Reducing Design Time to Develop Portable Image Processing Applications Using SDAccel

Uttam kumar Elango

CE-MS-2017-11

Abstract

Interventional X-Ray (iXR) systems require specialized accelerators and advanced image processing techniques to reduce noise levels in the output image produced by machine components and low radiation doses. Currently, the image processing chains are implemented on PCs which have an average life cycle of 3 to 5 years whereas, the life cycle of X-Ray systems is expected to be up to 20 or 25 years. This mismatch introduces the need to change the PC architecture during the lifetime of the medical system, for which the image processing chain has to be redeveloped and retested increasing the maintenance costs. A solution to the life cycle management challenge is to use Field Programmable Gate Arrays (FPGAs) since certain FPGAs are considered to have longer life cycles than PCs. However, the process of programming and integrating the FPGA hardware into existing systems is challenging for software developers. Moreover, portability needs to be ensured to reduce the development time when moving to different compute devices for improved functionality or performance. This thesis investigates the possibility to reduce the design time for FPGAs while maintaining portability. To achieve this, the SDAccel framework and a workflow combining Halide (a Domain Specific Language) and SDAccel were proposed and analyzed. The results indicate that high-performance image processing solutions can be implemented on FPGAs in a fraction of the time it takes to create manual RTL designs, while maintaining functional and performance portability.

Reducing Design Time to Develop Portable Image Processing Implementations on FPGAs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Uttam kumar Elango
born in Akividu, India

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

Reducing Design Time to Develop Portable Image Processing Implementations on FPGAs

by Uttam kumar Elango

Abstract

Interventional X-Ray (iXR) systems require specialized accelerators and advanced image processing techniques to reduce noise levels in the output image produced by machine components and low radiation doses. Currently, the image processing chains are implemented on PCs which have an average life cycle of 3 to 5 years whereas, the life cycle of X-Ray systems are expected to be up to 20 or 25 years. This mismatch introduces the need to change the PC architecture during the lifetime of the medical system, for which the image processing chain has to be redeveloped and retested increasing the maintenance costs.

A solution to the life cycle management challenge is to use FPGAs since certain FPGAs are considered to have longer life cycles than PCs. However, the process of programming and integrating the FPGA hardware into existing systems is challenging for software developers. Moreover, portability needs to be ensured to reduce the development time when moving to different compute devices for improved functionality or performance. This thesis investigates the possibility to reduce the design time for FPGAs while maintaining portability. To achieve this, the SDAccel framework and a workflow combining Halide (a Domain Specific Language) and SDAccel were proposed and analyzed. The results indicate that high-performance image processing solutions can be implemented on FPGAs in a fraction of the time it takes to create manual RTL designs, while maintaining functional and performance portability.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2017-11

Committee Members :

Advisor:	Dr. ir. Zaid Al-Ars, CE, TU Delft
Member:	Dr. ir. Rene van Leuken, CAS, TU Delft
Member:	ir. Steven van der Vlugt, Philips Healthcare
Member:	ing. Rob de Jong, Philips Healthcare

Dedicated to:
My parents for their motivation and support
My brother-in-law and sister for their helpful advice
My friends in Netherlands and India for their help and support

Contents

List of Figures

List of Tables

List of Acronyms

ALAP	As Late As Possible
ALMARVI	Algorithms, Design methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing
API	Application Programming interface
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BRAM	Block Random Access Memory
CDFG	Control/Data Flow Graph
CFG	Control Flow Graph
CLB	Configurable logic block
CPU	Central Processing Unit
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
DFG	Data Flow Graph
DRAM	Dynamic Random Access Memory
DSL	Domain Specific Language
DSP	Digital Signal Processor
EU	Execution unit
FIFO	First in First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GP	General Purpose
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HIPAcc	Heterogeneous Image Processing Acceleration

HLL	Higher Level Language
HLS	High Level Synthesis
IGT	Image Guided Therapy
ILP	Instruction Level Parallelism
IP	Intellectual Property
IR	Intermediate Representation
iXR	Interventional X-ray Systems
LCM	Life Cycle Management
LUT	Look-up Table
MRA	Multi Resolution Analysis
MRI	Magnetic Resonance Imaging
NDRange	N-dimensional range
OpenCL	Open Computing Language
OS	Operating System
PC	Personal Computer
PCIe	Peripheral Component Interconnect Express
QoR	Quality of Results
RTL	Register Transfer Level
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SP	Streaming Processor
SRAM	Static Random Access Memory
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Acknowledgements

This thesis marks the end of my master study programme at TU Delft. These past two years have been instrumental in my academic and personal growth. I would like to reflect on the people who have helped and supported me during this period.

First and foremost, I would like to thank my supervisor at the university, Zaid Al-Ars, for his guidance and support throughout my work. His door was always open for questions, and his advice helped me in making the right choices during my study.

This thesis was performed at Philips healthcare in Netherlands. My experience at Philips has been nothing short of amazing. I have been incredibly fortunate to have Steven van der Vlugt and Rob de Jong from Philips as my supervisors during my thesis work. I thank them for making me feel welcome and steering me in the right direction during my research. Their enthusiasm towards their work was inspiring. Rob's amazing insights and Steven's general advice helped me grow both personally and academically.

I would like to extend my gratitude to my colleagues Aries Thio Gunawan, Ruben Guerra Marin, Rachana Arun Kumar for their helpful advice and support. I thank my friends at TU Delft, for providing me the much-needed distraction during my thesis to blow off some steam.

Finally, I must express my profound gratitude to my family for providing me support and encouragement throughout my life without whom this accomplishment would not be possible.

Uttam kumar Elango
Delft, The Netherlands

1 Introduction

Image processing, mainly digital image processing is finding numerous applications in several industrial and research domains. Processing an image requires performing complex operations that are expensive both in time and resource usage. Images represent a large data set, and many tasks require several operations to be performed on each pixel in the image. Furthermore, when real-time constraints are specified, they must be performed at live video rates. Such requirements demand careful selection of hardware architectures and programming techniques.

In the computing world, there exists a plethora of different devices such as Central Processing Units (CPUs), Graphic Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs) etc. Each device has its own set of advantages and disadvantages that the programmer has to leverage to obtain maximum performance. This brings a need to create a unifying specification that enables the programmer to write the code once, and use it everywhere. This is called functional portability and has seen decades of research. One key issue here is that even though programs run functionally correct on different architectures, it doesn't necessarily mean that they obtain the maximum achievable performance. User intervention is still required to ensure that the program is leveraging the device capabilities to the maximum extent possible. A dream scenario would be to develop the application once and execute it on different platforms at the maximum achievable performance, automatically.

In this work, the focus will be on reconfigurable architectures, particularly FPGAs. FPGAs are reconfigurable devices that contain several programmable logic blocks that can be customized to generate the required hardware. However, hardware generation on the FPGA fabric is quite difficult, in the sense that, the effort it takes to implement the design, verify and validate is quite complex when compared to its software counterpart. The skill set required for a hardware programmer is vastly different than a software programmer. This results in a steep learning curve for a general-purpose software programmer to move to hardware programming. To address this issue, an abstraction layer is required that ensures that the programmer requires minimal (or ideally none) knowledge of the hardware, but is still able to perform hardware synthesis. As we achieve higher abstraction levels, we can also manage to unify CPU, GPU, and FPGA programming.

The focus of this thesis is to investigate the current state of tools that are available to abstract the low-level hardware details from the programmer, thereby facilitating ease of development on FPGAs. The rest of the report will describe the context within which the research will take place, provide background on the subject, and finally outline the implementation and results.

1.1 Context

This section will describe the context within which this thesis was written.

1.1.1 Philips healthcare

Koninklijke Philips N.V. (Royal Philips Electronics of the Netherlands, commonly known as Philips) is a Dutch technology company headquartered in Amsterdam. It was founded in Eindhoven in 1891 by Gerald Philips and Frederik Philips [1]. Philips is organized into two divisions: Philips Consumer Lifestyle and Philips Healthcare. As the main division in the enterprise, Philips Healthcare focuses on improving people's lives through meaningful innovation in the healthcare industry. Some of the popular health care products are Interventional X-Ray Systems, Computed Tomography (CT)-scan, Ultrasound, Magnetic Resonance Imaging (MRI), and Clinical Management Systems. Philips brand line is "innovation and you" which highlights its mission to innovate while understanding people's needs. Philips Healthcare, in particular, saves millions of lives through its exceptional innovation and state of the art healthcare systems.

This work was performed in the Image Guided Therapy (IGT) department of Philips Healthcare at Best, the Netherlands, where the focus is on development of real-time applications for minimally invasive medical procedures.

1.1.2 ALMARVI

Almarvi stands for: **Algorithms, Design Methods, and Many-core Execution Platforms for Low-Power Massive Data-Rate Video and Image Processing**. Almarvi is a European project with many different parties contributing deliverables worldwide. Philips Healthcare and TU Delft are also part of this project.

To reduce the overall system design cost, time-to-market, and to enable low-cost solutions for high volume markets, the Almarvi project was conceived [2]. Almarvi project focuses on: enabling cross domain re-use and interoperability for different product categories and application domains, facilitate predictable system and product properties, develop joint hardware-software techniques for resource and power management. The main goal of this thesis is to ease the programming effort for FPGAs and to ensure portability among different architectures which combines perfectly with the Almarvi mission statement: "to ease the programming of new technologies and cut production costs by reducing design time".

1.2 Imaging system

Fig. 1.1 shows an Interventional X-ray Systems (iXR) system designed at Philips. It generates high quality images at a fast rate with the help of advanced image processing techniques.

X-radiation, which is composed of X-rays, is a form of electromagnetic radiation. X-rays with high energies (> 10 keV) are called hard X-rays. These hard X-rays can

Figure 1.1: Azurion Interventional X-ray system [1]

penetrate the skin and are often used in medical radiography, like in the iXR medical X-ray systems.

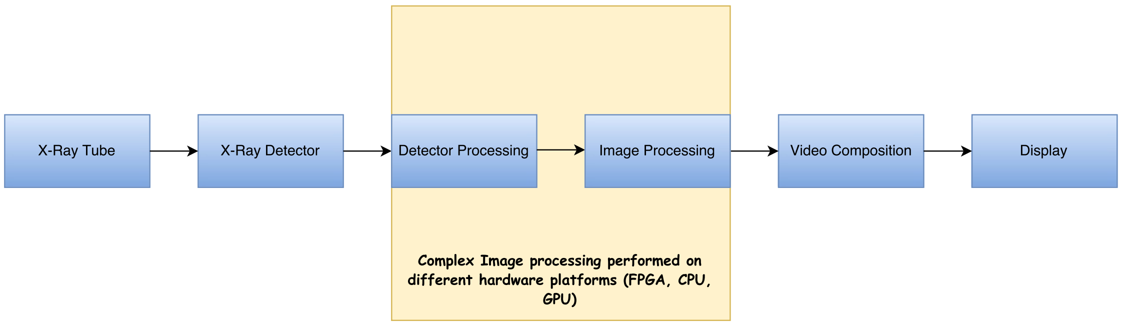

X-Ray Tube
X-Ray Detector
Detector Processing
Image Processing
Video Composition
Display
Complex Image processing performed on different hardware platforms (FPGA, CPU, GPU)

Figure 1.2: iXR – processing chain

Fig. 1.2 shows the iXR image processing chain. First, the X-ray tube generates the X-rays which fall on the patient lying on a detector plate. Based on the subjects X-ray absorption properties, a "shadow image" is formed on the detector plate. Some imperfections may exist during the capture of a raw X-ray image, but they can be removed by imaging algorithms. The detector processing block provides a dose control feedback loop for regulating the X-ray dose [3] and performs a minimal clean up of the raw images. Next, more advanced image processing algorithms are applied to improve the image quality, examples include noise reduction and contrast enhancement. After the image has been processed, it is streamed to the output to be viewed directly by the physician treating the patient. This video stream is either sent directly to an external monitor or to another PC which distributes the signal to different monitors. Both functional correctness and latency are crucial requirements for such an X-ray system. The physician needs to have timely feedback of the patient under study to avoid making potentially disastrous mistakes.

1.3 Problem discussion

Interventional X-Ray systems currently use PCs to implement the image processing pipelines. PCs are expected to have shorter life cycles (3 to 5 years) due to regular advancements in hardware architectures whereas X-Ray systems have an average life cycle of up to 20 or 25 years. Hence there is a need to redevelop and retest the imaging algorithms to target the next generation PCs during the life cycle of the medical system which increases the time to market and maintenance costs.

FPGAs are considered to be an attractive choice to address the Life Cycle Management (LCM) challenge since certain FPGAs have a much longer life cycle (15 years or more) than PC based platforms. They also offer deterministic performance benefits to satisfy real-time constraints. But to effectively use FPGAs, the designer has to be aware of low-level hardware details, which are unfamiliar to many application programmers. To ease the development process on FPGAs, research has focused on High Level Synthesis (HLS) tools, and several industries have come up with their own set of tools to bridge the gap between software and hardware synthesis. A common goal of these tools is to hide the low-level details from the developer. However, if software programs are directly mapped to hardware, then the generated design might not be the best model on FPGAs. Thus an efficient tool flow is required that could generate hardware with minimal knowledge of low-level implementation details.

Moreover, portability must be ensured among different platforms (CPU, GPU and FPGA) so that the same application can be used on other platforms, ideally without modifications. Providing separation between application and hardware will reduce redevelopment time and costs when using different computing platforms for improved functionality or performance.

From the above discussion, the following research question can be formulated,

Is it possible to generate hardware structures on FPGAs from higher level programming languages for image-processing algorithms, resulting in ease of development for programmers and ensuring portability among CPUs and GPUs?

To answer the above question the following sub-questions must be explored:

- What are the architectural differences between CPUs, GPUs, and FPGAs?
- What are the characteristics of image processing algorithms and what challenges are involved in porting these algorithms to FPGAs?
- What are the challenges involved in generating hardware from Higher Level Languages (HLLs)?

- What are the current tools available to achieve hardware generation on FPGAs from HLLs and do these tools result in an easier development cycle?
- What are the current research trends in obtaining portability across CPUs, GPUs and FPGAs?

Based on the above problems, we can formulate a set of global requirements for the project.

- A workflow in terms of tools and strategies needs to be developed. The output must be similar to the original working solution.
- The algorithm must be implemented once, and ideally be able to execute correctly on different accelerators with acceptable latency and throughput requirements. Since the domain is medical image processing, any changes to the algorithm itself might lead to undesirable outcomes.
- The workflow needs to ensure that complex hardware challenges are abstracted from the programmer and result in an easier development cycle.

Background 2

This project aims to investigate tools and techniques available to quickly obtain hardware designs on Field Programmable Gate Arrays (FPGAs) for image processing algorithms while maintaining portability. Ideally, the work flow should be adaptable to general-purpose tasks and should meet the requirements of the Philips use-case. To do this, it is necessary to look into the different computational platforms to understand the architectural requirements. This chapter starts with the analysis of different computational platforms namely Central Processing Units (CPUs), Graphic Processing Units (GPUs), and FPGAs. This is followed by analyzing the general characteristics of image-processing algorithms and selecting a suitable representation that emulates the Philips use-case.

2.1 Hardware architectures

We discussed that FPGAs might be a suitable platform to address the Life Cycle Management (LCM) challenge, but we also want the flexibility to move to other computational platforms like CPUs and GPUs with minimal development efforts. FPGAs are preferred since a custom-built hardware design will often have a higher throughput than its software counterpart which is a major factor in real-time applications. Secondly, several commonly used functions are available as Intellectual property (IP) cores which are highly optimized and can be readily integrated into existing designs alleviating design costs.

There is, of course, no "one platform fits all" solution available since all of them have very different capabilities and limitations. We will perform a short study to understand and compare these differences.

2.1.1 CPU

CPUs are the most well-known and widely used devices. The fundamental operation of CPUs is to execute a sequence of stored instructions. CPUs have grown from single-core to multi-core processors capable of exploiting parallelism both in data and instructions. They are capable of executing enormous amounts of different operations and tasks. A large unit is dedicated for managing and scheduling tasks which effectively optimizes performance bottlenecks such as branch prediction and instruction ordering. The development time on CPUs is relatively low. They can be targeted with a large range of programming languages making it easier to program.

2.1.2 GPU

GPUs are gaining popularity due to their ability to exploit massive parallelism in certain tasks. The capability of GPUs to be used as accelerators are being widely exploited in all compute intensive applications.The advantages of using GPUs are attributed to its high memory bandwidth and a large number of programmable cores. Graphics processors are either integrated with CPU or placed separately with the ability to communicate via a Peripheral Component Interconnect Express (PCIe) bus. The development time on GPUs is also low, thanks to the introduction of Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) which provides a C-based programming environment for programming GPUs.

2.1.3 FPGA

CPUs and GPUs are off-the-shelf products whose architectures cannot be changed, whereas FPGAs are reconfigurable devices. FPGAs are based around a matrix of Configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be programmed (and reprogrammed) to the desired application for functionality requirements after manufacturing. FPGAs are programmed using Hardware Description Languages (HDLs) like Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog which mandates knowledge of low-level hardware details. A synthesis, implementation and routing tool can translate a hardware design to digital logic that can be implemented in this array. Table 2.1 provides a comparative overview of the three hardware platforms.

	CPU	**GPU**	**FPGA**
Computation	Fixed Arithmetic Units 2 - 8 cores	Fixed Arithmetic Units 600 - 4000 cores	User-configurable Logic
Parallelization	MultiThreading (Pthreads/OpenMP) Vector Instructions	Highly Parallel Single Instruction Multiple Data (SIMD)	Pipeline Execution Multiple hardware units
Arithmetic	Versatile Floating-point and Integers	Faster than CPUs in certain operations single-precision FP > double-precision FP	Integer and fixed-point work well floating-point not advisable
Programmability	Easy to program C,C++,Java etc.	Relatively easier to program CUDA, OpenCL	Complicated HDLs-low-level details required
I/O	Fixed I/O	Fixed I/O	user-configurable I/O
Debugging	Many tools available Gdb, Valgrind etc.	Relatively easier to debug than FPGAs Intel Vtune amplifier, NVVP etc.	Complicated
Upgrades	Easy	Moderate	Complicated

Table 2.1: Features overview

2.1.4 Memory architecture

In this part, we explore the memory subsystem of the different computational devices discussed above. In almost all platforms, at the top of the memory hierarchy, there are on-chip storage elements called registers which have the lowest read/write latency. Registers are usually used to store intermediate results of operations for fast access. The second layer is formed by cache memories which also provide low latency accesses. Caches are split into multiple hierarchies named as L1, L2, and L3 caches. Each level in the cache is progressively slower. Thus L2 cache has a higher access time than L1 cache. This is followed by the Dynamic Random Access Memory (DRAM) (also called main memory) which are optimized for capacity but have relatively slower access times. Finally, the last layer in the hierarchy is referred to as the secondary memory (also named hard disks) that can be used to store huge amounts of data but has high access latency. All the different memories except secondary memory are volatile memories (data is remove once the device powers off).

In CPUs, registers are located within the processing units. There are general-purpose registers that the programmer can use and special-purpose registers (program counter, status register etc.) that are internal to the CPU. Fig. 2.1 shows the typical memory organization of modern CPUs.

