MASTER

Portable image processing implementation through domain-specific languages

Thio Gunawan, Aries

Award date:
2017

Link to publication
PORTABLE IMAGE PROCESSING IMPLEMENTATION THROUGH DOMAIN-SPECIFIC LANGUAGES

Master Thesis

Aries Thio Gunawan
Embedded Systems

Supervisors:
dr.ir. M.C.W. (Marc) Geilen (TU/e)
ir. S.C. (Steven) van der Vlugt (Philips Healthcare)
ing. R. (Rob) de Jong, (Philips Healthcare)

Eindhoven, August 2017
Abstract

Digital image processing has been widely adopted in a broad range of fields including in the healthcare domain. In medical devices, such as Philips iXR Azurion, image processing is needed to improve the quality of the resulting images. Also, it is important to obtain a low-latency image processing implementation as it is used in an interventional setting. Considering the high-performance and the accuracy requirement of the implementation, and the growing complexity of image processing algorithms, great development and verification costs and efforts would be required.

However, the main challenge is that this process has to be repeated several times during the life cycle of the medical system. The life cycle of a medical system is longer, can be up to twenty years, compared to the life cycle of its third-party components, e.g., computing platforms. Currently, PC-based platforms are employed to perform the image processing algorithms. This kind of component typically has a life cycle of 3 - 6 years resulting in the component unavailability during the life cycle of the system. As a result, re-development and re-testing of the image processing algorithms need to be performed targeting a new platform architecture.

In a broader context, the main objective of this study is to overcome the Lifecycle Management (LCM) challenge. In the previous study, an FPGA-based platform has been proposed as an alternative target platform considering the longer life cycle and the portability of this platform. However, the programmability of FPGAs poses huge barriers for software developers who do not have sufficient hardware design knowledge. Hence, this thesis aims to investigate a means to overcome the programmability challenges on FPGA platforms while still maintaining portability to other computing platforms.

To achieve this solution, Halide - a Domain-Specific Language (DSL) for the image processing domain - was proposed to achieve both functional and performance portability across several platforms including FPGA-based platforms. In addition, several limitations were identified during the research. Several extensions for Halide targeting FPGAs were proposed: an extension of arbitrary precision data types, a separation between the algorithm description and the data type specification in Halide, and an on-chip boundary conditions handling. The results demonstrate that it is possible to generate a high-performance FPGA implementation from a higher-level of abstraction with an additional advantage in reducing the development time. Moreover, the portability to other platforms are maintained. Therefore, this research has shown the potential of Halide to achieve a portable implementation towards FPGA platforms and to address the LCM challenge in general.
Preface

This thesis marks the end of my two-year study at TU Eindhoven. This project is the result of my graduation project which took place nearly nine months in the Image Guided Therapy (IGT) Department at Philips Healthcare, Best. I would like to express my gratitude to everybody who supported me during the course of this project.

Firstly, I would like to thank my supervisor at the university, Marc Geilen, for his guidance and critical opinions throughout this project. Furthermore, I would like to thank Steven van der Vlugt and Rob de Jong as my supervisors at Philips Healthcare in Best for their support, feedbacks, and discussion during this project. I am grateful to be able to learn lots of things during my internship.

Thanks to my colleagues, Uttam Kumar Erlanggo and Pooja Ravi Shankar, who were also doing their thesis project at Philips, for their helpful advice and supports. Also, I would also like to thank Ruben Guerra Marin and Rachana Arun Kumar for their companionship in the office. I am really grateful to Ruben who wanted to allocate some of his time to give short tutorials and was always ready to answer my questions. Thanks also to all of my friends, especially Christian Stevandy and Pramita Winata, for their support and some fun time together to blow off some steam.

Last but not least, special thanks to my family as this could not have been possible without their love and support.

Aries Thio Gunawan,
Eindhoven, The Netherlands
August, 13th 2017
## Contents

**List of Figures**
ix  
**List of Tables**
xi  
**Listings**
xiii  
**List of Abbreviations**
xv  

### 1 Introduction
1.1 Context ................................................. 1  
1.1.1 Philips Healthcare .................................. 1  
1.1.2 ALMARVI ........................................... 2  
1.1.3 Medical Imaging Technology ......................... 2  
1.2 Lifecycle Management and Embedded Platforms ........ 3  
1.3 Problem Description ................................... 5  
1.4 Outline of this thesis .................................. 6  

### 2 Background
2.1 Image Processing Algorithm Overview .................. 7  
2.1.1 Characteristics of Image Processing Algorithms .... 7  
2.2 Hardware Architectures Comparison .................... 9  
2.2.1 Multi-core CPU ..................................... 10  
2.2.2 GPU ............................................... 11  
2.2.3 FPGA ............................................ 11  
2.2.4 Opportunities and Programmability Challenges of FPGAs 13  
2.3 Portable Computing Solutions .......................... 14  
2.3.1 Open Computing Language (OpenCL) .................. 14  
2.3.2 Portable Computing Language (pocl) ................. 15  
2.3.3 High-Level Languages (HLLs) ....................... 15  
2.4 Quality Measurement ................................... 16  

### 3 Domain-Specific Languages
3.1 DSL Overview .......................................... 19  
3.2 Image Processing DSLs .................................. 19  
3.3 Current Limitations of Halide-HLS ....................... 24  

### 4 Implementation
4.1 Extending Arbitrary Precision Data Types on Halide .... 29  
4.2 Separation between Algorithm Description and Data types 31  
4.3 On-chip Boundary Handling .............................. 35
5 Results
  5.1 Evaluation Methodology ........................................ 37
  5.2 Evaluation Results .............................................. 39
    5.2.1 Design Space Exploration .................................... 39
  5.3 Results Summary .................................................. 43
  5.4 Comparison to OpenCV ........................................... 45
  5.5 Overhead of On-Chip Boundary Handling .......................... 45

6 Discussion .............................................................. 49
  6.1 Results Discussion ................................................ 49
  6.2 Research Questions Revisited ..................................... 50

7 Conclusions .............................................................. 53

Bibliography ............................................................... 55

Appendix ................................................................. 59

A Image Processing Applications in Halide ............................ 59
List of Figures

1.1 Interventional X-Ray System - Philips Azurion 7 [1] ........................................... 3
1.2 A simplified processing chain in a medical system .................................................. 3
1.3 A development process of PC-based platforms ......................................................... 4

2.1 Types of operators in image processing ................................................................. 8
2.2 An example of image processing pipeline ............................................................... 9
2.3 Simplified Generic Multi-core CPU Memory Architecture ...................................... 11
2.5 Nvidia Pascal Architecture [3] ................................................................................ 12
2.6 An FPGA Generic Memory Architecture ............................................................... 13
2.7 The portable computing solutions ecosystem ........................................................... 16
2.8 The portable computing solutions ecosystem ........................................................... 17

3.1 Error Distribution of 8-bit integer based Gaussian Filter ........................................... 25
3.2 Error Distribution of the 8-bit unsharp filter without scaling ..................................... 26
3.3 Error Distribution of the 8-bit unsharp filter without scaling ..................................... 26
3.4 Boundary Handling in a Convolution Filter ............................................................. 28

4.1 IR Nodes (a) Before and (b) After Transformation .................................................... 34
4.2 Image Partitions for Boundary Handling ................................................................. 36
4.3 Input Buffer Stage with repeat boundary condition for 2-D Local Operator ............... 36

5.1 Comparison of output images in the implementation of Unsharp Mask without and with the scaling respectively ................................................................. 41
5.2 Boundary Handling with Stencil size <1,1,1> with offsets <1,1,1,1> and <3,3,3,3> .......... 46
5.3 Boundary Handling with Stencil size <3,1,1> and <5,1,1> .......................................... 47
5.4 Boundary Handling with Stencil Size <3,3> and <5,5> ............................................ 47
## List of Tables

2.1 Comparison between multi-core Central Processing Units (CPUs), Graphical Processing Units (GPUs), and Field-Programmable Gate Arrays (FPGAs) ........................................ 10

3.1 Comparisons of different DSLs in the image processing domain ........................................ 23

3.2 Comparison of several implementations of Gaussian Filter in term of accuracy ............... 25

3.3 Comparisons of accuracy between several implementations of Unsharp Filter ............... 25

4.1 Mapping between C/C++ native data types and Halide data types ..................................... 30

4.2 Proposed Fixed-Point Data Types in Halide ......................................................................... 30

4.3 Proposed Fixed-Point Data Types in Halide ......................................................................... 30

4.4 Xilinx Vivado ArbitraryPrecision Data Types Mapping ......................................................... 32

5.1 Resources Available in the target FPGA .................................................................................. 37

5.2 Fixed-Point Configurations for Listing 5.1 ............................................................................ 39

5.3 Estimated Resource Utilization (Used and % of total) and Performance (Latency in clock and Initiation Interval) of several Gaussian Filter Implementations .............................. 40

5.4 Accuracy Comparisons between several Gaussian Filter Implementations ......................... 40

5.5 Fixed-Point Configurations for Listing 5.2 ............................................................................ 42

5.6 Estimated Resource Utilization (Used and % of total) and performance (Latency in clock) of several Unsharp Mask Implementations ......................................................... 42

5.7 Accuracy Comparisons between several Unsharp Filter Implementations ......................... 43

5.8 Resource Utilization Summary ............................................................................................... 44

5.9 Accuracy Summary .................................................................................................................. 44

5.10 Comparisons to OpenCV ....................................................................................................... 45

5.11 Resource utilization and latency results of several boundary conditions handling on FPGAs with window size 10x10 .................................................................................... 46
Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Algorithm description of a blur filter in Halide</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Gaussian Blur Filter in HIPAcc copied from [4]</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Unsharp Mask Operation in Darkroom</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>PolyMage DSL code for Unsharp Mask [5]</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>8-Int-based Gaussian Filter Implementation in Halide</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Floating-Point-based Gaussian Filter in Halide</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Proposed structure of Halide code</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Code Snippet of Sobel Filter in Halide with the separate data type</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Input buffer stage interface</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Data types specification for Gaussian Filter</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>Data types specification for Unsharp Mask</td>
<td>42</td>
</tr>
<tr>
<td>A.1</td>
<td>Sobel Filter</td>
<td>59</td>
</tr>
<tr>
<td>A.2</td>
<td>Gaussian Filter</td>
<td>61</td>
</tr>
<tr>
<td>A.3</td>
<td>Unsharp Mask</td>
<td>62</td>
</tr>
<tr>
<td>A.4</td>
<td>Bilateral Grid</td>
<td>64</td>
</tr>
</tbody>
</table>
List of Abbreviations

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

API Application Program Interface
AST Abstract Syntax Tree
BRAM Block Read Access Memory
CPU Central Processing Unit
CT Computed Tomography
CUDA Compute Unified Device Architecture
DAG Directed Acyclic Graph
DSL Domain Specific Language
DSP Digital Signal Processor
FA Full Adder
FF Flip-Flop
FPGA Field-Programmable Gate Array
FPU Floating-Point Unit
GPU Graphical Processing Unit
HDL Hardware Description Language
HLL High-Level Language
HLS High-Level Synthesis
IGT Image Guided Therapy
II Initiation Interval
IP Intellectual Property
IR Intermediate Representation
iXR Interventional X-Ray
LCM Lifecycle Management
LLVM Low Level Virtual Machine
LUT  Look-Up Table
MRI  Magnetic Resonance Imaging
MSE  Mean Squared Error
OpenCL  Open Computing Language
pocl  Portable Computing Language
PC  Personal Computer
R&D  Research and Development
RGB  Red-Green-Blue
RTL  Register Transfer Level
SIMD  Single Instruction Multiple Data
SSE  Streaming SIMD Extensions
Chapter 1

Introduction

Digital image processing has been extensively developed since the 1960s. Initially, it was used to enhance distorted pictures of the surface of the Earth’s moon [6]. This first successful pioneer leads to a steady stream of advances in digital image processing. Over the next couple of years, image processing starts to gain importance to produce high-quality images. This technique has been adopted in other application fields, such as photograph enhancement, object recognition, satellite imagery, and medical imaging. As the digital imaging equipment, such as camera, advances and capable of producing high quality raw digital signals, digital image processing has become more common and tremendously compute-intensive.

In the past, the processing power was still limited by the computation capability of CPUs of that era. As CPUs become more powerful and cheaper, digital image processing becomes widely available, because of its cheaper processing cost. Furthermore, the choice of processing platform nowadays is not limited to a single processing platform anymore. There are various kinds of potential computing platforms for image processing applications, such as GPUs and FPGAs. Depending on the characteristics of the algorithms, accelerating image processing applications on the non-CPU platforms can result in a high performance implementation.

The increasingly complex image processing algorithms and the plethora of computing platforms pose several challenges in the development and maintenance efforts and costs in the lifecycle of a system. Since each kind of platform has a specific hardware architecture and programming model, obtaining high-performance implementation requires several platform-specific optimization strategies. As a result, the implementation is not portable to other platforms anymore and often results in hard-to-maintain implementation. The aim of this thesis is to address this challenge in the context that will be described in the next section.

The next section describes the context of this thesis followed by the elaboration of the problem description and the research questions. Finally, the last section presents the outline of this thesis.

1.1 Context

1.1.1 Philips Healthcare

Koninklijke Philips N.V. or Philips is a diversified technology company based in Amsterdam. It was established in Eindhoven in 1891 by Gerald Philips and Frederik Philips [7]. The company currently consists of two primary divisions: healthcare and consumer lifestyle. As the main division in the enterprise, Philips Healthcare focuses on improving peoples lives through meaningful innovation in the areas of healthcare technology. Some of the popular healthcare products are interventional X-Ray, Computed Tomography (CT)-scan, Ultrasound, Magnetic Resonance Imag-
CHAPTER 1. INTRODUCTION

This work was performed in the Image Guided Therapy (IGT) department of Philips Healthcare. The location is in their Research and Development (R&D) in Best, the Netherlands.

1.1.2 ALMARVI

Algorithms, Design Methods, and Many-core Execution Platforms for Low-Power Massive Data-Rate Video and Image Processing (ALMARVI) [8] is a European project with the collaboration between several leading technical universities and companies. Both Philips Healthcare and TU Eindhoven are involved in this project. This project aims to address the complexity and challenges in the implementation of compute-intensive image and video processing algorithms.

One of the objectives of this project is to achieve software portability targeting CPU, GPU, and FPGA platforms from a common code base. This thesis aims to extend the research result of the ALMARVI project on that objective.

1.1.3 Medical Imaging Technology

Medical Imaging is a technique and process of creating visual representations of body parts, organs, or tissues, for clinical diagnosis, treatment, and medical intervention [9]. Over the past decades, the advancements in the medical imaging technology have played a major role in improving the health of all population groups. The current techniques can produce the digital images of internal body non-surgically improving the medical decision-making process without unnecessary procedures (e.g., surgery).

One of the medical branches that incorporate medical imaging techniques is Radiology. Some examples of commonly used medical imaging devices are X-Ray and MRI to diagnose and treat diseases within the body [10]. One of the subspecialties of radiology is Interventional Radiology which provides minimally invasive image-guided medical procedures. Making incisions into the body as typically performed in the traditional surgery can be avoided since the process to direct interventional instruments, such as needles inside the body, can be accurately carried out with the guidance of internal body images obtained in real-time from imaging modalities. The motivation of this technique is to minimize risks to the patient and to improve the health outcomes, such as fast recovery time, reduced complication risk, and less trauma effect [11].

In the IGT department, an Interventional X-Ray (iXR) system is developed to provide the doctors with real-time images of the patient’s blood vessels while performing an interventional medical procedure. Figure 1.1 shows the latest generation of an iXR system from Philips, Philips Azurion 7. This system has several benefits for both doctors and patients, such as improving the accuracy of the medical procedures, minimizing the number of invasive surgery, reducing the chance of infection after surgery, and improving the patients’ recovery time.

The iXR imaging system contains a large number of components; image processing is one of the crucial elements in the processing pipelines. The simplified medical image processing chain can be seen in Figure 1.2. The first block in the chain is a data acquisition system (e.g. X-Ray tube and detector) which produces the raw input images. The raw data must be processed to obtain high-quality images before being displayed. In the iXR system case, a low dose of X-Ray radiation is desirable to reduce hazardous radiation effects on the patient. However, a lower X-ray dose might negatively affect the contrast and the noises of the images. Hence, an image processing pipeline plays an important role to keep the radiation dose as low as possible while still being able to produce high-quality images. Typically, noise reduction, image enhancement, and sharpening are applied. At the end of the processing chain, the processed image is displayed or stored in the system.

