PolyMage: Automatic Optimization for Image Processing Pipelines

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Science (Engineering)

IN COMPUTER SCIENCE AND ENGINEERING

by

Ravi Teja Mullapudi

Department of Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

JULY 2015

Acknowledgements

I thank my advisor Dr. Uday Bondhugula for his invaluable guidance. He has been very supportive and gave ample freedom to pursue whichever direction I chose. This has benefited me in enormously. I was able to explore fascinating areas which molded my research interests. He always had the time for a discussion and gently prodded me in the right direction.

I have collaborated with Roshan Dathathri and Vinay Vasista during the course of my work. Both these collaborations have been quite fruitful and enjoyable. I would like to thank them for their patience and support. The Multicore Computing Lab was a fun and nice environment to work in, people were are always open to discussing ideas and giving constructive feedback. I would like to specially thank Vinay for surviving the very initial versions of the PolyMage tool and showing great enthusiasm for the project.

Dr. Matthew Jacob's course on computer architecture has taught me a lot about reading and analyzing research work. Attending his classes was fun and enriching. I thank him for offering such a course. Dr. Albert Cohen and Dr. R Govindarajan have been very supportive of my work and warmly encouraged me to pursue my academic goals. I am very thankful for their support.

I would like to thank all the staff in the department, including Mrs. Suguna, Mrs. Lalitha, and Mrs. Meenakshi, for ensuring that my time at the department was hassle free.

I thank my friends and family both in and outside IISc for their support. Lastly, I would like to thank my wife Vidya for being the one constant in my life.

Publications based on this Thesis

1. Ravi Teja Mullapudi, Vinay Vasista, Uday Bondhugula. *PolyMage: Automatic Optimization for Image Processing Pipelines*, International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Mar 2015, Istanbul, Turkey.

Abstract

Image processing pipelines are ubiquitous. Every image captured by a camera and every image uploaded on social networks like Google+ or Facebook is processed by a pipeline. Applications in a wide range of domains like computational photography, computer vision and medical imaging use image processing pipelines. Many of these applications demand high-performance which requires effective utilization of modern architectures. Given the proliferation of camera enabled devices and social networks optimizing these emerging workloads has become important both at the data center and the embedded device scales.

An image processing pipeline can be viewed as a graph of interconnected stages which process images successively. Each stage typically performs one of point-wise, stencil, sampling, reduction or data-dependent operations on image pixels. Individual stages in a pipeline typically exhibit abundant data parallelism that can be exploited with relative ease. However, the stages also require high memory bandwidth preventing effective utilization of parallelism available on modern architectures. The traditional options are using optimized libraries like OpenCV or to optimize manually. While using libraries precludes optimization across library routines, manual optimization accounting for both parallelism and locality is very tedious.

In this thesis, we present the design and implementation of PolyMage, a domain-specific language and compiler for image processing pipelines. The focus of the system is on automatically generating high-performance implementations of image processing pipelines expressed in a high-level declarative language. We achieve such automation with:

- tiling techniques to improve parallelism and locality by introducing redundant computation,

- a model-driven fusion heuristic which enables a trade-off between locality and redundant computations, and
- an autotuner which leverages the fusion heuristic to explore a small subset of pipeline implementations and find the best performing one.

Our optimization approach primarily relies on the transformation and code generation capabilities of the polyhedral compiler framework. To the best of our knowledge, this is the first model-driven compiler for image processing pipelines that performs complex fusion, tiling, and storage optimization fully automatically. We evaluate our framework on a modern multicore system using a set of seven benchmarks which vary widely in structure and complexity. Experimental results show that the performance of pipeline implementations generated by our approach is:

- up to 1.81× better than pipeline implementations manually tuned using Halide, a state-of-the-art language and compiler for image processing pipelines,
- on average 5.39× better than pipeline implementations automatically tuned using Halide and OpenTuner, and
- on average 3.3× better than naive pipeline implementations which only exploit parallelism without optimizing for locality.

We also demonstrate that the performance of PolyMage generated code is better or comparable to implementations using OpenCV, a state-of-the-art image processing and computer vision library.

Contents

List of Tables

Keywords

Domain-specific languages, image processing, polyhedral optimization, locality, parallelism, tiling, multicores, vectorization

List of Figures

List of Algorithms

Chapter 1

Introduction

This chapter introduces image processing pipelines, describes the need for optimized implementations of these pipelines and motivates our automatic optimization approach for image processing pipelines.

1.1 Image Processing Pipelines

Cameras and other imaging sensors are ubiquitous in today's world. Applications spanning across a wide range of domains like computational photography, computer vision and medical imaging rely on processing data acquired by these sensors. For example, enhancing the capabilities of digital cameras and emerging devices like Google Glass [22], autonomous driving cars, Magnetic Resonance Imaging (MRI) and analyzing astronomical data captured by telescopes. Figure 1.1 shows two specific applications, on the left is a scan of an eye used in the diagnosis of diabetic retinopathy and the right image shows a face detection algorithm in action.

Processing and analyzing the data generated from imaging systems often demands high performance. This need is due to: (a) the sheer volume of data compounded by high resolution and frame rates, (b) increasing complexity of algorithms used to process the data, and (c) potential real-time requirements of interactive and mission-critical applications. The emergence and evolution of multicore architectures, GPUs, FPGAs, single instruction

Fundus of the Eye (c) Ignis, CC By A-SA 3.0
Face detetction (c) Beatrice Murch, CC By 3.0

Figure 1.1: On the left is a scan of an eye used in the diagnosis of diabetic retinopathy. The image on the right shows the output of a face detection algorithm.

multiple data (SIMD) instruction sets through MMX, SSE, and AVX, are examples of advances on the hardware front that have benefited the image processing domain. Coping up with the increased demand in performance requires software to effectively utilize multiple cores, SIMD parallelism and caches.

A wide range of algorithms for processing image data can be viewed as *pipelines* consisting of several interconnected processing stages. Every image captured by a camera and every image uploaded on social networks like Google+ or Facebook is processed by a pipeline. For example, Figure 1.2 shows the result of the Google+ Auto Enhance pipeline on an image and Figure 1.3 is a panorama stitched together from several camera shots. Given the proliferation of camera enabled devices and social networks optimizing these emerging workloads has become important both at the data center and the embedded device scales.

Each pipeline can be represented as a directed acyclic graph, with the stages as nodes and producer-consumer relationships between the stages as edges. Pipeline structure can vary from a few stages, with only point-wise operations, to tens of stages having a combination of point-wise, stencil, sampling and data dependent access patterns. Individual stages in a pipeline typically exhibit abundant data parallelism that can be exploited with relative ease. However, the stages also require high memory bandwidth necessitating locality optimization. Manually exploiting both parallelism and locality on modern architectures

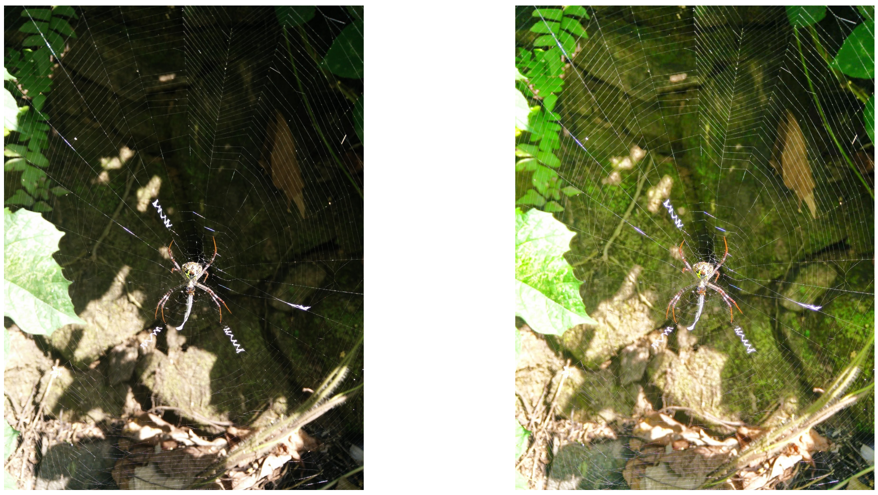


Figure 1.2: Google+ Auto Enhance

Golden Horn Panorama (c) Ggia, CC By A-SA 3.0

Figure 1.3: Panorama shot

for complex pipelines is a daunting task. However, implementations that account for both the factors can perform significantly better as illustrated by the execution times of implementations of harris corner detection show in Figure 1.5, where the hand-tuned implementation is more than 4× faster than a naive parallel implementation. Libraries such as OpenCV [39], CImg [15] and MATLAB image processing toolboxes do not completely solve the problem. They only provide tuned implementations for a limited set of algorithms on specific architectures. Even when optimized implementations of the individual stages of a pipeline are available, the inability to optimize across stages prevents effective utilization of the target architecture.