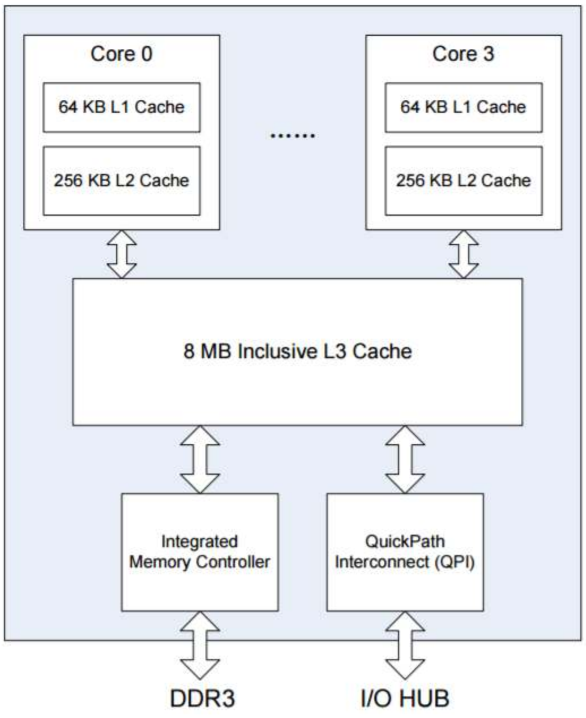


Figure 2.1: Memory architecture on CPU (Intel Nehalem Quad-Core CPU) [4]

Fig. 2.2 shows the memory organization on GPUs. The global memory is shared by the entire GPU. It is used for communication among cores and also for communication with the host (typically CPUs). Transferring data to and from the global memory is the most time-consuming operation on the GPUs. The global memory is implemented on off-chip DRAM. The shared memory is local to each Streaming Multiprocessor (SM) and can be shared by all the Streaming Processor (SP) in that SM. The access latency for this memory is quite low, and they are usually implemented in Static Random Access Memories (SRAMs). The next level is the local or private memory, which is specific to a single SP core. They are part of the global memory and are used when the registers are not capable of holding the thread data (a phenomenon called as register spilling). To achieve better access latency, modern GPUs cache portions of the local memory on-chip.

There are also two read-only memories (read-only for threads, CPU can read/write on these memories) named “constant memory” and “texture memory”. Constant memory is usually used to fill data during compile time to reduce the amount of data transfer between the host and GPUs during program execution. On the other hand, texture memories exhibit two-dimensional locality (spatial locality) thereby aiding in transferring blocks of data for fast access. Both constant and texture memory can be cached on-chip.

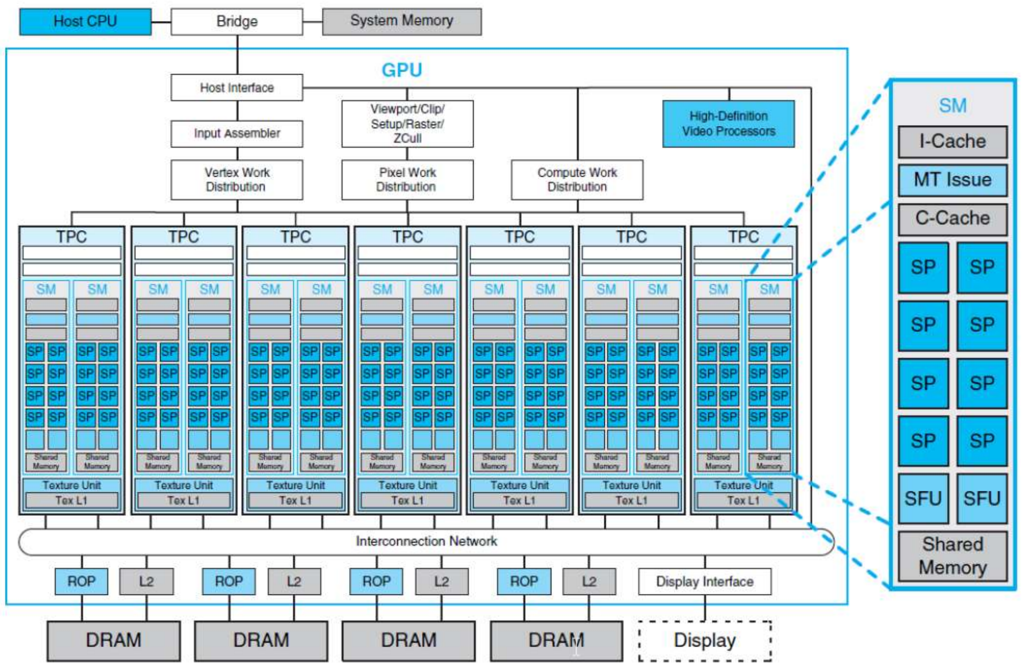


Figure 2.2: Memory Architecture of GPUs [5]

FPGAs are reconfigurable devices which the user can program to fit his application needs. On FPGAs, configurable logic blocks typically consists of Look-up Tables (LUTs) and Flip Flops (FFs). LUTs are made out of logic gates and can be used to store all possible outcomes of a particular function, thereby allowing fast access times. Memory modules on FPGAs are typically Block Random Access Memories (BRAMs) which are on-chip memory resources that can be configured for read or write accesses. Finally, there is the DRAM which is off-chip and can store relatively large amounts of data but has higher access latency.

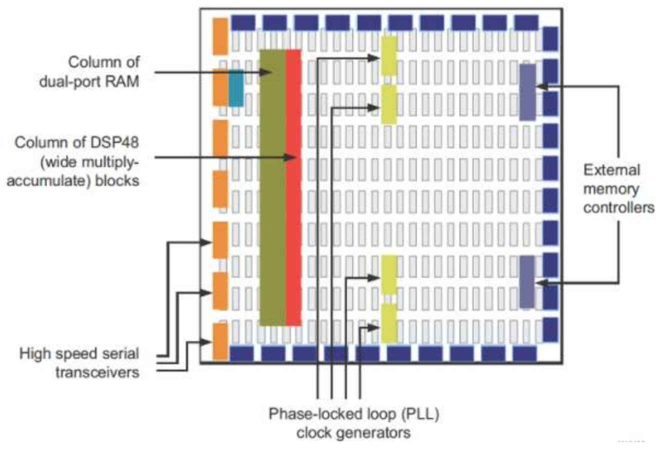

Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
Phase-locked loop (PLL) clock generators
External memory controllers

Figure 2.3: Memory architecture on FPGAs [6]

To make things clear, we will draw comparisons between the different platforms for some of the important metrics that will be considered throughout this work.

- *Performance portability*: This metric indicates the "performance maintainability" when the design is ported to a different device of the same kind.

- *Ease of development*: This metric indicates the development effort required to run applications on the platform.

- *Life cycle*: This metric indicates the average life cycle of the hardware platforms (average time before the product become obsolete).

- *Scalability*: This metric indicates whether adding more resources will increase the performance.

- *Energy efficiency*: This metric indicates the number of operations (GFLOPs) performed for one watt of power consumption (Operations/Watt). [7].

- *Timing latency*: This metric indicates the capability of achieving deterministic timing requirements [8].

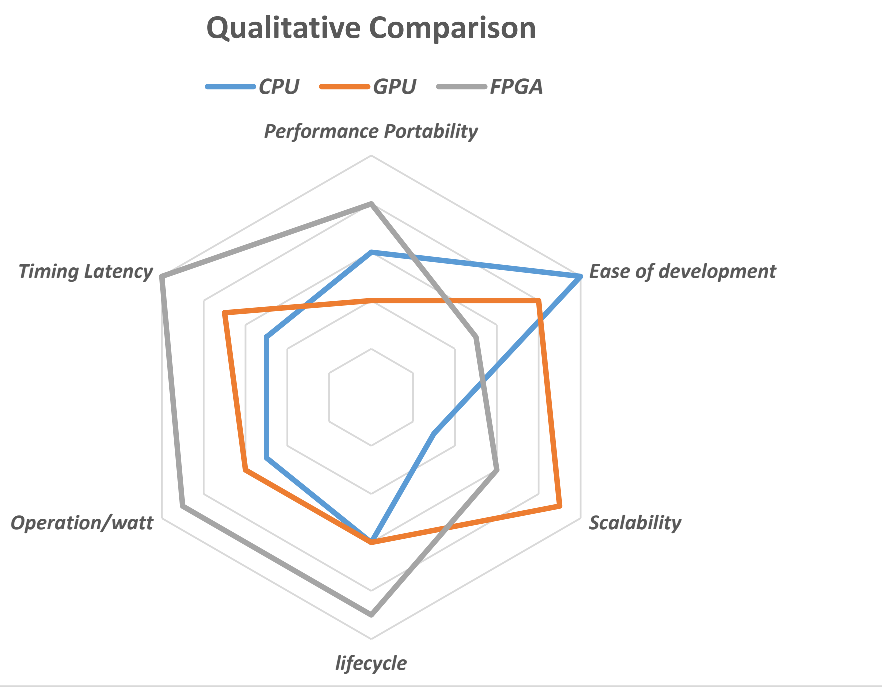


Figure 2.4: Comparison chart CPUs, GPUs, and FPGAs

Fig. 2.4 shows the comparison between the parameters discussed above for CPUs, GPUs and FPGAs. All comparisons are made with devices in the same price range. Please note that the farther the line is from the center, the better the device performs on that metric (e.g. FPGAs provide better timing latency followed by GPUs and then CPUs).

2.1.5 Discussion

The study performed in the previous sections shows that memory architecture is different for all the three platforms. E.g. GPUs provide shared memory (also known as scratchpad memory) local to an SM, which is unavailable on CPUs. Thus in CPUs, local variables (in registers) spill to caches which have higher access latency than shared memory in GPUs [9]. FPGAs, on the other hand, are user configurable. This means that the programmer can generate hardware structures and decide on the memory hierarchy. Multiple hardware structures can be created with memory elements between them, and data can be directly transferred without the need for off-chip memory. Understanding memory access patterns to effectively leverage temporal (once a memory word has been accessed, it is likely to be re accessed) or spatial locality (once a memory word has been accessed, nearby words are likely to be accessed) will lead to achieving good performances. Therefore either the programmer or the tool must be capable of identifying such patterns.

2.2 Image processing

In order to generate a good representation of an algorithm that is suitable for medical imaging and one that is representative for the Philips use-case, we need to understand the characteristics of image processing algorithms. In this section we will look at some of the characteristics, and classes of image processing algorithms.

2.2.1 Imaging operations

Medical image processing, and image processing in general, is mostly performed by applying a number of processing steps to an image, altering the contents of the image to the needs of the user. Image processing operations can be classified into three types: *point*, *window* and *global* operations. Fig. 2.5 shows how these operations are performed on an image.

- `Point operations`: In these type of operations, the output pixel value depends only on the value of the corresponding input pixel (e.g. Threshold operations).
- `Window operations`: In global operations, each pixel in the output image is produced by sliding an $N \times M$ window over the input image and computing an operation according to the input pixels under the window and the chosen window operator (e.g. Convolution filters).
- `Global operations`: In these operations, the output value of a pixel is dependent on the entire input image (e.g. Fast Fourier Transform (FFT) operations).

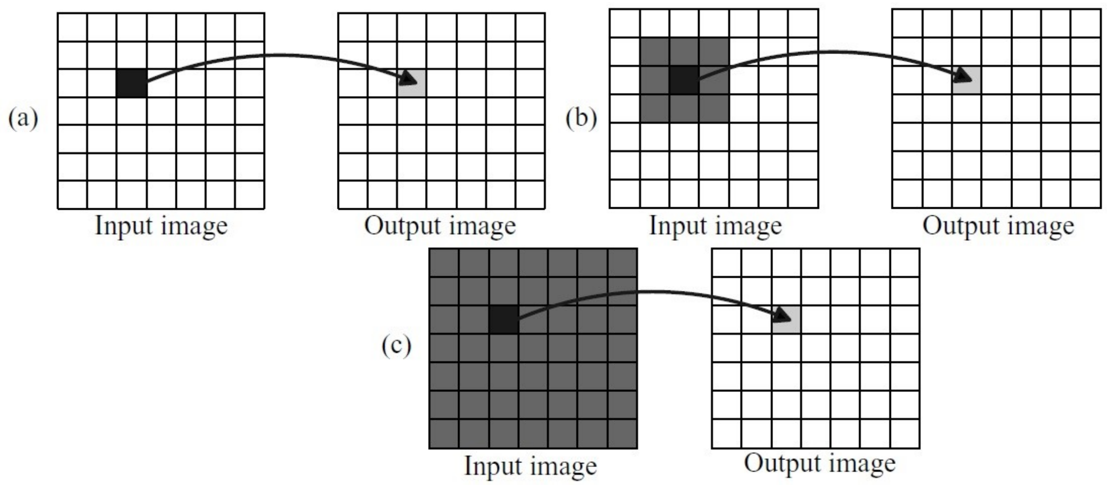


Figure 2.5: (a) Point Operation (b) Window Operation (c) Global Operation

There are also two types of image processing characterizations: spatial and temporal. Spatial characterization involves analyzing a single image (e.g. edge detection), whereas temporal characterization involves analyzing a series of images taken at different time instances (e.g. motion detection). The above discussed operations (point, window and global) are commonly used in these characterizations.

Table 2.2 summarizes the common classes of image processing algorithms that will be used to represent our use-case.

Classes	**Processing**	**Example**
Image Scaling	Algorithms are capable of resizing the image either by adding new pixels (upsample) between existing pixels or by removing pixels (downsample).	Bi-linear interpolation
Colour conversion	These algorithms alter the colour information in the image or enhance the colour information. Point based operations	Grayscale conversion
Filters	These algorithms are used to remove unwanted artifacts from the image resulting in smoothed, brightened image Window based operations.	Gaussian blur
Feature extraction	These algorithms are used to detect and isolate various desired portions of an image. Window based operations.	Canny edge detection

Table 2.2: Classes of image processing algorithms

2.2.2 Processing requirements

Since images are made out of pixels, it naturally lends itself to massive parallelism since each pixel can be processed independently. But when part or entire image (window and global) is used to compute an output pixel, we need to analyze the algorithm to exploit parallelism efficiently.

To adhere to real-time processing constraints, image data must be processed at high data rates. On CPUs and GPUs due to large memory sizes, data can be processed in frames at very high speeds. On FPGAs, due to resource constraints, data is usually processed as streams. This means that data is processed as soon as it is received, rather than waiting for the entire input image to be buffered before processing starts. Computations on a stream of data are performed by kernels, which are functions that operate on all elements of the input stream. Since stream elements are independent, kernels can operate in parallel exploiting data-level parallelism [10]. Moreover, in medical imaging, multiple algorithms are used to form the image processing chain. These can be implemented as multiple kernels running in parallel exploiting task level parallelism. A kernel can potentially take an arbitrary number of inputs and produce one or more output streams. Thus, generating the hardware in a streaming fashion would be beneficial to achieve a high throughput. Making the algorithm to work in a streaming fashion is a challenge since data needs to be buffered (memory considerations), and data order must be maintained.

2.2.3 Algorithm selection

To obtain a good representation of the image processing domain, we chose to implement Sobel and Gaussian filters. These filters are selected owing to their widespread use in the imaging domain. The operations performed by these filters can be extrapolated to other commonly used imaging filters thereby giving us a heuristic representation.

Sobel filter is a convolution filter that uses two kernels to calculate intensity changes in the horizontal and vertical direction. This information can be used to detect edges in the input image. The Gaussian filter is a low-pass filter that uses a single kernel to blur an image which reduces the noise in the input image. These two filters will serve as a good base case for our analysis of the tools and will help us understand portability challenges.

To obtain a good representation of a medical imaging algorithm, we looked at some of the commercial grade algorithms. First, most of the operations used in the algorithms are frame-based or pixel based. Since there are no global operations performed, the parallelization requirements on the workload are reduced because the entire image need not be buffered for the processing to start. Second, the image processing pipelines are often implemented in a pyramid fashion. A gray-scale image is used as an input to the imaging system. This image is then scaled down in multiple stages. In each stage, the image size is reduced by a factor of 2, thereby enabling working with a smaller dataset. The lowest resolution is then used to reconstruct the image in a pyramid fashion.

This is similar to the Multi Resolution Analysis (MRA) used in popular literature [11]. MRA is a mathematical method that is based on working on a problem at different levels of resolution. This method is currently being used in signal detection applications, PDEs solving, computer vision and image processing.

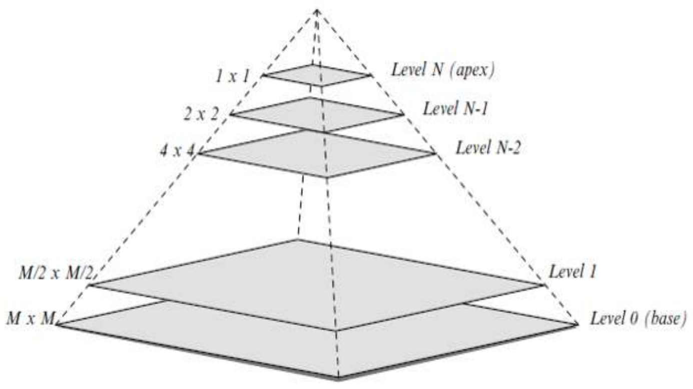


Figure 2.6: Image Pyramids [12]

Fig. 2.6 illustrates the MRA concept. As shown in the figure, the method starts at the full resolution (called the base) and continues to create a more coarse-grained representation of the data set. In each level, the same computational operations can be applied, affecting a different relative region size. Since they have the structure of a pyramid, they are also called image pyramids when used in image processing applications. Some advantages of using image pyramids are: First, they reduce computational costs of various image operations because the operations can be applied to a smaller dataset rather than the complete image. Second, pyramids enable image features to interact locally at higher levels of the pyramid, even though the features are far apart in the original image. Third, this algorithm can generate sets of low pass and band-pass filtered images at a fraction of the cost of performing Fast Fourier Transforms [11].

In MRA, at the input stage, downsampling is performed to scale down the image. Since downsampling introduces aliasing effects, the image is sent through a low-pass filter. This operation can be represented by the formula shown in Eqn. 2.1 for (2-D image). G_0 is the original image and G_N is the image at the top level of the pyramid.

$$G_l(i,j) = \sum_m \sum_n w(m,n) G_{l-1}(2i+m, 2j+n) \quad (0 < l <= N) \tag{2.1}$$

The weighing function *w(m,n)* is called the "generating kernel". The above formula can be simply represented as a "reduce" equation since each stage is scaled down by a factor of two. Reduce is a function that performs filtering and correspondingly downsamples the image.

$$G(l) = \begin{cases} I_0 & \text{if } l = 0 \\ Reduce[G_{l-1}] & \text{if } l > 0 \end{cases}$$

Since the weighing function resembles the gaussian density function, the decomposition phase is simply referred to as "Gaussian pyramid" in common literature.