Since the iXR system is used in image-guided medical operations or interventional settings, it is
important that the system is able to display the output images in real-time. This strict time constraint is related to the safety of the patient. In general, the minimum requirement for the latency in which no delay is perceived is 150 ms or less. In addition to the latency, the system must be able to produce continuous streams of processed images; the recommended minimum requirement is 30 frames per second. Since the latency can also be introduced by several components, such as the data acquisition part, it is important that the performance penalty to the overall system performance contributed by the image processing component is as low as possible.

1.2 Lifecycle Management and Embedded Platforms

Medical systems, such as the iXR system, generally have a much longer lifecycle compared to the lifecycle of the computing platforms. Currently, the computing platforms used for the implementation of the image/video processing algorithm employed in the iXR system are PC-based platforms. Typical PCs have an average life cycle of 3 - 6 years while the life cycle of the overall system can be up to twenty years.

Due to Moore’s law, the manufacturing technology and the processor micro-architecture keeps advancing resulting in fast obsolescence of a certain architecture generation. As a result, it can be expected that the original PC architecture employed in the system is not available anymore at some point during the life cycle of the system. Hence, there is a need to re-develop the same image processing algorithms to support the next generation of PC-based platforms. This process entails re-designing, re-development, re-verification, and re-certification of the system leading to the Lifecycle Management (LCM) challenge.

The current development process contributes to the efforts and costs of re-development and re-
CHAPTER 1. INTRODUCTION

verification in the LCM challenge. Figure 1.3 illustrates the development process of an image processing algorithm targeting different Personal Computer (PC) architectures. First, the image processing algorithms are developed and verified using a modeling language, such as MATLAB. The next process is to convert the MATLAB model into a compiled language implementation, such as C/ C++ based implementation. Further platform-specific optimizations are often required to obtain high-performance implementations. This process is time-consuming resulting in longer time-to-market and typically produces codes that are heavily coupled to the underlying hardware architecture of the platform resulting in a non-portable and difficult-to-maintain implementation. As a result, implementing the same image processing application targeting a different PC architecture requires re-development and re-verification.

In addition to the PC-based platforms, there are also other different target platforms that could also deliver high or even better performance. The previous study in Philips [12] has demonstrated that a high-performance image processing implementation can be achieved using a GPU platform. In another previous study [13], an FPGA-based implementation has been proposed as an alternative target platform. As the computational capacity of FPGAs has advanced, FPGAs have become a potential platform for image processing applications. With the proper architecture configuration (e.g., parallel and deep-pipeline architecture), the FPGA-based implementation can offer better performance (e.g., throughput) and satisfy the real-time requirement in the medical processing domain.

An additional benefit of FPGAs is that FPGAs targeting particular types of industry, such as automotive and defense, have a much longer lifecycle, up to fifteen years or more, than PC-based platforms resulting in a longer component availability. Also, the implementation in the form of soft Intellectual Property (IP) cores in a Register Transfer Level (RTL) format can be re-synthesized on different process technologies of FPGAs allowing a portable implementation among FPGAs. These advantages can significantly reduce the re-development and the maintenance costs of the system.

Since each computing platform has its own hardware architecture and programming paradigm, different optimization strategies are needed. These differences result in increases in the re-development and re-verification costs and efforts in the LCM challenge. Since there is no single ideal target platform for a wide range of image processing algorithms, it is desirable to be able to support these platforms. One approach to address the increasing LCM challenge due to the plethora of computing platforms is a portable computing solution. The ideal portable solu-

![Figure 1.3: A development process of PC-based platforms](image-url)
tion must be able to ensure functional correctness of a common implementation among different computing platforms and to maintain the performance in each kind of platform compared to the manually hand-optimized implementation. A portable implementation with the same code base can reduce the re-development and re-verification efforts and costs without any major changes in the source code.

1.3 Problem Description

As discussed in the previous section, a portable implementation is a promising solution towards addressing the increasing LCM challenge. Three different target platforms, i.e., multi-core CPUs, GPUs, and FPGAs, have been considered to be potential platforms for the implementation of image processing algorithms. Hence, the portable solution should be able to support these platforms. The study in [12] describes a portable implementation towards a multi-core CPU and a GPU through OpenCL. However, it remains a challenge for the portable solution to support FPGAs as an additional target platform due to the different programming paradigm and architecture of FPGAs.

Despite the potential of FPGAs, the development efforts targeting FPGAs are relatively high. The hardware design on FPGAs and software design have different programming paradigm. This gap poses a high barrier for software programmers to adopt FPGAs as the target platforms. The condition is also aggravated because the number of software programmers is ten times as many as the number of hardware programmers [14]. As a result, the implementation targeting an FPGA platform is more time-consuming and costly compared to the equivalence software implementation.

Traditionally, a RTL implementation on an FPGA platform is programmed directly using Hardware Description Languages (HDLs) such as Verilog and VHDL [15]. These programming languages are well-suited to capture the hardware structure design of an application. However, comparable to assembly language in software, the development process is laborious, error-prone, and difficult-to-maintain; it is obviously challenging for software developers who have little or no knowledge of hardware design.

Nowadays, a higher level of abstraction to program FPGAs can be achieved using C-based High-Level Synthesis (HLS) tools. In the previous study [13] which was performed in the context of ALMARVI, the high-level approach using C programming language and Xilinx Vivado HLS was used to implement a medical image processing algorithm in an FPGA platform. Despite the higher abstraction to RTL, programming an FPGA platform using C-based language still requires sufficient knowledge of hardware design. Several structural changes in the source code and the insertion of pragmas are still needed to obtain high-performance implementation. Naively mapping the implementation from the equivalent CPU-based code often results in low-performance implementation.

Thus, the portable solution should take into account the different programming model and hardware architecture of FPGAs to maintain the functional correctness and to achieve a high-performance implementation. Enabling a portable implementation to FPGAs can overcome the programmability challenges of FPGAs and maintenance challenges in the LCM. The aim of this thesis is to explore and investigate the methods and tools to achieve cross-platform portability targeting FPGAs while maintaining portability to CPUs and GPUs. Furthermore, any limitation in the methods and tools is identified and addressed. According to the problem description, the main research question is:

Are there any means to achieve cross-platform functional and performance portability targeting CPUs, GPUs, and FPGAs in the domain of medical image processing?

The following sub-research questions are formulated to answer the main research question.
1. What are the current tools or methods available to achieve cross-platform portability?

2. What is the most suitable solution (e.g. tool or technique) for the implementation of portable image processing algorithms in the context of Philips Healthcare?

3. Are there any limitations in the solution and how to improve the solution to address the limitations?

1.4 Outline of this thesis

This thesis consists of six further chapters organized as follows.

In Chapter 2, the background knowledge regarding the characteristics of image processing algorithms and comparison of FPGA architecture with CPUs and GPUs are described. In addition, several existing portable computing solutions are outlined in this chapter. Chapter 3 provides the critical review of existing domain-specific languages for image processing. Chapter 2 and Chapter 3 mainly answers sub-research question 1 and 2. Moreover, the limitations of the current framework are also described leading to some proposed extensions. In Chapter 4, the design and implementation of two proposed extensions are discussed. This chapter address sub-research question 3. Then, the implementation results and its analysis are discussed in Chapter 5 and Chapter 6 respectively. Finally, Chapter 7 concludes this thesis with the recommendations for the future work and the conclusions of this study.
Chapter 2

Background

This chapter addresses the basic concepts used throughout this report. Section 2.1 discusses the main characteristics of image processing algorithms. In the next section, we are going to explore relevant hardware architectures: CPUs, GPUs, and FPGAs, and compare FPGA platforms with the other platforms. Section 2.3 describes the existing portable computing solutions. Finally, the last section concludes this chapter with the quality measurement methodology used for the evaluation.

2.1 Image Processing Algorithm Overview

As mentioned in the previous chapter, digital image processing is one of the main components of the medical imaging system. Some of the most commonly used techniques in the medical domain are reducing image noises, improving image quality, and highlighting important features in a medical image. In the case of improving image quality, the raw images obtained from a sensor (e.g. camera and X-ray detector) need to be processed to remove the noises before being displayed. To have a better understanding of image processing algorithms, the characteristics of the algorithms are described in this section.

Image processing is a kind of signal processing in which the input signal is a two-dimensional (or more) signal in the form of an image and the output signal can be in the form of an image or information. There are two kinds of image processing: analog and digital. The type of image processing discussed in this thesis is digital image processing. In digital image processing, the images are typically represented as the collection of pixels. The processing operations applied to every pixel in an image can be mathematical or logical operations.

2.1.1 Characteristics of Image Processing Algorithms

Despite being applied in many different fields, image processing algorithms from various application domains typically share the same classification and characteristics. According to [16], image processing operations can generally be classified into three main types as illustrated in Figure 2.1. The types of image processing are as follows.

- Point Operator.
  Point operators are the most basic form of image processing algorithm. In this operation, the output pixel value at a specific pixel is determined by the input value at the same coordinate. Some examples of this operator are color conversion, brightness adjustment,
CHAPTER 2. BACKGROUND

Figure 2.1: Types of operators in image processing

and image thresholding. This operation is an embarrassingly parallel algorithm because each pixel can be independently computed.

- Local Operators
  This type of operator is common in image processing algorithms. In local operators, the output value at a specific coordinate is computed from the input values in the neighborhood of the same coordinate. An example of a local operator is a convolution filter. Although there are many kinds of convolution filters, they share the same characteristics. The computation of an output pixel is performed using a filter mask with a certain window size convolved with a group of pixels surrounding the central pixel. This kind of operation is performed on every pixel in the image. Hence, a local operator is more compute-intensive requiring multiple accesses to the same input pixels.

- Global Operator.
  In global operators, the output value at a specific coordinate is calculated from all of the pixels in the input image. Some examples of this operation are the calculation of the mean square error, the minimum or maximum pixel, and the sum of all pixels, which are usually used for image analysis purpose. Reduction operations, such as grayscale histogram, are also considered as global operators.

These operations are classified as spatial image processing, but there is also temporal image processing. Some examples are feature extraction, adaptive filtering, and optical flow. Although multiple frames are processed, this algorithm still applies the basic operations: pixel, local, and global operators. In this thesis, only these three operators are discussed.

Although image processing algorithms are too diverse to be generalized [17], general characteristics of image processing algorithms are sufficient to describe the characteristics of most of the algorithms. The general characteristics of image processing algorithms are as follows.

- Large size of memory buffers.
  Input images are usually stored in buffers before being processed. The size of the buffer depends on the size of the image; the larger the image size, the larger the buffer needed to store all of the pixels.

- Frequent access to memory buffers.
  These frequent accesses are also related to the image size and the window size in the case of local operators. Depending on the location of the buffer, frequent accesses to the memory can hinder the overall performance.

- Sequential access pattern.
  The memory accesses are typically accessed in a sequential order, e.g. row-major or column-major order.

- Intensive floating-point arithmetic operations.
  Arithmetic operations in most image processing algorithms use floating-point data type. The
use of floating-point representation is related to the accuracy of the computation. In the case of the local operator, the number of arithmetic operations is also multiplied by the size of the filter mask.

- Intensive logical operations
  Some algorithms involve bitwise logical operations on pixel values, such as AND, NOT, XOR, and bit-shifts. It is intensive because these operations are applied to every pixel in the image.

- A mixture of sub-algorithms/ pipeline processing
  It is also common to find an image processing algorithm which consists of several sub-steps in which the input of the next steps depends on the output from previous processing steps. This kind of operation is performed in a pipeline fashion or which is commonly known as image processing pipelines.

  Figure 2.2 illustrates an image processing pipeline consisting of two local operators. In each stage, intensive arithmetic operations are performed to the input pixels. In the image processing pipeline, the challenge of performing the computation in a stage is combined with the challenge of data dependencies between stages.

![Figure 2.2: An example of image processing pipeline](image)

Considering the characteristic of image processing algorithms, in general, there is an enormous opportunity to exploit the parallelism nature of the algorithm. Several levels of parallelism can be exploited such as data-parallelism and task-parallelism. In data-parallelism, massively independent pixels processing can be performed while in task parallelism, several processing steps can be carried out concurrently. The next section maps the parallelism opportunity in image processing algorithms onto several relevant hardware architectures.

### 2.2 Hardware Architectures Comparison

We distinguish three kinds of computing platforms for the implementation of image processing algorithms: multi-core CPU, GPU, and FPGA. This section starts with the brief discussion on those computing platforms to understand the characteristics of each platform. The comparison between platforms is summarized in Table 2.1. The comparison is followed by the discussion on the programmability challenges of FPGAs.
CHAPTER 2. BACKGROUND

2.2.1 Multi-core CPU

Prior hardware development was mainly focused on single core performance by using techniques such as pipelining, branch prediction, out-of-order execution, instruction-level parallelism, and other hardware-based optimization techniques [18]. Increasing the clock frequency of the core can also improve the performance. However, further improvement of the clock frequency has been limited by a power-wall as it would cause serious heat problem on the chip. As a result, the microprocessor design has moved towards a lower-speed multi-core design instead of a fast single-core processor.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Multi-core CPUs</th>
<th>GPUs</th>
<th>FPGAs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallelization Model</strong></td>
<td>- Multithreading</td>
<td>- Massive number of cores</td>
<td>- Deep-pipeline Execution</td>
</tr>
<tr>
<td></td>
<td>- Vectorization</td>
<td>- Multiple Thread (SIMT)</td>
<td>- Parallel Hardware Execution</td>
</tr>
<tr>
<td><strong>Data Type Operation</strong></td>
<td>- Integer Operation</td>
<td>- Integer Operation</td>
<td>- Arbitrary Precision Integer</td>
</tr>
<tr>
<td></td>
<td>- Floating-point Operation</td>
<td>- Floating-Point Operation</td>
<td>- Arbitrary Precision Fixed-Point</td>
</tr>
<tr>
<td><strong>Memory Hierarchy</strong></td>
<td>- Cache</td>
<td>- Local Memory</td>
<td>- On-chip Memory Banks</td>
</tr>
<tr>
<td></td>
<td>- Global Memory</td>
<td>- Texture Memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Global Memory</td>
<td></td>
</tr>
<tr>
<td><strong>Platform Specific Optimization</strong></td>
<td>- Hyper Threading Technology</td>
<td>- Coalesce Memory Accesses</td>
<td>- Stream-based Processing</td>
</tr>
<tr>
<td></td>
<td>- SIMD Instruction Sets</td>
<td>- Avoid Code Branches</td>
<td>- On-chip Memory Banks</td>
</tr>
<tr>
<td></td>
<td>- Data caching</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Memory Hierarchy</strong></td>
<td>- Cache</td>
<td>- Local Memory</td>
<td>- Block RAM (BRAM)</td>
</tr>
<tr>
<td></td>
<td>- Global Memory</td>
<td>- Texture Memory</td>
<td>- Distributed Memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Global Memory</td>
<td></td>
</tr>
<tr>
<td><strong>Important Metrics</strong></td>
<td>- Performance</td>
<td>- Performance</td>
<td>- Performance</td>
</tr>
<tr>
<td></td>
<td>- Power</td>
<td>- Power</td>
<td>- Resource Utilization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Global Memory</td>
<td>- Power</td>
</tr>
</tbody>
</table>

Figure 2.3 shows the simplified version of a multi-core CPU architecture. Contrary to a single-core architecture, different optimization strategies are performed to obtain high-performance implementation. Each core on a multi-core CPU can support Single Instruction Multiple Data (SIMD) instructions. With more than one core and SIMD instructions, higher data-level parallelism can be achieved because of larger numbers of computing cores. In addition to data-level parallelism, multi-core CPUs also enable task-level parallelism. To achieve task-level parallelism, each core in a multi-core CPU can work on a different task concurrently.

**Programming Models**

Compared to other platforms, CPUs execute an application in a sequential style. A well-known example of the programming languages for this platform is C. With the introduction of multi-core CPUs, some libraries or extensions for C are also introduced to be able to take advantages of the underlying platform. Several parallel programming libraries used for this purpose are OpenMP, pthread, platform-specific intrinsics e.g. Streaming SIMD Extensions (SSE), AVX, and ARM NEON. For instance, writing a multithreading application which utilizes the parallelism ability of the platform using those libraries is possible.
2.2.2 GPU

GPU was originally designed to handle graphics drawing applications, such as image rendering, video games, and CAD tools. As the hardware architecture of GPU changed from fixed graphics pipeline architecture into programmable architecture, people started to use this platform for general-purpose or non-graphical computations, such as scientific computation application. Implementing those general-purpose computation applications on GPU was challenging until the introduction of modern GPU architecture. The new architecture evolves from programmable pipelines into unified computing architecture which enables the GPU to be also used for general-purpose computing applications.