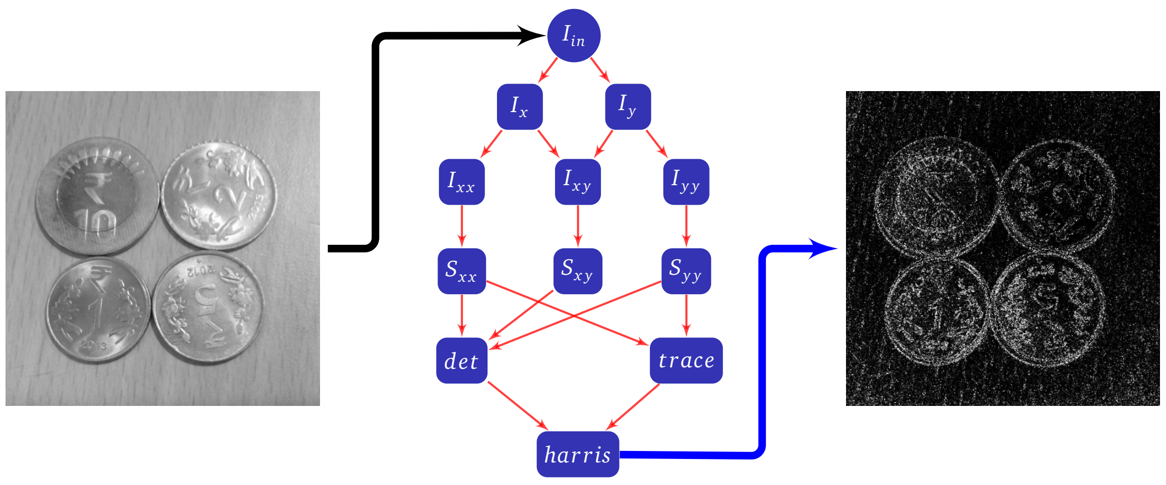
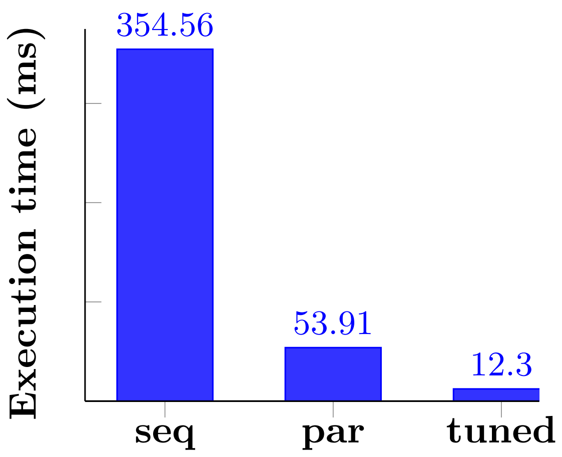


Figure 1.4: Harris corner detection

Figure 1.5: Performance of various implementations of the Harris corner detection pipeline.
seq – Naive sequential implementation in C
par – Naive parallel implementation using OpenMP and vector pragmas
tuned – Hand optimized parallel implementation accounting for locality and using vector intrinsics.



1.2 Domain-Specific Languages and Optimizers

A promising way to address the tension between ease of programming and performance is to provide a high-level domain-specific language (DSL) to express algorithms, and use an optimizing compiler to map them to a target architecture. Such an approach has been used successfully in the context of several DSLs [33, 19, 49]. For image processing, languages like CoreImage [42] and functional image synthesis [20] have focused on creating easy-to-use abstractions with minimal compiler optimization. Halide [45, 46] a recent domain-specific language and compiler for image processing pipelines focuses on both productivity and performance. However, the Halide compiler requires a schedule specification to generate an implementation. Determining an effective schedule requires manual effort and expertise, or relying on extensive and prolonged autotuning over a vast space of schedules.

1.3 Thesis Contributions and Organization

In this thesis, we describe our framework, PolyMage, comprising a DSL, an optimizer, and an autotuner, for generating high performance implementations of image processing pipelines. We first describe the input language in Chapter 2; it also serves the purpose of describing the class of image processing computations we currently handle. We then describe our automatic optimization framework, our main contribution, in Chapter 3. The key optimization techniques that we present are:

- a method for overlapped tiling tailored for heterogeneous image processing stages,
- a heuristic, modeling the trade-off between locality and redundant computation, for partitioning a pipeline into groups of stages that are later fused together with overlapping tiles,
- storage optimization and code generation for general-purpose multicores accounting for SIMD parallelism,
- and an autotuning mechanism for exploring a small parameter space resulting from our model-driven approach.

Chapter 4 details our experimental evaluation on a 16-core Intel Xeon (Sandybridge) server. We use a set of seven applications of varying structure and complexity to demonstrate the effectiveness of our approach when compared to:

- pipeline implementations manually tuned using Halide [45],
- pipeline implementations automatically tuned using Halide and OpenTuner,

- naive pipeline implementations which only exploit parallelism without optimizing for locality across image processing stages, and
- OpenCV, a widely used library for image processing and computer vision.

In cases where feasible, we also show that the schedule determined by our system when specified using Halide provides improved performance. In Chapter 5 we discuss related work, and conclusions are presented in Chapter 6.

Chapter 2

PolyMage Domain-Specific Language

In this chapter, we give the design rationale of our DSL, an overview of the language constructs, and the computation patterns that can be expressed with it.

2.1 Design Criteria

The primary goals of the language are (a) allowing expression of common patterns in image processing in an intuitive manner, and (b) enabling compiler analysis and optimization. To achieve these goals we target a common set of of patterns in image processing and design language constructs to effectively express these patterns.

Common computation patterns in image processing include point-wise operations, stencils, upsampling and downsampling, histograms, and time-iterated methods. Point-wise operations take one pixel from the input and produce an output pixel. An example of such an operation is gray scale conversion, where each pixel in the color image is converted into a gray scale pixel. Stencil operations take a small neighborhood of pixels around each pixel in the input image and use them to compute a pixel in the output image. Convolutions, correlations and image derivatives are some of the operations that follow the stencil pattern. They are used to perform tasks like edge detection [13], optical flow [34] and patch matching [6], which are in turn used in applications like object detection, tracking and segmentation.

Downsampling and Upsampling operations reduce and increase the resolution of an image respectively. These operations are used to construct image pyramids which provide a multi-scale representation of an image. These pyramids are integral in computing widely used feature descriptors like SIFT [37], SURF [8] and HOG [18]. Image pyramids are also used in several image manipulation and enhancement applications like panorama stitching and edge aware smoothing. Histograms are used to compute patch local as well as global statistics in an image. For example, histogram equalization uses the pixel intensity histogram for the entire image to adjust the contrast. The data access patterns corresponding to some of these operations are shown in Table 2.1. The Figure 2.1 illustrates examples of the operations on an image.

All these patterns can be captured by abstracting an image as a function on a multi-dimensional integer grid, i.e., it maps a multi-dimensional integer coordinate to an intensity value. Using this abstraction, new images can be constructed as expressions involving other images, thus enabling implicit expression of producer-consumer relationships that are a characteristic of image processing pipelines. Our language design is inspired by Halide [45] which provides a similar functional abstraction. However, our language design also enables powerful compiler analysis.

Instead of building a standalone language we chose to embed the language in python. This enables seamless integration with the vast collection libraries that have python interfaces (OpenCV, Numpy).

