Flowpix: DSL Compiler for Image Processing Pipelines

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
***Computer Science and Engineering** by Research*

by

Anish Gulati
20161213
anish.gulati@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
December 2023

Copyright © Anish Gulati, 2023
All Rights Reserved

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled **"Flowpix: DSL Compiler for Image Processing Pipelines"** by **Anish Gulati**, has been carried out under my supervision and is not submitted elsewhere for a degree.

____________	______________________________
Date	Adviser: Prof. Venkata Suresh Reddy Purini

Dedicated to

Mother **Smt. Manisha Rani**

&

Father **Shri. Krishan Gulati**

&

Sister **Anjali**

All that is good in me is because of them

Acknowledgments

I want to express my sincere appreciation to Prof. Suresh Purini for having faith and allowing me to learn and grow as a person under his guidance. He has been an excellent guide and supported me with his motivation, knowledge, and patience. I am truly honored and humbled to be the student of a great researcher and human being.

All this time of research journey with CSG, IIIT Hyderabad, has been astounding. I want to thank my mentor Ziaul Choudhury for helping me out during the entire project duration. The research work presented in this thesis would not have been possible without the expertise of him. I have spent countless hours of discussions and iterations over numerous experiments regarding the research with him. All this helped me clear my doubts and give me the right direction to grow far beyond what I would have managed had I been on my own. I would like to thank my friend, Bhumika Satpathy, for being there with me and helping me out whenever I was stuck on some problem. Next, I want to thank my roommate, Vashist Madiraju, who was always there to cheer me up with his positive mindset and cheerful nature. My other friends Aditya, Ayush, Bajaj, Bakshi, Chand, Joshi, Khatri, Kunal, Prusty, Pushpa, Sodhi, and Sumukh, supported me constantly. I express my gratitude towards them for giving me a fun and rewarding experience at IIIT.

Most importantly, I am very thankful to my family for their constant support and motivation.

Abstract

The exponential performance growth guaranteed by Moore's law has started to taper in recent years. At the same time, emerging applications like image processing demand heavy computational performance. These factors inevitably lead to the emergence of domain-specific accelerators (DSA) to fill the performance void left by conventional architectures. FPGAs are rapidly evolving towards becoming an alternative to custom ASICs for designing DSAs because of their low power consumption and a higher degree of parallelism. DSA design on FPGAs requires careful calibration of the FPGA compute and memory resources towards achieving optimal throughput.

Hardware Descriptive Languages (HDL) like Verilog have been traditionally used to design FPGA hardware. HDLs are not geared towards any domain, and the user has to put in much effort to describe the hardware at the register transfer level. Domain Specific Languages (DSLs) and compilers have been recently used to weave together handwritten HDLs templates targeting a specific domain. Recent efforts have designed DSAs with image-processing DSLs targeting FPGAs. Image computations in the DSL are lowered to pre-existing templates or lower-level languages like HLS-C. This approach requires expensive FPGA re-flashing for every new workload. In contrast to this fixed-function hardware approach, overlays are gaining traction. *Overlays* are DSAs resembling a processor, which is synthesized and flashed on the FPGA once but is flexible enough to process a broad class of computations through soft reconfiguration. Image processing algorithms vary in size and shape, ranging from simple blurring operations to complex pyramid systems. The primary challenge in designing an image-processing overlay is maintaining flexibility in mapping different algorithms.

This thesis proposes a domain-specific language (DSL)-based compiler, Flowpix, for image processing. Flowpix is capable of natively processing benchmarks and targeting FPGA implementation of these benchmarks using overlay configurations. This thesis discusses how an application can be expressed using the Flowpix language, and how the compiler processes the application to generate control words that enables the configuration of the hardware architecture. We also compare our results with the existing state-of-the-art frameworks: Polymage, Halide, Xilinx, Darkroom, and Rigel.

Contents

Chapter Page

1 Introduction . . . 1

1.1 Motivation . . . 1

1.2 Summary of Contributions . . . 2

1.3 Thesis Organization . . . 2

2 Image Processing and Background on Existing Frameworks . . . 4

2.1 Image Processing Pipelines . . . 4

2.2 Applications of Image Processing . . . 5

2.3 Existing Frameworks and Related Work . . . 6

2.3.1 Polymage . . . 6

2.3.2 Halide . . . 9

2.3.3 Darkroom and Rigel . . . 11

2.3.4 Ippro . . . 11

2.3.5 Summary . . . 12

3 Language Design and Compiler . . . 14

3.1 Flowpix Overlay Architecture . . . 14

3.2 Flowpix Front-end DSL . . . 15

3.3 FlowPix Compiler . . . 21

3.3.1 DAG Clustering . . . 21

3.3.2 Latency Matching 22

3.3.3 Control Word Generation 23

3.4 FlowPix Scheduler 24

3.5 Example 1 27

3.6 Example 2 27

4 Experiments and Results 30

4.1 Experimental Setup 30

4.2 Performance Analysis 31

4.3 Framework Analysis 35

5 Conclusions 37

5.1 Limitations and Future work 37

5.1.1 Improve Clustering Algorithm 37

5.1.2 Feedback from Scheduler 37

Bibliography 39

List of Figures

Figure Page

2.1 DAG representation of the Unsharp Mask pipeline. *Image credits:* Input image by Dan Diffendale [7]. 7

3.1 The figure above portrays the overall flow of program compilation and execution in FlowPix. The compiler takes in DSL code and produces a sequence of control words. The directed acyclic graph (DAG) is divided into smaller clusters, each of which is mapped to a processing engine (PE) in a compute unit (CU). The driver configures the overlay using the generated control words and streams the input image. The image is segmented into smaller strips that are processed simultaneously across the available CUs. 16

3.2 Computational DAG of the Unsharp Mask pipeline. 18

3.3 Computational DAG of the 2-level Guassian Pyramid pipeline. 19

3.4 The illustration depicts the various stages of the compilation process. The FlowPix compiler generates a sequence of control words for the input DAG. 20

3.5 The illustration depicts a dummy DAG. We use this DAG to provide proof of the scheduling algorithm. 25

3.6 The figure depicts a dummy application with 3x3 stencil, pointwise, and upsample nodes, as well as its clustering and execution schedule. The DAG clusters are executed over a CU with two memory banks. The number on each bank corresponds to the node whose output it presently stores. 28

3.7 The figure on the left shows the DAG of the Pyramid Blending benchmark. The execution schedule is shown on the right. The DAG is broken into six clusters. The number of each bank corresponds to the node whose output it presently stores. 29

4.1 Analysis of the FlowPix Compiler. 36

List of Tables

Table Page

3.1 Computations corresponding to USM Stages 18

3.2 Computations corresponding to 2-level Gaussian Pyramid Stages 19

4.1 A short description of the benchmarks implemented using FlowPix. 31

4.2 Lines of code comparison of the different frameworks for the benchmarks implemented. 32

4.3 FPGA resource consumed by the different architecture variants of our overlay. P_y is set to 8 in all the PE designs. 32

4.4 Comparing FlowPix with HeteroHalide. 32

4.5 Comparing FlowPix with VITIS-HLS. 33

4.6 Comparing FlowPix with Darkroom and Rigel. 33

4.7 Comparing FlowPix with PolyMage. 33

4.8 Comparison of FlowPix with IPPro. 33

Chapter 1

Introduction

1.1 Motivation

The exponential performance growth guaranteed by Moore's law has started to taper in recent years. At the same time, emerging applications like image processing demand heavy computational performance. These factors inevitably lead to the emergence of domain-specific accelerators (DSA) to fill the performance void left by conventional architectures. Field Programmable Gate Arrays (FPGAs) are rapidly evolving towards becoming an alternative to custom ASICs for designing DSAs because of their low power consumption and a higher degree of parallelism. DSA design on FPGAs requires careful calibration of the FPGA compute and memory resources towards achieving optimal throughput.

Traditionally, Hardware Descriptive Languages (HDLs) such as Verilog have been used to design FPGA hardware. However, HDLs are not domain-specific, and users must expend significant effort describing the hardware at the register transfer level. In contrast, Domain-Specific Languages (DSLs) and compilers have recently been used to weave together handwritten HDL templates that target a specific domain. Efforts have been made to design DSAs with image-processing DSLs targeting FPGAs. Image computations in the DSL are lowered to pre-existing templates or lower-level languages such as High-Level Synthesis C (HLS-C). However, this approach necessitates expensive FPGA re-flashing for each new workload.

Overlays are gaining traction as an alternative to this fixed-function hardware approach. Overlays are DSAs that resemble a processor and are synthesized and flashed on the FPGA once, but are flexible enough to process a broad range of computations through soft reconfiguration. Less work has been reported in the context of image processing overlays, and designing an image-processing overlay poses a primary challenge of maintaining flexibility in mapping different algorithms.

In this thesis, we present a compiler for DSL-based overlay accelerator called FlowPix for image processing applications. The DSL programs are expressed as pipelines, with each stage represent-

ing a computational step in the overall algorithm. We implemented 15 image-processing benchmarks using FlowPix on a Virtex-7-690t FPGA, ranging from simple blur operations to complex pipelines like Lucas-Kande optical flow. We compared FlowPix against existing DSL-to-FPGA frameworks like Hetero-Halide[14] and Vitis-HLS[1]. FlowPix achieves an average frame rate of 170 FPS on HD frames of 1920x1080 pixels in the implemented benchmarks.

1.2 Summary of Contributions

Chapter 3 of this thesis outlines the main contributions, which are summarized below:

- Firstly, we present a front-end domain-specific language (DSL) called Flowpix, built in Scala, which enables the programmer to express complex image processing pipelines using a minimal number of code lines.
- Secondly, we demonstrate that this DSL is capable of realizing the algorithm as a directed acyclic graph (DAG), which can then be further broken down into multiple DAG clusters in accordance with hardware architecture specifications. This thesis gives an overview of the architecture, but does not dive deep in the hardware architecture details and arrangement.
- Thirdly, we propose a generic scheduler that determines the order of execution of these clusters within specified constraints. We provide the proof of how these constraints are met in the given order of execution of clusters.
- Lastly, we demonstrate the effectiveness of our framework by processing 15 image processing benchmarks, including simple filters and complex pyramid systems. Our results are comparable to, and in some cases better than, existing DSL-to-FPGA frameworks that generate fixed-function hardware based on a front-end specification.

1.3 Thesis Organization

The thesis is further divided into the following chapters:

- **Chapter 2** gives an introduction to image processing and its applications in various real-life applications. It then talks about Image processing pipelines, its advantages and the data flow model. It will end by talking about various existing frameworks and pipelines in the image processing area, and some of their usecases.

- **Chapter 3** introduces Flowpix, a front-end domain-specific language (DSL) for expressing image processing pipelines. It describes the compiler that enables the language to represent any algorithm as a directed acyclic graph (DAG), cluster it, match the latencies, and schedule the multiple clusters in an order that satisfies specific constraints. Additionally, this chapter provides some example algorithms to illustrate how the DSL processes it.
- **Chapter 4** describes the experimental setup used for evaluating our design and presents the results obtained from our analysis.
- **Chapter 5** summarizes the major contributions of the thesis, discusses the key takeaways, and outlines the future scope of the problem.
- **Bibliography** contains all the referenced papers used in this thesis.

Chapter 2

Image Processing and Background on Existing Frameworks

This section introduces the concept of image processing pipelines and highlights the importance of optimizing these pipelines for efficient performance. It further explores various existing image processing frameworks, which play a crucial role in implementing these pipelines. By delving into the specifics of these frameworks, we can gain a deeper understanding of their relevance and potential benefits.