A second operation "expand", is defined to be the inverse of reduce. It expands an image by interpolating sample values between the given values. Let $G_{l,k}$ be the image obtained by applying "expand" to G_l, k times

$$G_{l,k} = Expand[G_{l,k-1}] \quad given \quad that \quad G_{l,0} = G_l \tag{2.2}$$

Note that the gaussian filter is a low-pass filter that removes the high frequency components at each successive level.

Bandpass images can be quite useful for image analysis. They can be obtained by subtracting each gaussian pyramid level from the next lower level in the pyramid. Because the resolutions are different, it is necessary to interpolate to expand the image (Expand operation can be used here). Since these operations resemble laplacian operators, they are named as "laplacian pyramid". The image pyramid is then constructed by repeated application of reduce and expand operations as shown in Fig. 2.7.

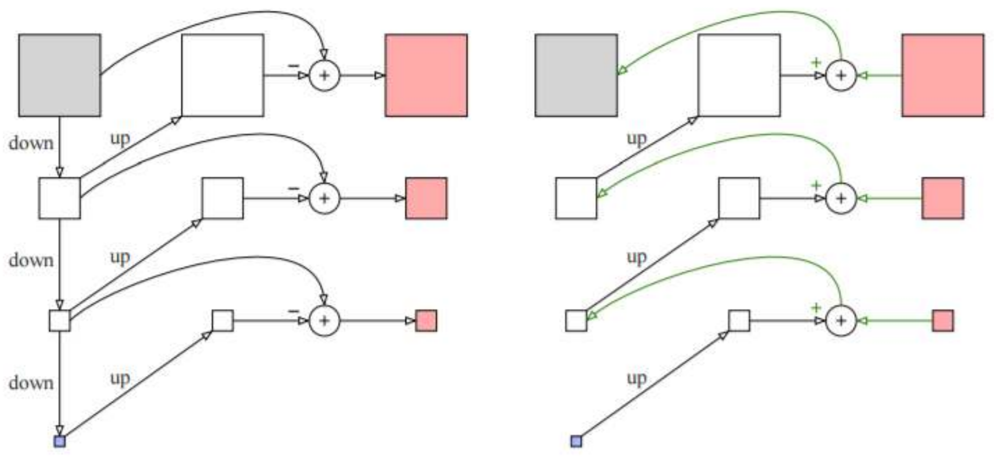


Figure 2.7: Gaussian and Laplacian pyramids [13]

Once we have the bandpass images from Laplacian pyramids, we can then use any filter to remove noise or extract information at different resolutions.

Analysis

The first insight we develop as mentioned above is that all algorithms used in the image processing pipeline use window based operations. This makes “parallel capability” analysis relatively easier since the entire image need not be stored and operated on (like global operations). Second, we see that the different stages can operate in a streaming fashion. This would allow concurrent execution and improve the arithmetic intensity of the kernel, thereby increasing the throughput. FPGAs offer massively parallel hardware architectures which can achieve best results with data streaming and pipelining. Each computational kernel in the MRA algorithm can be converted to hardware modules and laid out in parallel on FPGAs. The modules can then be interconnected by data streams to form a pipeline, through which data is streamed form one module to another. This structure can then effectively supply a continuous flow of output from a continuous delivery of input data achieving high throughout and low latency.

3 Software to Hardware

As discussed previously, designing hardware is time consuming and resource intensive. Moreover a completely different approach than traditional software programming is required, which introduces a steep learning curve for software programmers. This chapter deals with the challenges of moving to higher level languages [14] followed by a study in the current state of tools to help with our research.

3.1 Challenges

Today computing systems are designed as a mix of hardware and software. Compute-intensive and real-time problems are offloaded to hardware (Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs)) for speed and "timing predictability" advantages. Using Higher Level Languages (HLLs) for both architectures simplifies the work, but there are several challenges in moving to higher layers of abstraction for hardware programming. The following lists some of the challenges.

- **Concurrency**
  - Algorithms developed using HLLs are traditionally sequential. Concurrency can be exposed in higher-level languages through the use of so called "pragmas" that help the compiler in optimizing the code effectively. Even with these features, the programmer has to analyze the code carefully to understand where to place the pragmas.
- **Datatypes**
  - The base types in HLLs allow a minimum of one or more bytes (integer is usually 4 bytes) to be stored in memory. This is in contrast to hardware where single bit manipulations are performed frequently.
- **Timing**
  - The ability to specify detailed timing (clock cycles) is another fundamental requirement in hardware. Applications that should work real-time are usually implemented directly in hardware, and can lead to unwanted effects if timing requirements are not met. This is something that cannot be explicitly manipulated in HLLs.
- **Communication**
  - As discussed in Section 2.1, memory architectures of various platforms are quite different. When it comes to hardware design, the programmer can use

various techniques to build effective communication channels. In software programming, the programmer is faced with a limited set of interfaces.

- **Multiple design choices**
  - Every operation on hardware can be implemented in a variety of ways. E.g. multiplication on hardware can be implemented using DSP blocks (or) LUT and FF pairs. It all depends on trade-offs between area, cost and performance. Therefore, the translation process for hardware is more complicated than software.

3.2 High level synthesis

Elaborate research has led to the development of several tools that abstract the hardware and ease the development process on FPGAs [15][16][17]. The process of generating hardware from HLLs is termed as high-level synthesis in common literature.

High-level synthesis has been under research for many decades, and major vendors have come up with their own set of tools to achieve High Level Synthesis (HLS). This is an important direction of research since HLS will improve designed productivity and will make the use of FPGA technology viable for software programmers [12]. Since these tools predominately target FPGAs, let us look at some important requirements that must be satisfied for hardware generation on FPGAs [18].

- The implemented algorithm must use the resources properly. If the pipelined operations are not "speed matched", then the slowest operation will dominate the execution time. This is analogous to the problem of load balancing in a parallel computing model where, if a single thread operates on a large amount of data, then the other threads (in many applications) have to wait for its completion.
- Using appropriate precision for representing data allows a proportional increase in parallelism and hence performance. Moreover alternatives to perform floating point operations must be developed since these operations are costly in terms of area, time, and power.
- Timing is very important in FPGA designs (real-time processing) and timing constraints are to be considered as hard requirements.

3.3 RTL generation

This section briefly explains the HLS process for hardware generation.

The first step is identical to the traditional software flow model. It involves the compilation stage that performs common lexical and syntactic analysis that generates an Intermediate Representation (IR) which is an Abstract Syntax Tree (AST) that represents the source code. Several optimizations are applied to the IR for optimal FPGA mapping. IR is usually a Control/Data Flow Graph (CDFG) which is formed by combining a Control Flow Graph (CFG) and Data Flow Graph (DFG). CFG indicates the flow of control between basic blocks which are defined as a sequence of instructions which have a single entry point and which are executed until the end with a single exit point. If data dependencies exist between basic blocks then such dependencies are represented in a data flow graph.

The next step is the synthesis stage. The Register Transfer Level (RTL) design specifies the exact timing, and moreover data and operations are mapped to concrete hardware units. To implement the final digital system, HLS has to solve the following tasks:

3.3.1 Scheduling

In this step, all operations of the input CDFG are mapped to control steps. A schedule will be generated such that all the data and control dependencies are not violated and performance constraints are satisfied. Since scheduling determines the operation sequence, it affects the degree of concurrency of the resulting design. All operations that map to the same control step are executed in parallel thereby exploiting Instruction Level Parallelism (ILP). Different types of scheduling algorithms (As Soon As Possible (ASAP), As Late As Possible (ALAP), list scheduling) are used based on different requirements such as high performance or low resource usage.

3.3.2 Allocation and binding

After scheduling, hardware resources are bound to operations. Allocation determines the type and number of hardware resources for a given design. In the binding stage, the different operations in the application are mapped to individual cores from technology libraries. Optimizations on the number of hardware resources and registers required to execute the functions are performed. E.g., arithmetic units (Adder, multiplier) can be shared if two operations are not executed in the same clock cycle. One trade-off to note here is that even though resource sharing may reduce area usage, it may introduce delays (due to the usage of multiplexers) and increase the number of interconnects. HLS tools can be guided by programmer to customize resource allocation and binding to fit the user needs.

3.3.3 Controller synthesis

This step involves the derivation of the controller that sequences the design and controls the functional and storage units in the datapath. Finite State Machines (FSMs) are used as a basis to implement controllers for the design. First, the controller selection (single or hierarchical controllers) and the number of FSMs required is selected. Next, the controller generation stage decides whether to implement the FSM as a Moore or Mealy machine. Finally, the controller is implemented with registers holding the current state and combinational logic to generate the next state based on the machine model.

3.3.4 Optimizations

To obtain an optimal design, the programmer needs to expose high-levels of fine-grained as well as coarse-grained parallelism. The programmer can expose parallelism by understanding the underlying architecture and carefully exposing concurrent execution possibilities, but the idea is to allow the compiler to automatically find optimizations and apply them to generate effective hardware. Some of them are constant propagation, constant folding, loop transformations like unrolling, tiling, fusion, distribution, and strip-mining. Fig. 3.1 illustrates the process for RTL generation.

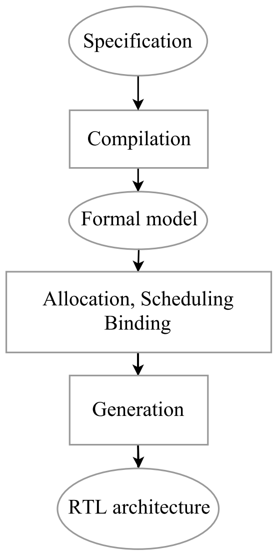


Figure 3.1: Front-end of HLS

After the RTL has been generated, the next phase is obtaining the bitstream to be uploaded on the FPGA. Fig. 3.2 shows the backend process. The first step performs lower-level synthesis. This results in a graph representation of hardware components such as gates and connecting signals called the netlist. The next step maps the netlist to hardware resources on FPGAs. Finally the hardware units are connected using the configurable routing resources in the "placement and routing stage". All the component and connection information is stored in a bitstream file which can be used to configure the FPGAs.

The separation of the synthesis process into frontend and backend can be attributed to easy generalization of frontend synthesis, whereas backend synthesis is bound to the technology. Frontend can be easily adapted to a different design environment, but backend has a short life cycle, because technologies change frequently.

3.4 Tools and techniques

We already discussed the various challenges involved in going from software to hardware. Extensive research has been done to address these challenges and many tools and frame-

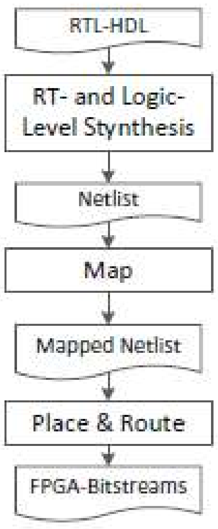


Figure 3.2: Hardware generation from RTL [12]

works have emerged in the market. In this section we will discuss and compare some of these tools to select suitable frameworks for our study.

The Altera OpenCL Software Development Kits (SDKs) [19] delivers a complete development to deployment solution for software programmers to design FPGA hardware using OpenCL language. To enable easy integration, the subsystem design is packaged into an OpenCL Board Support Package (BSP) and distributed along with the hardware. The SDK enables software emulation to verify the functionality of the design and also contains profiling tools for analyzing and debugging the design.

SDAccel [20] is a Xilinx framework that can accept OpenCL/C/C++ as input languages. This is also a complete development to deployment environment for hardware designs. We will discuss the framework in detail in Chapter 4.

Vivado HLS [21] is a Xilinx tool that accepts C/C++/System C as input languages to obtain hardware designs. It supports arbitrary-precision and fixed-point data-types using Xilinx libraries which is an advantage since the bit-width of compute variables can be customized resulting in reduced resource usages.

LegUP [16] is an open source tool developed at the university of toronto. LegUP accepts C programs as inputs, with constraints specified in a Tcl file. The tool can operate in two modes: pure hardware and hybrid. In hardware mode the input program is synthesized to a hardware circuit. In the hybrid mode, the program is synthesized, to target heterogeneous systems with a processor and an accelerator.

HIPAcc [12] provides a C++ based embedded DSL for the image processing domain. Several primitives are provided to implement imaging operations. The framework uses the clang/LLVM compiler infrastructure to generate an Abstract Syntax Tree (AST) that is operated on to produce a host code for managing kernels and device code for specific architectures (e.g CUDA, OpenCL). Recently, support for FPGAs was extended using the Vivado-HLS framework.

Halide [22] is also an image-processing DSL. It adopts a functional style description of image processing algorithms. The main feature of Halide is its separation between algorithm and schedule which will be discussed in detail in Section 3.6. They can target several architectures and recently, Halide was extended to support FPGA using the Vivado-HLS tool.

Table 3.3 summarizes the main features of the SDKs and Domain Specific Languages (DSLs) discussed above. Since exposing concurrency in the program is a fundamental requirement of HLS, these frameworks approach the problem in two ways. They either add parallel constructs to the programming language thereby forcing the programmer to expose concurrency or contain sophisticated compilers that automatically identify parallelizable functions in the source code. To enable working with fixed-point data these HLS tools provide features to specify the bit-width of variables. Design space exploration is also automatically performed based on tradeoffs specified by the user.

Although design tools start from a High-level language, they are rather a hardware description language than a high level tool. This is because of the additional features that are added to the language, that require hardware knowledge for effective utilization. Such HLS tools are given a low rating (-) for the level of abstraction they provide.

HLS tools that can automatically identify ILP and loop-level parallelism (loop unrolling, software pipelining) are marked with a high rating (++). A "+" rating is given for tools that cannot efficiently express parallelism in the user code.

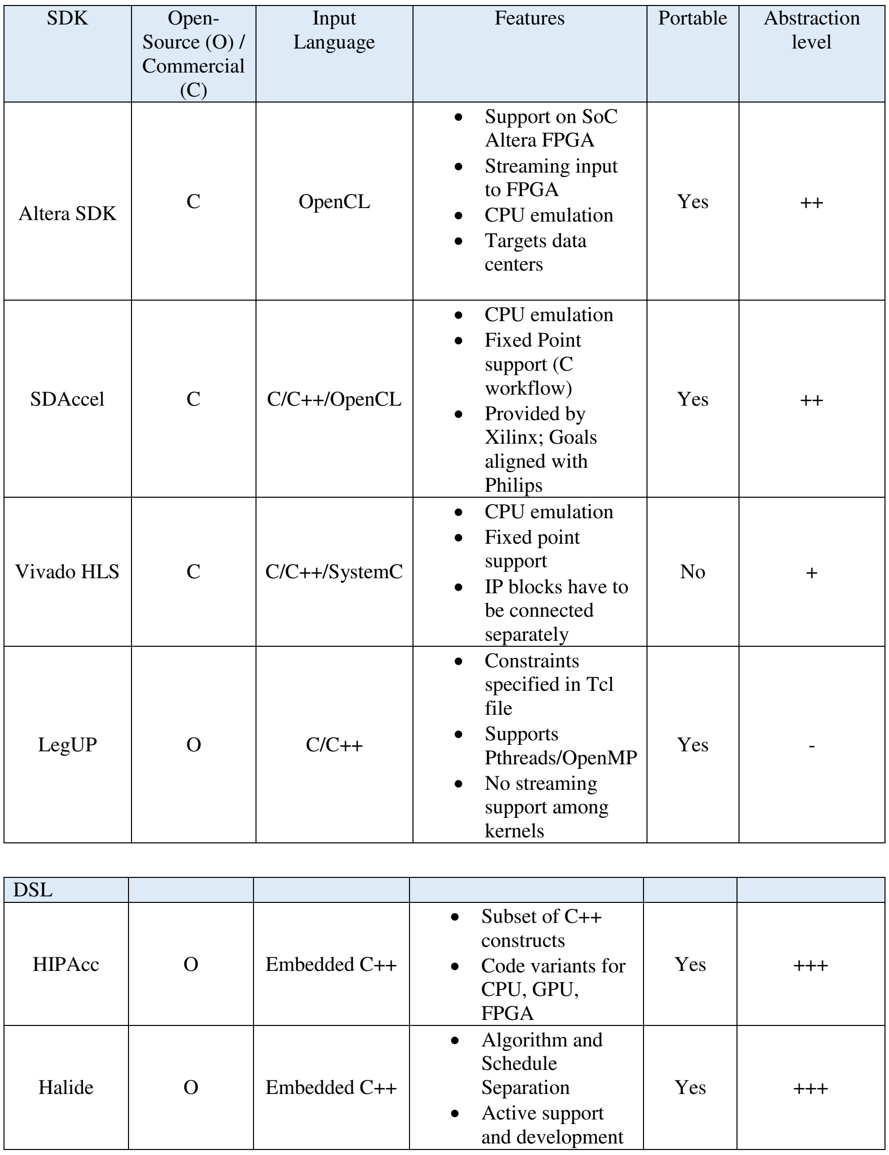
Advanced coding transformations for efficient data-reuse can increase the degree of parallelism. Tools which are created for specific domains with highly optimized instructions are given a very high level of abstraction rating (+++).

On studying the state of the art tools, we see that using Open Computing Language (OpenCL) as the base specification would be a good starting point because of its platform-independent model and active support by both industries and academia. Since OpenCL is a generic HLL, we also discussed about DSLs that can possibly perform better on our requirements. Halide and HIPAcc are two popular image processing DSLs. We chose to use Halide for this study because of its unique algorithm and schedule separation philosophy which will be discussed in the forthcoming sections.

3.5 OpenCL

The OpenCL standard is being developed by the Khronos Group industry consortium to address the challenges of programming multi-core and heterogeneous compute platforms. In 2013, Khronos released the OpenCL 2.0 specification in which a number of additional features such as nested parallelism, shared virtual spaces were added which simplify parallel application development and improves performance portability of applications. The OpenCL specification defines a single programming model that is supported by all hardware platforms conforming to the standard. The OpenCL specification is defined in three parts namely:

- Platform model
- Memory model



SDK	Open-Source (O) / Commercial (C)	Input Language	Features	Portable	Abstraction level
Altera SDK	C	OpenCL	• Support on SoC Altera FPGA • Streaming input to FPGA • CPU emulation • Targets data centers	Yes	++
SDAccel	C	C/C++/OpenCL	• CPU emulation • Fixed Point support (C workflow) • Provided by Xilinx; Goals aligned with Philips	Yes	++
Vivado HLS	C	C/C++/SystemC	• CPU emulation • Fixed point support • IP blocks have to be connected separately	No	+
LegUP	O	C/C++	• Constraints specified in Tcl file • Supports Pthreads/OpenMP • No streaming support among kernels	Yes	-
DSL					
HIPAcc	O	Embedded C++	• Subset of C++ constructs • Code variants for CPU, GPU, FPGA	Yes	+++
Halide	O	Embedded C++	• Algorithm and Schedule Separation • Active support and development	Yes	+++

Figure 3.3: SDKs and DSLs used for High Level Synthesis

- Execution model

3.5.1 Platform model

The platform model is defined by a combination of a host processor and one or more OpenCL compute devices. An OpenCL program always starts with a host processor. The host is responsible for managing the Operating System (OS), enabling drivers for all devices, setting up the global memory buffers, manage data transfers between host and device, and monitor the status of all the compute units in the system. The device is the hardware element on which the compute kernels of an OpenCL application are executed. An OpenCL kernel is often a compute-intensive function that the programmer wants to execute on an accelerator (OpenCL device). Each device is further divided into a set of compute units which are further subdivided into processing elements. A processing element is the fundamental computing engine in the compute unit, which is responsible for executing the operations of one work item. Fig. 3.4 illustrates the OpenCL platform model.