Operation	**Example**
Point-wise	$f(x,y) = g(x,y)$
Stencil	$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(x+\sigma_x, y+\sigma_y)$
Upsample	$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g((x+\sigma_x)/2, (y+\sigma_y)/2)$
Downsample	$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(2x+\sigma_x, 2y+\sigma_y)$
Histogram	$f(g(x)) += 1$
Time-iterated	$f(t,x,y) = g(f(t-1,x,y))$

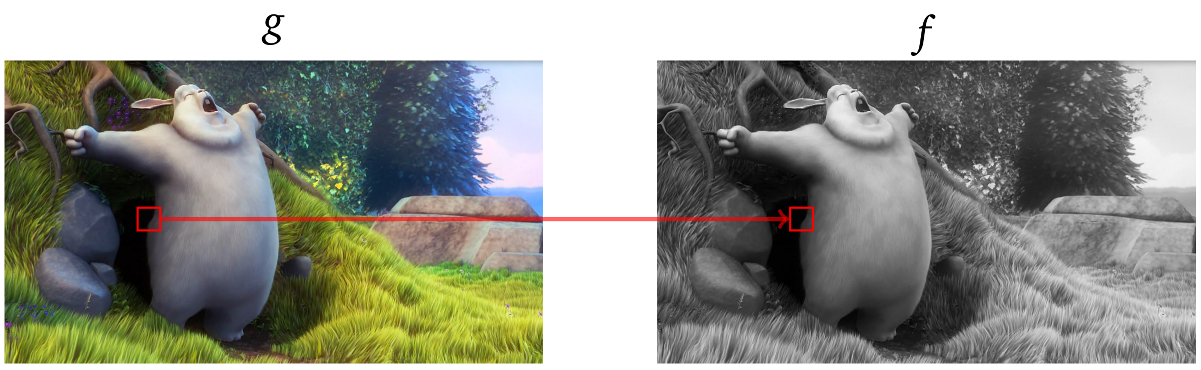
Table 2.1: Typical computation patterns in image processing

Language Construct	**Syntax**
Image	*Name* = Image(*type, list of dimensions*)
Parameter	*Name* = Parameter(*type*)
Interval	*Name* = Interval(*lower bound, upper bound, stride*)
Variable	*Name* = Variable()
Condition	*Name* = Condition(*expression, operator, expression*) where *operator* can be >, =, !=, <, >=, <=
Case	*Name* = Case(*condition, expression*)
Function	*Name* = Function(*(list of variables, list of intervals), type*) *Name*.defn = *List of cases*
Reduction	*Name* = Reduction(*(list of reduction domain variables, list of intervals), (list of variable domain variables, list of intervals), type*) *Name*.defn = Reduce(*Name(expression), expression, operation*) where *operation* can be Sum, Prod, And, Or *Name*.default = *constant* optional initialization of all values to a constant

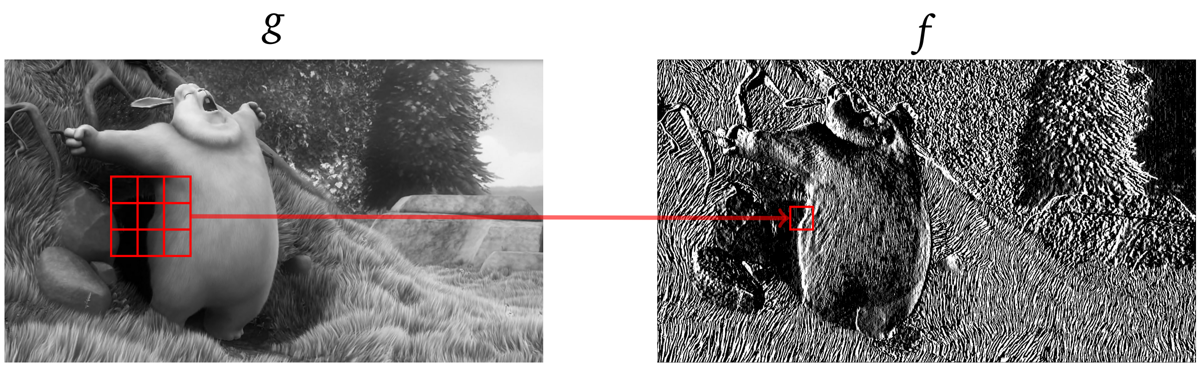
Table 2.2: Syntax of key constructs in the PolyMage DSL

2.2 Language Constructs

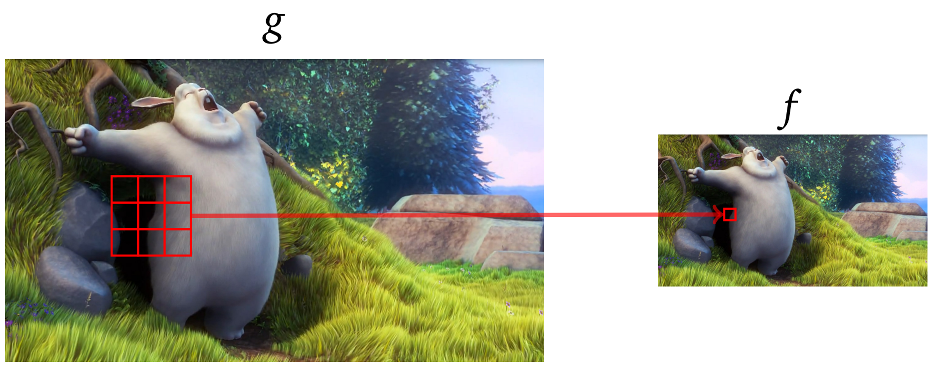
All the key constructs in the language and their syntax is listed in the table 2.2. Each of the constructs is explained in the context of two example pipelines that follow. The Harris Corner Detection [29] pipeline, description of the algorithm in the PolyMage DSL and the corresponding C pseudo code is shown in Figure 2.2. The DSL allows an user to specify the computation without needing to worry about the memory allocation, parallelism or the target architecture. To describe the pipeline in our DSL the user starts by specifying parameters, inputs and outputs. Parameters like image width, height, and other constants, which are inputs to the pipeline, can be declared using the `Parameter` construct as shown in Line 1. The input data to the pipeline is declared using the `Image` construct, as in Line 2, by specifying both its data type and its extent along each dimension. Extents are restricted to expressions involving parameters and constants. `Function` is a central construct in the



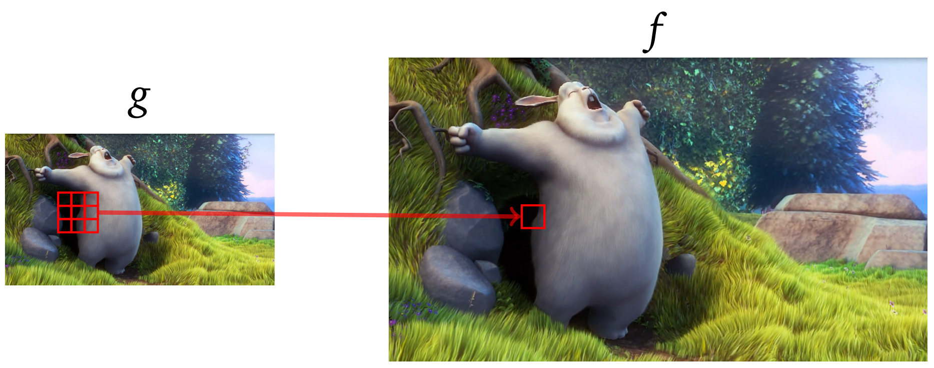
(a) Point-wise



(b) Stencil



(c) Downsample



(d) Upsample

Figure 2.1: Point-wise, Stencil, Downsampling and Upsampling operations

language, and is used to declare a function mapping a multi-dimensional integer domain to a scalar value. The domain of a function is a list of variables followed by their ranges. `Variable` is used to declare integer variables which serve as labels for function dimensions. The range of a variable is declared using the `Interval` construct. An interval is defined by a lower bound, an upper bound, and a step value. Lower and upper bounds are restricted to affine expressions involving constants and parameters. Lines 3 and 4 show how variables and intervals are created.

For a function, the expressions which define it over the domain need to be specified. A function can be defined in a piece-wise manner using a list of cases. Each `Case` construct takes a condition and an expression as arguments. Piece-wise definitions allow for expressing custom boundary conditions, interleaving, and other complex patterns. `Condition` can be used to specify constraints involving variables, function values, and parameters as shown in lines 6 and 8. Two conditions can be combined to form a disjunction or a conjunction using the operators `|` and `&` respectively. All the cases defining a function are expected to be mutually exclusive; otherwise, the function definition is considered ambiguous. The `Case` construct is optional for functions that are defined by a single expression over the entire domain. Expressions defining a function can involve its domain variables, parameters, and other function values. The `Stencil` construct is a compact way to specify a spatial filtering operation; it can also be expressed using simple arithmetic operations. Lines 31 and 34 show how the host language Python is used for meta-programming, enabling compact specification of complex pipelines. Function definitions allow referencing image values under the function being defined – this allows expression of important patterns like time-iterated computations and summed area tables [17].

The language allows expressing histograms and other reduction operations using a specialization of the function construct called `Reduction`. The reduction construct has two domains:

- a variable domain which much like the function's variable domain defines the extent of the reduction and

- a reduction domain on which the reduction is performed.

Figure 2.3 shows an reduction used to compute a histogram by counting the number of pixels of each intensity value, ranging from 0-255, in the image I.