2.1 Image Processing Pipelines

Image processing pipelines are a crucial component of modern image analysis, with applications spanning computer vision, medical imaging, remote sensing, and more. These pipelines employ a sequence of algorithms to extract useful information from digital images, encompassing four stages: image acquisition, pre-processing, feature extraction, and classification. In the first stage, the digital image is obtained from a camera or database, and pre-processing techniques are applied to improve image quality. Feature extraction identifies specific characteristics of the image, such as edges, shapes, colors, or textures, followed by the classification stage, which categorizes images based on their features. The pipeline's accuracy heavily depends on algorithm selection, parameter optimization, and application order, and machine learning can be used to enhance performance.

An image processing pipeline comprises a directed acyclic graph (DAG) of stages, with each node representing a computation stage that generates output pixels from input pixels. The DAG's edges represent producer-consumer relationships between stages, and each node can be a stencil or point-wise computation stage, and other types of computation stages. Stencil stages perform operations, such as convolution or max filters, on small pixel windows, while point-wise stages use input pixels from multiple predecessor stages to compute each output pixel. The pipeline's streaming data flow model is amenable to FPGA implementation, and stages can be executed in parallel, subject to data availability and dependency constraints.

Pointwise stages do not require buffering, as they require precisely one input pixel per prior stage output. In contrast, stencil operations require multiple input pixels, necessitating line buffering of at least two rows of the input frame before computation. This buffering ensures the intermediate values are transmitted from the producer to the consumer without external memory intervention, reducing energy consumption.

2.2 Applications of Image Processing

Image processing pipelines have diverse applications, such as medical imaging, where they identify tumors or anomalies in MRI and CT scans. They can also be used in computer vision to detect intruders in security camera images or identify objects. Remote sensing applications employ pipelines to analyze satellite images for crop growth, changes in land use, and natural disasters. In conclusion, image processing pipelines are essential tools for automated image analysis, and optimizing algorithm selection and application order can improve performance.

Image processing has become an essential tool in various fields such as medical imaging, remote sensing, computer vision, and more. It involves the analysis and manipulation of digital images to extract useful information and enhance their quality. In today's world, image processing is used in a wide range of applications, from diagnosing medical conditions to improving satellite imagery.

One of the most significant applications of image processing is in the medical field. Recent work has shown that image processing can aid in the early detection and diagnosis of diseases, such as cancer. For example, a study published in the Journal of Medical Imaging in 2021[16] explored the use of deep learning algorithms for the early detection of breast cancer in mammograms. The researchers used a pipeline that combined image preprocessing, feature extraction, and classification to achieve high accuracy in detecting breast cancer.

Image processing is also used in remote sensing to analyze satellite imagery and monitor changes in land use, crop growth, and natural disasters. A study titled "UAV-based forest fire detection and tracking using image processing techniques"[26] demonstrated the application of image processing in detecting and tracking forest fires. The researchers utilized unmanned aerial vehicles (UAVs) equipped with cameras to capture high-resolution images of forested areas. Through the implementation of image processing techniques such as fire detection algorithms and thermal imaging analysis, the study achieved accurate and real-time forest fire detection and tracking. The findings highlight the effectiveness of image processing in facilitating early response and intervention in forest fire management.

In computer vision, image processing is used to analyze images from security cameras, identify objects, and detect intruders. Recent work has focused on developing algorithms for video-based human

action recognition using deep learning techniques. For example, a study titled "Video-based human action recognition using deep learning[19] demonstrated the effectiveness of a pipeline that utilized deep learning algorithms for recognizing and interpreting complex human actions in videos. The study employed techniques such as convolutional neural networks and temporal feature aggregation to achieve high accuracy in video-based human action recognition.

In conclusion, image processing has numerous applications in today's world, ranging from medical imaging to computer vision and artistic applications. Recent work has shown the potential of image processing techniques such as deep learning algorithms and image segmentation and classification in improving the accuracy and efficiency of image processing pipelines.

2.3 Existing Frameworks and Related Work

2.3.1 Polymage

Polymage[17] is an Image rocessing pipeline that has been designed to optimize the performance of image processing algorithms on modern computer architectures. It was developed by a team of researchers from Multicore Computing Lab at the Indian Institute of Science, with the aim of reducing the computational complexity of image processing tasks by automatically generating specialized code for different hardware platforms.

The Polymage system addresses a significant challenge in image processing, which is the need to process large amounts of data in real-time with high efficiency. This is particularly important in applications such as computer vision, robotics, and autonomous systems, where real-time processing is essential for accurate and timely decision-making. It achieves its efficiency gains by breaking down image processing tasks into smaller, more specialized sub-tasks that can be optimized for specific hardware architectures. These sub-tasks are then combined into a pipeline that can process images in real-time with high efficiency. The system is highly customizable, allowing developers to tailor the pipeline to specific applications and hardware platforms.

The unsharp mask algorithm is a popular technique for image sharpening in digital image processing systems. It can be implemented efficiently in PolyMage using its powerful DSL. Listing 2.1 shows the code for unsharp mask in Polymage, and figure 2.1 shows the dependency graph generated for this algorithm.

The PolyMage DSL code for the unsharp mask algorithm is expressed as a composition of four pipelined stages. The first two stages apply a Gaussian blur to the original input image along the x and y directions, respectively, using PolyMage's $Stencil$ construct. This construct takes as input the

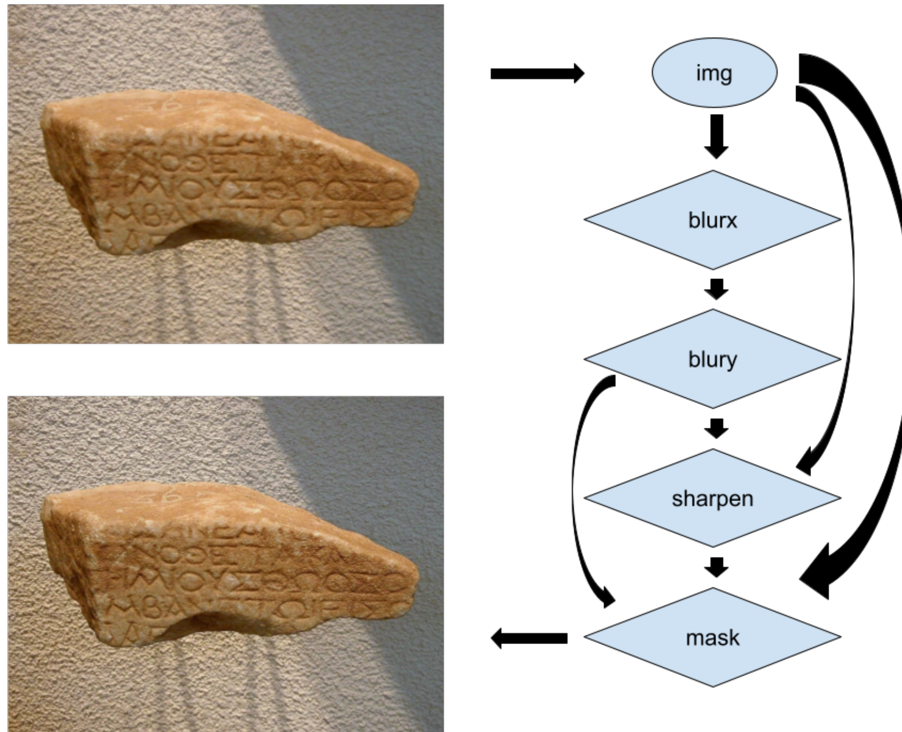


Figure 2.1 DAG representation of the Unsharp Mask pipeline. *Image credits:* Input image by Dan Diffendale [7].

point around which the stencil should be applied, a global weight for stencil output, and a stencil kernel defined using a list of lists. The $Stencil$ construct in PolyMage facilitates the efficient computation of Gaussian blurs by providing a customizable and optimized implementation of the stencil operation.

In the third stage of the unsharp mask algorithm, a weighted sum of the resulting blurs is computed to generate a sharpened image, also using the Stencil construct. The final stage of the algorithm selects pixels to be written to the output using PolyMage's $Select$ construct. This step chooses sharpened pixels based on whether the difference between the original and its blur image crosses a threshold value.

The system has been evaluated extensively in a range of applications, including real-time object detection and tracking, image filtering, and feature detection. In all cases, the system has demonstrated significant performance improvements over traditional image processing pipelines. For example, in one study, the Polymage system was shown to achieve a 9.5x speedup[17] over traditional image processing pipelines for object detection and tracking tasks.

```
# Params
R = Parameter(Int , "R")
C = Parameter(Int , "C")
thresh = Parameter(Float, "thresh")
weight = Parameter(Float, "weight")

# Vars
x = Variable(Int, "x")
y = Variable(Int, "y")
c = Variable(Int, "c")

# Input Image
img = Image(Float, "input" , [3, R+4, C+4])

# Intervals
cr = Interval(Int, 0, 2, 1)
xrow = Interval(Int, 2, R+1, 1)
xcol = Interval(Int, 0, C+3, 1)
yrow = Interval(Int, 2, R+1, 1)
ycol = Interval(Int, 2, C+1, 1)

# Pipeline
blurx = Function([c, x, y], [cr, xrow, xcol],
                 Float, "blurx")
blurx.defn = [Stencil(img (c, x, y), 1.0/16,
                      [[1], [4], [6], [4], [1]])]
blury = Function([c, x, y], [cr, yrow, ycol],
                 Float, "blury")
blury.defn = [Stencil(blurx(c, x, y), 1.0/16,
                       [[1 , 4, 6, 4, 1]])]

sharpen = Function([c, x, y], [cr, yrow, ycol],
                   Float, "sharpen")
sharpen.defn = [img(c, x, y) * (1 + weight) \
               + blury(c, x,y) * (- weight)]
masked = Function([c, x, y], [cr, yrow, ycol],
                  Float, "mask")
absv = Abs((img(c, x, y) - blury(c, x, y)))
masked.defn = [Select(Condition(absv, "<", thresh),
                      img(c, x, y),
                      sharpen(c, x, y))]
```

Listing 2.1 Polymage DSL code for Unsharp Mask pipeline.

2.3.2 Halide

Halide[21] is a programming language and compiler designed for image processing tasks. It allows developers to express image processing algorithms in a concise, high-level language that is easy to read and maintain. Halide aims to optimize image processing pipelines for high performance on modern hardware, including CPUs, GPUs, and specialized image processing hardware.

The Halide image processing pipeline consists of a series of stages that process the input image to generate the desired output image. Each stage in the pipeline is defined by a Halide function, which specifies the computation to be performed on the input image. Halide functions can be composed to create complex image processing pipelines, allowing developers to express their algorithms in a modular and flexible manner.

One of the key features of Halide is its ability to optimize image processing pipelines for high performance. The Halide compiler performs a range of optimizations, including loop tiling, loop fusion, and vectorization, to generate highly efficient code for the target hardware. This allows image processing pipelines written in Halide to achieve performance close to hand-optimized code, while still retaining the ease-of-use and maintainability of a high-level programming language. Listing 2.2 shows the implementation of Unsharp mask algorithm in Halide.