One of the distinguishing characteristics of GPUs is its massive numbers of cores enabling the computation of tens of thousands of data items in parallel. With this feature, the GPU is suitable for data-intensive computing applications. Image processing algorithms are one of the applications that can exploit the parallelism capabilities of GPUs.

There are mainly two different kinds of GPU devices, e.g. discrete and integrated GPU. A discrete GPU is a GPU device that is separated from the CPU. This kind of GPU is produced by, for example, NVIDIA and AMD. Another GPU type is an integrated GPU which is integrated with CPU cores in the same chipset. Some examples are Intel HD Graphics and AMD APU.

The advantage of an integrated GPU is its zero-copy data transfer from and to the CPU since both of them share the same physical memory. In contrast, a dedicated GPU has a separate memory system, and data transfer is performed through PCI express bus. The memory hierarchy in an Intel processor graphic gen9 and a Nvidia GPU can be seen in Figure 2.4 and Figure 2.5 respectively.

Programming Models
The emerging programming models for GPUs are Compute Unified Device Architecture (CUDA) and OpenCL. Both of them enable software programmers to program GPUs for general-purpose computing. Due to many-core-nature of GPUs, the programming languages are designed to be able to specify parallelism at a high-level of abstraction.

2.2.3 FPGA

FPGA is a semiconductor device that consists of configurable logic blocks connected using reprogrammable interconnects. The logic blocks typically consist of a Look-Up Table (LUT), Full
CHAPTER 2. BACKGROUND

Figure 2.4: Intel Processor Graphics gen9 Memory Hierarchy [2]

Figure 2.5: Nvidia Pascal Architecture [3]
Adder (FA), and a Flip-Flop (FF). In modern FPGA platforms, the logic blocks can also be implemented as several kinds of fixed function modules, such as generic Digital Signal Processor (DSP), Block Read Access Memory (BRAM), and multiplier blocks. The routing fabrics connect the logic blocks to form a specialized fixed function hardware. Hence, the elements in an FPGA platform can be reprogrammed after the fabrication process is complete. The generalized version of an FPGA architecture can be seen in Figure 2.6.

This platform becomes a potential accelerator due to its flexibility, programmability, inherent parallelism, and low power usage. Despite operating at a low clock frequency, high-performance implementation of image processing algorithms can still be achieved by fully utilizing their high degree of parallelism. This parallelism is possible since all of the computing resources can be used simultaneously enabling both data-level and task-level parallelism.

**Programming Models**

An implementation design on an FPGA platform can be programmed using a HDL. Some examples of HDLs are Verilog and VHDL. In addition, a higher abstraction approach is offered to ease the task of programming an FPGA. Xilinx offers an HLS tool, Xilinx Vivado [19] which is based on C language to create an FPGA design. More recently, OpenCL language is also used to program FPGAs which offers a higher level of abstraction by enabling programmers to use OpenCL constructs to create a design targeting FPGA platforms. Both SDAccel [20] from Xilinx and Intel FPGA SDK for OpenCL [21] from Intel, have supported this programming model.

### 2.2.4 Opportunities and Programmability Challenges of FPGAs

Implementing an application targeting an FPGA platform is quite challenging [15]. A naive porting of a software image processing implementation to FPGAs would result in an under-utilized hardware platform leading to a low-performance implementation. Sufficient knowledge of the underlying hardware architecture and a different programming paradigm are needed to implement application onto FPGAs. Instead of thinking in a sequential processing style, a parallel and deep-pipelines execution model should be adopted resulting in steep learning curves.

In FPGAs, resource utilization is also an important design metric. Since the hardware designers are responsible for the design and the implementation of the hardware structure, it is important to consider
By understanding several unique characteristics of FPGAs, high performance and efficient implementation in FPGAs can be obtained.

**Memory hierarchy.** Unlike other platforms, FPGAs do not have a fixed memory hierarchy. There are lots of re-programmable distributed memory blocks or BRAMs or LUTs on the hardware. Since these memories are located near the operation blocks, it offers huge memory bandwidth. Moreover, the memory blocks can be programmed into several memory banks which enable multiple data fetching in the same clock cycle. In the case of local operators, multiple operations to produce the value of an output pixel can be performed in one cycle. For the implementation of image processing pipelines, the memory hierarchy of FPGAs offers advantages since the data transfer between stages can be performed through on-chip memories reducing the latency of data accesses. By effectively utilizing the huge numbers of memory banks, high-performance implementations can be achieved.

**Stream-based processing.** One of the limitations of FPGAs is the limited size of memory resources. In the case of image processing, the typical execution model in multi-core CPUs and GPUs is a buffer-wise model in which the whole image frame is stored before being computed. Since the size of on-chip memories in FPGAs are typically too small to store the entire frame, accessing the larger off-chip memory results in increased latency. Hence, to achieve high performance, stream-based processing should be adopted in which the computation starts on-line as soon as enough data is available. Combined with deep-pipeline architecture, the throughput of the system can be improved.

**Arbitrary bit-width data types.** Another characteristic of FPGAs is their arbitrary precision arithmetic operations. In contrast to multi-core CPUs and GPUs which use data types with 8-bit boundaries, it is possible and desirable to define a specific bit-width for the operands. The effective use of bit-widths is crucial for obtaining efficient resource utilization in an FPGA.

Compared to CPUs and GPUs, FPGAs operate on a much lower clock frequency. Despite the low clock frequency, it is possible to achieve comparable high-performance implementation. However, it depends on the how the hardware structure is implemented in the code which results in huge gaps to the equivalence software implementation. A high-level abstraction solution should hide these hardware design knowledge from the application developers and apply the necessary optimizations implicitly.

### 2.3 Portable Computing Solutions

There have been several studies on the possibility to achieve cross-platform portable implementations. This section summarizes the existing portable computing solutions.

#### 2.3.1 OpenCL

Open Computing Language (OpenCL) is the standard platform-independent parallel programming model for multi-core and heterogeneous computing platforms. Originally, OpenCL was designed to ease the programming challenges targeting GPUs. As the specification is developed over time, the programming model aims to be a portable computing solution targeting different kinds of platforms as accelerators. Currently, it supports AMD and Nvidia GPUs, multi-core CPUs, AMD APUs, MIC (Intel Xeon Phi), DSPs, and FPGAs. Moreover, OpenCL is also used for custom architectures which are mainly developed in academic studies [22]. With this vast support, OpenCL is rapidly moving forward to become a true platform-independent programming model.

Although OpenCL enables cross-platform functional portability, the performance portability across devices is not guaranteed, especially across different kinds of platforms [18] [23] [24]. Functional portability is the ability of an implementation to run correctly across different target devices. On
CHAPTER 2. BACKGROUND

the other hand, performance portability concerns the ability of the common implementation to achieve the same performance compared to the manually-optimized code for the specific target device.

OpenCL can be considered as a low-level programming model because it still exposes the hardware capabilities to the software programmers. Platform-specific optimizations are still required to obtain high-performance implementations. Moreover, the approach taken by OpenCL is by providing a low-level, uniform, and device-independent platform model to the user and leaving the mapping to the OpenCL-supported physical devices to the device vendors. As a result, the performance is affected by the maturity of the OpenCL implementation on particular devices. In addition, the performance is vulnerable to changes in the target architecture. The experimental study in [24] has demonstrated the sensitivity of a particular target platform to the optimization targeted on different kinds of architecture. A generic implementation using OpenCL is possible resulting in a portable implementation, but it comes at the cost of relatively lower performance and complex code (e.g. using pre-processors to separate the code for each kind of architecture).

**Auto-tuning Approach**

Several studies in [23] and [24] proposed an auto-tuning approach to improving the performance portability of OpenCL. This method automatically tunes several options, such as work-item size, work-group size, and local memory usage, to improve the performance on GPUs using the same OpenCL code. The main drawback of this approach is that it is limited to a specific kind of devices. Some options may not be applicable to other architectures. For example, Single Work- Item OpenCL kernel is recommended in FPGAs, and as a result, the work-item and the work-group size do not apply anymore. In FPGAs, It might require structural changes in the OpenCL code instead of only some parameter changes.

**2.3.2 pocl**

The work in [25] proposed a portable OpenCL implementation called pocl. This framework improves the portability of OpenCL in both functionality and performance. By using an Intermediate Representation (IR) to retain the data parallelism information, further optimizations or transformations can be applied to the IR instead of the kernel code. The performance can be improved by applying platform-specific optimization strategies to the IR.

This framework has supported a wide range of architectures, both already commercialized and on those that are still under research. Compared to a direct source-to-source transformation solution, this approach is still able to maintain the parallelism information in the IR form. The IR code can be optimized later based on the target platform architecture to achieve performance portability.

To our knowledge, there is no study yet on performing the IR transformations in the pocl framework targeting FPGAs. It is not clear how effective the transformations on this low-level form is.

**2.3.3 HLLs**

Another solution is to use a higher level programming model. The motivation for using an HLL is to abstract the detailed knowledge of target architecture and coding techniques from the application programmers while trying to achieve high performance comparable to the hand-coded implementation. There are two kinds of HLLs described in this study:
Generic High-Level Languages

Some emerging high-level programming languages are OpenACC, OpenMP 4.0, Kokkos, RAJA, and SyCL. These programming models abstract several common coding techniques targeting multi-core CPUs and GPUs, such as memory transfer method from the host to the accelerator, kernel and host code separation, and synchronization. The performance portability depends on the compiler implementation which performs optimizations guided by the programmers using pragmas and general coding techniques. OpenACC and OpenMP 4.0 rely on the annotations provided by the users in the source code. On the other hand, RAJA, Kokkos, and SyCL provide library-based solutions (e.g. Application Program Interfaces (APIs)) to guide the compilers to generate optimum implementations.

Domain Specific Languages (DSLs)

Similar to generic HLLs, DSLs provides a higher level of abstraction that is close to a specific domain. By utilizing the specific domain knowledge and platform-specific knowledge, DSLs can ease the programming efforts and generate optimum implementations. The optimizations performed by DSLs are better than the ones performed by generic HLLs because of the nature of the specific target domain. As a result, functional and performance portability can be achieved with an extra benefit of productivity. With these advantages, DSLs seems to be a promising solution to achieve both functional and performance portability and ease the programmability challenges of FPGAs.

Some examples of the DSL in the domain of image processing are Halide [23], PolyMage [24], HIPAcc [25], and Darkroom [26]. These DSLs in the image processing domain will be elaborated in the next chapter.

Figure 2.7 shows the hierarchy view of the portable computing solutions presented in this section.

2.4 Quality Measurement

In this thesis, the reference image processing algorithms use floating-point arithmetic. Depending on the architecture of the target platform, this data type might not be a perfect choice. For instances, floating-point arithmetic is expensive in FPGAs due to lack of dedicated Floating-Point Units (FPUs) in some FPGA platforms. As a result, a conversion from floating-point to other data
types (e.g. arbitrary precision fixed-point or integer data type) is needed, but with the tradeoff of less accurate computation values.

In the case of medical domain, the accuracy of the computation value is of particular importance. To evaluate the accuracy of the implementation, the computation results from a different data type are converted into floating-point and stored into a 16-bit image. The reference output image is also generated from the same algorithm but using a floating-point data type. The accuracy can be evaluated by comparing the output images. The quality measurement setup between the floating-point based implementation and the fixed-point based implementation can be seen in Figure 2.8. To have an objective quality measurement, the following metrics are used for the evaluation:

1. Number of error pixels.
2. Minimum/ Maximum absolute error. This metric shows the dev
3. Mean Squared Error (MSE) which is computed by averaging the squared intensity of the original image (e.g. in floating-point) and the output image (e.g. in fixed-point). It is expressed in the Equation 2.1.

\[
\text{MSE} = \frac{1}{H \times W} \sum_{y=0}^{H-1} \sum_{x=0}^{W-1} [x_{\text{float}}(x,y) - x_{\text{fixed}}(x,y)]^2
\]  

(2.1)

where \( H \) and \( W \) represent the height and the width of the image respectively.

These metrics provide a comprehensive view of the accuracy of the output image. Since the exact accuracy requirement depend on the end users, these metrics allow the users to evaluate whether the accuracy of the results is acceptable or not.
Chapter 3

Domain-Specific Languages

As described in the previous chapter, the computing platforms for implementing image processing algorithms are not limited to CPUs anymore. Computing architectures, such as GPUs and FPGAs, are also commonly used with some benefits that are not offered by CPUs. This vast computing ecosystem leads to the challenge of portability. Every architecture has its own programming language and paradigm. Due to their unique architectures, an implementation for a particular architecture does not result in the same performance on other architectures. Domain-Specific Languages are proposed to alleviate this increasing challenge.

3.1 DSL Overview

A DSL is a specialized programming language for a particular application domain [26]. In this type of programming language, the design of the constructs or notations is based on the common vocabulary or challenges of the problem domain. The notations allow the program to be written in a high-level concept close to the domain. This design may lead to the limited expressiveness of the language compared to the general purpose languages, such as C, OpenCL, and Java. However, if the DSL is carefully designed, it can become a powerful language on a particular domain resulting in the decrement of overall development costs and improvement of the development time [27].

3.2 Image Processing DSLs

Halide

Halide [28] is a DSL designed for easily describing image processing pipelines and generating high-performance image processing implementation on various kinds of platforms. This language adopts a functional style description of image processing algorithms as a set of functions from coordinates to values. An example of an image processing pipeline implemented in Halide can be seen in Listing 3.1. It is considered as an embedded (internal) DSL in which the front-end constructs are implemented as a C++-based library. Hence, the notations in Halide are strictly tied to the C++ syntax.

The main feature of Halide is its separation between the algorithm description and the scheduling part. The algorithm description describes the definition of the image processing pipelines in a functional style which is platform-independent. On the other hand, the scheduling, which is closely related to the target platform, defines the constraints on the order of execution and placement of data between stages. Since it is platform-dependent, a different scheduling can be specified for
each kind of platform. With this separation, any changes or updates on the scheduling part will not affect the functional correctness of the algorithm. Hence, the functional correctness can still be maintained. The examples of scheduling primitives in Halide can also be seen in Listing 3.1. A more elaborate description of the scheduling primitives in Halide can be found in [28].

Initially, Halide only supported multi-core CPUs and GPUs and was able to generate highly optimized code on those platforms. Recently, Pu et al. [29] proposed a Halide-based DSL for generating image processing pipelines for heterogeneous systems, e.g. CPU-FPGA. Most of the scheduling language existing in Halide can be reused and only three new constructs (e.g. `linebuffer()`, `fifo_depth()`, `accelerate()`) for the scheduling are added in order to generate efficient FPGA implementations. In addition, several compilation passes are also added, such as loop perfecting which is helpful for the Xilinx Vivado HLS tool to generate more efficient codes, and data-flow-based Halide IR generation to generate line-buffer based image processing pipelines.

```c
// Algorithm description of 3x3 blur filter
blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);
blur_y(x,y) = blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1);

// Scheduling
// Some examples of scheduling.
// Compute the value of blur_x completely before blur_y.
// blur_x.compute_root();
// Compute the necessary region of blur_x at the start of each y iteration of blur_y.
// blur_x.compute_at(blur_y, y);
// blur_x.compute_at(blur_y, y);
```

Listing 3.1: Algorithm description of a blur filter in Halide

**HIPAcc**

HIPAcc [4] provides a C++-based embedded DSL for the image processing domain. Unlike Halide, the processing kernels in HIPAcc are implemented imperatively. The image processing kernels are encapsulated in C++ classes in which a particular implementation pattern should be followed. Listing 3.2 shows the implementation of the Gaussian blur filter in HIPAcc.

Several DSL primitives are provided to implement image processing algorithms: `Image` to describe the data storage of the input pixels, `IterationSpace` to describe the region of the output image computation, `Pyramid` to implement multi-resolution image processing applications, `BoundaryConditions` to perform boundary handling, and `Mask` and `Domain` to specify the filter coefficient and the region of interest respectively. These primitives guide the framework to generate proper code and to perform platform-specific optimizations.

The compiler framework is divided into two parts: front-end and back-end. The front-end is implemented on top of Clang, a C language family front-end for Low Level Virtual Machine (LLVM). Clang performs source-to-source translation to an IR in the form of Abstract Syntax Tree (AST). The next step is the responsibility of the back-end part of the framework. The next phase of the compilation performs some analysis on the AST nodes in which the information is used to perform target-platform-dependent optimizations. Finally, the transformed AST nodes are stored to a file as the target code.