Overall, our language allows intuitive and compact expression of image processing pipelines at the algorithm level. The number of lines of code required to specify such pipelines in our DSL is significantly less than that in an equivalent naive C/C++ implementation.

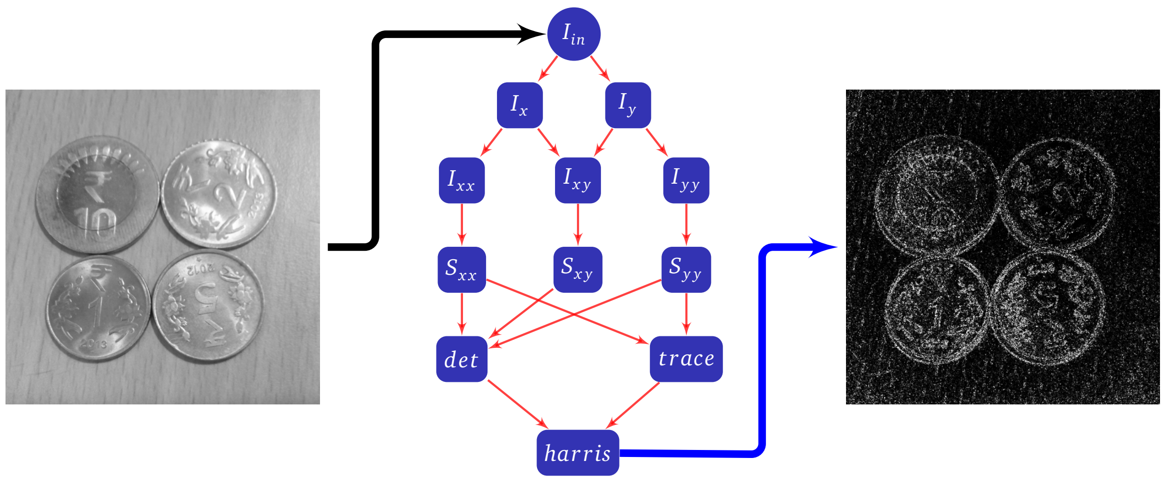


Figure 2.2: Harris corner detection

```
R, C = Parameter(Int), Parameter(Int)
I = Image(Float, [R+2, C+2])
x, y = Variable(), Variable()
row, col = Interval(0,R+1,1), Interval(0,C+1,1)

c = Condition(x,’>=’,1) & Condition(x,’<=’,R) & Condition(y,’>=’,1) &
    Condition(y,’<=’,C)
cb = Condition(x,’>=’,2) & Condition(x,’<=’,R-1) & Condition(y,’>=’,2) &
     Condition(y,’<=’,C-1)

Iy = Function(varDom = ([x,y],[row,col]),Float)
Iy.defn = [ Case(c, Stencil(I(x,y), 1.0/12, [[-1, -2, -1],
                                             [ 0,  0,  0],
                                             [ 1,  2,  1]]) ]
Ix = Function(varDom = ([x,y],[row,col]),Float)
Ix.defn = [ Case(c, Stencil(I(x,y), 1.0/12, [[-1, 0, 1],
                                             [-2, 0, 2],
                                             [-1, 0, 1]]) ]
Ixx = Function(varDom = ([x,y],[row,col]),Float)
Ixx.defn = [ Case(c, Ix(x,y) * Ix(x,y)) ]

Iyy = Function(varDom = ([x,y],[row,col]),Float)
Iyy.defn = [ Case(c, Iy(x,y) * Iy(x,y)) ]

Ixy = Function(varDom = ([x,y],[row,col]),Float)
Ixy.defn = [ Case(c, Ix(x,y) * Iy(x,y)) ]

Sxx = Function(varDom = ([x,y],[row,col]),Float)
Syy = Function(varDom = ([x,y],[row,col]),Float)
Sxy = Function(varDom = ([x,y],[row,col]),Float)
for pair in [(Sxx, Ixx), (Syy, Iyy), (Sxy, Ixy)]:
    pair[0].defn = [ Case(cb, Stencil(pair[1], 1, [[1, 1, 1],
                                                   [1, 1, 1],
                                                   [1, 1, 1]]) ]

det = Function(varDom = ([x,y],[row,col]),Float)
d =  Sxx(x,y) * Syy(x,y) - Sxy(x,y) * Sxy(x,y)
det.defn = [ Case(cb, d) ]

trace = Function(varDom = ([x,y],[row,col]),Float)
trace.defn = [ Case(cb, Sxx(x,y) + Syy(x,y)) ]

harris = Function(varDom = ([x,y],[row,col]),Float)
coarsity = det(x,y) - .04 * trace(x,y) * trace(x,y)
harris.defn = [ Case(cb, coarsity) ]
```

Figure 2.2: PolyMage specification for Harris corner detection

```
void pipe_harris(int C, int R, float* I, float*& harris)
{
   /* Allocate storage for output and intermediate stages */
   for (int x = 1; x <= R; x+=1)
      for (int y = 1; y <= C; y+=1){
            Ix[x][y] = 1.0/12*(-I[x-1][y-1] - 2*I[x-1][y] - I[x-1][y+1] +
                                 I[x+1][y-1] + 2*I[x+1][y] + I[x+1][y+1])
            Iy[x][y] = 1.0/12*(-I[x-1][y-1] - 2*I[x][y-1] - I[x+1][y-1] +
                                 I[x-1][y+1] + 2*I[x][y+1] + I[x+1][y+1])
      }
   for (int x = 1; x <= R; x+=1)
      for (int y = 1; y <= C; y+=1){
            Ixx[x][y] = Ix[x][y] * Ix[x][y]
            Iyy[x][y] = Iy[x][y] * Iy[x][y]
            Ixy[x][y] = Ix[x][y] * Iy[x][y]
      }
   for (int x = 2; x <= R-1; x+=1)
      for (xnt y = 2; y <= C-1; y+=1){
            Sxx[x][y] = Ixx[x-1][y-1] + Ixx[x-1][y] + Ixx[x-1][y+1] +
                        Ixx[x][y-1] + Ixx[x][y] + Ixx[x][y+1] +
                        Ixx[x+1][y-1] + Ixx[x+1][y] + Ixx[x+1][y+1]
            Syy[x][y] = Iyy[x-1][y-1] + Iyy[x-1][y] + Iyy[x-1][y+1] +
                        Iyy[x][y-1] + Iyy[x][y] + Iyy[x][y+1] +
                        Iyy[x+1][y-1] + Iyy[x+1][y] + Iyy[x+1][y+1]
            Sxy[x][y] = Ixy[x-1][y-1] + Ixy[x-1][y] + Ixy[x-1][y+1] +
                        Ixy[x][y-1] + Ixy[x][y] + Ixy[x][y+1] +
                        Ixy[x+1][y-1] + Ixy[x+1][y] + Ixy[x+1][y+1]
      }
   for (int x = 2; x <= R-1; x+=1)
      for (int y = 2; y <= C-1; y+=1){
            trace[x][y] = Sxx[x][y] + Syy[x][y]
            det[x][y] = Sxx[x][y] * Syy[x][y] - Sxy[x][y] * Sxy[x][y]
      }
   for (int x = 2; x <= R-1; x+=1)
      for (int y = 2; y <= C-1; y+=1)
            harris[x][y] = det[x][y] - 0.04 * trace[x][y] * trace[x][y]
   /* De-allocate storage for intermediate stages */
}
```

Figure 2.2: C pseudo code for the Harris Corner Detection pipeline

```
R, C = Parameter(Int), Parameter(Int)
I = Image(UChar, [R, C])
x, y = Variable(), Variable()

row, col = Interval(0, R, 1), Interval(0, C, 1)
bins =  Interval(0, 255, 1)
hist = Reduction(redDom = ([x,y], [row,col]),
                 varDom = ([x], [bins]), Int)
hist.defn = Reduce(hist(I(x,y)), 1, Sum)
```

Figure 2.3: Grayscale histogram

Chapter 3

Optimizing Pipeline Compiler

This chapter describes how our compiler translates pipelines specified in the PolyMage DSL into high-performance implementations. The sequence of compiler phases is shown in Figure 3.1. We first describe the front-end, which constructs a polyhedral representation of pipelines, performs static bounds checking and inlining. We then discuss the rationale behind our choice of tiling technique, which forms the core of our optimization, and describe a new approach for constructing overlapped tiles for a group of heterogeneous pipeline stages. Next, we detail the model-driven heuristic to decompose the pipeline into groups. Finally, we discuss the code generation and autotuning approach.