Halide also supports automatic scheduling, which allows the compiler to generate optimized schedules for a given pipeline automatically. Developers can specify constraints on the schedule, such as memory usage and execution time, to guide the compiler's optimization process. This feature allows developers to write image processing pipelines in a high-level language, while still maintaining the performance.

```
Var x("x"), y("y"), c("c");

const float kPi = 3.14159265358979310000f;
Func kernel("kernel");
kernel(x) = exp(-x * x / (2 * sigma * sigma)) / (sqrtf(2 * kPi) * sigma);

Func gray("gray");
gray(x, y) = (0.299f * input(x, y, 0) +
              0.587f * input(x, y, 1) +
              0.114f * input(x, y, 2));

Func blur_y("blur_y");
blur_y(x, y) = (kernel(0) * gray(x, y) +
                kernel(1) * (gray(x, y - 1) +
                             gray(x, y + 1)) +
                kernel(2) * (gray(x, y - 2) +
                             gray(x, y + 2)) +
                kernel(3) * (gray(x, y - 3) +
                             gray(x, y + 3)));

Func blur_x("blur_x");
blur_x(x, y) = (kernel(0) * blur_y(x, y) +
                kernel(1) * (blur_y(x - 1, y) +
                             blur_y(x + 1, y)) +
                kernel(2) * (blur_y(x - 2, y) +
                             blur_y(x + 2, y)) +
                kernel(3) * (blur_y(x - 3, y) +
                             blur_y(x + 3, y)));

Func sharpen("sharpen");
sharpen(x, y) = 2 * gray(x, y) - blur_x(x, y);

Func ratio("ratio");
ratio(x, y) = sharpen(x, y) / gray(x, y);

output(x, y, c) = ratio(x, y) * input(x, y, c);
```

Listing 2.2 Halide DSL algorithm for Unsharp Mask pipeline. *Code credits:* Code by Andrew Adams [3]

2.3.3 Darkroom and Rigel

Darkroom[10] and Rigel[11] are two image processing pipelines used in astronomical research to enhance the quality of astronomical images. Both Darkroom and Rigel are widely used in the field, but differ in their design, implementation, and availability.

Darkroom is an open-source image processing pipeline that aims to provide an accessible and user-friendly interface for processing raw astronomical images. Developed in Python, Darkroom is based on popular scientific Python libraries such as NumPy, SciPy, and Astropy. The pipeline offers a range of standard image processing techniques, including bias, dark, and flat correction, image registration, stacking, and post-processing. Moreover, Darkroom comes with a graphical user interface (GUI) that facilitates its use by amateur astronomers and astrophotographers.

In contrast, Rigel is a proprietary image processing pipeline developed by Apogee Instruments, a company that specializes in high-performance scientific imaging equipment for astronomical research. Designed to work with Apogee's line of CCD cameras, Rigel offers advanced image processing techniques, including automatic image alignment and deconvolution, as well as the ability to handle large image datasets. Unlike Darkroom, Rigel is a commercial product that requires a license for use.

Both Darkroom and Rigel are highly regarded in the astronomical community for their effectiveness in processing astronomical images. Darkroom's open-source nature makes it accessible to anyone interested in using it, while Rigel's proprietary nature provides commercial customers with access to more advanced features. However, the choice of which pipeline to use ultimately depends on the user's specific needs and preferences.

2.3.4 Ippro

IPPRO[24] (Image Processing PROcessor) is a high-performance image and video processing platform. Its custom-designed FPGA board, software libraries, and development tools provide a highly flexible and configurable solution for image and video processing tasks.

The IPPRO platform offers a number of benefits that make it an attractive choice for researchers and engineers working in the field of image and video processing. Firstly, the platform offers high processing speed and low latency, enabling it to perform real-time image processing tasks. This makes it ideal for a range of applications, including surveillance, robotics, medical imaging, and scientific research.

Secondly, the platform's FPGA-based architecture offers significant flexibility and configurability, allowing users to tailor the platform to their specific needs. The platform comes with a set of software libraries and development tools that enable users to program the FPGA for their specific applications.

Finally, the IPPRO platform has been used in a variety of research projects, including the development of real-time image processing systems for space exploration, autonomous vehicles, and medical imaging. Its flexibility, programmability, and high performance make it a valuable tool for a wide range of applications in image and video processing research and development.

In conclusion, the IPPRO platform is a powerful and flexible solution for high-performance image and video processing tasks. Its FPGA-based architecture, software libraries, and development tools make it an attractive choice for researchers and engineers working in the field of image and video processing. The platform's high processing speed and low latency, combined with its configurability and flexibility, make it a valuable tool for a wide range of applications.

2.3.5 Summary

Coming up with high-performance hardware using HLS [15, 6] is like using C to develop high-performance software [20]. This limitation is overcome by using domain-level abstractions on top of the high-level language in the form of DSLs with micro-architectural templates, generating hardware blocks relevant to the domain [2, 18, 9, 12]. Aetherling [8] is a DSL that compiles high-level image processing algorithms to hardware with the focus of exploring resource-vs-throughput trade-offs. DSAGEN[25] DSL annotates algorithms using pragmas and automatically searches a large architecture design space for a range of algorithms. Researchers have created image processing DSLs based on the line buffered pipeline model for the image processing domain. Darkroom [10], converted a pipeline of the stencil and pointwise computations into custom hardware using a synchronous data flow model [13] model. The resulting pipelines were statically scheduled, supporting single rate processing at a throughput of one pixel/cycle. Rigel[11], follow-up work on Darkroom, supported multi-rate pipelines, enabling the programmer to specify pipeline stages running at throughput other than one pixel/cycle. An integer linear programming formulation was used in both these frameworks to calculate the line buffer sizes. Rigel used FIFO buffers to handle the variability between two pipeline stages. FlowPix employs a more intuitive DAG formulation and data relaying techniques to find and optimize line buffer sizes in the pipeline. As against designing a new DSL, there have been works augmenting pre-existing DSLs targeted for CPU and GPU architectures with an FPGA back-end. The `Halide` HLS framework [20] extended the Halide compiler [21], allowing the user to accelerate their programs on the FPGA. The system provided high-level semantics to explore different hardware mappings of applications, including the flexibility of changing the throughput rate of the individual stage. The PolyMageHLS [4], compiler extended the PolyMage [17] DSL to target FPGAs. It exploited coarse-grain parallelism by creating multiple copies of the pipeline to process image tiles in parallel. PolyMage and Halide emit HLS-C and feed that output to Vivado HLS to generate the hardware. O. Reiche et al. [22], [28] explored Image

Processing and a source-to-source compiler called Hipacc that produces highly efficient hardware accelerators. By utilizing spatial and architectural information, Hipacc can generate tailored implementations that can compete with handwritten source codes, without requiring an expert in hardware architecture. The DSL's capabilities enable the definition of algorithms for a wide range of target platforms, and also allow for the exploitation of target-specific forms of parallelism.

Chapter 3

Language Design and Compiler

FlowPix comprises a DSL front-end, a compiler, and a backend hardware overlay. The DSL is embedded in the Scala language. It is possible to run a FlowPix DSL code as yet another Scala program that executes on the underlying CPU. This feature facilitates the programmer during the development cycle. One can use the CPU execution mode without incurring expensive hardware execution for debugging purposes. The FlowPix compiler constructs a Directed Acyclic Graph (DAG) from the source DSL program. It maps the DAG to the processing engines in the overlay and generates a control word sequence accordingly. The overlay is configured to process the DAG computations through these control words. A C++ driver streams the control words followed by the image frames to the overlay. The overlay continues to perform the same computations until it is reconfigured by the driver using a different control word sequence. Figure 3.1 summarises the compilation and execution flow of a FlowPix DSL program. We discuss the overview about the Flowpix overlay architecture below, and then dive deep into DSL and compiler in further sections.

3.1 Flowpix Overlay Architecture

The FlowPix overlay is designed as a collection of Compute units (CUs). Each CU is further made up of an array of Processing Engines (PE) connected in a pipeline via FIFO interfaces. Each PE, in turn, has a fully pipelined design. Thus, both *intra-PE* and *inter-PE* pipelined parallelism are exploited within a CU. The DAG is broken into smaller compute units called clusters before processing. Each DAG cluster is scheduled for execution on a single CU. The topologically sorted DAG clusters are mapped to PEs, facilitating streaming dataflow computation within a PE and across PEs in the same CU. If the available PEs are less than the number of clusters, we time multiplex by reconfiguring the PE. The generated intermediate data is internally buffered with no necessity for off-chip DDR memory accesses. Newer clusters are configured to run on the overlay through soft reconfiguration to process this intermediate

data to generate the final processed image. The reconfiguration is inexpensive since we can reset the PE control words in minimal cycles at runtime.

The overlay has a parametric design. The number of CUs and the size of the PE array inside each CU can be configured at hardware synthesis time depending on the FPGA and the workload characteristics. In the presence of more CUs, the same computations are instantiated across all the CUs. Based on the number of CUs, the image is vertically divided into an equal number of strips. The size of each strip is fixed based on the size of the overlay's internal memory. Thus the available *data parallelism* is exploited by processing the strips in parallel over the CUs. A single strip is processed over the PE array inside a CU. The overlay has a dedicated control memory for storing the control words. This memory is distributed across the PEs. A distribution logic routes the incoming control words to their correct location inside the control memory. The processed image from the overlay is streamed back to the driver, from where it is passed upwards to the application logic. The parallel CUs can also be used to process different workloads. In other words, the FlowPix framework can accelerate two or more similar image processing workloads at the same time.

3.2 Flowpix Front-end DSL

This section provides a brief of how Image processing pipelines can be expressed using the Flowpix DSL. Our DSL is embedded in the Scala Language. DSL programmers can use all Scala features with embedded DSL constructs for specifying image processing pipelines. All the input and intermediate images in the computational pipeline are stored as Scala objects of class type `Stage`. Point-wise computations on images are handled through operator overloading. Filter operations on images corresponding to stencil stages are obtained by instantiating the `Filter` class with suitable parameters. The currently supported filter operations in the DSL are `convolution`, `max`, `min`, `sum` and `average`. It is possible to support more filter operations in our DSL compiler infrastructure easily. We relieve the programmer from specifying the image dimensions at each computation step. Instead, our compiler automatically infers the dimensions through DAG analysis.

A standard image processing algorithm, Unsharp Mask (USM), used as a running example in this section, is a commonly used image sharpening technique in digital image processing systems. In a signal processing context, we can visualize it as a filter that amplifies an input signal's high-frequency components. The algorithm is expressed as a composition of three operations. Firstly, a $blur$ operation performs a Gaussian blur of the original input image along the y and the x directions. The second operation, $sharpen$, computes a weighted sum of the resultant blur and outputs a sharpened image. The final stage, $mask$, chooses pixels to be written to the output, between the original and the sharpened