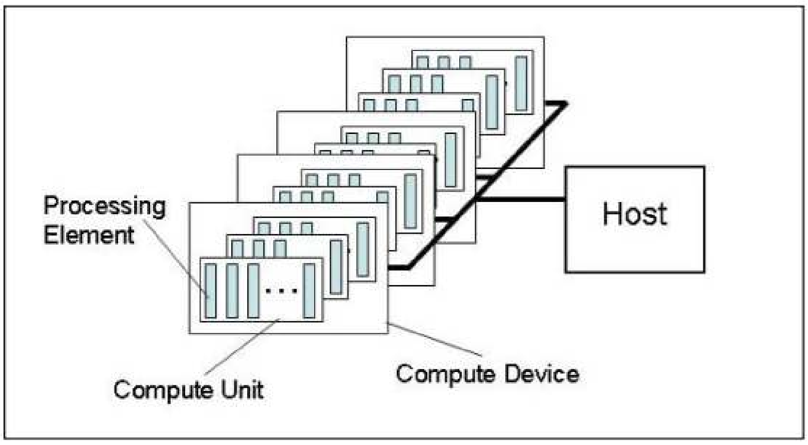


Figure 3.4: OpenCL platform model[23]

3.5.2 Memory model

As discussed previously, memory architecture varies widely between computing platforms. To overcome this challenge, OpenCL defines an abstract memory model that programmers can target when writing code, and that vendors can map to their actual memory.

3.5.2.1 Memory objects

OpenCL defines three types of memory objects namely: buffers, images and pipes. The memory for these objects are allocated using the host Application Programming interface (API). Buffers and images act as data storage that can be accessed by the host, whereas pipes serve as First in First Out (FIFO) objects between kernels and cannot be accessed by host.

- **Buffers**
  - With buffers, data elements are stored contiguously in memory. The OpenCL clCreateBuffer() API allocates space for a buffer and returns a memory object.
- **Images**
  - Images are multidimensional structures that are limited to a range of types. Image objects exist in OpenCL to offer access to "special function hardwares" on graphics processor that support highly efficient access to image data. Image objects are created using the clCreateImage() API.
- **Pipes**
  - Pipes allow transfer of data between kernels. It organizes data in a FIFO structure. Pipes are created using the clCreatePipe() API.

3.5.2.2 Memory regions

OpenCL divides device memory into four regions namely: Global memory, constant memory, Local memory and private memory.

- **Global memory**
  - Data in Global memory is accessible by both the host and device units. Transfer of data between the host and device takes place from the global memory. Global memory usually has the longest access time.
- **Constant Memory**
  - Constant memory is part of the global memory space that the host can read-write whereas the device can only read. It is typically used to store values that are used by all work-items (constant values).
- **Local Memory**
  - Local memory is a memory that is shared between work-items in a work group. Accesses to local memory has much shorter latency and much higher bandwidth than global memory. Operations on local memory are unordered between work-items but synchronization can be achieved using barriers.
- **Private memory**
  - Private memory is unique to a work-item. This memory is usually mapped to registers.

Fig. 3.5 illustrates the memory model

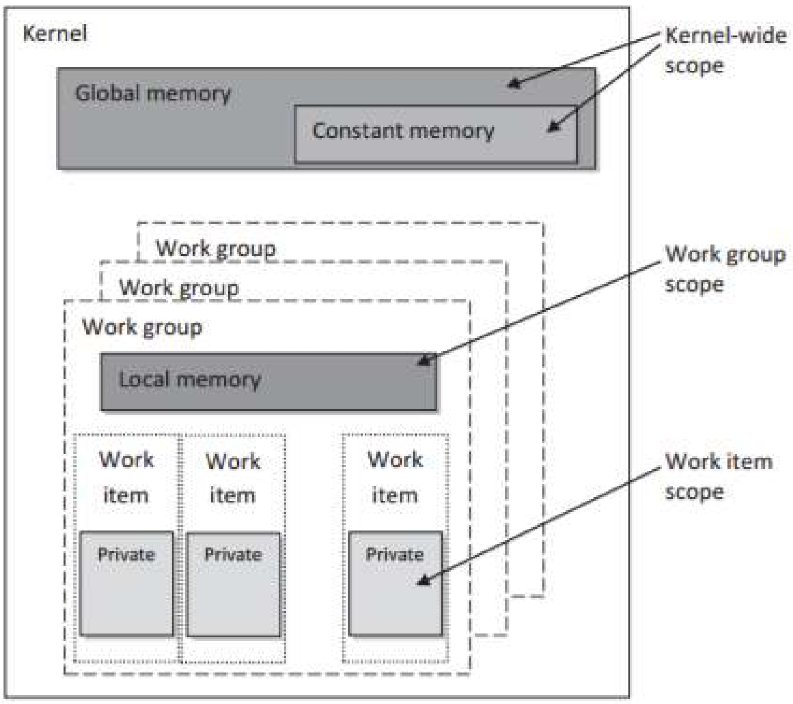


Figure 3.5: OpenCL memory model [23]

3.5.3 Execution model

The OpenCL execution model defines how kernels execute. A work-item is a unit of concurrent execution in OpenCL. Work-items map to processing elements and each work-item executes the kernel. When an OpenCL device begins executing a kernel, it provides intrinsic functions that allow the work-item to identify itself. OpenCL kernels execute within a predefined index space called N-dimensional range (NDRange) which can be one, two or three dimensional index space of work-items. The work-items are further divided into work groups. Work-items in a work group have access to a shared memory space as mentioned before.

Some key concepts in the OpenCL execution model are contexts, command queues and events. Contexts are used for managing all the objects specified for an accelerator device. Command-queues are the communication mechanism that the host uses to request action by a device. The host creates a command-queue for each device and submits commands to the proper command-queue. Events are used to specify dependencies between commands. Events also enable querying the execution status (Queued, submitted, ready, running, ended and complete) at any time.

3.5.4 Discussion

Since OpenCL is actively supported by many vendors and used commercially, it was chosen to accelerate the imaging algorithms on CPU, GPU, and FPGA. To achieve hardware generation from FPGA using OpenCL we have commercial tools from Altera and Xilinx. Since Philips and Xilinx have aligned goals and agreed to provide support, SDAccel was chosen as the development environment for this study.

3.6 Domain Specific Languages

In the previous section, we discussed HLS which is usually approached from generic programming languages (C/C++). Even though they provide a good abstraction over Hardware Description Languages (HDLs), the programmer still has to be aware of the lower-level details.

To address the programming and portability challenges DSLs provide higher levels of abstraction by utilizing domain knowledge and platform specific knowledge. The optimizations performed by DSLs are more involved than the optimizations performed by generic HLLs because of their targeting specific domains. Thus DSLs could be an attractive solution to mitigate portability and programmability challenges.

3.6.1 Halide

Halide [24] is an image-processing DSL designed to write high-performance image processing code on modern machines. Its front-end is coded in embedded C++ and targets a variety of hardware devices. Fig. 3.6 illustrates the compilation process for Halide language. Listing 3.1 shows a minimal example of a halide program. The algorithm

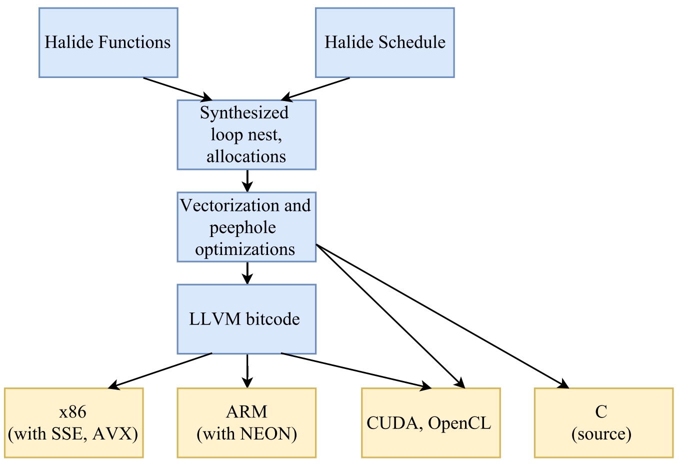


Figure 3.6: Halide Framework

description is coded in a functional style where images are pure functions that define the value at each point in terms of arithmetic operations. Different schedules can be explored to obtain optimal performance on target platforms.

```
Halide Algorithm:
blurx(x,y) = (in(x-1, y) +in(x, y) + in(x+1, y))/3;
blury(x,y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

Halide Schedule:
blury.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
blurx.comput_at(blury, x).store_at(blury,x).vectorize(x, 8);
```

Listing 3.1: Halide Description

3.6.2 Halide design philosophy

Writing efficient image processing implementation involves performing several platform specific optimizations on the algorithm which makes it complicated, unreadable and difficult to maintain.

Halide deals with this problem by separating the algorithm (what is computed) from the concerns of efficiently mapping to machine execution (decisions about storage and the ordering of computation) [22]. The choices of how to map an algorithm onto resources for a specific target platform is called the schedule.

This feature of separating the algorithm and schedule is interesting to our study because once the programmer has specified the algorithm, implemented it and tested the solution, a separate architecture expert can define the schedule without making modifications to the algorithm. In the medical domain, this will reduce the development cycle time thereby reducing development effort and costs. Design space exploration with different schedules can be performed quickly and easily with the assurance that the functional correctness of the program is not affected. Moreover, the Halide compiler can target different architectures ensuring performance portability. The above mentioned arguments provide compelling reasons to use Halide in our research.

A parallel research was conducted at Philips healthcare focusing on generating hardware structures on FPGAs from Halide. They analyzed a study [25] which proposed and implemented a Halide-HLS framework for generating image processing pipelines for heterogeneous systems (CPU-FPGA).

The research conducted at Philips looked at the limitations of the Halide-HLS framework and addressed some limitations. They used Vivado-HLS tool to synthesize the design on FPGA. In this study, we will look at adopting the framework for the SDAccel tool. The advantage of using SDAccel is the flexibility offered by the model to swap kernels dynamically on the OCL region allowing dynamic partial reconfiguration. Moreover, we will have a single entry point to design hardware, where as the Vivado generated Intellectual Property (IP) blocks have to be connected outside the tool.

3.7 Related study

HLS has been under research for several decades now. Elaborate studies [15][26][27] have been performed to generate hardware designs from HLLs. But as discussed previously, they mandate some knowledge of the underlying hardware architecture.

OpenCL is being actively developed by both academia and industries and recently it has been adopted to generate hardware structures on FPGA. Both Altera and Xilinx have developed their own set of tools to target FPGAs using OpenCL. One study with the help of Altera's SDK tools [28] shows that FPGA implementation offers about 5.5x speedup compared to Central Processing Units (CPUs) and Graphic Processing Units (GPUs) implementations for an information filtering algorithm. This takes into consideration that the algorithm itself can be deeply pipelined allowing multiple kernels to run in parallel which is an efficient fit for FPGAs. The programmer needs good knowledge of the underlying hardware to optimize such designs,

Streaming architectures can be implemented on FPGAs with elements of varying granularity. Different functions in a dataflow specification can be implemented on FPGAs to obtain high performances. OpenCL allows kernels to stream data without host intervention using the pipe semantic. In one study, the efficiency of pipe semantic was evaluated with Altera's SDK tools [13] for an image processing use case. The study proposes some methods to achieve better optimizations. One method is to use the global memory for data transfer while using OpenCL pipes for synchronization, and the second method is to build an OpenCL wrapper to efficiently overlap streaming data transfer and vision processing.

A previous study [29] conducted at Philips analyzed the Vivado-HLS tool to reduce the development time as compared to creating a manual RTL design. They estimated around 10 times decrease in the development time as compared to creating a manual RTL design. We will also compare their study with the SDAccel tool in Chapter 7.

DSLs combine domain-specific and platform-specific knowledge to obtain high-performance solutions. They offer higher levels of abstraction and studies have been performed to target FPGAs. HipAcc [30] and Halide are image processing DSLs that have compiler frameworks to generate RTLs designs that can be synthesized using commercial synthesizers. Active research is being pursued to use DSLs since they ease the FPGA development process.

To the best of our knowledge, the scope of related studies is limited to understanding the optimizations and performance benefits of using High-level languages to generate hardware structure on FPGAs. Even though these aspects are important for the Philips use-case, our motivation for using FPGAs are different. In this study, we mainly focus on the ease of development on FPGAs while maintaining portability to CPUs and GPUs with the help of Xilinx SDAccel tool and Halide programming frameworks.

In this chapter we discussed the challenges involved in developing code for FPGAs. We compared several tools and slected SDAccel and Halide programming framework to generate portable image processing implementations, and study the applicability of these tools to the Multi Resolution Analysis (MRA) algorithm. In the forthcoming chapters, we will propose a workflow using these tools, analyze them and provide conclusions.

Solution workflow 4

This chapter discusses the SDAccel environment in detail and also proposes a workflow to combine the halide framework and SDAccel tool to generate hardware on Field Programmable Gate Arrays (FPGAs).

4.1 SDAccel

The SDAccel Environment is a complete software development environment for creating, compiling, and optimizing OpenCL applications to be accelerated on Xilinx FPGAs [31]. SDAccel provides an environment for emulation on x86 based devices as well as deployment mechanisms for Xilinx FPGAs. It is a complete development environment from software development to deployment which is an advantage over other tools like Vivado HLS which only generates separate accelerators. All the concepts discussed in the previous section about OpenCL applies to Xilinx's OpenCL API. Some exceptions exist which will be documented later in this thesis. One notable difference from compilation on Central Processing Units (CPUs) and Graphic Processing Units (GPUs) is that the kernel code is always compiled offline in SDAccel. Just in time compilation of kernels is not supported in SDAccel due to long compilation process of generating bitfiles.

Fig. 4.1 shows the SDAccel environment. The x86-based server acts as the host and transfers data to the accelerators (FPGAs) through the PCI-e bus.

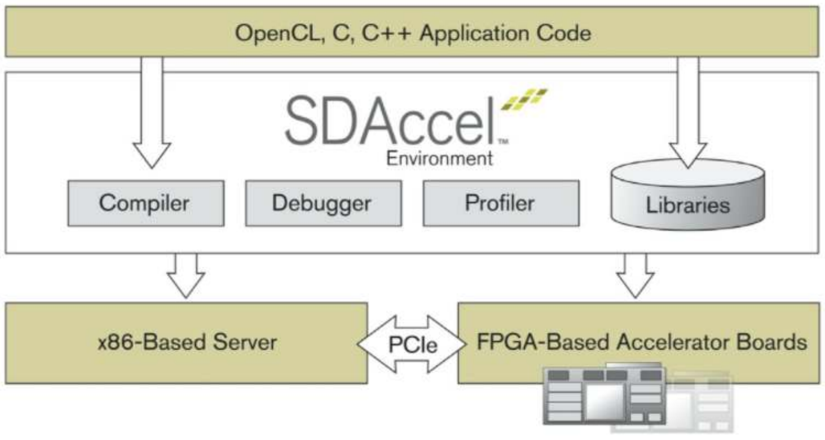


Figure 4.1: SDAccel environment [20]

4.1.1 Memory mapping

OpenCL memory specifications are mapped to FPGA as follows,

- Host memory is part of the host processor. In our case, it is x86 based CPUs.
- Global memories are usually SDRAM (outside the FPGA fabric) or BlockRAMs (within FPGA fabric). The host processor has access to these memories.
- Local and private memories are within the FPGA fabric and are typically implemented in BlockRAMs or registers.

Fig. 4.2 shows Xilinx's OpenCL memory mapping.

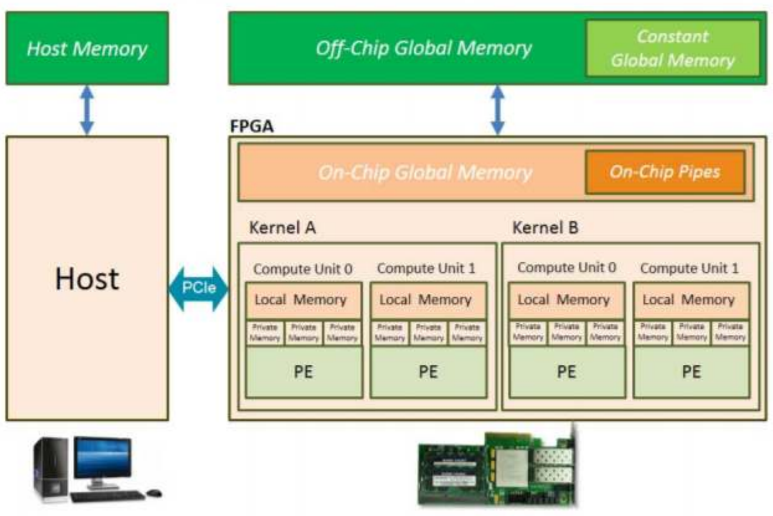


Figure 4.2: Xilinx-OpenCL memory mapping [31]

4.1.2 OCL region

SDAccel devices contain a dynamic reconfigurable area called the OCL region (Fig. 4.3). A reconfigurable area is a designated and physically constrained area on the FPGA. These areas are dynamic in an otherwise static FPGA implementation, meaning that bitstreams (functionality) for these areas can be swapped out without affecting the static part. For SDAccel devices the infrastructure is static, and kernels / compute units are dynamic. Both task-level parallelism (placing different kernels to operate on different tasks concurrently) and data-level parallelism (multiple identical kernels working on different data concurrently) can be exploited using this region. Thus hardware customization is possible, which allows the developer to leverage many opportunities which are not possible in fixed architectures like CPUs and GPUs.

4.1.3 SDAccel design

We mentioned that SDAccel supports only offline compilation. The reason for this decision stems from the fact that generation of optimized hardware architectures takes a longer time due to multiple design space explorations. While GPUs and CPUs support

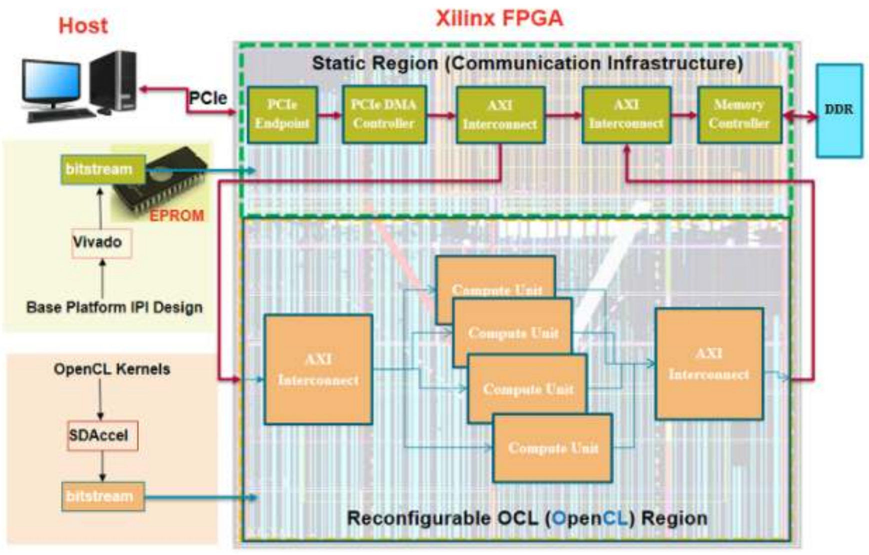


Figure 4.3: Xilinx-OpenCL region [31]

just-in-time compilation due to their fixed architectures, SDAccel tool exploits the offline compilation flow provided by the OpenCL standard. Thus binaries are pre-computed and loaded on to the OCL region. Xilinx has defined the Xilinx OpenCL compute unit binary format *.xclbin* that contains all the binaries of compute units (kernels) and decriptive metadata for compute units (Automatically generated by the tool).