The structure of the framework can be extended to support new target platforms. Initially, the framework is developed to generate optimized image processing implementations targeting CPUs and GPUs. Recently, FPGA platforms have been supported without changing or extending the DSL constructs [30].
CHAPTER 3. DOMAIN-SPECIFIC LANGUAGES

```cpp
Class LinearFilter : public Kernel<uchar> {
private:
    Accessor<uchar> &input ;
    Mask<float> &mask ;
    size_t size ;

public:
    LinearFilter(IterationSpace<uchar> &iter , Accessor<uchar>,
    Mask<float> &mask , size_t size ) :
    Kernel(iter), input(input), mask(mask), size(size) {
        add_accessor(&input); }

    void kernel() {
        float sum = 0;
        int range = size/2;
        for (int yf = -range; yf <= range; ++yf)
            for (int xf = -range; xf <= range; ++xf)
                sum += mask(xf , yf) * input(xf , yf);
        Output() = (uchar) sum ;
}
```

Listing 3.2: Gaussian Blur Filter in HIPAcc copied from [4]

**DarkRoom**

Darkroom [31] is a DSL designed to ease the implementation of image processing pipelines. The DSL provides the suitable semantics to generate a line-buffered pipelines implementation. This line-buffered description can later be synthesized into FPGA or ASIC, or CPU code. The programming model is similar to Halide in which each processing stage is specified as a function mapped from 2D coordinates to the value of those coordinates. Listing 3.3 shows the unsharp mask operation implemented in Darkroom DSL. Currently DarkRoom supports code generation for CPUs and FPGAs.

```cpp
bx = im(x,y) (I(x-1,y) + I(x,y) + I(x+1,y))/3 end
by = im(x,y) (bx(x,y-1) + bx(x,y) + bx(x,y+1))/3 end
difference = im(x,y) I(x,y)-by(x,y) end
scaled = im(x,y) 0.1 * difference(x,y) end
sharpened = im(x,y) I(x,y) + scaled(x,y) end
```

Listing 3.3: Unsharp Mask Operation in Darkroom

**PolyMage**

PolyMage [32] is a DSL and a compiler for automatic code optimization and generation of image processing pipelines. It is an internal DSL based on Python adopting functional programming style. Listing 3.4 shows the partial code for the implementation of an unsharp mask filter. The language provides several constructs such as Parameter to declare parameters like width and height, Variable to declare the labels for functions, Image for the input image, Interval to define the lower bound, upper bound, and step value used in Function, and Function to define the image processing stage. In addition, a Stencil construct can be used to easily define a spatial filtering operation.

Unlike other DSLs, PolyMage adopts and extends polyhedral analysis techniques to perform optimization on the generated code. Polyhedral is a framework for representing loop nests and
Listing 3.4: PolyMage DSL code for Unsharp Mask [5]

performing loop transformations.

In [5], PolyMage DSL framework is extended to support FPGA platforms using C-based HLS as the target. Similar to Halide-HLS, the PolyMage DSL code is translated into a Directed Acyclic Graph (DAG) which resembles the dataflow model of computation. Each function defined in the source code corresponds to a node in the graph with clear dependencies to the input and the output nodes. During the code generation process targeting FPGAs, the dependencies between stages are implemented using line buffers.

Discussion

There exist several DSLs for image processing pipeline with their own advantages and disadvantages. Several metrics are used to evaluate the suitable DSL for the implementation of image processing pipelines.

Programmability.

The learning barrier to a programming language determines how wide the language can be adopted. Fortunately, the functional-style programming model used by Halide and Darkroom allows an easier transition from the conventional programming languages. However, an extra learning time is needed to learn the scheduling primitives in Halide. HIPAcc, on the other hand, uses the imperative
approach in the form of C++ classes to implement the image processing computations. The development effort is also reduced using several proposed constructs which represent the common patterns of image processing operations. PolyMage also adopts the functional-style approach, but the resulting language design is less intuitive and more cluttered compared to Halide or Darkroom.

Development Activity and Community Support.
All of the DSLs discussed here are an open-source project. However, Halide is the only DSL framework that is actively developed by the community. During the time this thesis was being written, there were still some active works committed in the main repository. It is also used by Google to implement image processing applications for the Google Pixel phone. Moreover, the backend code generation is also being extended to support new target platforms, for example, Qualcomm Hexagon [33].

Flexibility.
The distinguish feature of Halide is the separation between the algorithm description and the scheduling part. The scheduling part represents the platform-specific optimization declaration; the programmers can manually specify how the image processing pipelines can be optimized. Other DSLs adopt a single-source code in which the platform-specific optimization process is the responsibility of the compiler. In this case, the application developers lose control of the optimization process performed by the compiler. Moreover, design space exploration is not possible anymore. As a result, Halide is more flexible from the perspective of the application developers. Halide DSL offers a balanced level of abstraction compared to the other DSLs.

Extendibility.
Technically, all of the DSL frameworks can be extended to support other new target platforms and embed new optimization strategies. However, the framework design might limit the ease of extending the framework. All of the DSL frameworks separate the front-end and the backend code generation to split the platform-independent and the platform-dependent optimization process. Most of the transformations are performed in an IR form allowing platform-independent optimizations. Halide, PolyMage, and Darkroom implement their own IR form, while HIPAcc utilizes the standard Clang AST as its IR. The framework of Halide and HIPAcc is implemented using C/ C++ while PolyMage and DarkRoom are implemented using Python and Terra language respectively. These differences in the programming language pose an extra challenge to extend the framework.

Target Platforms.
Halide and HIPAcc have supported all of the target platforms considered in this research: multi-core CPU, GPU, and FPGA. During this study, PolyMage and DarkRoom supports only CPUs and FPGAs.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Halide</th>
<th>HIPAcc</th>
<th>PolyMage</th>
<th>DarkRoom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmability</td>
<td>+++</td>
<td>+++</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Development Activity and Community Support</td>
<td>+++</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Flexibility</td>
<td>+++</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Extendibility</td>
<td>+++</td>
<td>+++</td>
<td>++</td>
<td>++</td>
</tr>
</tbody>
</table>

Table 3.1: Comparisons of different DSLs in the image processing domain.

Table 3.1 summarizes the comparisons between Halide, HIPAcc, PolyMage, and Darkroom. In general, Halide has the potential to provide a portable computing solution in the image processing domain based on the advantages in extendibility, flexibility, and development activity.
CHAPTER 3. DOMAIN SPECIFIC LANGUAGES

3.3 Current Limitations of Halide-HLS

Despite the benefits offered by Halide, there are several limitations identified during the experience using this DSL. In this section, the limitations of the current framework are described based on the prior experience in implementing image processing pipelines targeting FPGAs.

Fixed bit-width C-based Data types

Currently, Halide only supports standard C-based native data types, which are on 8-bit boundaries. Some examples of those data types are 8, 16, 32, and 64 bits integer. In addition to integer data type, floating-point both single (32-bit float) and double precision (64-bit float), which are commonly used in scientific applications, are also supported. Those native data types are sufficient to implement most of the applications on several modern target platforms, such as CPUs and GPUs.

In terms of performance, arithmetic operations using C-native data types are well-supported in common architectures. Good performance can be achieved using fast clock frequency. For floating-point data types, however, the computation is much more complex and require much longer time to complete. To alleviate this bottleneck, on modern target platforms, such as CPUs and GPUs, dedicated floating-point arithmetic units are embedded in parallel to the basic arithmetic units. Hence, the arithmetic operations using floating-point can be improved significantly.

Unlike CPUs and GPUs, floating-point implementation in FPGAs is more complicated due to the conflicting design goals: between high throughput and resources utilization. Different FPGA vendors offer different solutions to support floating-point operations on FPGAs. Intel Altera FPGAs embed hardened DSP blocks with floating-point units in the FPGA. While in Xilinx FPGAs, floating-point operations can be implemented in several ways, such as a combination of DSP resources and additional logic blocks, using provided floating-point IP core, and system generator. Regardless the implementation differences, it is widely accepted that the floating-point operations require a large amount of FPGA resources.

On the other hand, using fixed bit-width C-based native data types on an FPGA platform can lead to inefficient hardware implementation due to the limited bit-width choices. For instance, the result of the addition operation between two 16-bit integer variables can be assigned to either 16-bit or 32-bit output value depending on the type of the output variable. Defining a 16-bit output variable has the possibility of overflow while using a 32-bit output variable will result in hardware with excessive bits. More resources will be needed for a multiplication operation, e.g. the number of DSP48 blocks and other logic used is more than necessary.

Since the context of the implementation of the image processing algorithm is in the medical domain, it is important to maintain the accuracy result of the implementation compared to the reference design (e.g., MATLAB model). The implementation on FPGAs requires the tradeoffs between resource utilization and accuracy of the result. The limited choices of data types in Halide only offers a small range of implementation choices for the users. For instance, using floating-point data type may result in a high accuracy of result with the tradeoff of high resource utilization, while on the other hand using 8-bit integer may lead to large loss of accuracy but with much smaller resource utilization.

To illustrate the effect of data type choices on the results accuracy, several implementations of a Gaussian filter with floating-point, 8-bit integer, and 16-bit integer data type are compared. The results are compared with the reference image computed using the double-precision floating-point data type in MATLAB. Table 3.2 shows the comparisons of the accuracy between the Gaussian filter implementations with a window size of 5x5 and an image size of 512x512. The errors distribution of the 8-bit integer implementation can be seen in Figure 3.1. As shown in the table, the errors of the floating-based implementation are negligible; the number of error pixels are not zero since the computation in MATLAB employs double-precision floating-point numbers (i.e.,
80-bit floating-point). On the other hand, the accuracy errors in the 8-bit integer implementation is larger than the errors in the floating-point and 16-bit integer implementations. As the bit-width is reduced to 8 bits, the accuracy errors start to increase in term of minimum error, maximum error, and mean square error.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Gaussian Filter (Float)</th>
<th>Gaussian Filter (16-bit)</th>
<th>Gaussian Filter (8-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of different pixels</td>
<td>129539 (49%)</td>
<td>262144 (100%)</td>
<td>262144 (100%)</td>
</tr>
<tr>
<td>Minimum Absolute Error</td>
<td>0.000015</td>
<td>0.000015</td>
<td>0.003189</td>
</tr>
<tr>
<td>Maximum Absolute Error</td>
<td>0.000015</td>
<td>0.000107</td>
<td>0.032135</td>
</tr>
<tr>
<td>MSE</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000323</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of several implementations of Gaussian Filter in term of accuracy.

![Errors Distribution of Gaussian Filter 8-bit Integer](image1.png)

Figure 3.1: Error Distribution of 8-bit integer based Gaussian Filter.

The computation errors may increase as the applications become complex. Table 3.3 shows the comparisons of accuracy between several implementations of a Unsharp filter. This application contains more than one processing step, and as a result, the computation errors can be accumulated through each stage.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Unsharp Filter (Float)</th>
<th>Unsharp Filter (8-bit) without scaling</th>
<th>Unsharp Filter (8-bit) with scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of different pixels</td>
<td>260809 (99%)</td>
<td>262144 (100%)</td>
<td>261953 (99%)</td>
</tr>
<tr>
<td>Minimum Absolute Error</td>
<td>0.000015</td>
<td>0.039216</td>
<td>0.003922</td>
</tr>
<tr>
<td>Maximum Absolute Error</td>
<td>0.001968</td>
<td>0.576471</td>
<td>1.000000</td>
</tr>
<tr>
<td>MSE</td>
<td>0.000001</td>
<td>0.126008</td>
<td>0.000553</td>
</tr>
</tbody>
</table>

Table 3.3: Comparisons of accuracy between several implementations of Unsharp Filter.

In the unsharp filter application, the effect of reducing bit-widths is more significant. The absolute differences can reach up to 1 as the data types are changed from floating-point to an 8-bit integer operations. As shown in Figure 3.2, the number of pixels with significant error values (i.e., more than 10000) increases. This result represents an unsharp filter implementation in which the floating-point based implementation is naively converted into an 8-bit integer implementation.
The accuracy errors in an 8-bit integer implementation can be further reduced through a proper scaling technique, i.e., normalization. As can be seen in Table 3.3 and Figure 3.3, the results accuracy can be improved using this approach. However, this process leads to the next limitation.

Non-portable Code

Converting floating-point implementation to integer implementation is not a trivial task. As described previously, a naive conversion could negatively affect the accuracy of the computation. The conversion can break the portability benefit offered by Halide since the algorithm description
has to be modified. To illustrate this issue, consider the following code fragments.

```cpp
// Algorithm description for 5x5 convolution filter
// (using integer data types)
RDom win(-2, 5, -2, 5);

// Compute and normalize kernel value.
kernel_f(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);
kernel(x,y) = cast<uint8_t>( kernel_f(x) * 255 /
  (kernel_f(0) + kernel_f(1)*2 + kernel_f(2)*2));

// Define the convolution
clamped = BoundaryConditions::repeat_edge(input);
conv(x,y) += cast<uint32_t>(clamped(x+win.x,y+win.y) * kernel(win.x) *
  kernel(win.y));
conv1_shifted(x,y) = cast<uint8_t>(conv(x,y) >> 16);
output(x,y) = conv1_shifted(x,y);

Listing 3.5: 8-Int-based Gaussian Filter Implementation in Halide
```

```cpp
// Algorithm description for 5x5 convolution filter
//(using floating-point)
RDom win(-2, 5, -2, 5);

// Compute and normalize kernel value.
kernel_f(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);
kernel(x,y) = ( kernel_f(x) /
  (kernel_f(0) + kernel_f(1)*2 + kernel_f(2)*2));

// Define the convolution
clamped = BoundaryConditions::repeat_edge(input);
conv(x,y) += (clamped(x+win.x,y+win.y) * kernel(win.x) * kernel(win.y));
output(x,y) = conv1(x,y);

Listing 3.6: Floating-Point-based Gaussian Filter in Halide
```

The code fragment in Listing 3.5 shows the implementation of a 5x5 convolution filter using floating-point and integer types. The application consists of the computation of the kernel coefficients and the convolution operation. Compared to the floating-point implementation, several changes in the algorithm description can be observed: the uses of `cast<T>` to convert the coefficient value from floating-point to integer, the normalization of the converted value, and the scaling of the output value.

The original code shown in Listing 3.6 is implemented using floating-point data type, especially for the computation of the kernel coefficients. In contrast, the algorithm has to be re-implemented using integer data types in order to reduce the resource utilization in an FPGA. The effect is that the algorithm description is not fully independent from the platform anymore because it is affected by the choice of the data types. The generated code from a CPU-based platform, for instance, uses integer data types instead of floating-point. Hence, this limitation reduces the portability advantage of the framework.
CHAPTER 3. DOMAIN-SPECIFIC LANGUAGES

Off-chip Boundary Handling

Boundary conditions handling is common in image processing algorithms, especially in local operators. In some algorithms, instead of cropping the border-dependent pixels, it is better to obtain the approximate results. In this kind of application, the computation of the edge pixels depends on the value of the pixels outside the image boundaries. Figure 3.4 shows the boundary handling in a common convolution filter. The value of those pixels can be calculated in several ways, such as clamp, repeat, constant values, mirroring, and undefined. Each method produces different edge pixels value; hence, the algorithm designer usually determines the desirable method of boundary conditions.

![Out-of-border pixels](image)

Figure 3.4: Boundary Handling in a Convolution Filter.

In the current version of Halide, the boundary handling is performed by pre-calculating the values for the pixels outside the image boundaries and adjusting the computation domain. In the local operator cases, the computation domain will be larger than the image domains. With this approach, it can be guaranteed that the subsequent processing kernels always have the needed data for the processing. The same approach is also used in Halide-HLS in which the host code performs the calculation of the ghost zones before sending the input pixels to the kernel. Similarly, the kernel does not need to handle the boundary issues when performing the local operators resulting in smaller kernel codes without conditional statements.

The advantage of this approach is that the kernel can perform the computation at a constant rate without having to handle the boundary cases. The necessary computation domain is automatically adjusted to take into account the boundary-dependent pixels. In addition, it is easier to generate the structure of each kernel. The drawback of this method is that the whole input image has to be buffered to be pre-processed. This buffering process will increase the latency at least as large as the size of the input image.

However, this method cannot be applied in the context of streaming-based architecture on FPGAs. In this kind of architecture, the processing hardware kernel typically obtains streaming image data from the input sources directly. Since there is no pre-processing performed on the image data, handling the boundary conditions must be performed. In this case, the processing kernels implemented on an FPGA platform is responsible for this task.
Chapter 4

Implementation

In Chapter 3, several DSLs have been studied. After the review of those DSLs, Halide has been chosen as the most promising solution to address the portability and the programmability challenges. Additionally, the limitations of Halide targeting FPGAs have been identified. In this chapter, several extensions are proposed to address the limitations.

4.1 Extending Arbitrary Precision Data Types on Halide

Unlike CPUs and GPUs, FPGAs support arbitrary precision data types. The effective use of this capability can achieve the same accuracy of result while generating resource-friendly hardware implementations. Hence, it is desirable to support arbitrary precision data types in Halide.