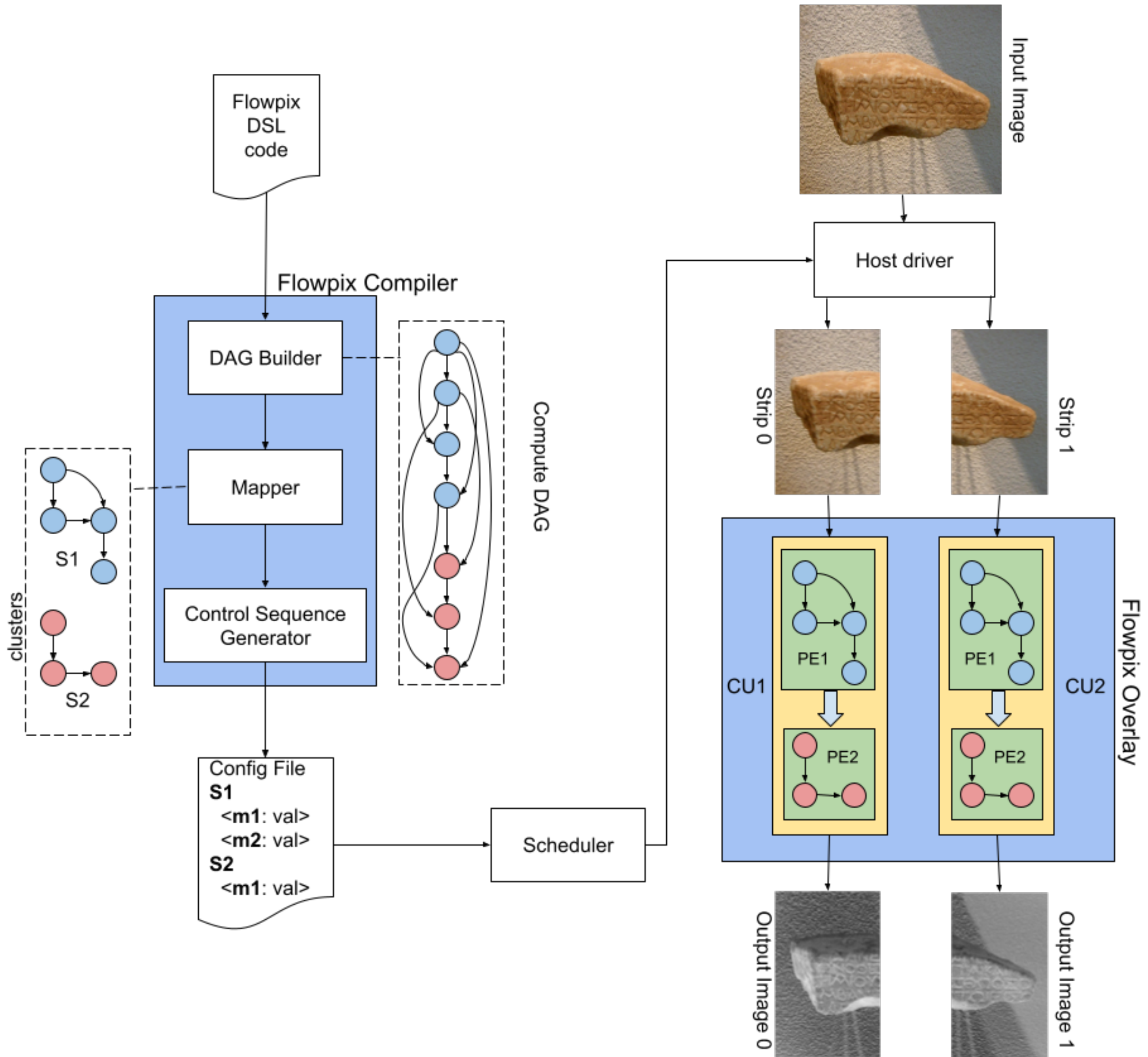


Figure 3.1 The figure above portrays the overall flow of program compilation and execution in FlowPix. The compiler takes in DSL code and produces a sequence of control words. The directed acyclic graph (DAG) is divided into smaller clusters, each of which is mapped to a processing engine (PE) in a compute unit (CU). The driver configures the overlay using the generated control words and streams the input image. The image is segmented into smaller strips that are processed simultaneously across the available CUs.

image. Sharpened pixels are chosen based on the difference between the original and its blur image, crossing a threshold value.

Figure 3.2 shows the computational DAG associated with the Unsharp Mask (USM) benchmark. Stencil and point-wise stages are denoted by diamond and circle symbols, respectively. The computations associated with each stage are given in Table 3.1. Listing 3.1 shows the FlowPix code for the USM benchmark. The input Stage object `img` (line 1) is instantiated by passing a valid image path to the `Stage` class. The other stage objects in the program `blur` (line 7), `sharp` (line 9), `thresh` (line 10) and `mask` (line 11) correspond to different stages in the DAG. The intermediate image at the `blur` stage is obtained by applying the filter `kernel` on the image `img` (line 7) using the `**` operator. `kernel` is a convolution filter obtained by instantiating the `Filter` class with a suitable stencil matrix as a parameter (line 3). The first two parameters in a filter instantiation correspond to the filter dimensions and filter strides, respectively. Filter dimension is a tuple of two integers that specifies the filter's number of rows and columns. A filter stride is a tuple of two integers corresponding to the row and column strides by which the filter moves over the image. The last parameter specifies the weights of the convolution operation. Point-wise operations on lines 9, 10, and 11 are written as expressions over image objects. The output Stage object `mask` is stored as an image by passing an image path to the `store_image` method provided by the `Stage` class.

```
1  var img = Stage("Path/to/image")
2  var kernel = Filter((3, 3), (1, 1),
3       filter_data = (1/16, 1/8, 1/16),
4                     (1/8, 1/4, 1/8),
5                     (1/16, 1/8, 1/16))
6  # stencil operation
7  var blur = img ** kernel
8  # pointwise operation
9  var sharp = img * 0.8 - blur
10 var thresh = img * 0.2 - blur
11 var mask = (thresh >= 0)? (img, sharp)
12 mask.store_image("Path/to/new/image")
```

Listing 3.1 FlowPix code for Unsharp Mask pipeline.

Figure 3.3 and Table 3.2 show the DAG and the associated computations for the 2-level Gaussian pyramid benchmark. This benchmark performs a downsampling operation using the `max` filter. The downsampling stage is represented using an inverted pyramid symbol in the DAG. Listing 3.2 shows the corresponding FlowPix code.

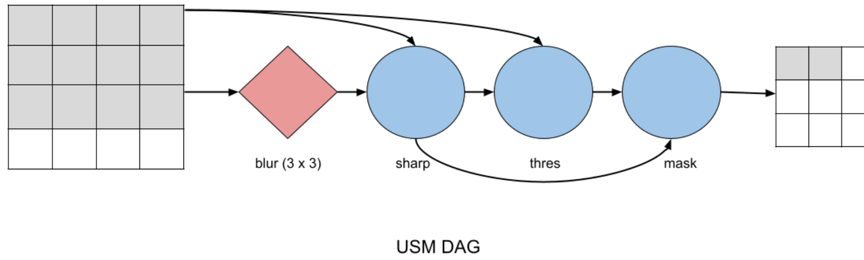


Figure 3.2 Computational DAG of the Unsharp Mask pipeline.

Stage	Computation
I	$Input\ Image$
$blur$	$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$
$sharp$	$0.8I\,(i,j) - blur\,(i,j)$
$thres$	$0.2I\,(i,j) - blur\,(i,j)$
$mask$	$thres\,(i,j) >= 0$
	$?I\,(i,j) : sharp\,(i,j)$

Table 3.1 Computations corresponding to USM Stages

Stage	Computation
I	$Input\ Image$
$blur_0$	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$
$down_0$	$\frac{I}{4}$
$blur_1$	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$
$down_1$	$\frac{I}{16}$

Table 3.2 Computations corresponding to 2-level Gaussian Pyramid Stages

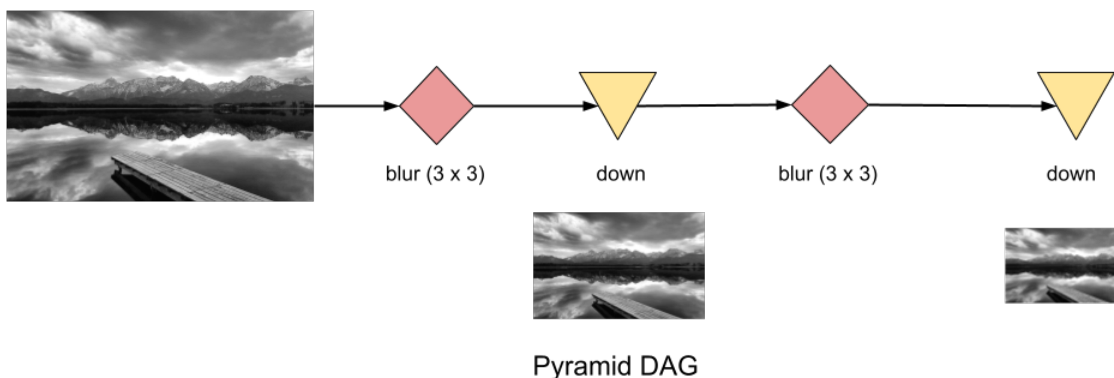


Figure 3.3 Computational DAG of the 2-level Guassian Pyramid pipeline.

```
var img = Stage("Path/to/Image")
var down = Filter((2, 2), (2, 2), "max")
var kernel = Filter((3, 3), (1, 1),
                   ((1/16, 1/8, 1/16),
                   (1/8, 1/4, 1/8),
                   (1/16, 1/8, 1/16))
                   )
for(i <- 0 until 2){
 var blur = img ** filter
 var reduce = blur ** down
 img = reduce
}
img.store_image("gaussian_pyramid")
```

Listing 3.2 Code for 2-level Pyramid.

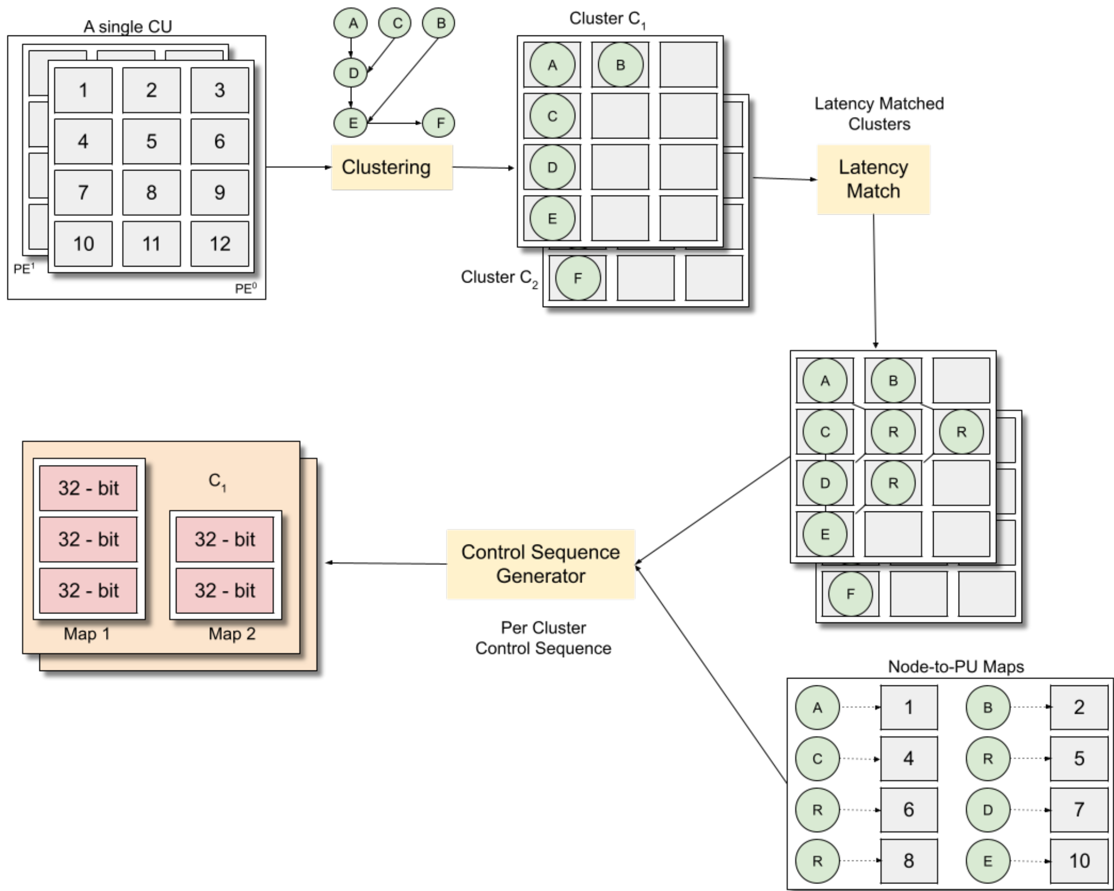


Figure 3.4 The illustration depicts the various stages of the compilation process. The FlowPix compiler generates a sequence of control words for the input DAG.