Three compilation flows are supported by the SDAccel tool:

- CPU emulation: Test functionality
- Hardware emulation: Emulate hardware design; check performance
- System: Generate bitstream to implement custom hardware.

Xilinx has recommended certain optimization strategies [32] that will be followed when we design our kernels. The optimizations are summarized in Table 4.1. The fourth column indicates whether OpenCL standard officially supports the optimization.

SDAccel supports on-chip global cache and pipes which have been introduced in OpenCL 2.0 specification. The host does not have control of over these memories and they can be used to transfer data between kernels without host intervention.

4.2 Halide on FPGA

A parallel study was conducted at Philips healthcare to ease the development process on FPGA using the Halide DSL. They analyzed a study [25] which proposed and implemented a Halide-HLS framework for generating image processing pipelines for heterogeneous systems (CPU-FPGA). They found two limitations in the current Halide-HLS framework (1) The current framework does not support arbitrary dataypes (2) The

Optimizations	**OpenCL attributes**	**Description**	**OpenCL support**
Loop unroll	opencl_unroll_hint()	- Exposes concurrency to the compiler - Increases resource usage	yes
Pipeline	xcl_pipeline_loop	- Enables concurrent execution of different operations - Automatically added if loop trip count <= 64 for the main loop	No
Pipeline work-items	xcl_pipeline_workitems	Pipelines kernel work items	No
Dataflow	xcl_dataflow	similar effect as pipelining but the function level(coarse-grain)	No

Table 4.1: SDAccel Optimization Techniques

framework performs off-chip boundary handling which is not suitable for generating streaming architectures. They proposed solutions for these challenges, implemented and tested them using the Vivado-HLS tool. In this study, we will try to combine their solutions to generate a workflow using the SDAccel tool. The advantage in using SDAccel is that we will have a single entry point to design, test and generate hardware.

Fig. 4.4 shows the Halide-HLS compilation flow which was added to the tool by a study conducted at Stanford university [25]. The blue blocks are the the new additions to the framework to generate HLS-C code. Since several image processing algorithms are convolution based, all of the data required for computing a single result pixel can fit within a small memory block. The Halide HLS framework generate streaming pipelines with line buffers inserted between different stages.

4.3 Workflows

As discussed previously, SDAccel and Halide programming frameworks will be used in this study. Based on the analysis of our literature study, we propose the workflow shown in Fig. 4.5 The first workflow uses OpenCL as the base language and generates code to be

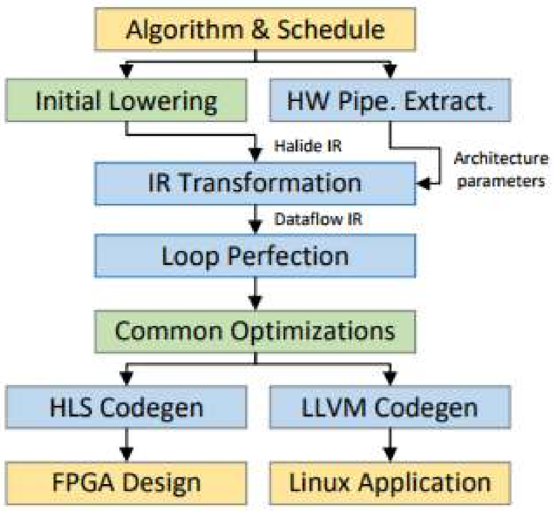


Figure 4.4: Halide compilation flow [25]

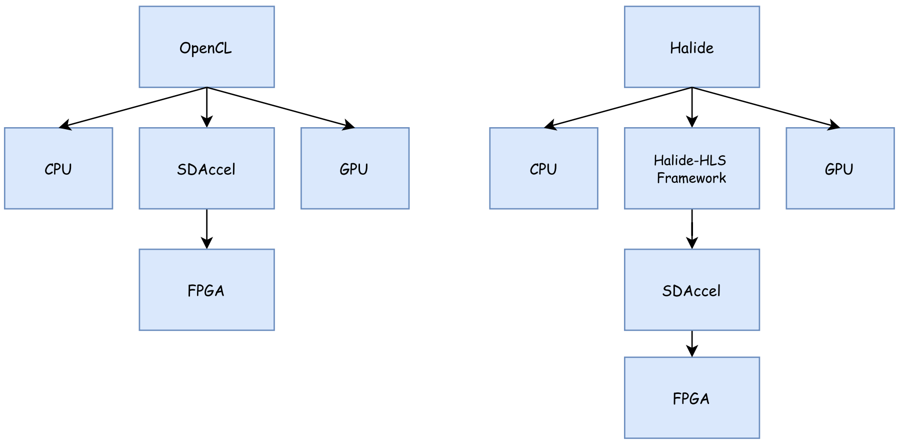


Figure 4.5: Solution Workflow

executed on CPU, GPU, FPGA. To execute the OpenCL binary on FPGAs we propose to use the SDAccel tool for reasons stated in Section 3.4. The second workflow uses the Halide-HLS framework to provide higher level of abstraction and uses the SDAccel tool to generate the final implementation on FPGA.

In order to generate efficient designs using SDAccel, Chapter 5 discusses the design choices that will be adopted while implementing our algorithms. Our main focus of using the SDAccel tool is to evaluate it for the Multi Resolution Analysis (MRA) algorithm and study the portability challenges.

Halide on the other hand has been already shown to be performance portable [24] on CPUs and GPUs. So in our second workflow we only evaluate the workflow for the FPGA platform.

4.4 Limitations and solutions

In this section, we identify the limitations of our workflows and suggest possible solutions.

4.4.1 SDAccel

SDAccel accepts C/C++/OpenCL as input languages and generates the RTL design. But, as discussed in previous chapters traditional software programming methodologies cannot be used to generate optimal hardware designs. Certain limitations in the current version of the SDAccel tool are detailed below.

4.4.1.1 Datatypes

OpenCL supports standard data types of 8, 16, 32 and 64 bits. Standard integer and floating point data types can be readily used in our design. Floating point designs provide the maximum accuracy, but consumes lots of resources and power.A solution to this challenge is use fixed point data types. Fixed point data types consume less power and resources, and can potentially produce same accuracy results as floating point for certain applications. However, in OpenCL we only use standard data-types. This means that the maximum bit-width we can use is 64 for the combination of integer and fractional part. This also means that, we cannot use arbitrary precision data types in our design. One possible way to handle this is to provide information to the compiler about the size of the variable.

```
int arbit_point = 0x3FFFF & ((in[i] & 0x1FFFF) + (in[i+1] &
                                                    0x1FFFF))
output = arbit_point & 0x3FFFF;
```

Listing 4.1: Data types precision

Listing 4.1 show addition of two integers [33]. If we already know the bit-width (in this case 17 bits) then we can mask the upper 15 bits. Since addition of two 17-bit numbers cannot exceed 18-bits we do a final mask with the result variable. This should produce 17-bit addition hardware instead of the full 32-bit addition. This solution works only for unsigned numbers and produces a lot of clutter. Hence, this method was not further pursed in this study.

Since this study focuses less on resource usage, the OpenCL designs were implemented with standard integer datatype. It is trivial to convert them to floating-point, but designs might exceed the maximum resource limit.

In this study, the MRA algorithm was developed in two separate versions. The first version uses C/C++ code and the second version uses OpenCL code. For the C/C++ version, optimized IP blocks using fixed-point datatypes was designed as part of the Almarvi project [29]. We will incorporate the fixed-point point version in SDAccel by using the Xilinx fixed point library. For the OpenCL version, standard integer datatypes are used.

4.4.1.2 Streaming architecture

Most of the image processing operations are convolution based. This means that the outputs can be calculated in a streaming fashion as soon as enough data is available (need not wait for the entire image). However, SDAccel does not support streaming inputs. To emulate streaming behavior we decided to load the image into Dynamic Random Access Memory (DRAM), split the design into multiple kernels and use pipes to stream data among them. Fig. 4.6 illustrates this process.

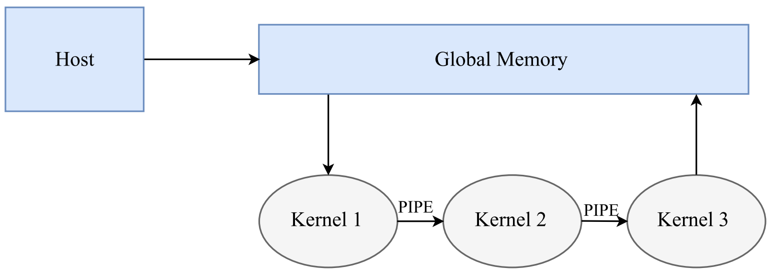


Figure 4.6: Streaming process in the device side

4.4.2 Halide

In our study, we use the Halide-HLS framework to generate synthesizable code and use SDAccel to obtain RTL designs. However, we have to ensure that the synthesized code is SDAccel compatible.

The generated HLS coded consists of an input stage, compute stage and an output stage. The input stage accepts data in a custom template function. To ease the data transfer from the host, we use standard data-types and convert them to the custom template at the input stage of the Halide-HLS code. To this end, a HLS synthesizable C++ template function is introduced in the generated code. The details will be discussed in Chapter 5.

Implementation 5

In this chapter, we first discuss the design choices made during the implementation phase. Second, the implementation of the Imaging filters to evaluate the portability challenges is discussed. Finally, the implementation of the Multi Resolution Analysis (MRA) algorithm using the SDAccel framework is detailed.

5.1 Design choices

In this section we will detail some of the design choices made in the implementations and explain the reasons for choosing them.

5.1.1 GPU

To ease the development of host code a helper library with OpenCL functions was written and documented.

A direct conversion from algorithm to OpenCL code is performed for the Sobel filter without considering any platform specific optimizations and executed on CPU, GPU, and FPGA. Different versions of the naive Sobel implementation are implemented on SDAccel using the optimizations discussed in Chapter 4. Xilinx has provided an optimized Sobel filter implementation for SDAccel which will be used in our research. This application will be modified to execute on CPUs and GPUs.

Next, the optimization of Gaussian filter on Graphic Processing Unit (GPU) using OpenCL will be performed based on the following checklist,

- Data transfer optimization.
- Datatypes optimization.
- Vector Operations.
- Hardware computation support.

5.1.1.1 Data Transfer

Inefficient memory transfers between the host and the device can become a major bottleneck to the whole design. GPU vendors (Nvidia, Intel, AMD) recommend using pinned memory (non-pageable memory) to achieve high data transfer rates. OpenCL does not guarantee pinned memory allocation, but Intel and Nvidia have provided guidelines to possibly achieve pinned memory allocation.

In Open Computing Language (OpenCL) the Application Programming interfaces (APIs) available for data transfer are detailed below:

- ClEnqueueReadBuffer(**)
- ClEnqueueWriteBuffer(**)
- ClEnqueueMapBuffer(**)
- ClEnqueueUnmapMemObject(**)

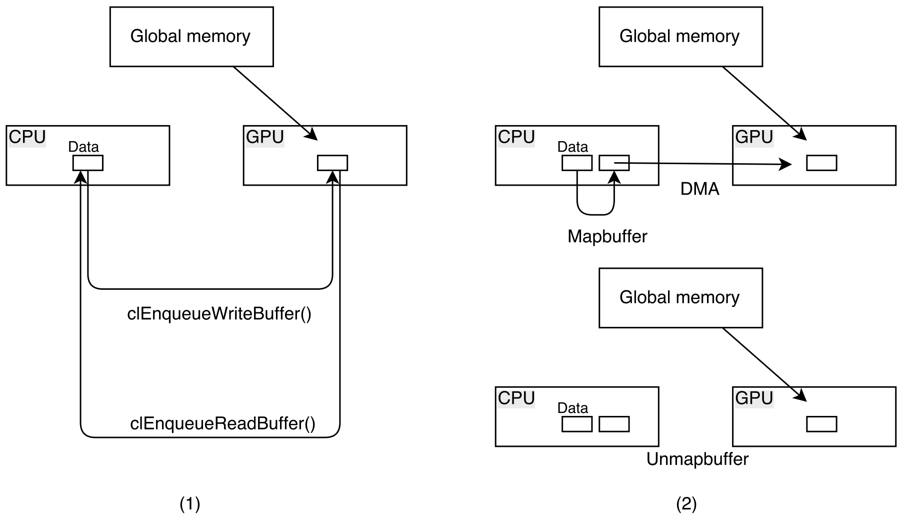


Figure 5.1: OpenCL Data Transfer (1) Read/Write buffers (2) Map/Unmap

Fig.5.3 shows the way data transfer is handled by OpenCL APIs. In (1), buffers are created on the device and data is transferred explicitly from the host to the device which is an inefficient way to transfer data. In (2) OpenCL allows the programmer to specify that a memory object should be allocated in "Host-accessible memory" using map/unmap buffers. This can be done by specifying `CL_MEM_ALLOC_HOST_PTR` or `CL_MEM_USE_HOST_PTR` flags while creating buffers. `CL_MEM_ALLOC_HOST_PTR` will automatically align buffers to achieve high data transfer rates. `CL_MEM_USE_HOST_PTR` flag requires the user to specify the boundary alignment. Intel recommends buffers allocated at a 4096 byte boundary and a total size that is a multiple of 64 bytes (cache line size). Xilinx also recommends using `CL_MEM_USE_HOST_PTR` to achieve high data transfer rates. The second scenario is efficient when system-on-chip devices (Integrated GPUs) are used, since they have a common shared memory between them. This could potentially lead to achieving zero costs on data transfers since data can be directly mapped and unmapped from the shared memory.

5.1.1.2 Datatypes

Choosing the right datatype for applications is important to attain the required precision and performance. Using the double data-type would provide the highest precision, but the application would become computationally intensive and consume more power and resources.

GPUs generally provide better single precision floating-point performance than integer or double performance because of dedicated floating point hardware units. Single precision performance is 441.6 GFLOPs whereas double precision stands at 110.4 GFLOPs [34]. Thus, using floating point computations on GPUs would provide better performances.

5.1.1.3 Vector operations

OpenCL supports vector data-types. Vector data-types are defined with the type name (int, char etc.) followed by a value which determines the number of elements. Since SSE and AVX instructions use vector registers, the programmer can effectively parallelize (SIMD) computations to achieve high speed-ups. Moreover, the Intel HD graphics has SIMD units that can execute up to four 32-bit floating-point operations per cycle or eight 16-bit integer operations [35]. Thus, using vector data-types would give us a natural advantage over using scalar data-types.

Xilinx also recommends using 32-bit 16 elements vectors for achieving high data transfer rates.

5.1.1.4 Built-in hardware computation support

Intel recommends using built-in functions like *mad* (multiply and add) instruction because of hardware support which would increase the performance. Xilinx also provides the optimized HLS math library [36] to be used in SDAccel designs.

5.1.2 Halide

The Halide-HLS framework generates streaming pipeline designs. In SDAccel, the integration is performed using the Dataflow model. The data transfer stage between the host and device is separated from the computation stage and the three stages are combined using the dataflow model.

5.1.3 SDAccel

To evaluate the tool comprehensively for the MRA algorithm, we make two structural design choices.

5.1.3.1 Dataflow model

Dataflow models are used to express parallelism at a coarse-grain level (function level). Fig. 5.3 illustrates the advantage of using a dataflow model as opposed to a sequential model. We see that designing the algorithm in a dataflow model can help us achieve better latency estimates. SDAccel allows us to specify such a model at the higher level by using the pragma `"pragma hls dataflow"`.

In this model, the generated HLS functions are connected with First in First Out (FIFO) memory circuits providing data-level synchronization. In such a scenario the wait time of the consumer functions to obtain new data for processing is called Initiation

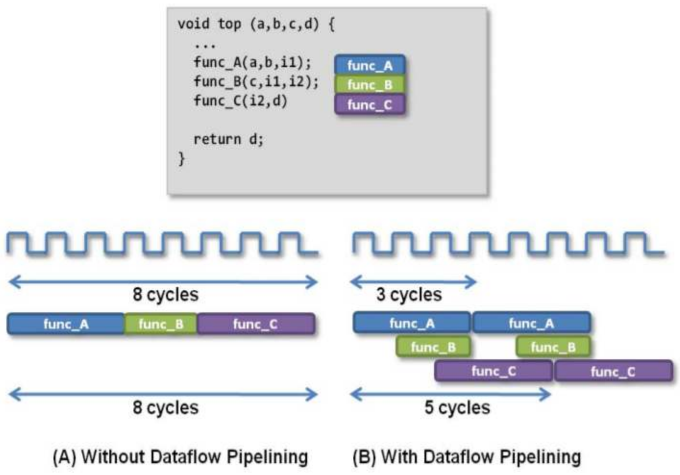


Figure 5.2: Dataflow model [37]

Interval (II). In our C design we will merge all the different stages in the MRA application into a dataflow model to achieve a low II.

5.1.3.2 Pipe objects

In Section 3.5 we discussed memory objects called pipes which can be used to transfer data between kernels (no host intervention). Pipes can be used in SDAccel using the standard OpenCL functions: `read_pipe()` and `write_pipe()` in non-blocking mode. Xilinx also provides `read_pipe_block()` and `write_pipe_block()` functions for blocking mode operations allowing automatic data synchronization between producer and consumer. In the OpenCL version the filters will be designed as seperate kernels and pipes are used to transfer data between them allowing concurrent kernel execution.

5.2 Portability

In this section, we discuss the implementation of Sobel and Gaussian filters to understand the portability challenges of SDAccel framework.

5.2.1 Sobel kernel

Listing 5.1 shows the Sobel kernel implementation using OpenCL. The code was executed with as many threads as the size of the input image. Boundary conditions were handled within the kernel code by repeating the edges of the image on all sides. Each thread computes the output of a single input pixel allowing parallel execution. A 960×960 image is taken as an input to the Sobel kernel and executed on CPU, GPU and FPGA (using SDAccel). To analyze the performance on SDAccel, two versions detailed below with some code modifications were developed on the SDAccel framework.