Using arbitrary precision data types in DSLs have been discussed in several studies. In [34], it is possible in HIPAcc to define arbitrary precision data types using pragmas to generate a resource friendly implementation in FPGAs. However, this approach assumes the operations are performed in integer data types instead of floating-point. Hence, it does not address the challenge in converting floating-point to fixed-point. While in PolyMage for FPGAs [5], it is not clear how the programmers can specify the use of arbitrary precision data types.

It has been identified and described in Chapter 3 that the image processing algorithms are generally designed using floating-point types. In addition, converting floating-point operations to integer operations for FPGAs is not a trivial task. Some additional stages are needed to properly adjust the computation results. In this study, arbitrary precision data types are introduced and extended into Halide.

In Halide, there is a one-to-one mapping of data types between Halide and C/C++. The Halide data types are characterized by the type name followed by the bit-width placed in parentheses. Table 4.1 shows the mapping of the data types and shows the legal data types in Halide. During the compilation phase, the native data types defined in the Halide code will be converted into the internal Halide data types.

To extend the arbitrary precision data types into Halide, two new classes representing both arbitrary precision integer data type and arbitrary precision fixed-point data type are created. The signatures of those data types can be seen in Table 4.2. By providing the information of the total bit-width and the bit-width of the integer part of the fixed-point type, the bit-width of the fractional part can be inferred automatically. If the total_bits is equal to int_bits, it represents the arbitrary precision integer data type. For example, a fixed-point data type with 2 bits for the integer part and 8 bits for the fractional part can be defined with either FixedPoint(11, 3) or UFixedPoint(10,2) depending on the sign of the value.
CHAPTER 4. IMPLEMENTATION

### C/C++ Native Data Types

| Signed Integer (e.g. int8_t, int16_t, int32_t, int64_t) | Int(8), Int(16), Int(32), Int(64) |
| Unsigned Integer (e.g. uint8_t, uint16_t, uint32_t, uint64_t) | UInt(8), UInt(16), UInt(32), UInt(64) |
| Floating Point 32-bit and 64-bit | Float(32), Float(64) |
| Pointer | Handle() |

Table 4.1: Mapping between C/C++ native data types and Halide data types.

<table>
<thead>
<tr>
<th>Halide Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FixedPoint</td>
<td>Signed Fixed-Point Type</td>
</tr>
<tr>
<td>(int total_bits, int int_bits)</td>
<td>total_bits: total bit-width.</td>
</tr>
<tr>
<td></td>
<td>int_bits: number of bits for the integer part.</td>
</tr>
<tr>
<td>UFixedPoint</td>
<td>Unsigned Fixed-Point Type</td>
</tr>
<tr>
<td>(int total_bits, int int_bits)</td>
<td>total_bits: total bit-width.</td>
</tr>
<tr>
<td></td>
<td>int_bits: number of bits for the integer part.</td>
</tr>
</tbody>
</table>

Table 4.2: Proposed Fixed-Point Data Types in Halide.

The data promotion rules in Halide have to be updated with the introduction of the arbitrary precision data types. Fixed-point data type has a higher priority than integer data type, but has lower priority than the floating-point type because of the accuracy of the data type. With this priority rule, the following new type promotion rules in Halide are applied.

1. Operation between floating-point (single and double) and fixed-point results in floating-point data type, unless it is explicitly specified to cast the output type into fixed-point.

2. Operation between integer (signed and unsigned) and fixed-point results in fixed-point data type. However, no conversion needs to be performed.

3. If one type is unsigned fixed-point and the other is signed fixed-point, then the unsigned fixed-point type will be promoted to signed fixed-point type with an extra 1 bit for the bit-width.

4. If both types are signed/unsigned fixed-point, then the resulting data type is signed/unsigned fixed-point data type. The bit-width of the output data type depends on the operation performed on the input types. By adjusting the bit-width of the resulting data type, the accuracy of the computation can be preserved. Table 6 shows the data promotion rules for fixed-point data types in respect to the performed operation.

<table>
<thead>
<tr>
<th>Input Type A</th>
<th>Input Type B</th>
<th>Operation</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FixedPoint (Ta, Ia)</td>
<td>FixedPoint (Tb, Ib)</td>
<td>Add</td>
<td>FixedPoint (max(Ta, Tb) + 1, max(Ia, Ib) + 1)</td>
</tr>
<tr>
<td>FixedPoint (Ta, Ia)</td>
<td>FixedPoint (Tb, Ib)</td>
<td>Mul</td>
<td>FixedPoint (Ta + Tb + 1, Ia + Ib + 1)</td>
</tr>
<tr>
<td>FixedPoint (Ta, Ia)</td>
<td>FixedPoint (Tb, Ib)</td>
<td>Sub</td>
<td>FixedPoint (max(Ta, Tb) + 1, max(Ia, Ib) + 1)</td>
</tr>
<tr>
<td>FixedPoint (Ta, Ia)</td>
<td>FixedPoint (Tb, Ib)</td>
<td>Div</td>
<td>FixedPoint (Ta + Tb + 1, Ia + Ib + 1)</td>
</tr>
</tbody>
</table>

Table 4.3: Proposed Fixed-Point Data Types in Halide.
Table 4.3 shows the data type promotion rule for different arithmetic operations. The extension of arbitrary precision data types in Halide will be combined with the proposed separation between the algorithm description and the data type specification.

### 4.2 Separation between Algorithm Description and Data types

The difference in data types for different target platforms results in non-portable implementation because there is a need to change the algorithm description. Moreover, the manual conversion is non-trivial and error-prone. To address this limitation, a separation between the algorithm description and the data types, in addition to scheduling, is proposed. The proposed extension aims to improve the portability offered by the current Halide framework with an addition scheduling for data types. This extension can reduce the programmability gap to FPGA platforms.

The current version of Halide has separated the algorithm description and the scheduling using separate functions, `generate()` and `schedule()`. To implement a separate region for specifying data types, an additional scheduling option is created which is called `type_schedule()`. The new structure of the Halide code can be seen in Listing 4.1.

```cpp
class ImageApp : Generator<ImageApp> {
    // Define Input and Output Arguments
    Input<T> arg_1{"arg_1"}
    Output<T> arg_2{"arg_2"}

    // Func/Expr declarations
    Func f1;
    Expr e1;

    void generate () {
        // Define the algorithm description
    }

    void schedule () {
        // Define the scheduling for each type of platform
        // e.g. CPU, GPU, FPGA
    }

    void type_schedule () {
        // Define the data types for each Func for each type of platform
        // e.g. CPU, GPU, FPGA
    }
}
```

Listing 4.1: Proposed structure of Halide code

With the separate data type scheduling, the algorithm description needs not to be changed anymore when the target platform is an FPGA. If the data types are not specified, then the data types will be inferred as usual using native data types. To generate different data types for a certain platform, the application developer has to manually specify the data types of the functions.

To manually specify the data type of a particular function, a new primitive called `cast_to (output_type, rhs_type)` is introduced. The first argument, `output_type`, specifies the output type of the function. This data type information will be propagated to the next processing stages which use the function. The next argument, `rhs_type`, is optional and it specifies a list of
Exprs or Funcs in the right-hand side of the assignment in which the type of these nodes need to be converted to the specified data type.

The new primitive can be applied to each Funcs in the Halide source code. If it is applied, the type of the Func node in the Halide IR AST will be converted into a fixed-point data type by adding a node representing a Cast<T> operation. The information of the new data type will be propagated to the next processing pipeline if the Func is called by the next stage. If there are any arithmetic operations, the type promotion rule for the fixed-point data type will be applied. This process is repeated until the last processing stage in the image processing pipelines. If the argument rhs_type is not empty, the AST nodes will be traversed from the output node until the input nodes. When the identifier of a node matches the identifier specified in the rhs_type list, a Cast<T> node will be added in front of the node.

Listing 4.2 shows the example of the data types scheduling for a Sobel filter application. The bit-width of the fixed-point type in each function can be manually specified using the primitive cast.to. During the code generation process, the framework may generate several intermediate variables. For instance, in conv_x and conv_y, some intermediate variables are generated to store the results of the multiplication operations. The bit-width of the intermediate variables is determined using the type promotion rule for the fixed-point data type. Using the primitive cast.to(), the growth of the bit-width in a function can be controlled. Figure 4.1 illustrates the IR nodes transformation for the function gray.

### Back-End Code Generation

During the code generation process, all nodes with fixed-point data types are converted into a specific construct for representing the arbitrary precision data types. In this work, the fixed-point library developed by Xilinx is used since the target platform is a Xilinx-based FPGA platform. The library supports many arithmetic operations using fixed-point data type and arbitrary precision integer data type. The mapping from the arbitrary precision data type in Halide to Xilinx library can be easily performed since it is a one-to-one mapping. Table 4.4 shows the mapping of the data types between Halide and Xilinx library.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Halide Type</th>
<th>Xilinx Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>(W, W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(W, I)</td>
<td></td>
<td>I: Bit-width of the integer part</td>
</tr>
</tbody>
</table>

Table 4.4: Xilinx Vivado Arbitrary Precision Data Types Mapping.
class SobelFilter : public Halide::Generator<SobelFilter> {

Expr R, G, B;

void generate() {
  R = 0.299f;
  G = 0.587f;
  B = 0.114f;
  clamped(x,y,c) = repeat_edge(input);
  gray(x,y) = (R*clamped(x,y,0) + (G*clamped(x,y,1) +
                     (B*clamped(x,y,2));
  conv_x(x,y) = -1 * gray(x-1, y-1) + (1 * gray(x+1, y-1) +
                 -2 * gray(x-1, y) + (2 * gray(x+1, y) +
                 -1 * gray(x-1, y+1) + (1 * gray(x+1, y+1);
  conv_y(x,y) = -1 * gray(x-1, y-1) + (1 * gray(x-1, y+1)) +
                 -2 * gray(x, y-1) + (2 * gray(x, y+1) +
                 -1 * gray(x+1, y-1) + (1 * gray(x+1, y+1);
  hw_output(x,y) = abs(conv_x(x, y)) + abs(conv_y(x, y));
}

void schedule() {
  // ...
}

void type_schedule() {
  if (get_target().has_hls_feature()) {
    // For FPGA Xilinx Vivado HLS
    clamped.cast_to(UFixedPoint(18,1));
    gray.cast_to(UFixedPoint(27,1), {
    {R, UFixedPoint(11, 1)},
    {G, UFixedPoint(11, 1)},
    {B, UFixedPoint(11, 1)});
    conv_y.cast_to(FixedPoint(28, 4));
    conv_x.cast_to(FixedPoint(28, 4));
    hw_output.cast_to(FixedPoint(27,3));
  }
  // ...
}

Listing 4.2: Code Snippet of Sobel Filter in Halide with the separate data type
Figure 4.1: IR Nodes (a) Before and (b) After Transformation.
4.3 On-chip Boundary Handling

There are two possible hardware structures of image processing pipelines to handle the boundary conditions.

In the first structure, the boundary handling is embedded in the processing kernel. This involves several conditions checking to handle all corner cases resulting in a larger processing kernel. In a large image processing pipeline, each processing stage needs to have redundant computation logic blocks for the boundary handling. As a result, the accumulated resources utilization will be high. Also, the structure is more challenging to be generated by the code generator while still maintaining the Initiation Interval (II) of 1.

Another approach is to put an additional input buffer stage preceding the processing pipeline. This stage is responsible for filling the necessary data in the row-buffer or line-buffer before sending it to the next processing step. The subsequent processing kernels will not have conditions for boundary handling. However, this increases the latency because of the larger size of computation domain.

Implementation

The implementation of the input buffer stage must not interrupt the continuous flow of the input data, e.g. maintaining the initiation interval of 1. In addition, the latency introduced by the implementation should be as low as possible.

To simplify the code generation, the input buffer stage is implemented as an HLS-synthesizable C++ template library, similar to the current implementation of line-buffering in Halide-HLS. The interface of the input buffer stage is shown in Listing 4.3. The code generator is responsible for generating the correct parameters for the input buffer stage, which depends on the boundary conditions and the window size of the local operators. The template-based design allows different optimizations for each situation with the same interface, e.g. line-buffer in a 2-D local operator and row-buffer in a 1-D local operator. In this thesis, the automatic code generation for the on-chip boundary handling is not implemented yet.

For the implementation of the input buffer stage, the input image is divided into three different regions as shown in Figure 4.2. The input buffer stage is designed for a stream-based processing environment.

Figure 4.3 illustrates the implementation of the input buffer stage for 2-D local operator and repeat boundary condition. Several important design consideration is as follows.

- Array partitioning to ensure that the flow of data is not limited by the memory read.
- Reuse of data by using local caches instead of reading data again from the memory.
- Loop unrolling to maintain II=1 in all corner cases.

```c
1 template < typename T, typename T_out , size_t IMG_EXTENT_0 ,
2     size_t IMG_EXTENT_1 , size_t EXTENT_0 , size_t EXTENT_1 , ... ,
3     size_t OFFSET_L , size_t OFFSET_R , size_t OFFSET_T , size_t OFFSET_B >
4 void input_buffer(hls::stream<T> &in_stream ,
5     hls::stream<PackedStencil<T_out , EXTENT_0 , EXTENT_1 , ... > >
6     &out_stream) {
7 //...
```

Listing 4.3: Input buffer stage interface
• Circular Buffering. (N-1) line buffering. Currently, only two kinds of boundary handling are supported: repeat and constant. Other types of boundary handling might require different methods to do it. For example mirroring/reflect. Since the interface is implemented as a template library, extending this type of boundary conditions is trivial.

Figure 4.3: Input Buffer Stage with repeat boundary condition for 2-D Local Operator.

Depending on the type of operator, different kinds of storages are used. In the case of local operators, which need 2-dimension window or 1-dimension vertical window, a line buffer and a window buffer are used to store the previous lines before producing the amount of pixels needed by the next processing step. For operations that require 1-dimensional horizontal window size, only a row buffer, which is implemented as one FIFO buffer, is needed.
Chapter 5

Results

This chapter presents the evaluation results of the proposed extensions. Firstly, the experimental setup is described. In the rest of this chapter, the evaluations for both the separation of data type specification and on-chip boundary handling are presented.

5.1 Evaluation Methodology

The evaluation process sets a Xilinx Zynq XC7Z020CLG484-1 board as the target platform. This platform is a low mid-range System on-chip (SoC). The information of the resources available in this platform can be seen in Table 5.1. For the evaluation, the target frequency is set to 100 MHz. The tool to synthesize the generated HLS code from Halide and to perform RTL co-simulation is Xilinx Vivado Design Suite 2016.4.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>280</td>
</tr>
<tr>
<td>DSP</td>
<td>220</td>
</tr>
<tr>
<td>FF</td>
<td>106400</td>
</tr>
<tr>
<td>LUT</td>
<td>53200</td>
</tr>
</tbody>
</table>

Table 5.1: Resources Available in the target FPGA.

Image Processing Algorithms

Four applications are used for the evaluation of the proposed extensions.

1. Sobel Filter.
   Sobel filter is an image processing used to detect edges in an image. This algorithm employs two 3x3 convolution kernels to calculate the approximation of gradient value in both horizontal and vertical direction at each pixel. In addition, an additional processing stage is added to convert an Red-Green-Blue (RGB) pixel into a grayscale pixel before performing the convolution. Hence, this application represents the combination of a point operator and two local operators.

2. Gaussian Convolution Filter.
   This application consists of a convolution operation between the filter coefficients and the input image. The local operator employed in this application involves a moderate number of
arithmetic operations. As a result, the accuracy of the computation is affected by the data types. For the evaluation, the effect of using different configurations of fixed-point data type can be observed.

3. Unsharp Filter.
The unsharp filter is a simple sharpening operator that enhances the high-frequency components, such as edges, in an image. The operation consists of a subtraction operation between the original and the blurred image resulting in an un-sharpened mask. In the next step, the mask is combined with the original image. This application represents the data dependencies between a local operator (i.e., Gaussian filter) and a point operator (i.e, subtraction and addition).

Among these image processing algorithms, bilateral grid application is the most complex one. It first constructs a 3D grid from the input image, then performs three 5x5 convolutions. In the next stage, several trilinear interpolation operations will be performed on the output from the last convolution stage. This application is chosen because it involves many floating-point arithmetic operations in the interpolation operations and it has a long processing pipeline.

In order to evaluate the proposed extensions, the following metrics are used for the evaluation:

1. Resource Utilization.
The resource utilization is evaluated based on the use of BRAM, DSP48, LUT, and FF.

2. Performance.
Parameters used for evaluation the performance are latency and Initiation Interval (II). The latency represents the number of clock cycles required to compute all output pixels. II represents the number of clock cycles before the function can accept new input data.