The image is first convolved using a 3×3 convolution filter `kernel` (Line 3) and then passed to a 2×2 `max` filter `down` (Line 2). The `max` filter outputs the pixel with the highest intensity within the pixel window. The window moves with a row and column stride of 2, shrinking the image by a factor of 4. In Line 2, the first tuple denotes the filter dimensions, and the second tuple denotes the row and column strides. The convolution and max-pooling stages are repeated to generate the final output image. Iterative computations are expressed using Scala's loop construct. The above computation sequence of convolving and down-sampling the image is put inside a loop such that the output image from one iteration acts as the input image for the next. The image is iteratively down-sampled to $\frac{1}{16}^{th}$ of its original dimension using a loop iteration count of two.

3.3 FlowPix Compiler

The FlowPix compiler generates a control word sequence from the DSL code. Generating these control words is similar to the code generation step in traditional compilers. The compiler can be broadly divided into three phases, as shown in Figure 3.1. The first phase, the *DAG Builder* phase, involves processing the DSL code to create a DAG representation, where nodes represent computations and edges represent producer-consumer relationships. This DAG is stored internally and used by the other two phases. The compiler then performs a range inference pass over the DAG to calculate the image dimensions at each node, which is necessary to generate the proper control words for each node and ensure correct hardware execution. The mapper is the second phase of the FlowPix compiler, which establishes a mapping between the DAG nodes and the compute units. The mapper has two sub-phases: clustering and latency matching. In the clustering phase, the DAG is broken into smaller sub-graphs, called clusters, that represent the maximum number of DAG nodes that a single CU can execute at once, subject to certain constraints. The formed clusters are then adjusted in the latency matching phase to account for variable latency nodes. In the final phase, the control sequence generator generates a control word sequence for each mapped node in the cluster. At this stage, the compiler has access to enough information about each DAG node, including its type, parent nodes, and CU mapping. For instance, in the case of a stencil node, the compiler has knowledge of the stencil size, stride, coefficients, and input and output image dimensions. The three phases, of the compiler work together seamlessly to transform DSL code into a control word sequence, enabling correct and efficient execution of an image processing pipeline.

$$\texttt{Ordering constraint} \rightarrow \; L(x) < L(y) \implies P(x) < P(y) \tag{3.1}$$

$$\texttt{Location constraint} \rightarrow \; P_{low}(t(x)) \leq P(x) < P_{high}(t(x)) \tag{3.2}$$

3.3.1 DAG Clustering

The clustering phase divides the DAG into clusters, which are smaller compute blocks that can be processed by a CU. The multiplicity of CUs is not considered during clustering because all image strips mapped to CUs are processed through the same cluster. The input DAG S is divided into distinct clusters S_1, S_2, and so forth up to S_n. The DAG nodes inside a cluster are then mapped to the PUs inside the PEs, assuming that the entire CU is accessible to execute the cluster. Initially, S is considered as a single cluster. The compiler then attempts to assign PUs to the nodes in S while adhering to a set of constraints. The assigned nodes form the cluster S_1. This procedure is recursively applied to the remaining DAG

$S - S_1$ to create the next cluster S_2. The process continues until S is empty and no further clusters are formed, thus achieving convergence.

To assign PUs, the DAG nodes are first sorted in topological order and segregated into different levels, denoted as L_0 through L_n. For instance, in Figure 3.4, the DAG is segmented into four levels, $L_0 = [A, B, C]$, $L_1 = [D]$, $L_2 = [E]$, and $L_3 = [F]$. The DAG levels are matched against the available PU arrays, represented as P_0 through P_m. Recall that the PU arrays within a PE are distributed across the stencil, pointwise, and upsample/downsample stages. The type of a node x, denoted as $t(x)$, determines which PE stage or PU arrays can process it. The range of permissible PU arrays to process a given node type is given by $P_{low}(t(x))$ to $P_{high}(t(x))$. For stencil and up/downsample nodes, P_{low} and P_{high} values are the same, since they are assigned to a single PU array. Only the PU array at the base of the multiply-and-accumulate tree is considered for stencil nodes.

To ensure correct mapping, the PU-to-node assignments must satisfy two constraints: the ordering constraint and the location constraint, represented by equation (3.1) and equation (3.2), respectively. The ordering constraint guarantees that DAG nodes are mapped to PU arrays in level order. If two nodes, x and y, belong to DAG levels $L(x)$ and $L(y)$ respectively, and $L(x) < L(y)$, then the node x must be assigned to a PU array, given by $P(x)$, with a lower index. This constraint ensures that data flows in the same direction inside the CU as in the DAG. The location constraint restricts the set of PU arrays to which a node of a particular type can be assigned. After determining the correct PU array, the assignment is completed by assigning the next available PU within that array. The compiler maintains an availability map of all the PUs within the PE. If a suitable PU cannot be found for a given node, the compiler considers the next available PE within the CU. If all PEs are exhausted, the compiler consolidates the current cluster, re-initializes the mapping data structures, and creates the next cluster with the remaining DAG nodes.

3.3.2 Latency Matching

Assume, we have a cluster mapped to the CU, and inside the cluster, we have three nodes: x, v, and w. The nodes v and w send data to node x, and they are mapped to the ith and j^{th} PU arrays, respectively, where $j - i > 2$. Node x is mapped to the $j + 1^{th}$ PU array inside the CU. During execution, data is directly exchanged between the PUs assigned for nodes w and x because they are in adjoining PU arrays. However, there is no direct connection between nodes v and x. To ensure the execution is correct, a buffer of size $j - i$ must be placed on the route between v and x. Unfortunately, the PE does not have a buffering capacity between PUs, so extra PUs relay the data from the ith to the jth array. The compiler creates relay nodes in the cluster and distributes them, one PU per array, across the $j - i - 2$ arrays in between.

Relaying is a method used to handle variable latency across the input edges of a mapped node. To ensure successful relaying, each PU array inside the PE has a reserved set of PUs that act as forwarding units. These forwarding units are lightweight and do not carry out any computing. The number of such PUs is equal to the length of the array. The compiler configures relay nodes based on the difference in the edge latencies across DAG nodes. An example of relaying can be seen in Figure 3.4. The node E mapped to the fourth PU array receives input from node B, located at the first PU array. Therefore, two relay nodes are configured between them. However, no relaying is required for the other node input E since the data is produced at the preceding PU array. The same applies to node D with inputs from nodes A and C.

In certain situations, data transmission across all edges within a cluster can be hindered by a lack of relay nodes. Let's examine a scenario where a PU consists of four compute levels, each with a single relay node. We observe that cluster C_1 (depicted in Figure 3.4) cannot be mapped to the PU due to an insufficient number of relay nodes. This problem arises because the second level of C_1 requires two relay nodes, which exceeds the available quantity. To tackle this issue, the compiler divides the cluster further, ensuring that smaller clusters get a sufficient number of relay nodes. In the present example, C_1 is split into two clusters: one encompassing nodes A, C, and D, and the other comprising nodes B and E. The division of the cluster involves removing nodes from C_1 to create two new clusters, namely C_1^a (the reduced cluster with adequate relay connectivity) and C_1^b (containing the deleted nodes). This process is recursively applied to C_1^b. The elimination of nodes occurs by selecting them in reverse topological order (in our example, starting with node E). All dependent nodes of the deleted node are also removed. Furthermore, any node whose output is no longer utilized by any other node within the cluster is eliminated (such as node B).

3.3.3 Control Word Generation

In the final phase, the compiler produces 32-bit control words that correspond to each PU-to-node mapping. Each mapping is achieved by a series of control words, the number of which varies based on the type of node being mapped. The compiler pre-stores the sequence of control words for each mapping type in a lookup table as templates. These templates are then given specific values during the compilation process. For instance, if a PU is being mapped to an add operation, three control words are needed. The initial control word configures the scalar unit to perform the add operation, while the other two words establish the input ports of the PU with the corresponding input source indices. These sources could be either other PUs in the case of intra-cluster dependency or the PE input vector in the case of inter-cluster dependency.

When it comes to mapping stencil nodes, the compiler generates a single control word sequence to configure the reduction tree for processing all stencils together. Information per stencil, like stencil size, stride and image dimensions are a part of the generated control words. The stencil coefficients are configured into the control words depending on the size of the stencils. The PUs in the stencil stage are pre-set to perform the multiply and accumulate operation. The PUs can be also configured with min/max operation to process min/max filters respectively. Finally, the output module of the stencil stage is configured to pass the correct output downwards to the next stage.

3.4 FlowPix Scheduler

The FlowPix scheduler, which uses the driver API, is responsible for scheduling clusters created by the compiler on the overlay during runtime. A cluster is created assuming that the entire CU memory banks are available for its execution, but during runtime, the CU memory banks are shared across all cluster executions. The scheduler generates an optimal/acceptable execution schedule that minimizes the external memory traffic between the host and FPGA by streaming the input image only once and keeping partial data created by clusters inside the CU memory banks. However, if there are not enough memory banks to meet the requirements of a schedule, the scheduler raises an exception, and the user must re-synthesize the overlay with more memory banks. To generate an execution schedule, the scheduler uses the cluster graph and per cluster control word sequence as input. The cluster graph defines the interdependence among clusters, with each cluster having one or more parents. The cluster graph is a DAG that can also be viewed as a tree, with the cluster that generates the final output serving as the root. Every cluster serves as a root of a subtree, and to make a cluster eligible for execution, all the clusters within its corresponding subtree must be processed. The scheduler tracks which compute nodes within a cluster are mapped to which memory bank using a dynamic map. Before discussing the scheduling algorithm in more detail, let's define a few terms that will be used throughout this discussion.

Definitions: The number of banks necessary to store the output of cluster M is denoted by $B_o(M)$, while the number of banks from which cluster M reads its inputs is indicated by $B_i(M)$. Considering all the clusters belonging to the subtree rooted at cluster M, the maximum value of B_i is referred to as $B_{imax}(M)$. Essentially, B_{imax} identifies the highest number of input bank requirements needed to compute all the clusters in the subtree. The bank utilization factor $U(M)$ for a cluster is given by $B_o(M) - B_{imax}(M)$. This value is an approximation for the effective number of banks used by a cluster.

Theorem 1. Given a set of clusters eligible for execution, the overall bank utilization is minimized by processing the clusters in the increasing order of their bank utilization factor.