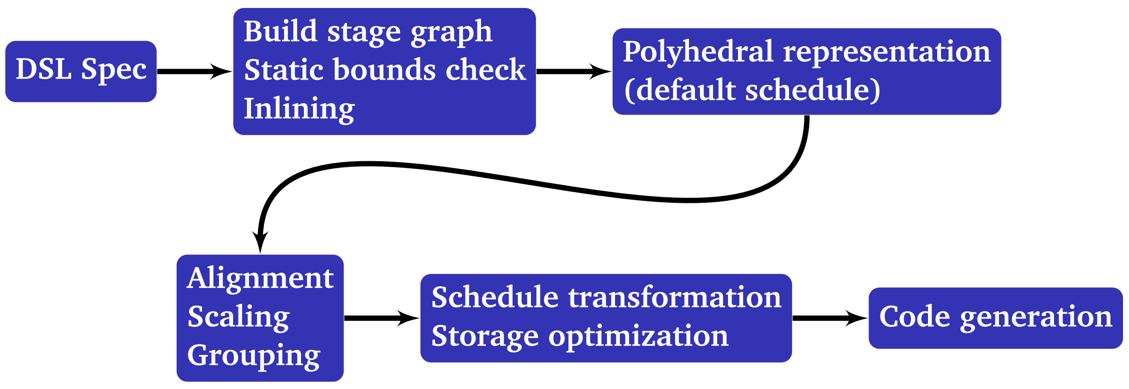


Figure 3.1: Phases of the PolyMage compiler

The PolyMage compiler takes the pipeline specification and the names of live-out functions as input. Pipelines are represented as a directed acyclic graph (DAG), where each stage (a function or an accumulator) in the user specification is mapped to a node, and the producer-consumer relations among the stages are captured by the edges between nodes. In the rest of the discussion, we use the terms function and stage interchangeably to refer to a stage in the pipeline. The *pipeline graph* is automatically extracted from the input specification; Figure 1.4 shows the pipeline graph for Harris corner detection discussed earlier. Cycles in the pipeline graph result in an invalid specification. After extracting the pipeline graph, the compiler statically checks if the values of a function used in defining other functions are within its domain. Function accesses which are affine combinations of variables and parameters are the only accesses analyzed. References to values outside the domain of a function are considered invalid and reported to the user.

Inlining substitutes producer function definitions into consumer functions. In the Harris corner detection example, the function `Ix` can be substituted into both the consumers `Ixx` and `Ixy`, resulting in `Ix` being evaluated twice. Inlining functions trades-off redundant computation for improved locality. For point-wise functions, `Ixx`, `Ixy`, `Iyy`, `det`, and `trace` in Figure 2.2, inlining is an obvious choice since it introduces minimal or no redundant computation. However, for stencil or sampling operations as consumer functions, the redundant computation introduced by inlining can be quite significant. Therefore, we restrict our inlining to cases where the consumer functions are point-wise functions, and rely on our schedule transformations to enhance locality for the other operations.

3.1 Polyhedral Representation of Pipelines

The polyhedral model is a mathematical framework well-suited to represent and transform loop nests. Image processing computations have regular dependence patterns which are amenable to polyhedral analysis. The strengths of the polyhedral model are in enabling complex transformations, precise dependence analysis, and code generation to realize the complex transformations. The PolyMage language allows a user to express pipelines naturally while capturing the essential details required to extract a polyhedral representation. A function *domain* in the language directly maps to a parametric integer set. The domain of `harris` function in Figure 2.2 is represented by the following integer set:

$$\texttt{harris}_{dom} = \{\ (x, y) \mid x \geq 2\ \wedge\ x \leq R-1\ \wedge\ y \geq 2\ \wedge\ y \leq C-1\ \}.$$

A geometric view of pipeline functions is shown in Figure 3.3. The functions f_1, f_2, and f_{out} are represented on the vertical axis, and the individual points in each function's domain are shown along the horizontal axis. We omit the bounds on the domain of a function when they are evident from the context, or not relevant to the discussion.

Schedules in the polyhedral framework can be represented as parametric relations from one integer set to another. The domain of the relation corresponds to a function domain, and the range to a multi-dimensional time stamp, whose lexicographic ordering gives a schedule for evaluating the function. The following shows a scheduling relation and the corresponding evaluation order for the `harris` function in the corner detection example:

$$\texttt{harris}_{sched} = \{(x, y) \rightarrow (y, x) \mid x \geq 2\ \wedge\ x \leq R-1\ \wedge\ y \geq 2\ \wedge\ y \leq C-1\ \}$$

```
for y in [2 ... C-1]:
    for x in [2 ... R-1]:
        harris(x, y)
```

A schedule can alternatively be described using hyperplanes, which provide better geometric intuition when dealing with tiling transformations. A hyperplane is an $n-1$ dimensional affine subspace of an n-dimensional space. It maps an n-dimensional vector, which corresponds to a point in a pipeline function's domain, to a scalar value. When viewed as a scheduling hyperplane, the scalar value is the time stamp at which the function value will be evaluated. A k-dimensional schedule is defined by k scalars, each given by the hyperplane corresponding to the dimension. Equation 3.1 describes a scheduling hyperplane for a function f. If $\vec{i_f} = (x, y)$ is a point in the function's domain, $\vec{h}$ is the normal to the hyperplane, and h_0 is the translation or the constant shift component, then

$$\phi(\vec{i_f}) = \vec{h} \cdot \vec{i_f} + h_0. \tag{3.1}$$

The scheduling hyperplanes corresponding to the relation $\texttt{harris}_{sched}$, representing a 2-dimensional schedule, are $\vec{h}_1 = [0\ 1]$ and $\vec{h}_2 = [1\ 0]$ with no translation ($h_0 = 0$).

After extracting the domain for a function, the compiler builds an initial schedule by using both the pipeline graph and the domain order in the function definition. The leading dimension of the initial schedule for a function is given by its *level* in a topological sort of the pipeline graph, and the remaining ones are given by its domain variables. The initial schedules for the functions `Ix`, `Ixx`, and `Sxx` in Figure 2.2 are as follows:

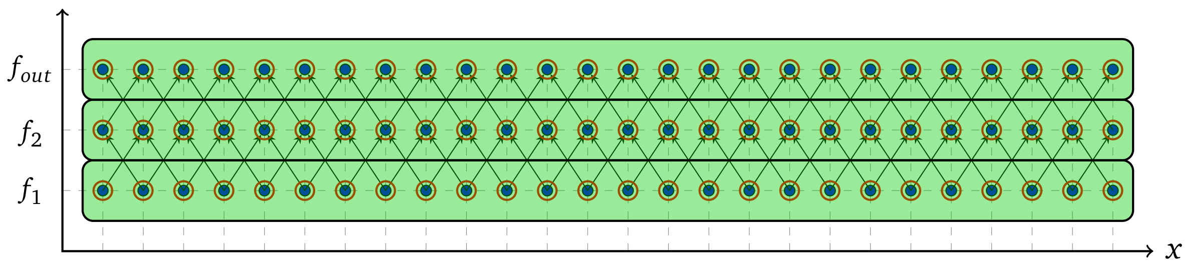
$$\begin{aligned}
\texttt{Ix}_{sched} &= \{(x,y)\rightarrow(0,x,y) \mid x\geq 1 \ \wedge \ x\leq R \ \wedge \ y\geq 1 \ \wedge \ y\leq C \ \} \\
\texttt{Ixx}_{sched} &= \{(x,y)\rightarrow(1,x,y) \mid x\geq 1 \ \wedge \ x\leq R \ \wedge \ y\geq 1 \ \wedge \ y\leq C \ \} \\
\texttt{Sxx}_{sched} &= \{(x,y)\rightarrow(2,x,y) \mid x\geq 2 \ \wedge \ x\leq R-1 \ \wedge \ y\geq 2 \ \wedge \ y\leq C-1 \ \}.
\end{aligned}$$

The compiler uses the initial schedule, which is implicit from the pipeline specification, and derives dependence information from it.

Dependences between consumer and producer functions, which are determined by analyzing the function definitions, are captured using *dependence vectors*. Initial function schedules give the time stamps at which function values are produced and consumed. The dependence vectors are computed by subtracting the time stamp at which a value is produced from the time stamp at which it is consumed. For example, the function `Sxx` at $(2,x,y)$ consumes the values of `Ixx` produced at $(1,x-1,y-1)$, $(1,x+1,y-1)$, $(1,x-1,y+1)$ and $(1,x+1,y+1)$: this is captured by the dependence vectors $(1,1,1)$, $(1,-1,1)$, $(1,1,-1)$ and $(1,-1,-1)$. Figures 3.4, 3.3 and 3.6 show functions, schedules, and corresponding dependence vectors.