In this thesis, we are only considering the estimated results obtained from the Xilinx Vivado HLS tool. The results are considered to be representative enough for this evaluation. Moreover, further optimizations are performed by the tool during the later process (e.g. Post synthesis and Place and Route (PAR) process). Hence, better performance and resource utilization can be obtained.

The verification also includes the accuracy evaluation of the fixed-point-based implementations. The measurement of the accuracy is performed with the output image from MATLAB as the golden reference. The MATLAB model uses double-precision floating-point arithmetic operations and generates a 16-bit lossless PNG image. The use of 16-bit images allows higher dynamic range and lower error tolerance (i.e. the maximum error tolerance is $1.5 \times 10^{-5}$). The objective of the evaluation is to obtain an implementation with the accuracy errors as low as possible. As discussed in Section 2.4, the following metrics are used:

1. Number of error pixels.
2. Minimum absolute error in floating-point.
3. Maximum absolute error in floating-point.
4. MSE.

This process can be performed using any C/C++ compiler with the fixed-point library from Xilinx. The simulation can be done bit-accurately using the fixed-point library provided by Xilinx. To perform the comparison, the computation values from the fixed-point implementation are converted back into floating-point and then stored into a 16-bit output image. After that, the pixels value can be compared with the golden reference as illustrated in Figure 2.8.
5.2 Evaluation Results

5.2.1 Design Space Exploration

With the separation between the data type specification and the algorithm description, it is possible to perform design space exploration of the FPGA implementations of Halide. To evaluate the advantages of the separation between the algorithm description and the data types, design space exploration in terms of resource utilization, performance, and accuracy of results were performed.

For the exploration, the default implementation generated from Halide-HLS using an integer data type was compared with the floating-point-based implementation and several configurations for fixed-point implementations. Below we elaborate the design process exploration process on the Gaussian Filter and Unsharp Filter. For the sobel filter and the bilateral grid, the results are summarized in Section 5.3.

Application 1: Gaussian Filter

Gaussian Filter is implemented as a one-step image processing pipeline. The algorithm description and the scheduling implementation of the Gaussian Filter can be seen in Listing A.2. The experiment setup is as follows:

- Image: 512x512 PNG Image
- Window Size: 5x5

Listing 5.1 shows the data types scheduling for this application. There are only three variables that can be configured for the exploration. Several combinations of T can be seen in Table 5.2.

```c
void type_schedule() {
  if (get_target().has_hls_feature()) {
    // Only apply to (Vivado) HLS
    clamped.cast_to(T1);
    kernel.cast_to(T2);
    conv1.cast_to(T3);
  }
}
```

Listing 5.1: Data types specification for Gaussian Filter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Fixed-Point A</th>
<th>Fixed-Point B</th>
<th>Fixed-Point C</th>
<th>Fixed-Point D</th>
<th>Fixed-Point E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clamped (T1)</td>
<td>UFixed &lt;25,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
<td>UFixed &lt;10,1&gt;</td>
<td>UFixed &lt;16,1&gt;</td>
<td>UFixed &lt;16,1&gt;</td>
</tr>
<tr>
<td>kernel (T2)</td>
<td>UFixed &lt;25,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
<td>UFixed &lt;10,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
</tr>
<tr>
<td>conv1 (T3)</td>
<td>UFixed &lt;50,1&gt;</td>
<td>UFixed &lt;40,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
<td>UFixed &lt;36,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
</tr>
</tbody>
</table>

Table 5.2: Fixed-Point Configurations for Listing 5.1.

Using Vivado HLS, the estimated resource utilization and performance can be seen in Table 5.3. The BRAM utilization depends on the width of the image and the window size. After the bit-width of the input pixel (clamped) is reduced from 32-bit to less than 16-bit, the BRAM utilization
CHAPTER 5. RESULTS

is reduced to 50%. A large reduction in the utilization of DSP, FF, and LUT can be obtained when converting the data types from floating-point to fixed-point or integer data types. Note that the DSP utilization in the 8-bit implementation is slightly higher than in the Fixed-Point C implementation. This result solely depends on the Xilinx Vivado HLS tool to decide whether to implement the multiplication operations using DSP blocks or other approaches (i.e., LUTs).

<table>
<thead>
<tr>
<th>Application</th>
<th>Resource Utilization</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BRAM</td>
<td>DSP</td>
</tr>
<tr>
<td>Floating-Point</td>
<td>8</td>
<td>2.86%</td>
</tr>
<tr>
<td>16-bit Integer</td>
<td>4</td>
<td>1.43%</td>
</tr>
<tr>
<td>8-bit Integer</td>
<td>4</td>
<td>1.43%</td>
</tr>
<tr>
<td>Fixed-Point A</td>
<td>8</td>
<td>2.86%</td>
</tr>
<tr>
<td>Fixed-Point B</td>
<td>8</td>
<td>2.86%</td>
</tr>
<tr>
<td>Fixed-Point C</td>
<td>4</td>
<td>1.43%</td>
</tr>
<tr>
<td>Fixed-Point D</td>
<td>4</td>
<td>1.43%</td>
</tr>
<tr>
<td>Fixed-Point E</td>
<td>4</td>
<td>1.43%</td>
</tr>
</tbody>
</table>

Table 5.3: Estimated Resource Utilization (Used and % of total) and Performance (Latency in clock and Initiation Interval) of several Gaussian Filter Implementations.

The accuracy of results can be obtained by comparing the output images (in 16-bit PNG files) with the golden reference output image from MATLAB. The results can be seen in Table 5.4. The benefits of using fixed-point data types instead of floating-point data types can be seen by comparing the results of resource utilization in Table 5.3 with the accuracy in Table 5.4. Assuming that the required MSE is 0, we can obtain an acceptable accuracy of the results with a much lower resource utilization of DSP, FF, and LUT by 83%, 98%, and 92% respectively. Depending on the minimum accuracy requirement, a further resource usage reduction can be obtained through the use of smaller bit-widths, e.g. using the configuration Fixed-Point C.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Number of error pixels</th>
<th>Min Absolute Error</th>
<th>Max Absolute Error</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit Integer</td>
<td>262144 (100%)</td>
<td>0.002426</td>
<td>0.029068</td>
<td>0.000259</td>
</tr>
<tr>
<td>8-bit Integer</td>
<td>262144 (100%)</td>
<td>0.005203</td>
<td>0.057282</td>
<td>0.001042</td>
</tr>
<tr>
<td>Fixed-Point A</td>
<td>136824 (52%)</td>
<td>0.000015</td>
<td>0.000015</td>
<td>0.000000</td>
</tr>
<tr>
<td>Fixed-Point B</td>
<td>250914 (95%)</td>
<td>0.000015</td>
<td>0.000031</td>
<td>0.000000</td>
</tr>
<tr>
<td>Fixed-Point C</td>
<td>262144 (100%)</td>
<td>0.002136</td>
<td>0.024949</td>
<td>0.000197</td>
</tr>
<tr>
<td>Fixed-Point D</td>
<td>262143 (99%)</td>
<td>0.000015</td>
<td>0.000061</td>
<td>0.000000</td>
</tr>
<tr>
<td>Fixed-Point E</td>
<td>256647 (100%)</td>
<td>0.000031</td>
<td>0.000092</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

Table 5.4: Accuracy Comparisons between several Gaussian Filter Implementations.
CHAPTER 5. RESULTS

(a) Visible Artifacts
(b) Without Artifact

Figure 5.1: Comparison of output images in the implementation of Unsharp Mask without and with the scaling respectively.

Application 2: Unsharp Mask

In this project, the unsharp mask is implemented into a 3-stage image processing pipeline; the first stage is a blurring step in the horizontal direction followed by a blurring in the vertical direction and the last stage is a scaling step to avoid a broken output image. Figure 5.1 shows the comparison of the output images of the integer-based implementation without and with the scaling step respectively.

The algorithm description and the scheduling can be seen in A.3. The scaling step is not needed anymore for the fixed-point based implementation. Conversion from floating-point to fixed-point can be done by specifying the specification of data types in type scheduling() as shown in Listing 5.2. The experiment setup for the design exploration of the Unsharp Filter is as follows:

- Image: 512x512 PNG Image
- Window Size: 9x9

There are six variables in which the fixed-point bit-width can be configured. Table 5.5 shows four explored bit-width combinations in this experiment. The results in term of resource utilization and performance can be seen in Table 5.6. The choice of the bit-widths of the variables used in the unsharp mask affects the number of used BRAMs. On the other hand, the other hardware resources depends on not only the bit-width values, but also the data type. The floating-point implementation consumes a significant number of DSP and LUT resources.

The accuracy comparisons can be seen in Table 5.7. In the Fixed-Point C configuration, the bit-widths used for the clamped and kernel variables are 8-bit, while the bit-widths for the...
CHAPTER 5. RESULTS

```c
void type_schedule() {
    if (get_target().has_hls_feature()) {
        // Only apply to Xilinx (Vivado) HLS
        kernel.cast_to(T1);
        clamped.cast_to(T2);
        blur_x.cast_to(T3);
        blur_y.cast_to(T4);
        sharpen.cast_to(T5);
        hw_output.cast_to(T6);
    }
}
```

Listing 5.2: Data types specification for Unsharp Mask

other variables are 10 bits. Despite the small bit-widths, it can be seen that the accuracy of the
Fixed-Point C implementation results in more accurate results in terms of MSE and maximum absolute value compared to the accuracy of the 8-bit integer implementation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Fixed-Point A</th>
<th>Fixed-Point B</th>
<th>Fixed-Point C</th>
<th>Fixed-Point D</th>
</tr>
</thead>
<tbody>
<tr>
<td>clamped (T1)</td>
<td>UFixed &lt;20,1&gt;</td>
<td>UFixed &lt;16,1&gt;</td>
<td>UFixed &lt;8,1&gt;</td>
<td>UFixed &lt;14,1&gt;</td>
</tr>
<tr>
<td>kernel (T2)</td>
<td>UFixed &lt;25,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
<td>UFixed &lt;8,1&gt;</td>
<td>UFixed &lt;14,1&gt;</td>
</tr>
<tr>
<td>blur_x (T3)</td>
<td>UFixed &lt;25,1&gt;</td>
<td>UFixed &lt;21,1&gt;</td>
<td>UFixed &lt;10,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
</tr>
<tr>
<td>blur_y (T4)</td>
<td>UFixed &lt;25,1&gt;</td>
<td>UFixed &lt;21,1&gt;</td>
<td>UFixed &lt;10,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
</tr>
<tr>
<td>sharpen (T5)</td>
<td>UFixed &lt;25,1&gt;</td>
<td>UFixed &lt;21,1&gt;</td>
<td>UFixed &lt;10,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
</tr>
<tr>
<td>hw_output (T6)</td>
<td>UFixed &lt;26,1&gt;</td>
<td>UFixed &lt;22,1&gt;</td>
<td>UFixed &lt;10,1&gt;</td>
<td>UFixed &lt;20,1&gt;</td>
</tr>
</tbody>
</table>

Table 5.5: Fixed-Point Configurations for Listing 5.2.

<table>
<thead>
<tr>
<th>Application</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-Point</td>
<td>33</td>
<td>11.79%</td>
<td>94</td>
<td>42.73%</td>
<td>10283</td>
</tr>
<tr>
<td>16-bit Integer</td>
<td>13</td>
<td>4.64%</td>
<td>64</td>
<td>29.09%</td>
<td>3534</td>
</tr>
<tr>
<td>8-bit Integer</td>
<td>13</td>
<td>4.64%</td>
<td>4</td>
<td>1.43%</td>
<td>2130</td>
</tr>
<tr>
<td>Fixed-Point A</td>
<td>27</td>
<td>9.64%</td>
<td>32</td>
<td>14.54%</td>
<td>2583</td>
</tr>
<tr>
<td>Fixed-Point B</td>
<td>17</td>
<td>6.07%</td>
<td>18</td>
<td>6.43%</td>
<td>1737</td>
</tr>
<tr>
<td>Fixed-Point C</td>
<td>13</td>
<td>4.64%</td>
<td>4</td>
<td>1.43%</td>
<td>1015</td>
</tr>
<tr>
<td>Fixed-Point D</td>
<td>16</td>
<td>5.71%</td>
<td>14</td>
<td>6.36%</td>
<td>1570</td>
</tr>
</tbody>
</table>

Table 5.6: Estimated Resource Utilization (Used and % of total) and performance (Latency in clock) of several Unsharp Mask Implementations.

The estimated resource utilization in Table 5.6 clearly shows the benefits of utilizing fixed-point data types instead of floating-point types. Compared to the accuracy result in Table 5.7, the resource utilization can be reduced significantly, especially the utilization of DSPs and LUTs while achieving a MSE of 0.

42
CHAPTER 5. RESULTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Number of error pixels</th>
<th>Min Absolute Error</th>
<th>Max Absolute Error</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit Integer</td>
<td>256309 (97%)</td>
<td>0.000015</td>
<td>0.87451</td>
<td>0.003309</td>
</tr>
<tr>
<td>16-bit Integer</td>
<td>237884 (90%)</td>
<td>0.000015</td>
<td>0.87451</td>
<td>0.000642</td>
</tr>
<tr>
<td>Fixed-Point A</td>
<td>2297 (10%)</td>
<td>0.000015</td>
<td>0.000015</td>
<td>0.000000</td>
</tr>
<tr>
<td>Fixed-Point B</td>
<td>92883 (45%)</td>
<td>0.000015</td>
<td>0.000048</td>
<td>0.000000</td>
</tr>
<tr>
<td>Fixed-Point C</td>
<td>261983 (99%)</td>
<td>0.000015</td>
<td>0.069154</td>
<td>0.001284</td>
</tr>
<tr>
<td>Fixed-Point D</td>
<td>274668 (100%)</td>
<td>0.000015</td>
<td>0.001129</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

Table 5.7: Accuracy Comparisons between several Unsharp Filter Implementations.

Note that the latency is not affected by the data types. This occurs because the latency of a line-buffer stage is larger than the one of the blur stages. With a dataflow implementation, the latency of the arithmetic operations is hidden.

5.3 Results Summary

The same design space exploration process is performed for implementing the Sobel Filter and Bilateral Grid application. Table 5.8 and Table 5.9 show the final implementation results of all evaluated applications in term of resource utilization and accuracy respectively. The complete implementation source code can be seen in Appendix A.

Overall the results in Table 5.8 demonstrates that the resource utilization on FPGAs can be reduced through the use of floating-point data types. Moreover, the fixed-point implementations can achieve better accuracy compared to the integer-based implementations. In the bilateral grid implementation, the resource utilization of DSP and LUT exceed the total resources available in the platform. This is expected since the image size used for the evaluation is a 3K image. Moreover, the LUT utilization in the integer-based implementation is already close to 100%. However, the same pattern in the resource utilization can be observed in the fixed-point implementation.

An additional anomaly in the bilateral grid application can also be observed in the latency of the floating-point implementation. First, the significant increase in the latency occurs because the Xilinx Vivado HLS tool stops the optimization process during synthesis as the all of the DSP resources have been consumed. As a result, the initiation interval of 1 cannot be achieved. In addition, a pipelined bilateral grid implementation cannot be produced because of the linebuffer in the histogram stage (i.e., see line 112 in Listing A.4).
### CHAPTER 5. RESULTS

#### Application Resource Utilization

<table>
<thead>
<tr>
<th>Application</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
<th>Latency [cycle]</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integer</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Floating-Point</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fixed-Point</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table 5.8: Resource Utilization Summary.

#### Application Accuracy Summary

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of error pixels</th>
<th>Min Absolute Error</th>
<th>Max Absolute Error</th>
<th>Mean Square Error (MSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integer</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Floating-Point</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fixed-Point</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table 5.9: Accuracy Summary.
5.4 Comparison to OpenCV

To evaluate the performance and resource utilization of the generated kernels from Halide, they will be compared with off-the-shelf implementations from Vivado OpenCV Library. This library has provided an implementation for Sobel filter and Gaussian filter. Since Unsharp mask and Bilateral grid implementation are not available yet in the OpenCV library, they will not be evaluated. The HLS code generated from Halide is implemented using fixed-point data type while the OpenCV implementation uses fixed precision data types (i.e. unsigned 8-bit integer). Note that the OpenCV implementations of the sobel filter and the Gaussian filter are specifically re-developed for FPGA platforms.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Target Frequency</th>
<th>Application</th>
<th>Resource Utilization</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>BRAM</td>
<td>DSP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Val</td>
<td>%</td>
</tr>
<tr>
<td>1920 x 1080</td>
<td>100 MHz</td>
<td>Sobel Filter Halide</td>
<td>4</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sobel Filter OpenCV</td>
<td>3</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gaussian Filter Halide</td>
<td>8</td>
<td>2.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gaussian Filter OpenCV</td>
<td>10</td>
<td>3.57</td>
</tr>
</tbody>
</table>

Table 5.10: Comparisons to OpenCV.