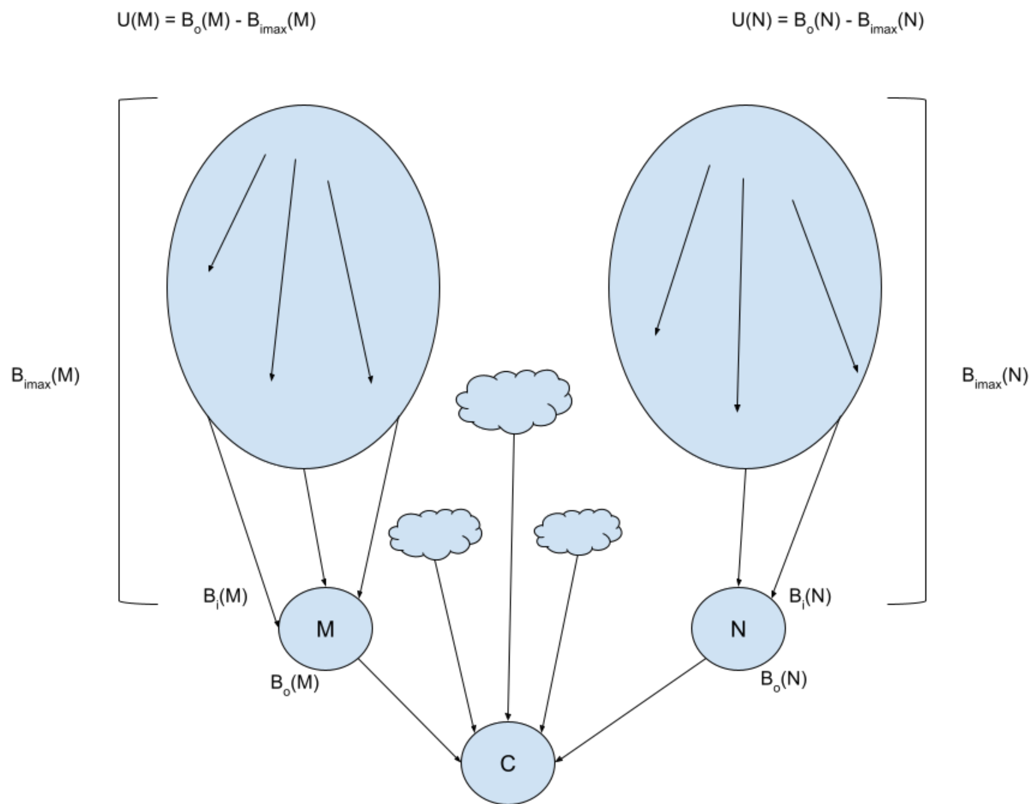


Figure 3.5 The illustration depicts a dummy DAG. We use this DAG to provide proof of the scheduling algorithm.

Proof. Assume a cluster C with two parent clusters, M and N. We can make this assumption without losing the generalization of the problem. It is necessary to compute M and N before computing C. We prove that if $U(M) > U(N)$, then the bank utilization is lesser if we compute N first followed by M.

We calculate the maximum number of banks utilized when M is computed first followed by N. $B_{imax}(M)$ is the maximum number of input banks required to compute M. The output of M is stored in $B_o(M)$ banks. Next, to compute N, $B_{imax}(N)$ banks are utilized. Therefore the maximum number of banks utilized in this compute sequence is $\max\{B_{imax}(M), B_o(M) + B_{imax}(N)\}$ which is equal to $B_o(M) + B_{imax}(N)$. This can be proved by simple substitution using our initial assumption that $U(M) > U(N)$.

$$
\begin{aligned}
& B_o(M) - B_{imax}(M) > B_o(N) - B_{imax}(N) && \text{Initial assumption} \\
\implies & B_o(M) + B_{imax}(N) > B_o(N) + B_{imax}(M) && \\
\implies & B_o(M) + B_{imax}(N) > B_{imax}(M) && \text{since } B_0(N) > 0
\end{aligned}
$$

Now, when N is computed first followed by M, the maximum number of banks in use will be $\max\{B_{imax}(N), B_o(N) + B_{imax}(M)\}$. There are two case to consider here.

1. $\mathbf{B_{imax}(N) > B_o(N) + B_{imax}(M)}$: Therefore maximum banks utilized is $B_{imax}(N)$. If executing M first leads to a lower bank utilization, then $B_o(M) + B_{imax}(N)$ must be less than $B_{imax}(N)$. This implies that $B_o(M) < 0$ which is not possible.

2. $\mathbf{B_{imax}(N) < B_o(N) + B_{imax}(M)}$: Therefore maximum banks utilized is $B_o(N)+B_{imax}(M)$. If executing M first leads to a lower bank utilization, then $B_o(M) + B_{imax}(N)$ must be less than $B_o(N) + B_{imax}(M)$. By simple rearrangement,

$$
\begin{aligned}
& B_o(M) + B_{imax}(N) < B_o(N) + B_{imax}(M) \\
\implies & B_o(M) - B_{imax}(M) < B_o(N) - B_{imax}(N)
\end{aligned}
$$

   This contradicts out initial assumption.

Therefore, we conclude that if $U(M) > U(N)$, then N must be computed first to achieve a lower overall bank utilization. □

Scheduling Algorithm: The FlowPix scheduler generates an acceptable schedule (if one exists) by consistently selecting the cluster with the lowest bank utilization factor. The algorithm starts at the root

of the cluster DAG. It recursively visits the subtrees whose root nodes have the lowest utilization factors. The generated cluster execution sequence ensures that the CU memory banks are used judiciously (see Theorem 1).

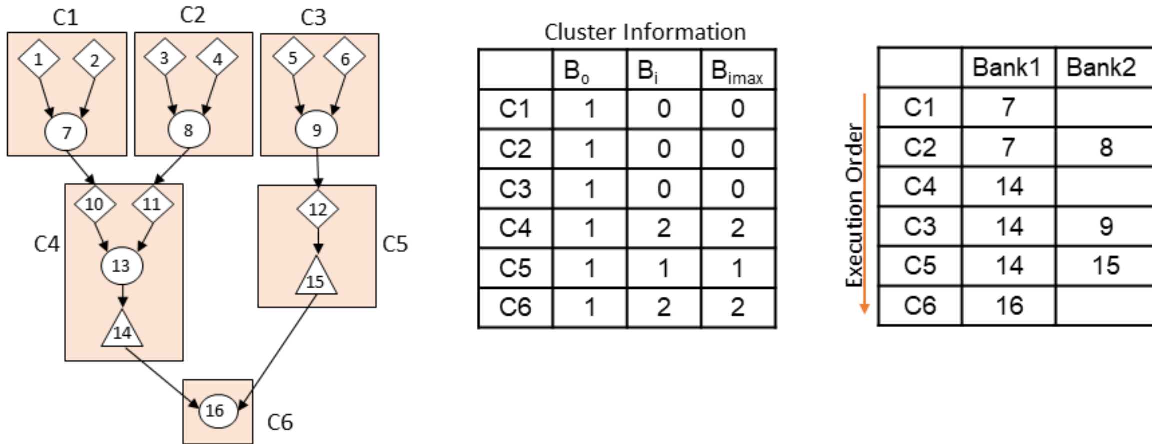
3.5 Example 1

We use a dummy benchmark, see Figure 3.6, to demonstrate the clustering and execution in FlowPix. We configure the overlay with a single CU having one PE. The PE is configured with $P_x = 6$ and $P_y = 4$. The CU can process a maximum of two 3×3 stencils nodes in parallel. The clustering step results in the creation of 6 clusters labelled S_1 through S_6. DAG nodes 1, 2 and 7 occupy the same cluster as 1 and 2 are parallel stencil nodes and can fit inside the same cluster. The pointwise node 7 depends on the output from 1 and 2. Clusters S_2 and S_3 are formed similarly. The stencil nodes 10 and 11 do not fit in clusters S_1 or S_2 as they receive input from a pointwise node, and there is no data path inside the PE to transfer data from a pointwise to a stencil node. The upsample nodes 14 and 15 belong to separate clusters because of the limitation of the PE to process a single upsample operation. The scheduler-generated cluster execution sequence minimizes the overall bank utilization. The scheduler first considers S_6 for processing as it is the root of the cluster DAG. S_6 is dependent on S_4 and S_5. Between S_4 and S_5, the scheduler picks up S_4 as it has a lower bank utilization factor ($-1 < 0$) than S_5. S_4 depends on S_1 and S_2, and both clusters have the same utilization factor. The source cluster S_1 followed by S_2 is executed first. Following this, S_4 is executed. A similar execution pattern is followed inside the subtree rooted at S_5 that results in the execution of the clusters S_3 and S_5. Finally, S_6 is processed, and the output image is stored in a single memory bank.

3.6 Example 2

We demonstrate the clustering and execution of the pyramid blending benchmark. Pyramid blending [27] is a widely used technique for blending two images using laplacian pyramids. The algorithm blends the laplacian images from each level of the pyramid corresponding to both images. We use a simple blending function that uses a blending ratio α, which determines the influence of each input image in the output. The blended images from all levels are upsampled and merged using pixel addition to create the final blended image. We use 2-level laplacian pyramids. We configure the overlay with single CU having one PE. The PE is configured with $P_x = 6$ and $P_y = 4$.

Figure 3.7 shows the DAG of the pyramid blending benchmark and the generated clusters. The four $3X3$ stencils nodes 1, 2, 3, and 4 are split across clusters 1 and 2 respectively since 1 cluster can fit 2

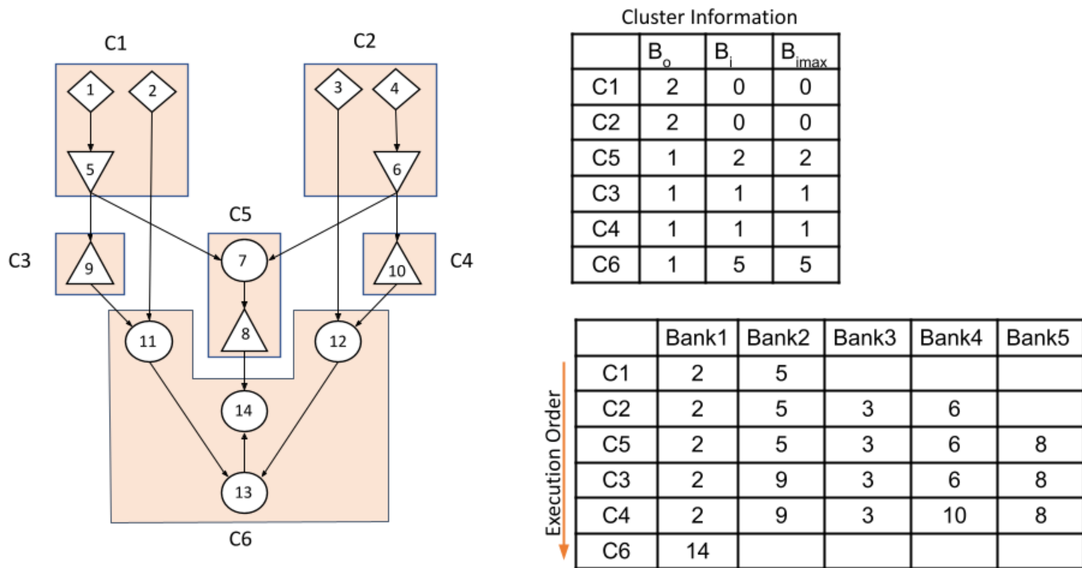


	B_o	B_i	B_{imax}
C1	1	0	0
C2	1	0	0
C3	1	0	0
C4	1	2	2
C5	1	1	1
C6	1	2	2

	Bank1	Bank2
C1	7	
C2	7	8
C4	14	
C3	14	9
C5	14	15
C6	16	

Figure 3.6 The figure depicts a dummy application with 3x3 stencil, pointwise, and upsample nodes, as well as its clustering and execution schedule. The DAG clusters are executed over a CU with two memory banks. The number on each bank corresponds to the node whose output it presently stores.

stencils parallely. The downsample node 5 depends on the output from 1 and 2. The upsample node 9 does not fit in cluster C1 as only one kind out of upsample/downsample node can reside within a cluster. The pointwise node 7 does not fit in clusters C1 or C2 as it receives input from a downsample node, and there is no data path inside the PE to transfer data from downsample node to a pointwise node. Similarly, the pointwise node 14 does not fit in cluster C5 as it receives input from an upsample node, and there is no such data path inside the PE. Following the above constraint, nodes 9 and 10 have their own separate clusters. Nodes 11, 12, 13, and 14 fit inside a single cluster as all of them are pointwise nodes with overall depth of 3, which fits inside a cluster. The scheduler-generated cluster execution sequence minimizes the overall bank utilization. The scheduler first considers C6 for processing as it is the root of the cluster DAG. C6 is dependent on C3, C4 and C5. Between all these, the scheduler pics up C5 as it has a lower bank utilization factor ($-1 < 0$) than the remaining both clusters. C5 depends on C1 and C2. Both of these have the same bank utilization factor. The source cluster C1 followed by C2 is executed first. Following this, C5 is executed. This brings the scheduler to C3 and C4 which are ready for scheduling. Having the same utilization factor, C3 followed by C4 is executed in this order. Finally, C6 is processed, and the output image is stored in a single memory bank.