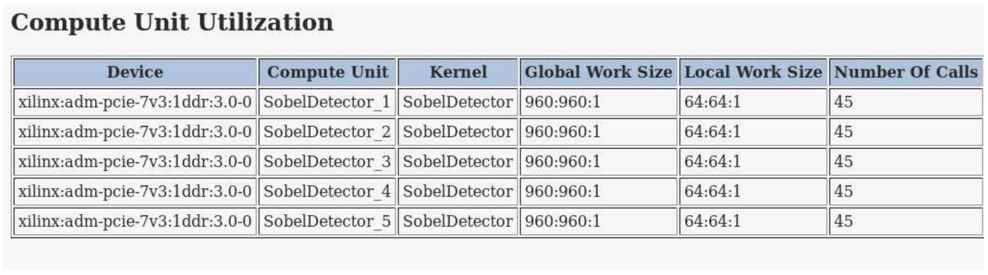
```
        int Gx = input[xx + yy * width]
                        + 2 * input[x + yy * width]
                        + input[xx1 + yy * width]
                        - input[xx + yy1 * width]
                        - 2 * input[x + yy1 * width]
                        - input[xx1 + yy1 * width];

        int Gy = -input[xx + yy * width]
                        + input[xx1 + yy * width]
                        - 2 * input[xx + y * width]
                        + 2 * input[xx1 + y * width]
                        - input[xx + yy1 * width]
                        + input[xx1 + yy1 * width];
```

Listing 5.1: Sobel kernel OpenCL

5.2.1.1 Design space exploration on SDAccel

The input image size is 960×960. Therefore the image can be split into multiple workgroups and all the work items in each workgroup can be pipelined. A local size of 64×64 is chosen which means that we have a total of $\frac{960}{64} \times \frac{960}{64} = 225$ workgroups. Listing 5.2 shows the modified Sobel filter implementation. The global index is mapped to the local index and the computations are pipelined using the "xcl_pipeline_workitems" attribute.

SDAccel implements the kernel as one single compute unit which does not make use of task or data-level parallelism since the computations are performed sequentially. Therefore multiple compute units are instantiated on the FPGA by modifying the compilation parameters. The code shown in Listing 5.2 is reused, but five compute units are started instead of a single compute unit. Fig. 5.3 shows the workgroup split that SDAccel performed for the Sobel filter. Each compute unit now works on forty five workgroups.


Compute Unit Utilization

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls
xilinx:adm-pcie-7v3:1ddr:3.0-0	SobelDetector_1	SobelDetector	960:960:1	64:64:1	45
xilinx:adm-pcie-7v3:1ddr:3.0-0	SobelDetector_2	SobelDetector	960:960:1	64:64:1	45
xilinx:adm-pcie-7v3:1ddr:3.0-0	SobelDetector_3	SobelDetector	960:960:1	64:64:1	45
xilinx:adm-pcie-7v3:1ddr:3.0-0	SobelDetector_4	SobelDetector	960:960:1	64:64:1	45
xilinx:adm-pcie-7v3:1ddr:3.0-0	SobelDetector_5	SobelDetector	960:960:1	64:64:1	45

Figure 5.3: Sobel multiple compute units

5.2.2 Gaussian kernel

A naive 3×3 Gaussian filter was implemented in OpenCL and iteratively optimized for GPU using the Intel optimization guide [38] and the steps detailed in Section 5.1. The

```
__kernel __attribute__((reqd_work_group_size(64,64,1))) void
   SobelDetector(__global int* input, __global int* output){
//local memory
__local int  local_workset[64*64];

//Get the global index
        int x1 = get_global_id(0);
        int y1 = get_global_id(1);
//Get the local index
        int x = get_local_id(0);
        int y = get_local_id(1);

        int index1 = x1 + y1 * 960;
        int index2 = x + y * 64;

//Handle boundaries
        int xx =  max(0, x - 1);
        int yy =  max(0, y - 1);
        int xx1 = min(width - 1, x + 1);
        int yy1 = min(height - 1, y + 1);

//Get the input and pipeline workitems
 __attribute__((xcl_pipeline_workitems)){
        local_workset[index2] = input[index1];
}
  __attribute__((xcl_pipeline_workitems)){
        //kernel computation
        }
}
```

Listing 5.2: Sobel pipelined

optimization steps are discussed below,

5.2.2.1 Data transfer

In section 5.1 we explained the benefits of using map/unmap functions to achieve high data transfer rates in integrated GPUs. Listing 5.3 shows the usage of map and unmap functions. Once the data is mapped on the host, it is immediately unmapped to enable the device to use the memory region.

5.2.2.2 Vector datatypes

Vector-datatypes can be used to perform parallel computations with SIMD registers. Moreover, the data-transfer is optimized since multiple elements are transferred packed in one vector variable.

Listing 5.4 shows the range on which the kernel is started along with the kernel arguments.

```
void* data = clEnqueueMapBuffer(Setall.command_queue, buf,
    CL_FALSE, CL_MAP_WRITE, 0, size, 0, NULL, NULL, &err);
clFinish(Setall.command_queue);

int err = clEnqueueUnmapMemObject(Setall.command_queue, buf,
    (unsigned char*)map_obj, 0, NULL, NULL);
clFinish(Setall.command_queue);
```

Listing 5.3: Data transfer

```
cl_ulong duration = PCL_Run(Setall, krnl, (input_image.cols/16),
    (input_image.rows/16), 1);
__kernel void gaus(__global uchar16* input, __global uchar16*
    output)
```

Listing 5.4: Vector data types

5.2.2.3 Floating point operations

Floating point computations are accelerated on Intel GPUs. Therefore the computations are performed by converting to floating point data-types. Modern Intel CPUs supporting advanced vector extensions are also capable of accelerating floating point computations. Moreover, the kernel computations are performed in a loop with 16 iterations, increasing the workload of each thread. Finally, The *restric* qualifier is applied to the kernel arguments which helps the compiler minimize pointer aliasing. The optimized Gaussian filter was then run on CPU, GPU and FPGA without further platform specific optimizations. The comparisons are performed in Chapter 6.

5.3 Halide

The Sobel and Gaussian filters were written in Halide and compiled with the Halide-HLS framework to generate the HLS-C code. The generated code used custom templates in its computation. Since the data is transferred using standard data-types (int, floats), we need to convert them to the custom templates. The conversion should not affect the streaming flow of input data maintaining an initiation interval of 1. To this end, an HLS C++ synthesizable template function shown in Listing 5.5 is introduced at the input stage of the generated HLS-C output . In the SDAccel environment, the data is transferred to a separate stage and streamed to the Halide generated HLS-C output. The data is collected in a separate output stage and sent back to host. To achieve streaming computation all the stages are connected in a dataflow model. The results are compared with the original implementation executed with Vivado-HLS tool.

```
template < typename T, typename T_out , size_t IMG_EXTENT_0 ,
    size_t IMG_EXTENT_1 ,size_t EXTENT_0 , size_t EXTENT_1 , size_t
    EXTENT_2 , size_t EXTENT_3 ,>
void inToStencil(hls::stream<T> &in_stream ,
hls::stream<PackedStencil<T_out , EXTENT_0 , EXTENT_1 , EXTENT_2 ,
    EXTENT_3> > &out_stream){
//
}
```

Listing 5.5: Type conversion

5.4 MRA

We discussed the structural design choices for the MRA application in Section 5.1 for the C/C++ and the OpenCL version. Here, we will discuss the optimizations performed in the individual design blocks. Fig. 5.4 illustrates the MRA design implemented in this study.

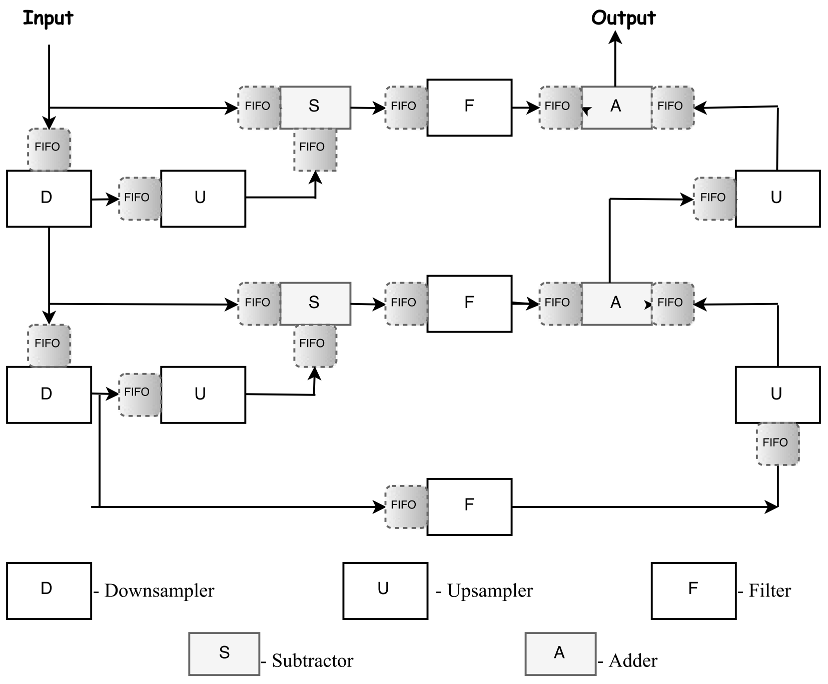


Figure 5.4: MRA model

5.4.1 Buffers

All the kernels in the MRA algorithm are implemented as streaming models. Therefore, FIFO buffers are placed at the interfaces as shown in Fig. 5.4. When these buffers are full, they block and data stops propagating in the network. This phenomenon is called *back pressure*. Insufficient buffer sizes can lead to deadlocks resulting in subpar hardware designs. In our MRA design, we experimented with different buffer sizes and obtained a deadlock free design. Please note that this may not be the most optimal design. To obtain optimal FIFO buffer sizes for a deadlock free design, a study is being conducted in the Almarvi project to automatically obtain the buffer sizes. Therefore this process can be automated in the near future.

5.4.2 Work group sizes

Providing workgroup sizes is an important requirement in OpenCL based designs. In our MRA design, we chose workgroup sizes of (1,1,1). The reason for choosing this size is detailed below.

In Listing 5.6, the programmer has more control over design choices like pipelining, loop unrolling, and can help the compiler optimize designs effectively. In Listing 5.7, the overhead of creating multiple workgroups adds to the resource usage and latency. In Listing 5.8, since the work-group sizes are not specified, the tool creates a generic hardware design leading to increased resource usage. Table 5.1 shows the resource usage and latency differences in choosing different workgroup sizes. In the third implementation, the tool is not able to determine the expected latency due to unspecified work-group sizes. This experiment shows that optimizing the design with work-group size of 1 provides the best results in SDAccel.

```
__kernel void __attribute__((reqd_work_group_size(1,1,1))
swg(__global int* in, __global int* in1, __global int* out){
        for(int i = 0; i < 4096; i++)
                out[i] = in[i] + in1[i];
}
```

Listing 5.6: Implementation one

```
__kernel void __attribute__((reqd_work_group_size(4096,1,1))
swg(__global int* in, __global int* in1, __global int* out){
out[get_global_id(0)] = in[get_global_id(0)] +
                        in1[get_global_id(0)];
}
```

Listing 5.7: Implementation two

```
__kernel
swg(__global int* in, __global int* in1, __global int* out){
out[get_global_id(0)] = in[get_global_id(0)] +
                        in1[get_global_id(0)];
}
```

Listing 5.8: Implementation three

Resources and Performance	Implementation 1	Implementation 2	Implementation 3
BRAM	2	2	2
DSP	0	0	14
FF	1566	1974	2600
LUT	1713	2219	2800
Latency	0.07 ms	0.127 ms	Latency undefined

Table 5.1: Workgroup sizes effect on resource usage

```
int lineBuffer[3][480]
    __attribute__((xcl_array_partition(block,3,1)));
```

Listing 5.9: Memory partition in SDAccel

5.4.3 Memory partitioning

To increase the memory bandwidth, the physical layout of memories is partitioned using certain pragmas and attributes. There are three types of memory partitioning in the tool chain (1) block (2) cyclic and (3)complete.

Arrays are usually implemented as BRAMs. If the entire array is implemented as one BRAM, then there might be stalls in reading the data which can increase the initiation interval. Therefore at the cost of extra resources we can partition the array to be implemented in multiple BRAMs.

In our design, we use a 2D arrays to capture three rows of the image data in the downsampler. Listing 5.9 shows the process. We partition the rows of the array to obtain access to all three rows simultaneously.

5.4.4 Optimizing arithmetic computations

Multiplication and division operations can be implemented as simple shift operations to save resources. Listing 5.10 shows how the multiplication and division operations are transformed.

A left shift represents multiplication by 2 whereas a right shift represents division by 2. So to multiply by 3, we just left shift once and add the same pixel again. Such

```
//Normal Convolution kernel 1/8 [1 3 3 1]
output = (pixel1 * 1 + (pixel2 + pixel3) * 3 + pixel4 * 1)/8;

//Optimized
output = (pixel1 + ((pixel2 + pixel3) << 1) + pixel2 + pixel3 +
    pixel4) >> 3;
```

Listing 5.10: Arithmetic optimizations

operations are transformed to signal connections, costing no additional clock cycles.

5.4.5 Implementation

In Section 5.1, the design model of the MRA implementation was elaborated. SDAccel currently does not support streaming inputs, therefore the data transfer stage was separated from the computation stages. The image was transferred to an input stage which streams it to the compute stage, and the output stage collects the results and transfers them back to the host.

In the MRA algorithm the data required to compute an output pixel can be captured in a small memory blocks called linebuffers. Using linebuffers also enables re-use of shared data resulting in minimal data fetching.

5.4.5.1 Filters

In the downsampler, a 1×4 and a 4×1 kernel is used for the horizontal and vertical convolutions respectively. Therefore to start the horizontal convolution, we only need 4 input pixels. To start the first vertical convolution the first “3 rows output pixels” from the horizontal convolution are needed. Thus a 2D line buffer of 3 rows and 960 columns is required. Once the first vertical convolution is completed, the output is streamed to the next stages in the algorithm. One can already see the advantage of using such a small memory block, rather than buffering the entire image which will increase the latency and consume more resources.

The upsampler also works in the same pattern capturing the required workset in a line buffer. Here, only 2 pixels are needed to calculate the required output pixels for horizontal and vertical convolution. Thus, a linebuffer of 2 rows is sufficient to start the computations. The filter stages just emulate a convolution process, and perform no modifications to the input data.

5.4.5.2 C/C++

As part of the Almarvi project [29], fixed-point implementations of the downsampler and upsampler were developed in C/C++. As discussed previously, since SDAccel also supports C/C++ kernels, these implementations can be directly used by including the Xilinx fixed-point library.

The kernels are modelled in a streaming fashion using the dataflow model. It is important to note that each stream can only have a single producer and consumer. Fig. 5.5 shows the dataflow model generated by SDAccel.

5.4.5.3 OpenCL

Initially the horizontal and vertical convolutions in each filter were designed as separate OpenCL kernels. On examining the design reports of the generated hardware, it was found that each OpenCL kernel was connected to the master AXI bus even though no host intervention was required to transfer data between kernels. Since the number of OpenCL kernels on a virtex 7 FPGA was limited to 10, this design did not fit the hardware. Therefore a different FPGA was chosen, and the filter stages in the middle were removed to fit the design to hardware. The horizontal and vertical convolutions were combined in a single kernel.

The reason for performing this experiment is to study the the effort needed to generate such algorithms using OpenCL as the base language. The C/C++ version breaks portability when moving to different platforms, but if the design is programmed using OpenCL, then we can ensure functional portability across different platforms. Since the algorithm is programmed as separate kernels, the tool allows to add or swap kernels using the dynamic reconfiguration flow.

Host code modifications

To enable concurrent execution of different kernels, out of order command queues are started from the host side. Listing 5.11 shows the method to enable out of order command queue and launch the kernels.

```
//Creating out of order command queue
Setall.command_queue = clCreateCommandQueue(Setall.context,
    Setall.device_id, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE |
    CL_QUEUE_PROFILING_ENABLE, &err);

//Enqueue the kernels in the command queue and launch
clEnqueueTask(Setall.command_queue, in, 0, NULL, NULL);
clEnqueueTask(Setall.command_queue, out, 0, NULL, NULL);
clEnqueueTask(Setall.command_queue, krnl, 0, NULL, NULL);
clEnqueueTask(Setall.command_queue, krnl2, 0, NULL, NULL);
clEnqueueTask(Setall.command_queue, krnl3, 0, NULL, NULL);
clEnqueueTask(Setall.command_queue, krnl4, 0, NULL, NULL);
clFinish(Setall.command_queue);
```

Listing 5.11: Command queues

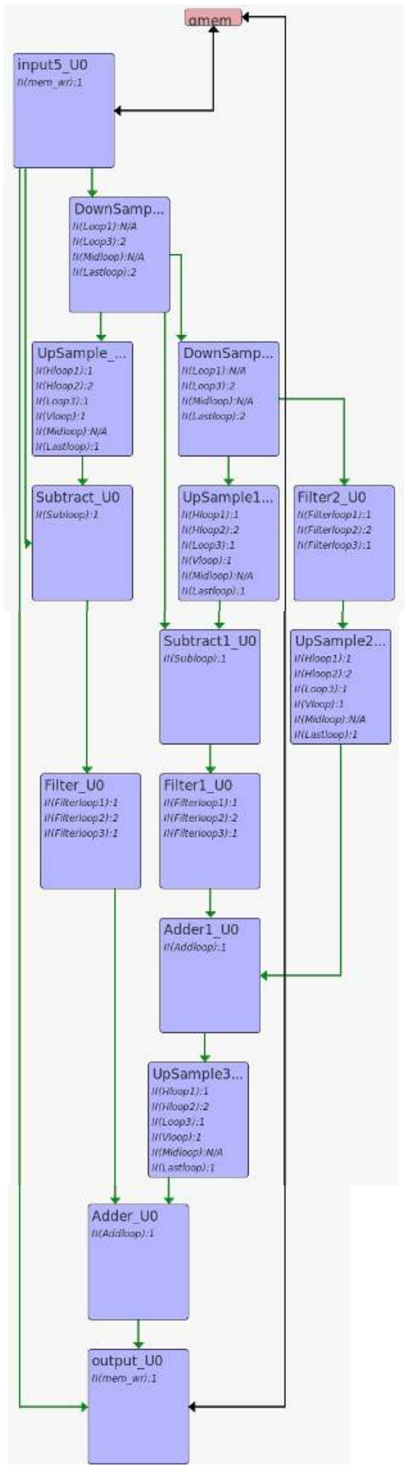


Figure 5.5: Dataflow view generated by SDAccel

Results 6

This chapter presents the results of the implementations described in Chapter 5. The evaluation methodology is described first, followed by comparisons and analysis of results.

6.1 Evaluation methodology

The CPU, GPU and FPGA devices used in this research are listed in Table 6.1. The final MRA OpenCL implementation is emulated on a different FPGA device for reasons stated in Section 5.4. For the latency estimates, the OpenCL kernels were pre-compiled and the offline compilation flow was used for CPU and GPU.

	CPU	GPU	FPGA	FPGA (OpenCL)
Name	Intel Core i7-6820HQ	Intel HD Graphics 530	Virtex 7 XC7VX690T	Kintex KU060
Features	2.70 GHz 4 cores 8 Threads	1150 MHz 24 EU 7 threads per EU	200MHz 2940 BRAMs 3600 DSP slices 866400 FF 433200 LUT	200MHz 2160 BRAMs 2760 DSP slices 663360 FF 331680 LUT

Table 6.1: Device details

To analyze the *portability challenges*, a naive Sobel filter is run on all three target platforms (CPU, GPU, and FPGA) and a Sobel filter optimized for FPGA is executed on CPU and GPU. To test the *functional portability* of the SDAccel tool, a Gaussian filter is optimized on GPU and run on FPGA using the SDAccel tool.