As can be seen in Table 5.10, the resource utilization of the generated designs in most cases is better than the result generated by Xilinx OpenCV library. Sobel Filter Halide uses more BRAM because it utilizes fixed-point data types while OpenCV uses 8-bit integer data types. In the second application, Halide-generated design consumes less BRAM because the OpenCV library version does not use an optimal size for the line buffer. For instance, only four lines of buffers are needed in a 5x5 convolution filter. To conclude, the Halide-generated designs are competitive in term of resource utilitzation and performance compared to the optimized library designs.

5.5 Overhead of On-Chip Boundary Handling

To evaluate and verify the implementation of the on-chip boundary handling, several types of boundary conditions (e.g. input sizes and stencil size) are evaluated. The following metrics are used for the evaluation:

- Estimated Resources Utilization (BRAM, DSP, FF, and LUT).
- Estimated Latency and Initiation Interval

Table 5.11 shows the estimated overhead resource utilization of the extra boundary handling step. As can be seen, the overhead consumes no more than 1% of the total resources available in the target FPGA.
CHAPTER 5. RESULTS

<table>
<thead>
<tr>
<th>Boundary Condition</th>
<th>Stencil Dimension &lt;X,Y,C&gt;</th>
<th>Resource Utilization</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat</td>
<td>&lt;1,1,1&gt;</td>
<td>Val %</td>
<td>Val %</td>
</tr>
<tr>
<td>Repeat</td>
<td>&lt;1,1,1&gt;</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Repeat</td>
<td>&lt;1,1,1&gt;</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Repeat</td>
<td>&lt;1,1,1&gt;</td>
<td>2</td>
<td>0.71</td>
</tr>
<tr>
<td>Repeat</td>
<td>&lt;1,1,1&gt;</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.11: Resource utilization and latency results of several boundary conditions handling on FPGAs with window size 10x10.

Figure 5.2 shows the increase of resource utilization in respect to the image size and the image offsets in all four directions.

The numbers of BRAMs, FFs, and LUTs are expected to rise as the input image and the window size increase. In all cases, there is no DSP usage.

The resource utilization with 1D boundary conditions (e.g. <3,1,1>and <5,1,1>) in respect to several input images can be seen in Figure 5.3. The increase trend of the resource usage is similar to the first boundary condition in Figure 5.2.

The overhead of 2D boundary condition is the largest among other boundary conditions. In this case as can be seen in Figure 5.4, the number of BRAMs, LUTs, and FFs increase proportionally to both the size of the input image and the size of the window.

In all cases, the introduction of on-chip boundary handling does not affect the throughput of the overall hardware implementation since the II of 1 can be achieved. The overhead of the resource utilization is negligible as it occupies close to 0% of the total resources.
Figure 5.3: Boundary Handling with Stencil size $<3,1,1>$ and $<5,1,1>$.

Figure 5.4: Boundary Handling with Stencil Size $<3,3>$ and $<5,5>$. 
Chapter 6

Discussion

In this work, several extensions have been proposed and implemented: the introduction of arbitrary precision data types into Halide, the separation between the algorithm description and the data type specification, and on-chip boundary handling. The evaluation results of the proposed extensions have also been reported in Chapter 5. This chapter presents the discussion of the results and revisits the research questions introduced in Chapter 1.

6.1 Results Discussion

It is evident from the results that the use of arbitrary precision fixed-point data types is beneficial for an FPGA platform in term of resource utilization and performance. In all applications, the resource utilization decreases significantly as the data types are changed from floating-point to fixed-point. In an application as complex as the bilateral grid, the latency of the fixed-point implementation is also reduced compared to the floating-point implementation. Although this design choice leads to loss of accuracy, it is still possible to achieve an acceptable level of accuracy compared to the floating-point operations. However, a much better accurate result can be obtained compared to the original implementation using fixed bit-width integer. On the whole, this extension can improve the code generation with Halide-HLS for FPGA platforms in term of resource utilization in general.

The separation between the algorithm description and the data type specification in addition to the scheduling in Halide improves both functional and performance portability of the framework. The extension enables the use of different data types for each kind of platform. In this case, the fixed-point-based implementation will be generated only for FPGAs while the floating-point data types are used for CPUs or GPUs. All of this can be achieved without having to change the algorithm description. From software engineers perspective, this obviously reduces the development gap between the software development paradigm and hardware development paradigm.

Moreover, this extension eases the design solution exploration targeting FPGAs using different fixed-point bit-widths. The experimental results in Chapter 5 show that the computation accuracy vs the resource utilization in an FPGA platform can be explored easily just by changing the data type specification. Compared to the implementation using C-HLS or RTL language in which the bit-width of every variable has to be manually configured, this approach needs the bit-width information of much smaller numbers of variables. The intermediate variables are generated automatically by the framework.

In addition, an on-chip boundary conditions handling was also proposed. The results show that the implementation of the boundary handling introduces some overhead for the resource utilization and the latency. However, the overhead can be considered to be negligible compared to the overall
CHAPTER 6. DISCUSSION

resource utilization and latency. As for the throughput, the required initiation interval of 1 can still be achieved with a careful implementation of the boundary handling block.

The results have demonstrated the benefits of these proposed extensions on improving the portability and programmability of Halide. By supporting arbitrary precision data types, the development gap to FPGA platforms can be further reduced. The on-chip boundary handling improves the applicability of the generated FPGA code in a streaming-based processing environment, i.e. FPGAs directly receive input pixels from sensors through on-chip memory without the interventions of CPUs.

This preliminary implementation of the proposed extensions shows promising results; however, several improvements can still be done. The implication of using a high-level of abstraction is the loss of fine-grain control over the implementation. In the current implementation, the data types for the intermediate variables are determined based on the types of input and output nodes and the type promotion rule in such a way that the accuracy can be maintained. This decision, however, may lead to unnecessary bit-width growth increasing the resource utilization of the FPGA. To control the bit-width growth, several fixed-point arithmetic modes, such as overflow mode and saturation mode, should be taken into account.

In addition, the current bit-widths of fixed-point data types are specified manually by the programmers. Determining the optimal bit-widths is not a trivial task and require a lot of experiments. An automatic bit-width infer will be beneficial to improve the programmability and ease-of-use of this DSL.

Future work should also include the automatic generation of the on-chip boundary handling kernel. Since the kernel is implemented as a template C++ library, it can be easily generated by the framework. This improvement should enhance the applicability of Halide to generate FPGA implementations.

6.2 Research Questions Revisited

In this section, we re-visit the research questions that were introduced in Chapter 1. The main research question of this thesis is as follows.

**Are there any means to achieve cross-platform functional and performance portability targeting CPUs, GPUs, and FPGAs in the domain of medical image processing?**

The following sub-research questions are formulated to answer the main research question.

1. **What are the current tools or methods available to achieve cross-platform portability?**
   
   There are several potential portable computing solutions which were identified during the literature study. The solutions consisting of OpenCL, pocl, high-level languages, and domain-specific languages, have been discussed in Chapter 2. In Chapter 3, the evaluation of domain-specific languages in the domain of image processing was described.

2. **What is the most suitable solution (e.g. tool or technique) for the implementation of portable image processing algorithms in the context of Philips Healthcare?**
   
   As described in Chapter 2, the low-level approach using OpenCL can offer functional portability but does not offer performance portability. A portable implementation of OpenCL called pocl aims to improve the performance portability of OpenCL. Relying on the code transformation in the IR form, there is only limited number of possible optimizations considering an FPGA platform generally has a different code structure, i.e. deep-pipeline vs parallel-style structure.
In this study, we focus on the domain-specific language approach to achieve cross-platform portability. Several DSLs in the domain of image processing have been evaluated in Chapter 3. From the evaluation, Halide seems to be the most promising solution to achieve cross-platform portability. Unlike the other DSLs, such as PolyMage, DarkRoom, and HIPAcc, Halide offers a balanced level of abstraction. In Halide, the algorithm description is separated from the platform-dependent scheduling part which allows the application developers to be able to specify platform-specific optimizations. The changes in the scheduling part do not break the functional correctness of the algorithm. In addition, this allows the developers to easily explore several design solutions to obtain high-performance implementation using high-level intuitive primitives. As a result, the development, testing and maintenance efforts and costs can be reduced.

3. Are there any limitations in the solution and how to address the limitations to improve the solution?

Two limitations that are related to generating FPGA implementations were identified during the study. The first one is that Halide only supports C-based data types which are on 8-bit boundaries. With this limited support of data types, the implementation on FPGAs leads to either inefficient resource utilization or low accuracy. Also, the portability of code offered in Halide is broken due to the need to convert from floating-point to fixed-point implementations.

In order to address this limitation, arbitrary precision data types and separation between data type specification and algorithm description were proposed. The benefits of these extensions on generating image processing kernels on FPGAs has been shown in the previous chapter: the improvement of the accuracy result with the tradeoff of resources utilization, the improvement of the functional and performance portability of Halide targeting FPGA platforms.

Another limitation is that the boundary handling which is common in image processing algorithms is not performed on FPGAs. In the use case of Philips Healthcare, streaming-based architecture in which the processing kernel receives streaming input data directly from the sources is adopted. In order to reduce the latency, the boundary handling should be performed directly as the input data is read by the processing kernel in the FPGA. To address this limitation, an on-chip boundary handling is proposed as an additional stage in the FPGA.

As the sub-research questions have been answered, the main research question can be addressed. This research demonstrates the suitability of Halide to achieve cross-platform portability and ease-of-development for implementing processing algorithms. Several extensions also have been proposed and implemented to improve the applicability of using Halide to generate code for FPGA platforms.

For the use case from Philips, the ability to have a common implementation of image processing algorithms and support multiple platforms at the same time can greatly reduce the development costs and address the lifecycle management challenge in general. In the upcoming years, heterogeneous-based systems are going to be common and become a potential target platform for implementing more advanced image processing algorithms. This obviously will increase the development and maintenance costs. By using higher-level of abstraction programming model offered by Halide, the challenge can be addressed.

Despite the advantages of Halide, there are also several limitations. Clearly, by targeting a higher level of abstraction, the flexibility to perform platform-specific optimization strategies is reduced and even hidden from the application developers. The costs for the lifecycle management will be moved to the development and maintenance of Halide framework instead. However, since this is a domain-specific solution, the long-term benefit still exceed the development and maintenance costs of the framework, such as reducing the development time and costs, improving testability, maintainability, and flexibility.
Chapter 7

Conclusions

The plethora of computing platforms, i.e., multi-core CPU, GPU, and FPGA, poses challenges on the lifecycle management (LCM) of a medical system. Since the high-performance requirement of the medical image processing algorithms requires highly optimized implementations, the implementations would be heavily coupled to the architecture of a specific platform. As a result, porting the same implementations towards other target platforms requires a long and costly re-development and re-verification process. Due to the differences in the architecture, programming model, and programming tool, the re-development of the same algorithm targeting FPGAs is more challenging. A portable computing solution has been proposed as a solution towards addressing this increasing LCM challenge. In this study, we focus on achieving portability targeting FPGA platforms while maintaining portability to CPUs and GPUs.

Among several portable computing solutions which were identified in this study, a domain-specific language approach was proposed as a suitable solution to achieve portability. From the evaluation of several DSLs in the image processing domain, it was concluded that Halide has the potential to address the main challenge. This framework offers an effective level of abstraction through the separation between the algorithm description and the scheduling. Hence the performance of the implementation targeting a specific platform can be tuned in the scheduling part while the functional correctness can still be guaranteed. In addition, currently Halide has an active community support; the framework is still improved and extended to support new target platforms and new optimization techniques. Recently, Halide has been extended to support FPGA platforms [29]. We showed that the FPGA implementations generated by Halide-HLS are comparable to the Xilinx HLS OpenCV library.

In this work, we also identified several limitations in the current Halide-HLS framework which is targeting FPGA platforms: data types are limited to C++ based data types which are on 8-bit boundaries, less portable FPGA implementations due to the data types, and off-chip boundary handling. To address the first two limitations, this thesis proposed an arbitrary precision data types extension and the separation between the algorithm and the data types. The experimental results in this study showed the advantages of using the arbitrary precision data types in reducing the resource utilization in the FPGA platform. In addition, the separation between the algorithm description and the data type specification could improve the portability of the framework towards FPGAs. Lastly, the third extension implemented an on-chip boundary handing. The results showed that the overhead of the implementation is negligible.

In conclusion, this study proposed a domain-specific language-based approach to overcome the programmability challenges of FPGA platforms and to achieve portability towards multi-core CPUs and GPUs. The effective level of abstraction offered by Halide enables the engineers to work from a higher level abstraction which improves the design productivity and still allows controls on performing platform-dependent optimizations. Despite the limitations in the current work and
CHAPTER 7. CONCLUSIONS

the current version of Halide, the structure of the framework is extendable in the sense that as the tool matures, the performance of the generated code can be better.

This study provided the roadmap towards achieving a portable implementation of image processing algorithms. From the result of this study, there are several recommendations for further improvement of the proposed framework:

**FPGA-based streaming input and output in multi-rate image processing hardware.** Currently, Halide-HLS works well with image processing pipelines with a data rate of one pixel per cycle. Both the input and the output can be consumed and produced in a streaming-manner. It is desirable to send the output pixels directly to IO (e.g., interface or display hardware). As a result, for a multi-rate processing application, such as 2D upsampling, the output pixels have to be reordered before being streaming out. An on-chip reordering kernel has to be implemented to produce the output pixels in a streaming manner. This work would require an effective use of on-chip memories to store the output pixels of the next lines until the whole pixels in the first line can be streamed out. In addition, it is important to maintain the performance of the implementation in terms of latency and initiation interval.

**Support for multi-resolution image processing algorithm.** Pyramid implementations are commonly used in advanced image processing algorithms. During the research, it turns out that the framework has not supported the implementation of multi-resolution pipelines, e.g., Gaussian pyramid and Laplacian pyramid, yet. In this kind of application, the task dependencies are more extensive compare to a single-resolution image processing algorithm. Hence, a proper configuration of memory buffers in each resolution level is needed to prevent deadlocks. The result of this implementation will improve the applicability of Halide.