	B_o	B_i	B_{imax}
C1	2	0	0
C2	2	0	0
C5	1	2	2
C3	1	1	1
C4	1	1	1
C6	1	5	5

	Bank1	Bank2	Bank3	Bank4	Bank5
C1	2	5			
C2	2	5	3	6	
C5	2	5	3	6	8
C3	2	9	3	6	8
C4	2	9	3	10	8
C6	14				

Figure 3.7 The figure on the left shows the DAG of the Pyramid Blending benchmark. The execution schedule is shown on the right. The DAG is broken into six clusters. The number of each bank corresponds to the node whose output it presently stores.

Chapter 4

Experiments and Results

To evaluate the effectiveness of our framework, we consider several standard image processing algorithms. Although these algorithms are well-known, they can vary in their implementations. We executed 15 image-processing benchmarks using Flowpix. The benchmarks encompass a range of structures and complexities, from simple pipelines such as Unsharp mask to relatively complicated pipelines such as Optical Flow with multiple input images. We also implemented iterative pyramid systems like Gaussian Pyramids and Up-Down sampling. We compared FlowPix with several existing FPGA-based image processing frameworks, including Polymage [17], DarkRoom [10], HeteroHalide [14], and Vitis-HLS [5]. These frameworks generate fixed-function hardware from a DSL specification. Our benchmark suite is described in Table 4.1. Furthermore, we compared FlowPix with IPpro [23], which employs an overlay-based approach.

4.1 Experimental Setup

The overlay is synthesized on a Virtex-7 690t FPGA with 3600 DSP blocks and 6.4 MB of on-chip BRAM. This FPGA card is connected to an Intel Core-i5 processor via a PCIe-8x link. The host driver, which streams the image frames in row-major order, is written in C++ and runs on the CPU. We use the Xillybus PCIe core (`http://xillybus.com/doc/revision-b-xl`) to connect the host input with our overlay. The Xillybus core can achieve an ideal data bandwidth of 6.4 GB/s each direction (read from and write to the host) when operating on the Virtex-7 device with 8x Gen3 lanes, utilizing the Gen3 Integrated Block for PCI Express v3.0. For details, please refer to `http://xillybus.com/doc/xillybus-bandwidth`. To match the datatype and filter sizes used in various benchmarks across different frameworks, we created four variants of the PE by altering both the datatype and the PE size. The characteristics of these PE variants are listed in Table 4.3. Although other design variants were possible, we chose these four specific ones. The number of CUs and PEs within a CU were determined

SL No	Benchmark	Acronym	Description
1	Harris Corner	HCD	Corner Detection Using 3 x 3 Stencil and Point-wise Operations.
2	Canny Edge	CED	Edge Detection Using 5 x 5 Stencil and Point-wise Operations.
3	Gaussian Filter	GAF	Application of a 3 x 3 Gaussian Filter over an Image.
4	Blur	BLU	Average operation over a 3 x 3 window of pixels.
5	Linear Blur	LB	Blur operation followed by 2 linear transformation operation.
6	Pyramid	PYR	Pyramid of Up-sampling or Down-sampling Gaussian Filters
7	Down-Up	DUS	Single Down-sampling followed by Up-sampling operation.
8	Erosion	ERS	Minimum operation over a 3 x 3 window of pixels.
9	Median	MED	Median operation over a 3 x 3 window of pixels.
10	Dilation	DIL	Maximum operation over a 3 x 3 window of pixels.
11	Convolution	CONV	A 8 x 8 convolution operation over an Image.
12	Lucas Kanade	LK	Single iteration of Lucas-Kanade Optical flow Algorithm.
13	Stencil Chain	SC	3x3 Stencil Operation repeated 3 times in a pipeline.
14	Gaussian Diff	GAD	Difference of a single and double Gaussian Blur.
15	Pyramid Blend	PYRB	Blending of two images using 2 level Image Pyramids.

Table 4.1 A short description of the benchmarks implemented using FlowPix.

based on the benchmark and framework being used. In order to minimize the comparative latency and LUT consumption, we synthesized the overlay using one of the PE variants and increased the number of CUs to match the pixel throughput. For example, when comparing FlowPix with a framework that generates 32 pixels per cycle for a given benchmark, we synthesized the overlay with a CU count of 8, considering a single PE within the CU generates 4 pixels per cycle.

4.2 Performance Analysis

Within this section, we conduct a comparative analysis of FlowPix alongside other frameworks, with a focus on their respective throughput and FPGA resource consumption across various benchmarks. Table 4.2 compares the Lines of Code of FlowPix with the other frameworks. To ensure a fair and objective comparison, we process each benchmark under the same settings as those reported by the framework in question. The configuration of each benchmark includes specifications such as image width (W), height (H), data type, and the number of pixels produced per cycle (P/C).

Framework	HCD	CED	GAF	USM	BLU	LB	SC	DIL	ERS	MED	CONV	PYR	LK	DUS
Flowpix	15	8	13	6	1	11	3	2	2	2	1	19	23	10
Halide	15	-	15	7	2	9	3	-	-	-	-	16	-	-
Polymage	43	-	-	16	-	-	-	-	-	-	-	71	75	50
Hetero Halide	26	-	8	13	2	11	15	2	2	2	-	-	-	-

Table 4.2 Lines of code comparison of the different frameworks for the benchmarks implemented.

Arch	Datatype	P_x	LUT	FF	DSP	Frequency MHz
A1	16 bit	8	9970	16984	176	200
A2	8 bit	8	5062	8673	88	250
A3	16 bit	16	13681	22111	352	200
A4	8 bit	16	5721	9971	176	250

Table 4.3 FPGA resource consumed by the different architecture variants of our overlay. P_y is set to 8 in all the PE designs.

	Input				Hetero Halide			FlowPix					Relative Performance		
	W	**H**	**Data Type**	**P/C**	**LUTs**	**FFs**	**Latency (ms)**	**LUTs**	**FFs**	**Latency (ms)**	**PEs , CUs**	**PE**	**xLUTs**	**xFF**	**xLatency**
HCD	2448	3264	UInt8	32	55198	64427	1.00	91536	159536	1.00	1,16	A4	1.66	2.48	1
GAF	2160	3840	UInt8	32	67298	41496	1.04	45768	79768	1.04	1,8	A4	0.68	1.92	1
USM	2448	3264	UInt8	32	47683	33114	3.00	91536	159536	3.00	1,16	A4	1.92	4.82	1
BLU	648	482	UInt16	16	6821	8209	0.08	39880	39880	0.10	1,4	A3	5.85	4.86	1.25
LB	768	1280	Float32	8	31049	39369	1.47	27362	44222	1.84	1,2	A3	0.88	1.12	1.25
SC	1536	2560	UInt16	16	61230	46174	0.98	109448	176888	1.23	1,8	A3	1.79	3.83	1.25
DIL	6480	4820	UInt16	32	13046	12114	3.90	13681	22111	4.88	1,4	A3	1.05	1.83	1.25
MED	6480	4820	UInt16	32	14388	10066	3.90	22884	39884	4.88	1,4	A3	1.59	3.96	1.25

Table 4.4 Comparing FlowPix with HeteroHalide.

	Input				VITIS-HLS			FlowPix					Relative Performance		
	W	H	Data Type	P/C	LUTs	FFs	Latency (ms)	LUTs	FFs	Latency (ms)	PEs, CUs	PE	xLUT	xFFs	xLatency
HCD	1080	1920	UInt8	8	13222	9330	1.7	22884	39884	2.07	1,4	A4	1.73	4.27	1.22
GAF	1080	1920	UInt8	8	2791	3641	7	5062	8673	8.28	1,1	A2	1.81	2.38	1.18
PYR-D	1920	1080	UInt8	1	1171	1238	6.99	5062	8673	8.29	1,1	A2	4.32	7.01	1.19
PYR-U	1920	1080	UInt8	1	1124	1199	27.82	5062	8673	33.14	1,1	A2	4.50	7.23	1.19
LK	3840	2160	UInt8	1	7730	11984	28.01	5062	8673	32.18	2,1	A4	1.48	1.66	1.14
CED	1080	1920	UInt8	8	6518	4899	8.5	11442	19942	2.07	1,2	A4	1.76	4.07	0.24

Table 4.5 Comparing FlowPix with VITIS-HLS.

	Input			DarkRoom+Rigel		FlowPix				Relative Performance	
	W	H	P/C	LUTs	Latency (ms)	LUTs	Latency (ms)	PEs, CUs	PE	xLUTs	xLatency
HCD	1080	1920	1	**12208**	13.83	13681	20.60	1,1	A3	1.12	1.49
CED	1080	1920	1	15696	15.10	13681	40.20	1,1	A3	0.87	2.66
LK	1080	1920	1	222000	11.91	13681	10.34	2,1	A3	0.12	0.87
CONV	1080	1920	4	20748	4.39	22884	2.50	1,4	A4	1.10	0.57
PYR-D	384	384	4	45220	0.55	22884	0.20	1,4	A4	0.51	0.36

Table 4.6 Comparing FlowPix with Darkroom and Rigel.

	Input				PolyMage			FlowPix					Relative Performance		
	W	H	Size	P/C	LUTs	FFs	Latency	LUTs	FFs	Latency	PEs, CUs	PE	xLUTs	xFFs	xLatency
HCD	1920	1080	24 bits	1	11314	5422	1.83	9970	16984	20.71	1,1	A1	0.88	3.13	11.32
USM	1920	1080	24 bits	3	6883	2500	5.85	9970	16984	10.36	1,1	A1	1.45	6.79	1.77
DUS	1920	1080	24 bits	3	6115	2352	5.39	9970	16984	10.36	1,1	A1	1.63	7.22	1.92

Table 4.7 Comparing FlowPix with PolyMage.

		FlowPix			IPPro
Micro Benchmark	**Description**	**Control Cycles**	**Compute Cycles**	**Latency** μs	**Latency** μs
CONV	A single 3x3 Gaussian filter	168	22	0.76	0.14
POLY	Degree-2 polynomial with non-zero constants	68	5	0.29	3.29.
FIR	5-tap Finite Impulse Response Filter	116	18	0.53	5.34

Table 4.8 Comparison of FlowPix with IPPro.

Table 4.4 presents a comparison between FlowPix and the hetero-halide framework across several reported benchmarks, all of which were processed at a pixel throughput greater than 1. The aim of this experiment is to assess FlowPix's ability to scale towards higher throughputs. For 8-bit cases, FlowPix's latency is comparable to that of hetero-halide, achieved with an average 1.7x increase in FPGA LUT consumption. In other benchmarks, the latency is decreased by 25%, but with an average 1.6x increase in FPGA LUT consumption. Notably, the `BLU` benchmark shows the highest increase in LUT consumption as it is computed with 16-bit precision. Conversely, the `GAF` benchmark achieves better LUT utilization than hetero-halide, given that it is computed using the 8-bit PE version.