The Halide-SDAccel Sobel and Gaussian implementations are compared with Vivado-HLS OpenCV implementations. The results of the MRA implementation with SDAccel tool are detailed and analyzed.

6.2 Portability

The first comparison analyzes the portability of the SDAccel tool. Table 6.2 shows the performance obtained on different platforms for the naive and optimized Sobel implementations. In Section 5.2, we discussed the implementation of two versions of the Sobel filter on the SDAccel framework. Table 6.3 shows the resource utilizations on FPGA for the different Sobel implementations.

Sobel Filter - 960 × 960			
Application	**CPU**	**GPU**	**FPGA (Virtex 7)**
Naive	0.65 ms	0.68 ms	319 ms
Optimized for FPGA	4.8 ms	300 ms	1.4 ms

Table 6.2: Sobel Filter - Latency Estimates

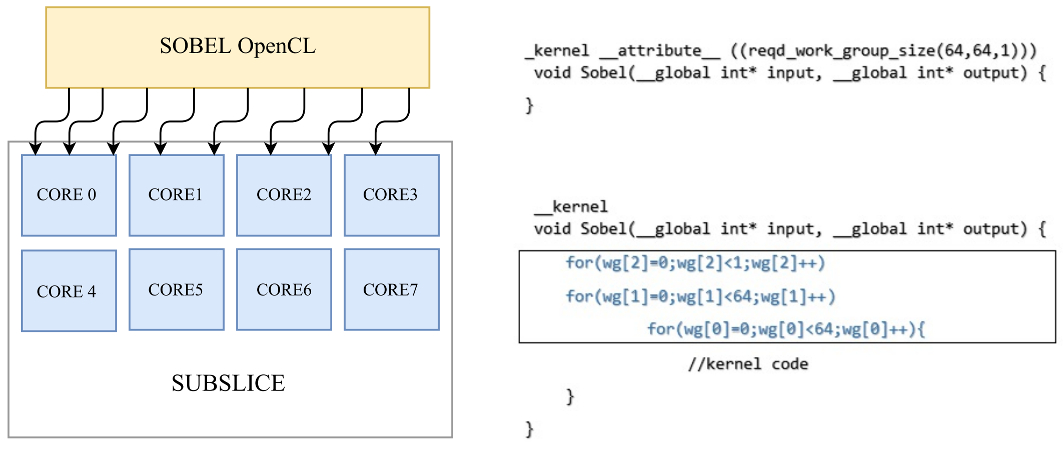
Image Size	960 x 960								
Target Frequency	200 MHz								
Application	**Resource Utilization**								**Performance**
	BRAM		DSP		FF		LUT		Latency
	Val	%	Val	%	Val	%	Val	%	
Sobel Filter Naive	2	0.06	0	0	3183	0.36	3897	0.89	319 ms
Sobel Filter pipeline	35	1.1	0	0	2425	0.27	2914	0.6	190 ms
Sobel Filter Multiple CU (5)	93	3.1	0	0	7014	0.8	20364	4.7	300 ms
Sobel Filter Opti-mized	10	0.34	0	0	1772	0.20	2339	0.53	1.4 ms

Table 6.3: Sobel Filter - Resource utilization

Table 6.2 shows that the naive Sobel implementation performs sub-optimal on FPGA whereas the optimized version on FPGA performs sub-optimal on CPU and GPU. The reasons for obtaining those numbers are detailed below.

6.2.1 Parallelism

GPUs contain multiple processing elements that perform the same operation on different data (*SIMD* model). FPGAs on the other hand use pipelines to perform concurrent processing (different instructions execute different work-items concurrently). Fig. 6.1 illustrates this process for the Sobel implementation. On GPUs, the individual work-items of the Sobel filter are mapped to compute-units (cores) for parallel processing. On FPGA, the naive Sobel kernel is transformed into for loops based on the work-group size and implemented as one single compute unit. To calculate each output pixel, the kernel sequentially accesses the off-chip global memory which becomes a bottleneck to the entire design. Also, several redundant computations are performed since every pixel is computed from the source. This leads to subpar latency estimates on FPGA.



```
_kernel __attribute__ ((reqd_work_group_size(64,64,1)))
 void Sobel(__global int* input, __global int* output) {
}


 __kernel
 void Sobel(__global int* input, __global int* output) {
     for(wg[2]=0;wg[2]<1;wg[2]++)
     for(wg[1]=0;wg[1]<64;wg[1]++)
             for(wg[0]=0;wg[0]<64;wg[0]++){
                 //kernel code
     }
}
```

(a) GPU

(b) FPGA

Figure 6.1: Mapping of OpenCL on GPU and FPGA

Fig. 6.2 illustrates the resource usage and latencies of the different Sobel implementations on FPGA. In the Sobel pipelined implementation, the input and computation stages are explicitly pipelined using OpenCL attributes. A *1.7x* speedup is obtained over the naive implementation due to concurrent processing of work-items, but the BRAM usage has increased due to caching the image data in the local memory.

Next, five compute units were instantiated on the FPGA to process workgroups concurrently. The resource usage is considerably increased due to additional hardware required to process multiple workgroups. The latency estimates are higher compared to the pipelined version, due to the overhead of creating multiple workgroups, and sequential accesses from the global memory (due to limited I/O ports) for all the compute units. Also, SDAccel only allows a maximum of 16 kernels on a FPGA, including memory ports. Hence arbitrarily instantiating multiple compute units on the device is not advisable.

The optimized Sobel version on FPGA uses workgroup size of (1,1,1). Line buffers (implemented in BRAMs) are used to capture the working set (pixels are shifted every clock cycle and old values are reused) exploiting pipeline parallelism on FPGA. However, on CPUs and GPUs, this means that all the computations are serialized resulting in low arithmetic intensity (Number of FLOPs/byte). The optimized Sobel implementation on FPGA is particularly bad for GPUs, since only a single thread is started resulting in sub-optimal usage of the multiple cores on GPUs.

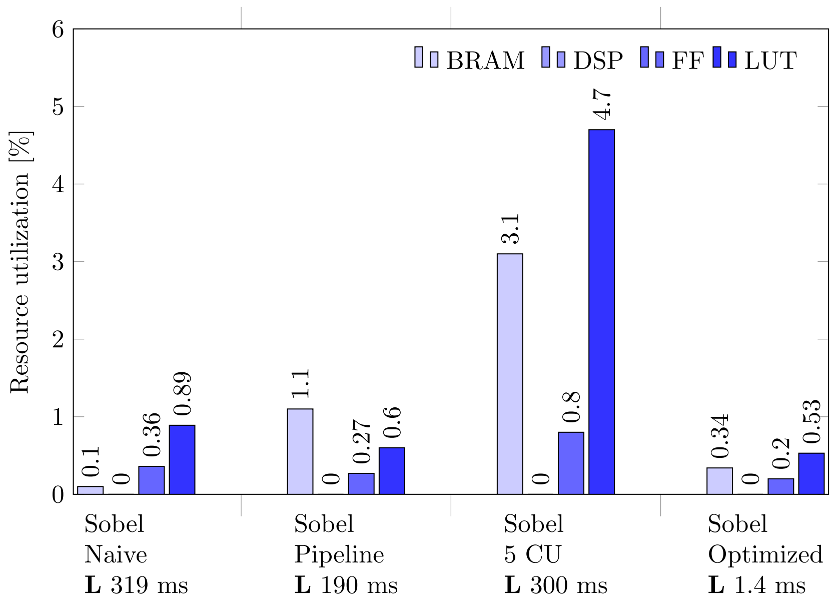


Figure 6.2: Sobel-SDAccel

6.2.2 Branching

On GPUs branching becomes a costly operation. Since all the work-items on the GPU must correctly execute the data path, masking operations are used to enable or disable each work-item based on the evaluated condition. In case of Sobel implementation, if the border cases were handled using conditional statements, then for a 960 × 960 the entire input image has to assessed in these statements. However, only 4000 pixels (on the border) needs to be evaluated. To overcome this challenge, the min and max functions are used to duplicate the edge pixels. But, complex applications with more branches become a bottleneck on GPUs. On FPGAs, since dedicated hardware units are available on the code-path all the branch conditions can be executed concurrently.

6.2.3 Gaussian

Table 6.4 shows the latency estimates for the Gaussian filter. The optimized Gaussian filter for GPU, achieved a *6x* speedup compared to the naive implementation. The arguments presented for the Sobel filter is applicable for the performance differences in the naive implementation of the Gaussian filter.

The SDAccel tool was not able to generate the hardware design for the optimized Gaussian filter. The optimized version used *convert_T()* OpenCL functions to convert the datatypes to float. It turned out that these functions are not implemented in the current version of the SDAccel framework. The convert functions could have been manually implemented on SDAccel, but a fundamental difference needs to be noted here. On FPGAs, floating point operations are not advisable since they are expensive in terms of area and power. An alternative is to use fixed-point datatypes which are not a good fit for

GPUs. Therefore, performance portability cannot be guaranteed even if these functions are implemented for SDAccel. Moreover, multiple accesses to the global memory are detrimental to FPGA designs.

Implementing an optimized Gaussian filter on FPGA will suffer from the same limitations as the optimized Sobel implementation discussed previously. Therefore, these designs were not explored further.

A couple of observations based on the Gaussian implementation are summarized below.

- Floating point operations
  - On GPUs, floating point computations are accelerated. It was observed that there were some differences in the floating point outputs of CPUs and GPUs. The reason for such minute differences could be the way floating point calculations are implemented on a particular architecture. E.g. a multiply and add operation "$((A\times B)+C)$" can be calculated in two ways (1) $round((A\times B)+C)$ or (2) $round(round(A \times B) + C)$ leading to small difference in output values [39].

    On the other hand, it is preferred to use fixed-point arithmetic on FPGAs. Thus there is a fundamental difference in optimizing for these architecures.
- Data transfers
  - The common characteristic we can deduce for the three platforms is the data transfer optimization. Using map/unmap functions (on SoC platforms) can potentially achieve zero-copy data transfer. Moreover, for the SDAccel environment using vector datatypes for data transfer can potentially saturate the AXI-bus alleviating data transfer bottlenecks (char16 can receive 128 bits packed in one variable).

Gaussian Filter - 960×960			
Application	**CPU**	**GPU**	**FPGA (Virtex 7)**
Naive	0.52 ms	0.63 ms	327 ms
Optimized for GPU	0.57 ms	0.10 ms	N/A

Table 6.4: Gaussian Filter - Latency Estimates

6.3 Halide-HLS

The Halide-HLS to SDAccel implementation is compared with the original implementation on Vivado HLS tool. Table 6.5 shows the latency and resource utilization on FPGA for the Sobel filter. The increase in BRAM utilization for the SDAccel version is due to the addition of data-transfer stages to provide isolation from the computation stages. Moreover, FIFO channels are implemented in the SDAccel version to stream data from the data-transfer stages to the computation stage resulting in additional resource usage compared to the Vivado implementation. To provide an estimate with the Xilinx optimized libraries, the OpenCV HLS optimized Sobel implementation is utilized. It is to be noted that the standard OpenCV HLS implementation uses fixed-point datatypes whereas the other implementations use integers. Xilinx OpenCV implementations might internally use optimized math functions resulting in DSP usage. Fig. 6.3 illustrates the resource utilization and latency estimates for the Sobel filter.

Table 6.6 shows the latency and resource utilization on FPGA for the Gaussian filter. The same trend as Sobel filter for the resource utilization and latency can be seen here as well. The reason for the increase in BRAM usage for the OpenCV implementation is because these libraries do not use the optimum sizes for line buffers. Fig. 6.4 illustrates the resource utilization and latency estimates for the Gaussian filter.

Based on these comparisons we can conclude that the Halide-SDAccel filter implementations are comparable to standard designs in terms of performance and resource utilization.

Image Size	960 x 960								
Target Frequency	200 MHz								
Application	**Resource Utilization**								**Performance**
	BRAM		DSP		FF		LUT		Latency [clks]
	Val	%	Val	%	Val	%	Val	%	
Sobel Halide Vivado	4	0.14	0	0	1240	0.13	2395	0.55	928331
Sobel HLS OpenCV	3	0.1	17	0.47	2103	0.24	3164	0.73	932173
Sobel Halide SDAccel	6	0.2	0	0	2476	0.28	3686	0.85	929302

Table 6.5: Resource utilization on FPGA - Sobel

Image Size	960 x 960								
Target Frequency	200 MHz								
Application	**Resource Utilization**								**Performance**
	BRAM		DSP		FF		LUT		Latency [clks]
	Val	%	Val	%	Val	%	Val	%	
Gaussian Halide Vivado	4	0.14	0	0	1240	0.1	2395	0.32	928331
Gaussian HLS OpenCV	6	0.2	8	0.22	1150	0.13	1480	0.34	932184
Gaussian Halide SDAccel	6	0.2	0	0	2042	0.23	2836	0.65	929302

Table 6.6: Resource utilization on FPGA - Gaussian

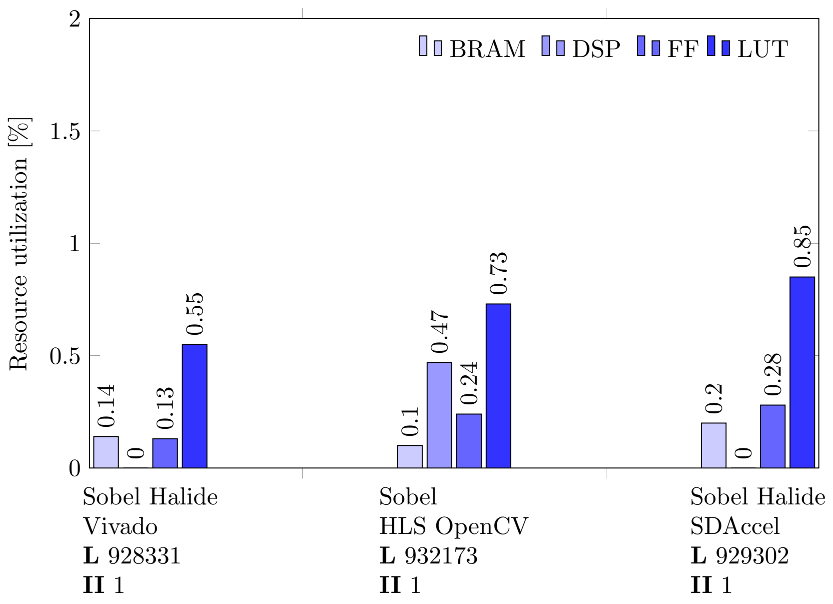


Figure 6.3: Sobel on FPGA

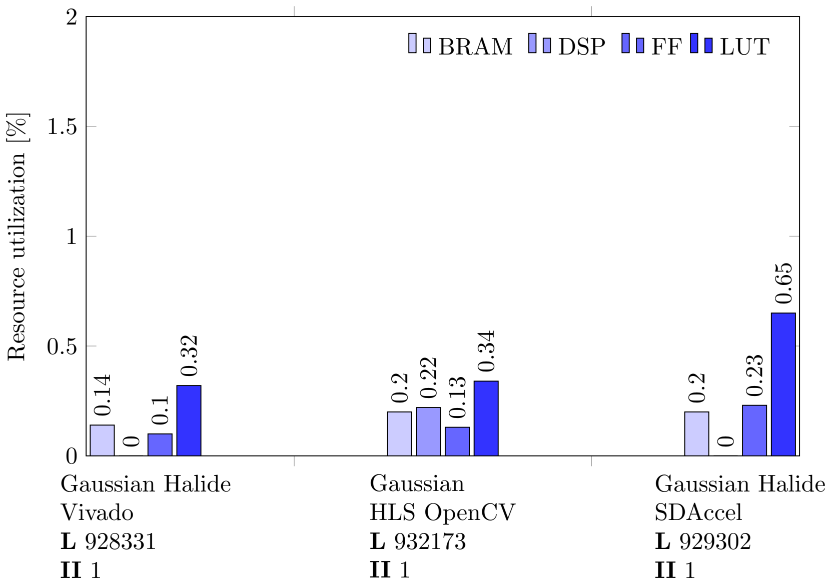


Figure 6.4: Gaussian on FPGA

6.4 MRA

This section details the results obtained for the C/C++ and OpenCL version of the MRA algorithm.

SDAccel was able to successfully synthesize the multi-resolution algorithm. Table 6.7 shows the resource utilization and latency values obtained for individual functions.

The downsampler, upsampler and filter stages use linebuffers that are implemented in BRAMs. The subtract and adder blocks directly process the pixels (no buffers) and achieve an II of 1 which means that they can process a pixel every clock cycle. Therefore, if the time taken to calculate the address and get an input pixel is 2 cycles, then for a pipelined block, the latency is $((960 \times 960) + 2)$ which is 921603 as obtained in the Table 6.7. The II achieved for each internal loop in a block is indicated in Fig. 5.5 (previous chapter). Even though individual blocks achieve an II of one, the II of the whole design is determined by the highest II which in this case is two. Therefore this design can produce an output pixel every two clock cycles.

The operating frequency of our FPGA is 200 MHz. If the design can output a pixel every two cycles, then we can process 100 million pixels every second resulting in a throughput of 108 frames/second. The design reports in SDAccel show that the total kernel computation time is 9.434 ms which is approximately 106 frames/second. It has to be noted that the computation time only includes the kernel computation time and does not detail the data transfer times. For streaming applications (data is received pixel by pixel), the data transfer rates become a bottleneck for the entire design in the current version of SDAccel. A recommendation to allow streaming inputs (without host intervention) has been requested to Xilinx to include in future releases of SDAccel. Note that for the whole design the resource utilization has increased considerably due to the implementation of FIFO buffers for streaming data from one block to the next.