**Automatic arbitrary bit-widths exploration for hardware generation on FPGAs.** Currently, there have been some studies on automatically producing the right schedule for the scheduling parts. Since the bit-widths are important factors in implementing hardware on FPGAs, the auto-tuning method should take this factor into account. An automatic bit-widths inference based on the precision requirements is an interesting topic for further exploration.
Bibliography


Appendix A

Image Processing Applications in Halide

Listing A.1. Sobel Filter

```cpp
#include "Halide.h"
using namespace Halide;

namespace {

class SobelConv : public Halide::Generator<SobelConv> {

public:
    Input<Buffer<float>> input {"input", 3};
    Output<Buffer<float>> output {"output", 2};

    RDom win;

    // Algorithm Description
    void generate() {
        win = RDom(-1, 3, -1, 3);
        // RGB Conversion
        R_const = 0.299f;
        G_const = 0.587f;
        B_const = 0.114f;
        zero_f = 0.0f;
        one_f = 1.0f;
        gray(x, y) = ((R_const * clamped(x, y, 0)) + (G_const * clamped(x, y, 1))
                      + (B_const * clamped(x, y, 2)));
        conv_x(x, y) = ((-1 * gray(x-1, y-1)) + (1 * gray(x+1, y-1)) +
                        (-2 * gray(x-1, y)) + (2 * gray(x+1, y)) +
                        (-1 * gray(x-1, y+1)) + (1 * gray(x+1, y+1)));
        conv_y(x, y) = ((-1 * gray(x-1, y-1)) + (-2 * gray(x, y-1)) +
                        (-1 * gray(x+1, y-1)) + (1 * gray(x-1, y+1)) +
                        (2 * gray(x, y+1)) + (1 * gray(x+1, y+1)));
```
APPENDIX A. IMAGE PROCESSING APPLICATIONS IN HALIDE

val = abs(conv_x(x, y)) + abs(conv_y(x, y));

hw_output(x, y) = clamp(val, zero_f, one_f);
output(x, y) = cast<float>(hw_output(x, y));

void schedule() {
    input.dim(2).set_bounds(0, 3);
    output.dim(0).set_stride(1);

    if (get_target().has_hls_feature()) {
        std::cout << "compiling HLS code..." << std::endl;
        clamped.compute_root(); // prepare the input for the whole image
        // HLS schedule: make a hw pipeline producing 'hw_output', taking
        // inputs of 'clamped', buffering intermediates at (output, xo) loop
        // level
        hw_output.compute_root();
        hw_output.tile(x, y, xo, yo, xi, yi, 1920, 1080).reorder(xi, yi, xo, yo);
        hw_output.accelerate({clamped}, xi, xo); // define the inputs and
        // the output
        conv_x.linebuffer();
        conv_x.unroll(x).unroll(y);
        conv_y.linebuffer();
        conv_y.unroll(x).unroll(y);
        // Linebuffering gray
        gray.linebuffer().compute_at(hw_output, xi);
    } else {
        std::cout << "compiling CPU code..." << std::endl;
        gray.compute_root();
        output.tile(x, y, xo, yo, xi, yi, 256, 256);
        output.fuse(xo, yo, xo).parallel(xo);
        output.vectorize(xi, 8);
        conv_x.compute_at(output, xo).vectorize(x, 8);
        conv_y.compute_at(output, xo).vectorize(x, 8);
    }
}

Expr R_const, G_const, B_const, zero_f, one_f;
Func gray("gray");
Func clamped("clamped"), conv_x("conv_x"), conv_y("conv_y"), hw_output("hw_output");
Var x("x"), y("y"), c("c");
Var xo("xo"), xi("xi"), yi("yi"), yo("yo");
Expr val("val");
std::vector<Argument> args;
HALIDE_REGISTER_GENERATOR(SobelConv, "sobel_conv");
} // namespace
Listing A.1: Sobel Filter

```cpp
#include "Halide.h"

using namespace Halide;

namespace {

class GaussianConv : public Halide::Generator<GaussianConv> {

public:
    Input<Buffer<float>> input {"input", 2};
    Input<Buffer<uint8_t>> weight {"weight", 2};
    Input<float> bias {"bias"};
    Output<Buffer<float>> output {"output", 2};

RDom win;

// Algorithm Description
void generate() {
    win = RDom(-2, 5, -2, 5);

    // Define a 9x9 Gaussian Blur with a repeat-edge boundary condition.
    float sigma = 1.5f;
    kernel_f(x, y) = (exp(-(x*x + y*y)/(2*sigma*sigma)) / 
                     (float)(2*M_PI*sigma*sigma));
    kernel(x, y) = kernel_f(x, y) / 
                  (kernel_f(0, 0) + kernel_f(1, 0) * 4 + 
                   kernel_f(2, 0) * 4 + kernel_f(1, 1) * 4 + 
                   kernel_f(1, 2) * 4 + kernel_f(2, 1) * 4 + 
                   kernel_f(2, 2) * 4);

    // define the algorithm
    clamped(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
    conv1(x, y) += clamped(x+win.x, y+win.y) * kernel(win.x, win.y);

    hw_output(x, y) = conv1(x, y);
    output(x, y) = cast<float>(hw_output(x, y));
}

void schedule() {
    if (get_target().has_hls_feature()) {
        std::cout << "\ncompiling HLS code..." << std::endl;
        clamped.compute_root(); // prepare the input for the whole image

        // HLS schedule: make a hw pipeline producing 'hw_output', taking
        // inputs of 'clamped', buffering intermediates at (output, xo) loop
        // level
        hw_output.compute_root();
        hw_output.tile(x, y, xo, yo, xi, yi, 512, 512).reorder(xi, yi, xo, yo);
    }
}
};
}
```

Listing A.2. Gaussian Filter

```cpp
#include "Halide.h"

using namespace Halide;

namespace {

class GaussianConv: public Halide::Generator<GaussianConv> {

public:
    Input<Buffer<float>> input {"input", 2};
    Input<Buffer<uint8_t>> weight {"weight", 2};
    Input<float> bias {"bias"};
    Output<Buffer<float>> output {"output", 2};

RDom win;

// Algorithm Description
void generate() {
    win = RDom(-2, 5, -2, 5);

    // Define a 9x9 Gaussian Blur with a repeat-edge boundary condition.
    float sigma = 1.5f;
    kernel_f(x, y) = (exp(-(x*x + y*y)/(2*sigma*sigma)) / 
                     (float)(2*M_PI*sigma*sigma));
    kernel(x, y) = kernel_f(x, y) / 
                  (kernel_f(0, 0) + kernel_f(1, 0) * 4 + 
                   kernel_f(2, 0) * 4 + kernel_f(1, 1) * 4 + 
                   kernel_f(1, 2) * 4 + kernel_f(2, 1) * 4 + 
                   kernel_f(2, 2) * 4);

    // define the algorithm
    clamped(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
    conv1(x, y) += clamped(x+win.x, y+win.y) * kernel(win.x, win.y);

    hw_output(x, y) = conv1(x, y);
    output(x, y) = cast<float>(hw_output(x, y));
}

void schedule() {
    if (get_target().has_hls_feature()) {
        std::cout << "\ncompiling HLS code..." << std::endl;
        clamped.compute_root(); // prepare the input for the whole image

        // HLS schedule: make a hw pipeline producing 'hw_output', taking
        // inputs of 'clamped', buffering intermediates at (output, xo) loop
        // level
        hw_output.compute_root();
        hw_output.tile(x, y, xo, yo, xi, yi, 512, 512).reorder(xi, yi, xo, yo);
    }
};
}
```
APPENDIX A. IMAGE PROCESSING APPLICATIONS IN HALIDE

Listing A.2: Gaussian Filter

```cpp
#include "Halide.h"

using namespace Halide;

namespace {

class UnsharpFilter : public Halide::Generator<UnsharpFilter> {
public:

  Input<Buffer<float>> input {"input", 2};
  Output<Buffer<float>> output {"output", 2};

  RDom win;

  // Algorithm Description
  void generate() {
    win = RDom(-4, 9);

    // Define a 9x9 Gaussian Blur with a repeat-edge boundary condition.
    float sigma = 1.5f;
    kernel_f(x) = exp(-x*x/(2*sigma*sigma)) / (sqrtf(2*M_PI)*sigma);

    HALIDE_REGISTER_GENERATOR(GaussianConv, "gaussian_conv");
  }
};

} // namespace
```

Listing A.3. Unsharp Mask
```c
// normalize and convert to 8 bit fixed point
kernel(x) = kernel_f(x) / 
  (kernel_f(0) + kernel_f(1) * 2 + kernel_f(2) * 2 + kernel_f(3) * 
   2 + kernel_f(4) * 2);

// define the algorithm
clamped(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
gray(x, y) = clamped(x, y);
blur_y(x, y) += gray(x, y+win.x) * kernel(win.x);
blur_x(x, y) += blur_y(x + win.x, y) * kernel(win.x);
sharpen(x, y) = clamp(clamped(x, y) - blur_x(x, y), 0.0f, 1.0f);
hw_output(x, y) = clamp(sharpen(x, y) + clamped(x, y), 0.0f, 1.0f);
output(x, y) = cast<float>(hw_output(x, y));
}

} // end of schedule

void schedule() {
if (get_target().has_hls_feature()) {
  std::cout << "compiling HLS code..." << std::endl;
  hw_output.compute_root();
  hw_output.tile(x, y, xo, yo, xi, yi, 512, 512).reorder(xi, yi, xo, yo);
  clamped.compute_root();
  blur_y.update(0).unroll(win.x);
  blur_x.update(0).unroll(win.x);
  hw_output.accelerate({clamped}, xi, xo);
  blur_y.linebuffer();
  gray.linebuffer();
  clamped.fifo_depth(hw_output, 512*9); // hw input bounds
} else {
  output.tile(x, y, xo, yo, xi, yi, 64, 64);
  // sharpen.compute_at(output, xo).vectorize(x, 8);
  output.vectorize(xi, 8).reorder(xi, yi, xo, yo);
  blur_x.update(0).unroll(win.x);
  blur_y.update(0).unroll(win.x);
  output.fuse(xo, yo, xo).parallel(xo);
}

Func blur_y("blur_y"), blur_x("blur_x"), gray("gray");
Func kernel_f("kernel_f"), kernel("kernel"), clamped("clamped");
Func sharpen("sharpen"), ratio("ratio");
Func hw_output("hw_output");
Var x("x"), y("y"), c("c");
Var xo("xo"), xi("xi"), yi("yi"), yo("yo");
}

HALIDE_REGISTER_GENERATOR(UnsharpFilter, "unsharp_filter");
```
#include "Halide.h"

using namespace Halide;

namespace {
    class BilateralGrid : public Halide::Generator<BilateralGrid> {
        public:
            GeneratorParam<int> s_sigma("s_sigma", 8);

            Input<Buffer<float>> input("input", 2);
            Input<float> r_sigma("r_sigma");
            Output<Buffer<float>> output("output", 2);

        // Algorithm Description
        void generate() {
            r = RDom(0, s_sigma, 0, s_sigma);
            sigma = s_sigma;

            // Add a boundary condition
            clamped(x, y) = BoundaryConditions::repeat_edge(input)(x, y);

            // Construct the bilateral grid
            Expr val = clamped(x * s_sigma + r.x - s_sigma/2, y * s_sigma + r.y -
                s_sigma/2);
            zero_f = 0.0f;
            one_f = 1.0f;
            val = clamp(val, 0.0f, 1.0f);

            val_const_1 = 10.0f;
            val_const_2 = 0.5f;
            Expr zi = cast<int>(val * (1.0f/0.1f) + 0.5f);

            // Histogram
            histogram(x, y, z, c) = 0.0f;
            histogram(x, y, zi, c) += select(c == 0, val, 1.0f);

            // Blur the grid using a five-tap filter
            blurrz(x, y, z, c) = (histogram(x, y, z-2, c) +
                histogram(x, y, z-1, c)*4 +
                histogram(x, y, z, c)*6 +
                histogram(x, y, z+1, c)*4 +
                histogram(x, y, z+2, c));
            blurrz(x, y, z, c) = (blurx(x-2, y, z, c) +
                blurrz(x-1, y, z, c)*4 +
                blurrz(x, y, z, c)*6 +
                blurrz(x+1, y, z, c)*4 +
                blurrz(x+2, y, z, c));
            blury(x, y, z, c) = (blurx(x, y-2, z, c) +
                blurrz(x-2, y, z, c) +
                blurrz(x-1, y, z, c)*4 +
                blurrz(x, y, z, c)*6 +
                blurrz(x+1, y, z, c)*4 +
                blurrz(x+2, y, z, c));
        }
    }
}

Listing A.3: Unsharp Mask

Listing A.4. Bilateral Grid
APPENDIX A. IMAGE PROCESSING APPLICATIONS IN HALIDE

```c
blurx(x, y-1, z, c)*4 +
blurx(x, y , z, c)*6 +
blurx(x, y+1, z, c)*4 +
blurx(x, y+2, z, c));

// Take trilinear samples to compute the output
int2(input2(x, y) = input(x, y);
val = clamp(input2(x, y), 0.0f, 1.0f);
Expr zv = val * (1.0f/0.1f);
zi = cast<int>(zv);
zf = zv - zi;
xf = cast<float>(x % sigma) / sigma;
yf = cast<float>(y % sigma) / sigma;
Expr xi = x/sigma;
Expr yi = y/sigma;

interpolated(x, y, c) =
  lerp(lerp(lerp(blury(xi , yi , zi , c), blury(xi +1 , yi , zi , c), xf),
    lerp(blury(xi , yi+1 , zi , c), blury(xi +1 , yi+1 , zi , c), xf),
    yf),
  lerp(lerp(blury(xi , yi , zi+1 , c), blury(xi +1 , yi , zi+1 , c), xf),
    lerp(blury(xi , yi+1 , zi+1 , c), blury(xi +1 , yi+1 , zi+1 , c),
    xf), yf), zf);

// Normalize and return the output.
output(x,y) = cast<float> (bilateral_grid(x,y));
```

// Scheduling
```c
void schedule () {
    // int s_sigma = 8;
    if (get_target ().has_gpu_feature ()) {
        // The GPU schedule
        Var xi("xi"), yi("yi"), zi("zi");
        // Schedule blurz in 8x8 tiles. This is a tile in
        // grid-space, which means it represents something like
        // 64x64 pixels in the input (if s_sigma is 8).
        blurz.compute_root().reorder(c, z, x, y).gpu_tile(x, y, xi , yi , 8, 8);
        // Schedule histogram to happen per-tile of blurz, with
        // intermediate results in shared memory. This means histogram
        // and blurz makes a three-stage kernel:
        // 1) Zero out the 8x8 set of histograms
        // 2) Compute those histogram by iterating over lots of the input
        // image
        // 3) Blur the set of histograms in z
        histogram.reorder(c, z, x, y).compute_at(blurz, x).gpu_threads(x, y);
        histogram.update().reorder(c, r.x, r.y, x, y).gpu_threads(x, y).unroll(c);
        // An alternative schedule for histogram that doesn't use shared
        // memory:
        // histogram.compute_root().reorder(c, z, x, y).gpu_tile(x, y, xi,
        // yi, 8, 8);
```
APPENDIX A. IMAGE PROCESSING APPLICATIONS IN HALIDE

```c
// histogram . update (). reorder (c, r. x, r. y, x, y). gpu_tile (x, y, xi, 8). unroll (c);

// Schedule the remaining blurs and the sampling at the end similarly .
blurx . compute_root (). gpu_tile (x, y, z, xi, yi, 8, 8, 1);
blury . compute_root (). gpu_tile (x, y, z, xi, yi, 8, 8, 1);
bilateral_grid . compute_root (). gpu_tile (x, y, xi, yi, s_sigma, s_sigma);

} else if ( get_target (). has_hls_feature () ) {

blurx . linebuffer (). compute_at ( bilateral_grid , x_in );
blury . linebuffer (). compute_at ( bilateral_grid , x_in );
blurz . linebuffer (). compute_at ( bilateral_grid , x_in );

histogram . linebuffer (). compute_at ( blurz , x_in ). reorder (c, z, x, y). unroll (c). unroll (z);

histogram . update (). reorder (c, r.x, r.y, x, y). unroll (c);

clamped . compute_root ();
input2 . compute_root ();

bilateral_grid . tile (x, y, xo, yo, x_in, y_in, 1536, 2560);
bilateral_grid . tile (x_in, y_in, x_grid, y_grid, x_in, y_in, 8, 8);
bilateral_grid . compute_root ();
bilateral_grid . accelerate ({ clamped , input2 }, x_grid, xo);
}
else {

// The CPU schedule .
blurz . compute_root (). reorder (c, z, x, y). parallel (y). vectorize (x, 8). unroll (c);
histogram . compute_at (blurz, y);
histogram . update () . reorder (c, r.x, r.y, x, y). unroll (c);
blurx . compute_root () . reorder (c, x, y, z). parallel (z). vectorize (x, 8). unroll (c);
blury . compute_root () . reorder (c, x, y, z). parallel (z). vectorize (x, 8). unroll (c);
bilateral_grid . compute_root () . parallel (y). vectorize (x, 8);
}

void type_schedule () {

if ( get_target (). has_hls_feature () ) {
clamped . cast_to ( UFixedPoint (15,1) );
input2 . cast_to ( UFixedPoint (15,1) );
histogram . cast_to ( UFixedPoint (20,7), // 0 - 64
    {{ one_f, UFixedPoint (1,1) }},
    {{ val_const_1, UFixedPoint (4,4) }, { val_const_2, UFixedPoint (1,0) }});

// Blur applications .
blurz . cast_to ( UFixedPoint (22,12) );
blurx . cast_to ( UFixedPoint (24,16) );
blury . cast_to ( UFixedPoint (26,20) );
```
interpolated.cast_to(UFixedPoint(28,20),
    {{xf, UFixedPoint(9,1)}, {zf, UFixedPoint(9,1)}, {yf,
    UFixedPoint(9,1)},
    {val_const_1, UFixedPoint(4,4)}},
    bilateral_grid.cast_to(UFixedPoint(10,1)));
}
}
Expr val_const_1, val_const_2;
Expr zf, yf, xf;
Expr sigma, div_sigma;
Expr one_f, zero_f;
Func clamped("clamped"), histogram("histogram");
Func input2("input2");
Func bilateral_grid("bilateral_grid");
Func blurx("blurx"), blury("blury"), blurz("blurz"),
    interpolated("interpolated");
Var x("x"), y("y"), z("z"), c("c");
Var x_in("x_in"), y_in("y_in"), xo("xo"), yo("yo"), x_grid("x_grid"),
    y_grid("y_grid");
RDom r;

// Halide::RegisterGenerator<BilateralGrid> register_me("bilateral_grid");
HALIDE_REGISTER_GENERATOR(BilateralGrid, "bilateral_grid");

} // namespace