In Table 4.5, FlowPix is compared with the industry standard Vitis-HLS library, which provides a software interface for computer vision functions accelerated on an FPGA device. Like their OpenCV equivalent, the library function (kernel) is compiled into a bitstream and runs on the FPGA. The library is optimized for area and performance. The benchmarks show an approximately 20% degradation in latency for FlowPix compared to Vitis-HLS. This difference in latency can be attributed to the fact that almost all the Vitis-HLS kernels are synthesized at a frequency greater than 300 MHz compared to FlowPix which is synthesized at a maximum frequency of 250 MHz. For the pyramid benchmarks, the relative LUTs consumption of FlowPix is over 4x that of Vitis-HLS for a single-level pyramid utilizing 5×5 filter sizes. However, since the benchmark involves just a single convolution followed by an upsample or downsample operation, the Vitis library does better at creating an area-efficient design. In the case of `LK` optical flow, pipelined parallelism is employed by setting the PE count to 2 since the LUTs increase is within the threshold. This parallelism benefits the processing of `LK` over two clusters. For `CED` edge detection, FlowPix is 4x faster than Vitis-HLS when processed using two 3×3 filters. A single PE can support four instances of the `CED` benchmark since the `A4` type PE is deployed. To achieve a $P/C = 1$, two parallel CU instances are utilized.

Table 4.6 compares FlowPix with the Darkroom and Rigel frameworks. Darkroom creates line-buffered pipelines with the stencil and pointwise computations. Rigel is an extension of Darkroom that can generate multi-rate pipelines containing upsample and downsample operations. Unlike Darkroom, Rigel pipelines can produce more than one pixel per cycle. FlowPix has a 1.5x to 2.7x higher latency for the `HCD` and `CED` benchmarks than Darkroom for single-pixel throughput. The latency increase is because these benchmarks are processed using a single PE. The downside of having a single PE is that more number of clusters are generated and are time multiplexed over the same PE. The `HCD` and `CED` benchmarks are processed using two clusters, executed one after another over the same PE. Increasing PEs was not an option here since it crosses the area threshold regarding LUT usage. In fact, for `CED`, the LUT consumption of FlowPix is 23% better than Darkroom.

FlowPix outperforms Rigel in both latency and LUT consumption. Specifically, when processing the `LK` benchmark with 2 PEs, FlowPix achieves 1.25 times better latency than Rigel because the relative LUT increment is within the set threshold of 50%. For the `CONV` and `PYR-D` benchmarks, which use 8×8 filters and produce 4 pixels per cycle, FlowPix processes them using 4 CUs containing the `A4` PE type, resulting in a latency improvement of 1.75x and 2.8x, respectively. When it comes to LUT usage, FlowPix fares better than Rigel in the `PYR-D` benchmark, utilizing 50% fewer LUTs. However, in the case of `CONV`, Rigel used 10% less LUT than FlowPix. Despite this, FlowPix still outperforms Rigel in terms of latency for both benchmarks.

Table 4.7 provides a comparison between FlowPix and the PolyMage framework. All three benchmarks are implemented on a single CU with the A1 PE type since PolyMage computes these benchmarks on 24-bit fixed-point precision. For the `HCD` benchmark, the latency with FlowPix is 11 times higher. This is because `HCD` is processed in 2 clusters over the single PE. For both `USM` and `DUS`, channel parallelism is exploited by using a single 3×3 filter in parallel, allowing all 3 channels to be processed on a single PE. While `DUS` is processed over 2 clusters due to the presence of a downsample followed by an upsample operation, the upsample is processed in the second cluster. As a result, the latency increase with FlowPix compared to PolyMage is considerably high for `DUS` but not as significant for `USM`.

After transitioning from frameworks that generate fixed-function hardware, we proceed to compare FlowPix with the IPPro instruction set-based processor. In this experiment, we selected three functions that produces a single output on a set of inputs and executed them on one CU with the A1 PE type. Our goal was to highlight the control overhead involved in processing a single operation and compare it against the processor's latency. As shown in Table 4.8, we found that the control cycles were 7x-10x higher than the compute cycles. However, this is a one-time overhead, and after the PE configuration, there will be no control cycles, and only compute cycles will be used until all the input is processed. This differs from a processor, where the pipeline stalls and control cycles are spent re-calibrating the pipeline for each instruction execution with one cycle latency. With FlowPix, we were able to process the `POLY` function and `FIR` filter at a considerably lower latency than IPPro.

4.3 Framework Analysis

In this section, we examine the capability of the Flowpix compiler to handle larger designs consisting of hundred of nodes. In this experiment, we generated DAGs with a varying number of compute nodes, ranging from 32 to 1024. To create these DAGs, we employed a random graph generator that labelled the nodes as stencil, pointwise, upsample or downsample nodes. The labeling is done using a set of constraints. For instance, only nodes with a single input are allowed to be labeled as stencil,

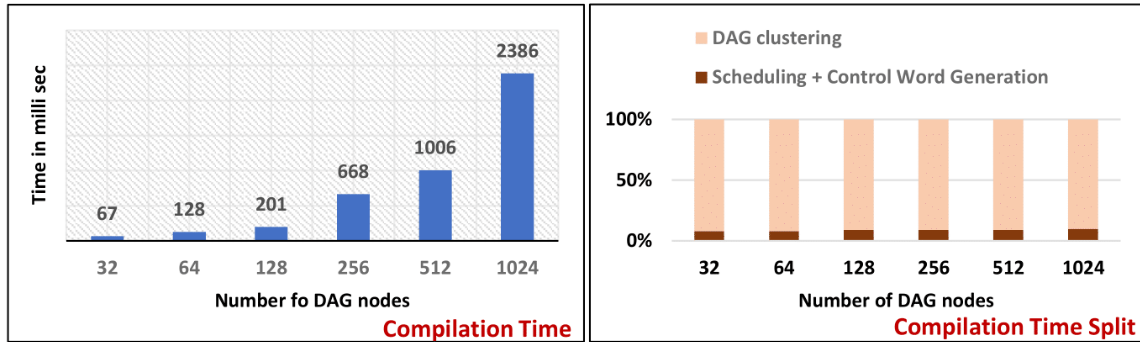


Figure 4.1 Analysis of the FlowPix Compiler.

downsample, or upsample, while nodes with two input edges could be labeled as pointwise. These constraints ensured that the generated DAGs were capable of processing an input image to produce an output image. Figure 4.1 illustrates that the compilation time increases proportionally with the size of the input DAG. The clustering phase accounts for approximately 90% of the compilation time, while the remaining 10% is spent on generating the control world and determining the scheduling order of the clusters. The clustering phase is particularly significant because as the size of the DAG increases and hence, the number of clusters that needs to be evaluated also increases.

Chapter 5

Conclusions

We described a DSL-based compiler called Flowpix for image processing applications, which achieves comparable performance to existing fixed function DSL-to-FPGA frameworks while maintaining flexibility in mapping different algorithms. We then show how the two algorithms of clustering and then scheduling the DAG enables in executing the application over the overlay.

5.1 Limitations and Future work

While initial results are promising, our work is far from done. I expect the greatest impact to come from user feedback and addressing any concerns raised.

5.1.1 Improve Clustering Algorithm

The DAG clustering algorithm described in section 3.3.1 generates clusters based on the given DAG. We will explore to further optimize the algorithm to generate the minimum number of viable clusters while also following the ordering and location constraints. Minimizing the clusters will lead to compartively reduced memory consumption in executing the algorithms over the overlay architecture.

5.1.2 Feedback from Scheduler

The Flowpix scheduler, described in section 3.4 generates an order of execution for the available clusters depending on the available memory banks. If there are not enough memory banks to meet the requirements of the schedule, the scheduler raises an exception, and asks the user to re-synthesize the overlay with more memory banks. As a future work, we will explore to provide the user with more specific information before re-synthesizing like: the minimum number of banks required to run the given clusters.

Related Publications

- Ziaul Choudhury, Anish Gulati, and Suresh Purini. 2023. ”Flowpix: Accelerating Image Processing Pipelines on an FPGA Overlay using a Domain Specific Compiler”. Submitted at the ACM Transactions on Architecture and Code Optimization, May 2023.
  [Submitted, In review]

Bibliography

[1] Vitis hls. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Vitis-HLS-Libraries-Reference.

[2] *Fifth International Workshop on Computer Architectures for Machine Perception (CAMP 2000), September 11-13, 2000, Padova, Italy*. IEEE Computer Society, 2000.

[3] A. Adams. Unsharp mask halide code. https://github.com/halide/Halide/blob/main/apps/unsharp/unsharp_generator.cpp.

[4] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. A DSL compiler for accelerating image processing pipelines on FPGAs. In *International Conference on Parallel Architectures and Compilation (PACT)*, pages 327–338, 2016.

[5] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang. Fpga hls today: Successes, challenges, and opportunities. *ACM Trans. Reconfigurable Technol. Syst.*, 15(4), aug 2022.

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE TCAD*, page 491, 2011.

[7] D. Diffendale. Isolympic inscription. https://www.flickr.com/photos/dandiffendale/6898855697/in/photostream/. License: CC By-NC-SA 2.0 https://creativecommons.org/licenses/by-nc-sa/2.0/.

[8] D. Durst, M. Feldman, D. Huff, D. Akeley, R. Daly, G. L. Bernstein, M. Patrignani, K. Fatahalian, and P. Hanrahan. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–422, 2020.

[9] Z. Guo, W. Najjar, and B. Buyukkurt. Efficient hardware code generation for fpgas. *ACM Trans. Archit. Code Optim.*, 5(1):6:1–6:26, May 2008.

[10] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[11] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Trans. Graph.*, 35(4):85:1–85:11, July 2016.

[12] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 115–127, June 2016.

[13] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987.

[14] J. Li, Y. Chi, and J. Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. pages 51–57, 02 2020.

[15] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009.

[16] M. Mehmood, E. Ayub, F. Ahmad, M. Alruwaili, Z. A. Alrowaili, S. Alanazi, M. Humayun, M. Rizwan, S. Naseem, and T. Alyas. Machine learning enabled early detection of breast cancer by structural analysis of mammograms. *Comput. Mater. Contin*, 67:641–657, 2021.

[17] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, Mar. 2015.

[18] W. A. Najjar, A. P. W. Böhm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8):63–69, 2003.

[19] H. H. Pham, L. Khoudour, A. Crouzil, P. Zegers, and S. A. Velastin. Video-based human action recognition using deep learning: a review. *arXiv preprint arXiv:2208.03775*, 2022.

[20] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Trans. Archit. Code Optim.*, 14(3):26:1–26:25, Aug. 2017.

[21] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, Dec. 2017.

[22] O. Reiche, M. A. Özkan, R. Membarth, J. Teich, and F. Hannig. Generating fpga-based image processing accelerators with hipacc. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1026–1033. IEEE, 2017.

[23] F. Siddiqui, M. Russell, B. Bardak, R. Woods, and K. Rafferty. Ippro: Fpga based image processing processor. 10 2014.

[24] F. M. Siddiqui, M. Russell, B. Bardak, R. Woods, and K. Rafferty. Ippro: Fpga based image processing processor. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2014.

[25] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

[26] C. Yuan, Z. Liu, and Y. Zhang. Uav-based forest fire detection and tracking using image processing techniques. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 639–643. IEEE, 2015.

[27] M. Zhao. Image blending using laplacian pyramids. `https://becominghuman.ai/image-blending-using-laplacian-pyramids-2f8e9982077f`.

[28] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich. Fpga-based accelerator design from a domain-specific language. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, Aug 2016.