Image Size	960 x 960								
Target Frequency	200 MHz								
Application	**Resource Utilization**								**Performance**
	BRAM		DSP		FF		LUT		Latency [clks]
	Val	%	Val	%	Val	%	Val	%	
Down sampler 960 × 960	4	0.13	0	0	2664	0.3	4075	0.9	927364
Down sampler 480 × 480	4	0.13	0	0	2687	0.3	4161	0.9	233284
Upsampler 480 × 480	6	0.2	0	0	1383	0.15	2721	0.6	928314
Upsampler 240 × 240	6	0.2	0	0	1360	0.15	2692	0.6	233754
Subtract 960 × 960	0	0	0	0	65	0.007	191	0.04	921603
Subtract 480 × 480	0	0	0	0	63	0.007	187	0.04	230403
Reconstruction									
Upsampler 240 × 240	6	0.2	0	0	1360	0.15	2692	0.6	233754
Upsampler 480 × 480	6	0.2	0	0	1383	0.15	2721	0.6	928314
Adder 480 × 480	0	0	0	0	63	0.007	218	0.05	230403
Adder 960 × 960	0	0	0	0	62	0.007	220	0.05	921603
Filters									
Filter 960 × 960	8	0.2	0	0	308	0.03	485	0.1	1843207
Filter 480 × 480	4	0.13	0	0	302	0.03	475	0.1	460807
Filter 240 × 240	2	0.06	0	0	296	0.03	463	0.1	115207
Complete (including input and output stages)									
MRA	1168	39.7	0	0	21936	2.5	63024	14.5	4605199

Table 6.7: MRA C version - Resource utilization and Latency

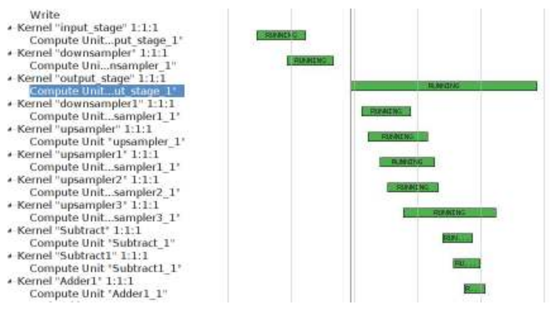
OpenCL

Table 6.8 shows the results for the OpenCL version. The latency estimates are similar to the C/C++ version. The same arguments put forth in the previous section can be extended for the OpenCL version. Since hardware emulation is a detailed simulation, the input dataset was reduced to a 32×32 image. The results obtained can be extrapolated to bigger images as it has been proven using the C version of the MRA algorithm. A few insights can be gained with the obtained results.

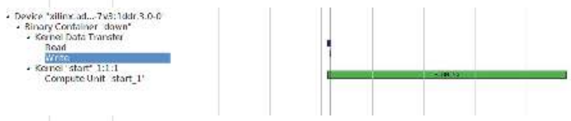
In the OpenCL implementation, pipe objects are used to stream data from one kernel to another. On FPGAs, pipes can be used to generate on-chip buffers for fast data accesses. However on GPUs, buffers are implemented in the global memory which can become a bottleneck to the design. But, modern GPUs are being designed to exploit concurrent kernel execution using pipes which may ensure portability among different platforms.

Fig. 6.5 shows the timeline trace for the OpenCL and C versions. In the OpenCL implementation, multiple kernels are started and execute concurrently. Each kernel requires some time to setup and start execution, hence we can see some delay in the start times. In the C implementation, only one kernel is executed since the design in modelled as multiple functions in a dataflow structure.

In the OpenCL implementations, there is also an additional overhead of creating AXI interfaces for all the kernels. Therefore, using pipes in the current version of SDAccel is not advisable for such an algorithm.


Write
Kernel "input_stage" 1:1:1
Compute Unit...put_stage_1"
Kernel "downsampler" 1:1:1
Compute Uni...nsampler_1"
Kernel "output_stage" 1:1:1
Compute Unit...ut_stage_1"
Kernel "downsampler1" 1:1:1
Compute Unit...sampler1_1"
Kernel "upsampler" 1:1:1
Compute Unit "upsampler_1"
Kernel "upsampler1" 1:1:1
Compute Unit...sampler1_1"
Kernel "upsampler2" 1:1:1
Compute Unit...sampler2_1"
Kernel "upsampler3" 1:1:1
Compute Unit...sampler3_1"
Kernel "Subtract" 1:1:1
Compute Unit "Subtract_1"
Kernel "Subtract1" 1:1:1
Compute Unit "Subtract1_1"
Kernel "Adder1" 1:1:1
Compute Unit "Adder1_1"
RUNNING

(a) OpenCL



(b) C

Figure 6.5: Timeline trace results

Image Size	32 x 32								
Target Frequency	200 MHz								
Application	**Resource Utilization**								**Performance**
	BRAM		DSP		FF		LUT		Latency [clks]
	Val	%	Val	%	Val	%	Val	%	
Down sampler 32×32	4	0.1	0	0	3240	0.4	3942	1.1	1202
Down sampler 16×16	4	0.1	0	0	3149	0.4	3720	1.1	364
Upsampler 16×16	6	0.2	0	0	1273	0.1	2077	0.6	1181
Upsampler 8×8	6	0.2	0	0	1256	0.1	2054	0.6	333
Subtract 32×32	0	0	0	0	336	0.05	564	0.1	1026
Subtract 16×16	0	0	0	0	334	0.05	559	0.1	258
Reconstruction									
Upsampler 8×8	6	0.2	0	0	1256	0.1	2054	0.6	333
Upsampler 16×16	6	0.2	0	0	1273	0.1	2077	0.6	1181
Adder 32×32	0	0	0	0	336	0.05	564	0.1	1026
Adder 16×16	0	0	0	0	334	0.05	559	0.1	258

Table 6.8: MRA OpenCL version - Resource utilization and Latency

7 Discussion

In Chapter 1 the problem statement and goals for this project were defined. The extent to which these goals have been addressed and a summary of the performed work is detailed in this chapter.

7.1 Summary

In Chapter 6, different comparisons were made to evaluate the proposed workflows. First, the portability challenges of the SDAccel workflow were assessed. Second, the method to generate hardware using Halide and SDAccel was evaluated. Third, the applicability of the SDAccel tool for the multi-resolution algorithms was analyzed. In Section 3.7 we mentioned a previous study conducted in Philips that evaluated the Vivado-HLS tool for reducing the development time of FPGAs. The comparison between the various approches is summarized in Table 7.1. The feature to enable streaming inputs has been proposed to Xilinx for the future releases of SDAccel.

One important challenge in SDAccel is optimizing the data transfers between the host and the device. In all our designs, we isolated the computation stage from the data transfer stages to emulate streaming designs. The reason for performing this isolation is to allow easy integration for future releases of SDAccel (maybe allowing streaming inputs), by just modifying the data transfer stages.

7.2 Research question re-visited

In Chapter 1 we had introduced the following research question:
“Is it possible to generate hardware structures on FPGAs for image-processing algorithms, resulting in ease of development for programmers and ensuring portability among CPUs and GPUs?”

We will discuss this research question based on the requirements we had identified for the solution workflows, which are reproduced below:

- *Requirement 1*: A workflow in terms of tools and strategies needs to be developed. The output must be similar to the original working solution
- *Requirement 2*: The algorithm must be implemented once, and ideally be able to execute correctly on different accelerators with acceptable latency and throughput requirements. Since the domain is medical image processing, any changes to the algorithm itself might lead to undesirable outcomes.
- *Requirement 3*: The workflow needs to ensure that complex hardware challenges are abstracted from the programmer and result in an easier development cycle.

7.2.1 Requirement 1

A workflow using existing tools and strategies was proposed and evaluated in this study. Open Computing Language (OpenCL) and Halide languages were used to obtain portable image processing implementations. The outputs were compared with existing solutions to verify the correctness of the implementations.

7.2.2 Requirement 2

Imaging filters were implemented once and executed on all three platforms. Even though functional portability was ensured, the performance decayed when moving to other platforms while using SDAccel. The reasons for the performance degradation were analyzed and explained.

In the Halide-SDAccel workflow, both functional and performance portability were ensured with minimal code changes because of the combination of architectural knowledge and domain specific knowledge in the Halide-HLS framework.

Concerning the Philips use-case, the Multi Resolution Analysis (MRA) algorithm was successfully implemented in SDAccel.

In the Halide-SDAccel workflow, the current version of the Halide-HLS framework does not support generation of HLS-C code for the MRA algorithm, since the compiler cannot generate buffers with proper configuration at each level of the MRA algorithm. Therefore, we have partially addressed requirement 2 with certain limitations in both the workflows.

7.2.3 Requirement 3

In SDAccel to obtain optimal performance, the programmer has to have knowledge of the underlying architecture, but with Halide-SDAccel workflow these challenges were completely abstracted from the application programmer. Concerning the MRA algorithm the integration to the final hardware was performed at a faster pace than using the Vivado workflow.

Hence, portable image processing implementations can be implemented using the proposed workflows. These workflows are extendable to other compute platforms since we use OpenCL and Halide as the base language. These implementations are also easy to maintain due to fast debug and testing process.

Parameters	**Vivado-HLS**	**SDAccel**	**Halide-SDAccel**
Functional portability	No	• Partial • Several OpenCL functions not supported in the current framework	Yes
Performance portability	No	No	• Yes • Change the schedule • Algorithm untouched
Applicability (MRA implementation)	Yes	Yes	• No • Cannot generate different configurations for memory blocks on different levels
Streaming inputs	Yes	• No • Use pipes on device or dataflow model within kernels	• No • Feature not available in the current SDAccel version
Ease of development	Hardware knowledge required	• Hardware knowledge required • Optimized code is similar for OpenCL and C/C++ versions • Vector operations simplified in OpenCL	• No hardware knowledge required • Scheduling can be done by architecture expert
Development and support	+++	+++	+++

Table 7.1: Frameworks summary

Conclusion 8

In the computing world, there exist a plethora of different devices like CPUs, GPUs and FPGAs. Each computing platform poses its own set of challenges that the programmer has to identify to obtain optimal performances. This results in a need to redevelop and retest the same algorithm, increasing the development time and maintenance costs.

In this work, Field Programmable Gate Arrays (FPGAs) are identified as a potential platform to address the Life Cycle Management (LCM) challenge. Due to differences in the programming models, the development process on FPGAs is more challenging than other platforms. To ease the development process and maintain portability OpenCL and Halide were chosen as the base programming languages. To synthesize the designs on FPGA, the SDAccel framework was chosen.

This thesis shows that the SDAccel tool does not guarantee performance portability but the design time for FPGAs can be considerably reduced compared to creating a manual RTL design. SDAccel framework can be used to generate streaming designs to adhere to real-time constraints. A proof of concept was shown by implementing a multi-resolution algorithm that is functionally correct, easily maintainable and offers comparable performances to manually created RTL models. FPGAs are a good fit for streaming models compared to CPUs or GPUs due to the capability to generate on-chip memory buffers allowing fast accesses to data, and enabling concurrent execution of kernels. They are also extendable in the sense that, as long as resources are available on FPGA, new designs can be added for better functionality. We also produced a workflow using Halide and SDAccel which was shown to produce comparable performances to the standard Xilinx HLS OpenCV library. Using Halide as the base specification offers higher levels of abstraction while maintaining performance portability since the schedule can be tuned to specific target platforms. Moreover, since the SDAccel programming flow uses partial reconfiguration capabilities, kernels can be swapped during run-time providing additional advantages to the programmer.

Hence, this study proposes the SDAccel and Halide-HLS framework to ease the development process on FPGAs while maintaining portability to other compute platforms. Several limitations were identified in both the workflows, but since the frameworks are extendable, they can be addressed in future versions.

During this research, several improvement areas were identified that can be addressed in future research work:

Halide-OpenCL-SDAccel

The current work uses the Halide-HLS C framework to generate hardware designs. A workflow that automatically generates Halide-HLS OpenCL code can be explored to further reduce the design time. Such a workflow could allow a larger design space exploration by allowing the user to vary work-group sizes.

The current version of Halide generates OpenCL code for GPUs. This code can be analyzed to check the applicability to the SDAccel tool. Thereafter, a model similar to the Halide-HLS C framework can be explored to generate optimized OpenCL code.

Halide-HLS MRA

The current Halide-HLS framework does not support the generation of multi-resolution algorithms. These algorithms require a different configuration of memory blocks for each level. Extending the framework to support such use-cases can improve the applicability of the framework.

SDAccel-streaming inputs

FPGAs are capable of generating streaming models to obtain high-performance solutions. In the current work, pipes are used to stream data between kernels, but SDAccel does not allow streaming data directly from the I/O interfaces. Therefore, data transfer between the host and the device becomes a bottleneck to the entire design. Providing capabilities to stream input directly from I/O interfaces results in performance and power efficiency gains.

SDAccel-pipes

SDAccel allows only 16 kernels to be generated on the FPGA. In streaming designs, the computing kernels are not required to interact with the host. Hence, the tool can be modified to identify such use-cases resulting in improved functionality.

The SDAccel tool is still in its early stages. The above recommendations for SDAccel were provided to Xilinx to implement in future releases

Bibliography

[1] Philips, "Our heritage - Company - About — Philips." [Online]. Available: http://www.philips.com/a-w/about/company/our-heritage.html

[2] Almarvi, "Almarvi." [Online]. Available: http://www.almarvi.eu/

[3] "D3 . 2 Automatic Generation of Hardware Accelerators and Configurations [confidential]," pp. 1–32, 2016.

[4] H. Fu, "Accelerating Scientific Computing Through GPUs and FPGAs." [Online]. Available: http://cees.stanford.edu/docs/CEESWorkshop8-HFu.pdf

[5] J. L. Hennessy, D. A. Patterson, and A. C. Arpaci-Dusseau, *Computer architecture : a quantitative approach.* Elsevier/Morgan Kaufmann Publishers, 2007. [Online]. Available: https://books.google.co.uk/books/about/Computer{_}Architecture.html?id=pqYl3SWkA64C{&}redir{_}esc=y

[6] "Understanding FPGA Architecture." [Online]. Available: https://www.xilinx.com/html{_}docs/xilinx2017{_}1/sdaccel{_}doc/topics/devices/con-fpga-architecture.html

[7] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.407.5162{&}rep=rep1{&}type=pdf

[8] "WHITE PAPER GPU vs FPGA Performance Comparison," 2016. [Online]. Available: http://www.bertendsp.com/pdf/whitepaper/BWP001{_}GPU{_}vs{_}FPGA{_}Performance{_}Comparison{_}v1.0.pdf

[9] J. Hestness, S. W. Keckler, and D. A. Wood, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior." [Online]. Available: https://www.cs.utexas.edu/users/skeckler/pubs/IISWC{_}2014{_}Characterization.pdf

[10] F. Plavec, Z. Vranesic, and S. Brown, "Towards compilation of streaming programs into FPGA hardware," in *2008 Forum on Specification, Verification and Design Languages.* IEEE, sep 2008, pp. 67–72. [Online]. Available: http://ieeexplore.ieee.org/document/4641423/

[11] A. Rosenfeld, Ed., *Multiresolution Image Processing and Analysis*, ser. Springer Series in Information Sciences. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, vol. 12. [Online]. Available: http://link.springer.com/10.1007/978-3-642-51590-3

[12] D. Koch, F. Hannig, and D. Ziener, *FPGAs for Software Programmers*, 2016. [Online]. Available: http://www.amazon.com/FPGAs-Software-Programmers-Dirk-Koch/dp/3319264060/

[13] A. Momeni, H. Tabkhi, Y. Ukidave, G. Schirner, and D. Kaeli, "Exploring the Efficiency of the OpenCL Pipe Semantic on an FPGA," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 4, pp. 52–57, apr 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2927964.2927974

[14] S. Edwards, "The Challenges of Hardware Synthesis from C-Like Languages," in *Design, Automation and Test in Europe.* IEEE, pp. 66–67. [Online]. Available: http://ieeexplore.ieee.org/document/1395531/

[15] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design, 2003. Proceedings.* IEEE Comput. Soc, pp. 461–466. [Online]. Available: http://ieeexplore.ieee.org/document/1183177/

[16] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11.* New York, New York, USA: ACM Press, 2011, p. 33. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1950413.1950423

[17] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, jul 2009. [Online]. Available: http://ieeexplore.ieee.org/document/5209959/

[18] T. V. Court and M. C. Herbordt, "Requirements for any HPC/FPGA Application Development Tool Flow (that gets more than a small fraction of potential performance) *."

[19] "Intel ® FPGA SDK for OpenCL Programming Guide Last updated for Intel ® Quartus ® Prime Design Suite: 17.0," 2017. [Online]. Available: https://www.altera.com/en{_}US/pdfs/literature/hb/opencl-sdk/aocl{_}programming{_}guide.pdf

[20] "With SDAccel, Xilinx Embraces OpenCL — Berkeley Design Technology, Inc." [Online]. Available: https://www.bdti.com/InsideDSP/2015/01/22/Xilinx

[21] Xilinx and Inc, "Xilinx Introduction to FPGA Design with Vivado High-Level Synthesis (UG998)." [Online]. Available: https://www.xilinx.com/support/documentation/sw{_}manuals/ug998-vivado-intro-fpga-design-hls.pdf

[22] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines." [Online]. Available: http://people.csail.mit.edu/jrk/halide12/halide12.pdf

[23] D. R. Kaeli, *Heterogeneous computing with OpenCL 2.0.* [Online]. Available: http://www.sciencedirect.com.tudelft.idm.oclc.org/science/book/9780128014141

[24] “Halide.” [Online]. Available: http://halide-lang.org/

[25] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, “Programming Heterogeneous Systems from an Image Processing DSL,” oct 2016. [Online]. Available: http://arxiv.org/abs/1610.09405

[26] F. Plavec and Franjo, *Stream computing on FPGAs.* University of Toronto, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=2231435

[27] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang, “High-Level Synthesis for FPGAs: From Prototyping to Deployment,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, apr 2011. [Online]. Available: http://ieeexplore.ieee.org/document/5737854/

[28] D. Chen and D. Singh, “Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering,” in *22nd International Conference on Field Programmable Logic and Applications (FPL).* IEEE, aug 2012, pp. 5–12. [Online]. Available: http://ieeexplore.ieee.org/document/6339171/

[29] S. P. Metman, “MSc THESIS Software To Hardware : Alternatives For Reducing Design Time Of Optimized FPGA Implementations In Medical Devices [Confidential],” Tech. Rep., 2016.

[30] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, “A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs,” in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation - PACT ’16.* New York, New York, USA: ACM Press, 2016, pp. 327–338. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2967938.2967969

[31] “SDaccel Release Notes and Supported Hardware.” [Online]. Available: https://www.xilinx.com/html{_}docs/xilinx2017{_}1/sdaccel{_}doc/topics/introduction/concept-supported-boards-release-notes.html

[32] Xilinx and Inc, “SDAccel Environment Optimization Guide (UG1207),” 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw{_}manuals/xilinx2017{_}2/ug1207-sdaccel-optimization-guide.pdf

[33] A. Corporation, “Altera SDK for OpenCL Best Practices Guide.” [Online]. Available: https://www.altera.com/en{_}US/pdfs/literature/hb/opencl-sdk/aocl{_}optimization{_}guide.pdf

[34] “Gen9 - Microarchitectures - Intel - WikiChip.” [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/gen9{#}Performance

[35] "The Compute Architecture of Intel ® Processor Graphics Gen9 External Revision History." [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf

[36] "Using Optimized Built-in Math Functions from HLS MATH Library." [Online]. Available: https://www.xilinx.com/html{_}docs/xilinx2017{_}1/sdaccel{_}doc/topics/design-flows/concept-optimized-built-in-math-functions.html?hl=hls{%}2Clibrary

[37] Xilinx and Inc, "Vivado Design Suite User Guide: High-Level Synthesis (UG902)," 2014. [Online]. Available: https://www.xilinx.com/support/documentation/sw{_}manuals/xilinx2014{_}1/ug902-vivado-high-level-synthesis.pdf

[38] "Optimizing OpenCL Usage with Intel® Processor Graphics — Intel® Software." [Online]. Available: https://software.intel.com/en-us/node/540440

[39] N. Whitehead and A. Fit-Florea, "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs." [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf