as local_work_size[0] * local_work_size[work_dim 1] is greater than the value specified by CL_DEVICE_MAX_WORK_GROUP_SIZE in the table of OpenCL Device Queries for clGetDeviceInfo.if local_work_size is NULL and the __attribute__ ((reqd_work_group_size(X, Y, Z))) qualifier is used to declare the work-group size for kernel in the program source.
-55	CL_INVALID_WORK_ITEM _SIZE		if the number of work-items specified in any of local_work_size[0], local_work_size[work_dim 1] is greater than the corresponding values specified by CL_DEVICE_MAX_WORK_ITEM_ SIZES[0], CL_DEVICE_MAX_WORK _ITEM_SIZES[work_dim 1].

-56	CL_INVALID_GLOBAL_ OFFSET		if the value specified in global_work_size + the corresponding values in global_work_offset for any dimensions is greater than the sizeof(size_t) for the device on which the kernel execution will be enqueued.
-57	CL_INVALID_EVENT_WAIT_ LIST		if event_wait_list is NULL and num_events_in_wait_list >0, or event_wait_list is not NULL and num_events_in_wait_list is 0, or if event objects in event_wait_list are not valid events.
-58	CL_INVALID_EVENT		if event objects specified in event_list are not valid event objects.
-59	CL_INVALID_OPERATION		if interoperability is specified by setting CL_CONTEXT_ADAPTER_D3D9_KHR, CL_CONTEXT_ADAPTER_D3D9EX _KHR or CL_CONTEXT_ADAPTER _DXVA_KHR to a non-NULL value, and interoperability with another graphics API is also specified. (only if the cl_khr_dx9_media_sharing extension is supported).
-60	CL_INVALID_GL_OBJECT		if texture is not a GL texture object whose type matches texture_target, if the specified miplevel of texture is not defined, or if the width or height of the specified miplevel is zero.
-61	CL_INVALID_BUFFER_SIZE	clCreateBuffer, clCreateSubBuffer	if size is 0.Implementations may return CL_INVALID_BUFFER_SIZE if size is greater than the CL_DEVICE_MAX_MEM_ALLOC_SIZE value specified in the table of allowed values for param_name for clGetDeviceInfo for all devices in context.
-62	CL_INVALID_MIP_LEVEL	OpenGL-functions	if miplevel is greater than zero and the OpenGL implementation does not support creating from non-zero mipmap levels.
-63	CL_INVALID_GLOBAL_WORK_SIZE		if global_work_size is NULL, or if any of the values specified in global_work_size[0], global_work_size [work_dim 1] are 0 or exceed the range given by the sizeof(size_t) for the device on which the kernel execution will be enqueued.
-64	CL_INVALID_PROPERTY	clCreateContext	Vague error, depends on the function
-65	CL_INVALID_IMAGE _DESCRIPTOR	clCreateImage	if values specified in image_desc are not valid or if image_desc is NULL.

-66	CL_INVALID_COMPILER _OPTIONS	clCompileProgram	if the compiler options specified by options are invalid.
-67	CL_INVALID_LINKER _OPTIONS	clLinkProgram	if the linker options specified by options are invalid.
-68	CL_INVALID_DEVICE _PARTITION_COUNT	clCreateSubDevices	if the partition name specified in properties is CL_DEVICE_PARTITION_BY_COUNTS and the number of sub-devices requested exceeds CL_DEVICE_PARTITION_MAX_SUB_ DEVICES or the total number of compute units requested exceeds CL_DEVICE_PARTITION_MAX_ COMPUTE_UNITS for in_device, or the number of compute units requested for one or more sub-devices is less than zero or the number of sub-devices requested exceeds CL_DEVICE_PARTITION_MAX_ COMPUTE_UNITS for in_device.

Table A.1: OpenCL Error Codes [14]