3.2 Transformation Criteria

While optimizing schedules for functions in a pipeline, one needs to account for the key factors of parallelism and locality. If the function values are computed in the order given by the default schedule as shown in Figure 3.2, i.e., each function is fully computed before moving to the next. Such schedules have abundant parallelism while evaluating each function. However, they suffer from poor locality since the intermediate values might move out of faster levels of the memory hierarchy before they are used. The default naive schedule also requires storage for all the intermediate values which are used in computing the next stage of the pipeline. In the Figure 3.2 all the values of intermediate stages f_1 and f_2 need



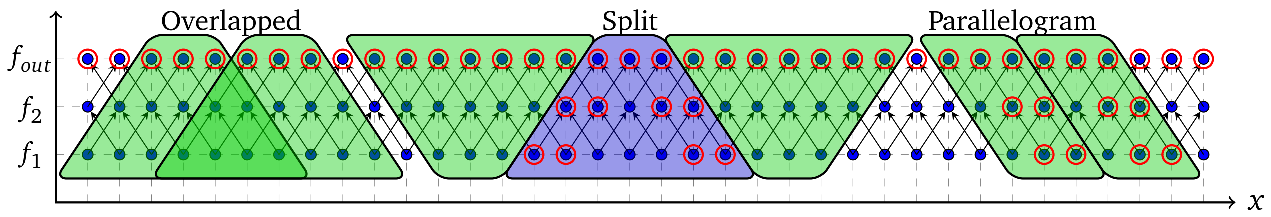
Function	Schedule	Dependence Vectors
$f_{out}(x) = f_2(x-1) \cdot f_2(x+1)$	$(x) \rightarrow (2, x)$	$(1, 1), (1, -1)$
$f_2(x) = f_1(x-1) + f_1(x+1)$	$(x) \rightarrow (1, x)$	$(1, 1), (1, -1)$
$f_1(x) = f_{in}(x)$	$(x) \rightarrow (0, x)$	

Figure 3.2: The table on the bottom shows the function definitions, dependence vectors and default schedules. The diagram on the top shows functions along the y-axis and function values along the x-axis. The dependences between the values are shown as arrows. Function values are computed in the default schedule order which required storing all the function values encircled in red.

to be stored to compute f_{out}. Reducing the storage used to implement a pipeline gives significant performance benefits which again is due to reduced working set sizes and in turn improved locality.

Parallelism and locality have been studied well in the context of time-iterated stencils, which are closely related to stencil functions in image processing pipelines. Several tiling techniques have been developed for time-iterated stencils to allow for a high degree of concurrent execution while preserving locality. Among these techniques, parallelogram [54, 10], split [31], overlapped [32, 35], diamond [5], and hexagonal [25] tiling use the polyhedral model. Figure 3.3 shows overlapped, split, and parallelogram tiling for a group of pipeline functions. Each of the tiling strategies provide different trade-offs with respect to parallelism, locality, redundant computation and ease of storage optimization.

Parallelogram tiling improves locality but only allows for wavefront parallelism, which effectively reduces to sequential execution of the tiles due to the small number of functions relative to the spatial tile size. This can be seen in Figure 3.3 where the second parallelogram tile is dependent on the first. Split tiling evaluates functions in two phases. The tiles with a larger base (upward pointing) are scheduled in the first phase, and the remaining

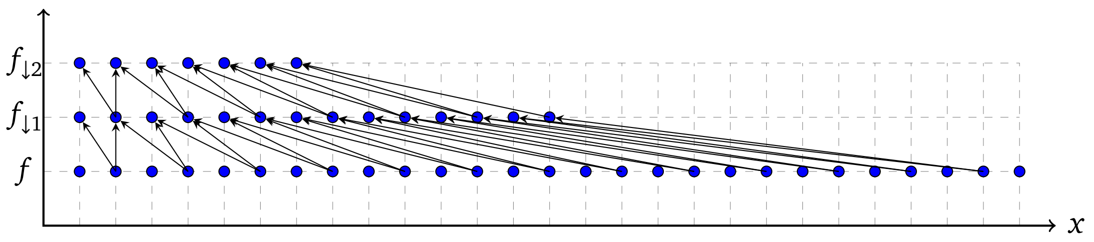


	Overlapped	Split	Parallelogram
Parallelism	√	√	×
Locality	√	√	√
Redundancy	√	×	×

Figure 3.3: Functions fused using overlapped, split and parallelogram tiling (left to right). Live-outs at tile boundaries are circled. Characteristics of the tiling techniques are shown in the table below.

ones (downward pointing) are scheduled next. All tiles in a single phase can be processed in parallel. Tiles in the second phase consume values produced at the boundaries of tiles in the first phase (encircled in the figure); hence, these values have to be kept live for consumption in the second phase. Overlapped tiling recomputes function values which are in the intersecting region of two neighboring tiles. Since the required values are recomputed within each tile, all the tiles can be executed in parallel without any communication across tile boundaries. This key difference allows for aggressive storage optimization making overlapped tiling a more suitable choice for image processing pipelines.

Tiling shown in Figure 3.3 is across several functions rather than multiple time iterations of the same function. Functions that describe complex pipelines are heterogeneous in nature as they potentially involve stencil, sampling and data dependent references to function values. Current techniques for overlapped tiling [32, 35] are designed only for time-iterated stencil dependence patterns and cannot be directly applied in our context. We now discuss the schedule transformations required to enable overlapped tiling for a group of heterogeneous functions.



Function	Schedule
$f_{\downarrow 2}(x) = f_{\downarrow 1}(2x-1) \cdot f_{\downarrow 1}(2x+1)$	$(x) \rightarrow (2, x)$
$f_{\downarrow 1}(x) = f(2x-1) \cdot f(2x+1) \cdot f(2x)$	$(x) \rightarrow (1, x)$
$f(x) = f_{in}(x)$	$(x) \rightarrow (0, x)$

Figure 3.4: The diagram shows non-stencil heterogeneous dependences across stages. The table below shows the functions and their default schedules. The dependence vectors between the stages are non-constant. Hence, the stages are not amenable to traditional tiling.

3.3 Alignment and Scaling of Functions

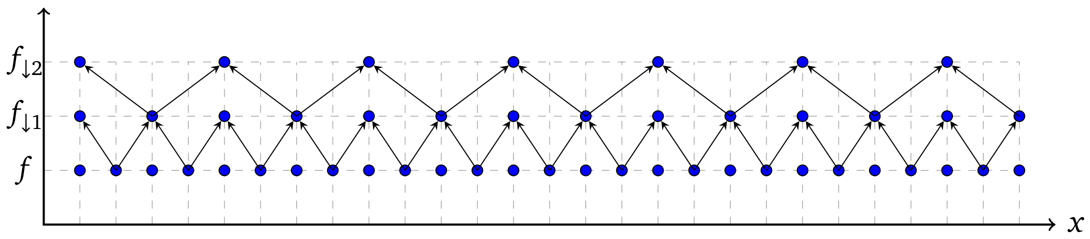
Constructing overlapped tiles for a group of functions is only possible when the dependences can be captured by constant vectors, as shown in Figures 3.3,3.5 and 3.6. In general, a group of heterogeneous functions can have different dimensions and complex access patterns, as shown in Figure 3.4. The schedules can be aligned and scaled to make the dependence vectors constant. The effect of scaling is illustrated in Figure 3.5. Consider the following example which shows a color to gray scale conversion.

```
gray(x,y) = 0.299×I(2,x,y) + 0.587×I(1,x,y) + 0.114×I(0,x,y)
```

`I` represents a color image where the first dimension corresponds to the color channel `c`, and the others to the spatial coordinates `x` and `y`. The initial schedules for `gray` and `I` are $(x, y) \rightarrow (1, x, y, 0)$ and $(c, x, y) \rightarrow (0, c, x, y)$ respectively. According to the initial schedule, the value `I(0,x,y)` required to compute `gray(x,y)` at $(1, x, y, 0)$ is produced at $(0, 0, x, y)$, resulting in the non-constant dependence vector $(1, x, y-x, -y)$. However, if the schedule for `gray` is transformed to $(x, y) \rightarrow (1, 0, x, y)$, the dependence vector becomes $(1, 0, 0, 0)$.

For the functions in Figure 3.6, value dependences are not near-neighbor like in stencils; evaluating $f_{out}(x)$ and $f_{\downarrow 1}(x)$ requires the values $f_{\uparrow}(x/2)$ and $f(2x-1)$ respectively.

Under the initial schedule of these functions, both dependences cannot be captured by constant dependence vectors. Dependences of this form are characteristic of upsampling and downsampling operations. These dependences can be made near-neighbor by scaling the function schedules appropriately, as shown in Figure 3.6. The compiler determines the schedule alignment and scaling factors for each function by analyzing the accesses to other function values in the function definitions. It may not always be possible to align and scale schedules to make the value dependence vectors constant, for instance, for the functions $f(x,y) = g(x,y) + g(y,x)$ and $f(x) = g(x/2) + g(x/4)$. Our grouping heuristic, which is presented in Section 3.5, takes the scaling and alignment factors into account while partitioning the pipeline into groups. Only functions whose schedules can be scaled and aligned to make the dependences near-neighbor are grouped together. We now present the method to construct overlapped tiles for a group of functions.

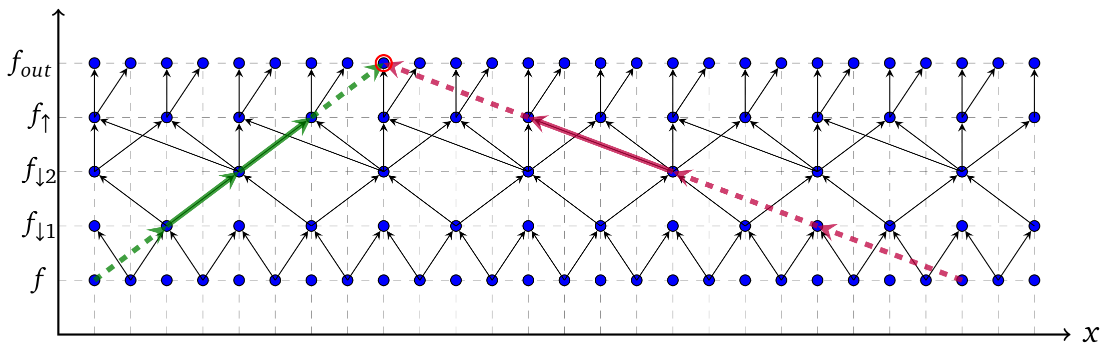


Function	Schedule
$f_{\downarrow 2}(x) = f_{\downarrow 1}(2x-1) \cdot f_{\downarrow 1}(2x+1)$	$(x) \rightarrow (2, 4x)$
$f_{\downarrow 1}(x) = f(2x-1) \cdot f(2x+1) \cdot f(2x)$	$(x) \rightarrow (1, 2x)$
$f(x) = f_{in}(x)$	$(x) \rightarrow (0, x)$

Figure 3.5: The diagram shows dependences across stages post scaling the function schedules. The table below shows the functions and their scaled schedules. The dependence vectors between the stages are constant thus enabling tiling with a fixed tile shape.

3.4 Generating Schedules for Overlapped Tiling

Tiling is only relevant for a group of functions whose dependence vectors are constant in at least one dimension after scaling and alignment transformations, as shown in Figure 3.6. The compiler constructs schedules for overlapped tiling of functions, one dimension after



Function	Schedule
$f_{out}(x) = f_{\uparrow}(x/2)$	$(x) \to (4, x)$
$f_{\uparrow}(x) = f_{\downarrow 2}(x/2) \cdot f_{\downarrow 2}(x/2 + 1)$	$(x) \to (3, 2x)$
$f_{\downarrow 2}(x) = f_{\downarrow 1}(2x - 1) \cdot f_{\downarrow 1}(2x + 1)$	$(x) \to (2, 4x)$
$f_{\downarrow 1}(x) = f(2x - 1) \cdot f(2x + 1)$	$(x) \to (1, 2x)$
$f(x) = f_{in}(x)$	$(x) \to (0, x)$

Figure 3.6: The diagram shows a group of functions with heterogeneous dependences patterns across stages. The schedules are scaled to make the dependence vectors constant. Conservative dependence analysis will result in a larger region to compute the circled live-out at the top.

another. For each dimension, the shape of an overlapped tile is determined by analyzing the dependence vectors. A naive approach to determine the tile shape is to assume that every dependence vector can exist uniformly at every point in the space. The over-approximate dependence analysis, illustrated in Figure 3.6, shows the region required to compute the live-out at the top. Such an analysis over-approximates the dependence cone increasing the redundant computation by a significant amount, as shown in Figure 3.7. The tile shape is given by the left and right bounding hyperplanes denoted by ϕ_l and ϕ_r respectively. For the tile shape to be valid, the cone formed by ϕ_l and ϕ_r from any of the live-out values should contain all the values required to compute it. Figure 3.7 shows hyperplanes which define a valid tile shape. It is desirable to minimize the redundant computation by determining the tightest *slope* possible for ϕ_l and ϕ_r. The red region in the Figure 3.7 shows the additional region that will be computed if the over-approximate dependence analysis is used.

Our compiler accounts for the heterogeneity of the functions, and determines the tile

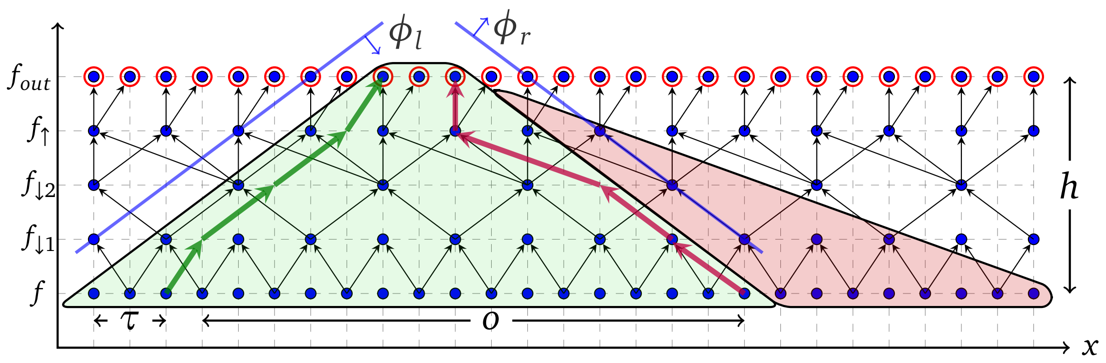


Figure 3.7: Anatomy of an overlapped tile for a group of heterogeneous functions. A tight tile shape is computed by analyzing dependence vectors between the stages. Extended region shows overlap with over-approximation. O is the amount of overlap, τ is the tile size and h is the height of the group.

shape by examining the dependence vectors between two *levels* in isolation from the other dependence vectors. *Level* refers to the level in a topological sort of the pipeline DAG formed by the functions in the group, it is also the first dimension in every function's initial schedule. To determine ϕ_l for a particular dimension, only the dependence vectors with non-negative components in that dimension are considered. Similarly, only the vectors with non-positive components are considered for ϕ_r. In Figure 3.7, the thick arrows shown at the left tile boundary are the maximum non-negative dependence vectors at each level. Similarly, the minimum non-positive vectors at each level are shown at the right tile boundary.

Algorithm 1 shows the procedure for computing the slopes of the hyperplanes which define the tile shape. The procedure starts by analyzing between the live-out function at the top and the previous level, f_{out} and $f_\uparrow$ in Figure 3.7. The initial iteration sets the slopes for ϕ_l and ϕ_r to accommodate the maximum non-negative and minimum non-positive dependence vectors, respectively, between the live-out level and the previous level. In the next iteration, the procedure moves to the next lower level and checks if the current hyperplanes contain all the dependence vectors between the current and the next level. In the case that the current slopes do not accommodate all the dependence vectors, they are adjusted to do so. This procedure is repeated till it reaches level zero, and the final slopes for ϕ_l and ϕ_r are computed.

Algorithm 1: Procedure to compute ϕ_l and ϕ_r for a group for a dimension

```
   Input : Height of the group, $H$; Group dependence vectors in the dimension being
           tiled, $D$
   /* Initially, the slopes for $\phi_l$ and $\phi_r$ are set to zero */
1  $\phi_l \leftarrow 0$, $\phi_r \leftarrow 0$
   /* Initially, left and right tile width are set to zero */
2  $w_l \leftarrow 0$, $w_r \leftarrow 0$
   /* $H$ is the level corresponding to the last stage in the group and is the live-out of
      the group */
3  for $h \leftarrow H$ to 1 do
       /* Get the dependence vectors between the current ($h$) and the previous ($h-1$)
          level */
4      $curr\_deps \leftarrow$ getDependencesAtCurrentLevel($h$, $D$)
5      $left\_extent \leftarrow 0$
6      $right\_extent \leftarrow 0$
7      for each $dep$ in $curr\_deps$ do
           /* Since we consider a single dimension at a time, the dependences vectors
              are reduced to positive or negative scalars */
8          $left\_extent = \max(dep, left\_extent)$
9          $right\_extent = \min(dep, right\_extent)$
10     $w_l \leftarrow w_l + abs(left\_extent)$
11     $w_r \leftarrow w_r + abs(right\_extent)$
       /* Check if the cone formed by the left and right hyperplanes encompasses all
          the dependences. If not adjust the hyperplanes to do so */
12     $\phi_l \leftarrow$ checkAndAdjustLeftHyperplane($h$, $\phi_l$, $w_l$)
13     $\phi_r \leftarrow$ checkAndAdjustRightHyperplane($h$, $\phi_r$, $w_r$)
14 return $(\phi_l, \phi_r)$
```

In each iteration, the slope computation algorithm ensures that all values requires to compute the live-out at the top are within the cone formed by ϕ_l and ϕ_r. By maintaining this invariant the algorithm ensures the correctness of the tiling transformation irrespective of the tile origin ,i.e, the left corner of the tile shown in Figure 3.7. It might be possible to find a tighter tile shape by constraining the tile origin. However, the improvements by doing such a restriction can be marginal over the current method and will complicate the code generation step. The traditional tiling conditions only allow dependence vectors to go out from the tiles. For overlapped tiles constructed using our method dependence vectors can go inwards into a tile. However, these values are not used within a tile for computing live-outs and do not effect correctness.

Once ϕ_l and ϕ_r are determined, the overlapped tile schedule for each function f_k in the group is constructed as follows. Let the scaled and aligned schedule for a function f_k in the group be $(\vec{i_k}) \rightarrow (\vec{s_k})$. For a tile, let h be the tile height which is one less than number of levels in the group, l and r be the slopes corresponding to ϕ_l and ϕ_r respectively. The amount of overlap for a dimension, denoted by o, is given by:

$$o = h \cdot (|l| + |r|).$$

With traditional tiling where both the lower and the upper bounding faces are parallel to each other and given by a single hyperplane, ϕ, the tiling constraints [2, 55] are given by:

$$\tau \cdot T \leq \phi(\vec{s_k}) \leq \tau \cdot (T+1) - 1,$$

where T is a newly added dimension corresponding to the iterator on the tile space, and τ is the tile size. For an overlapped tile with ϕ_l and ϕ_r as its lower and upper bounding faces respectively, the constraints are now given by the conjunction:

$$\begin{aligned} \tau \cdot T \leq \phi_l(\vec{s_k}) \leq \tau \cdot (T+1) + o - 1 \ \wedge \\ \tau \cdot T \leq \phi_r(\vec{s_k}) \leq \tau \cdot (T+1) + o - 1. \end{aligned} \tag{3.2}$$

Note that o, h and τ are known at code generation time. The schedule for f_k is updated to $(\vec{i_k}) \rightarrow (T, \vec{s_k})$, and the constraints in Equation (3.2) are added to the schedule relation. Since the overlapped tile is for the entire group of stages, the schedules for all the functions in the group are modified similarly.

3.5 Grouping

The schedule transformation and tiling techniques discussed earlier are applicable to a group of functions. However, the input to the PolyMage compiler is a full pipeline which has to be partitioned into groups of functions that cab be tiled. In doing the partitioning

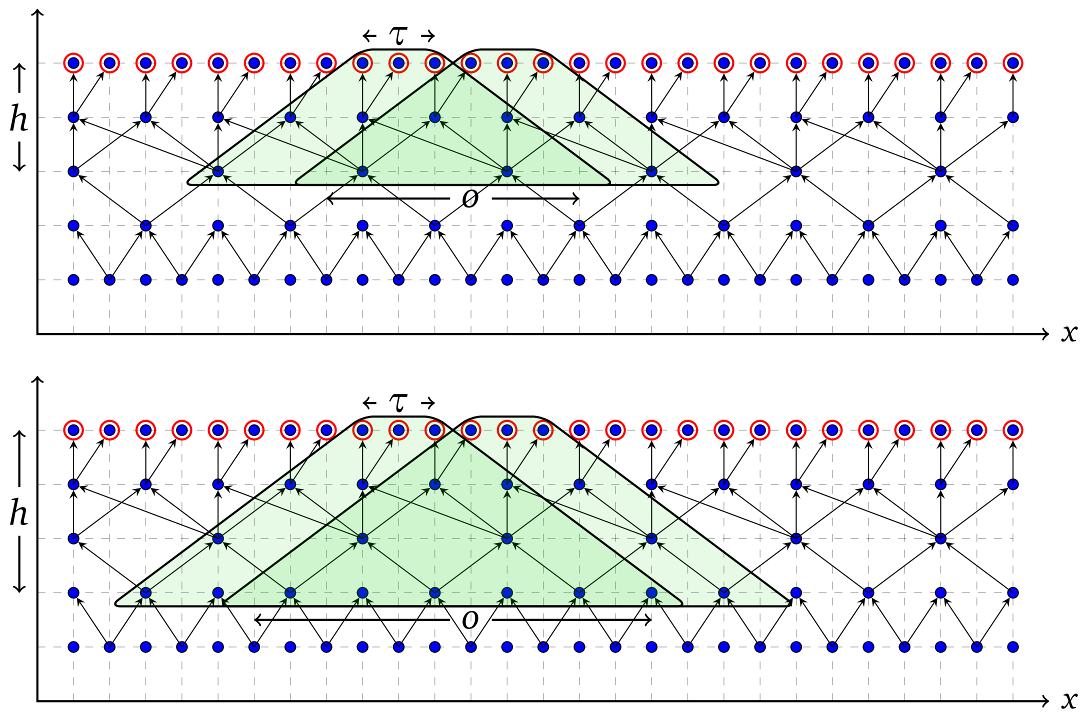


Figure 3.8: Trade-off between reuse and redundant computation. The tile on the top shows a smaller number of functions being fused. Thus, incurring lesser overlap relative to the tile at the bottom. O is the amount of overlap, τ is the tile size and h is the height of the group.

we have to account for the following factors:

- the possibility of overlap tiling a group of functions i.e., it should be possible to make the dependence vectors between functions in a group constant by scaling and aligning,
- the trade-off between redundant computation and reuse. This trade-off is illustrated in Figure 3.8 where the top half shows lesser fusion resulting in lesser overlap and reuse relative to the bottom half, and
- the fact that the number of valid grouping increases exponentially with the number of stages in the pipeline. This renders a brute force approach of enumerating all possible groupings ineffective for larger pipelines.

Algorithm 2: Iterative grouping of stages

```
Input : DAG of stages, $(S, E)$; parameter estimates, $P$; tile sizes, $T$; overlap
        threshold, $o_{thresh}$
/* Initially, each stage is in a separate group                                  */
1  $G \leftarrow \emptyset$
2  for $s \in S$ do
3      $G \leftarrow G \cup \{s\}$
4  repeat
5      $converge \leftarrow true$
       /* Find all the groups which only have one child                           */
6      $cand\_set \leftarrow$ getGroupsWithSingleChild($G$, $E$)
       /* Sort the groups by size to give priority larger groups                  */
7      $ord\_list \leftarrow$ sortGroupsBySize($cand\_set$, $P$)
8      for each $g$ in $ord\_list$ do
9          $child =$ getChildGroup($g$, $E$)
           /* Check if the stages group $g$ and the $child$ group can be scaled and aligned
              to have constant dependence vectors                                 */
10         if hasConstantDependenceVectors($g$, $child$) then
               /* Get the ratio of overlap area to tile area                      */
11             $o_r \leftarrow$ estimateRelativeOverlap($g$, $child$, $T$)
12             if $o_r < o_{thresh}$ then
13                 $merge \leftarrow g \cup child$
14                 $G \leftarrow G - g - child$
15                 $G \leftarrow G \cup merge$
16                 $converge \leftarrow false$
17                 break
18 until $converge = true$
19 return $G$
```

Our algorithm for grouping stages is shown as Algorithm 2. The algorithm takes the directed acyclic graph, (S, E), where S is the set of stages or functions and E, the set of edges, the tile sizes, an overlap threshold and the approximate estimates of all pipeline parameters as input. Typically, the user has an idea of the range of image dimensions on which the processing algorithm will be applied. The generated pipeline is optimized for the parameter values around the estimates. However, the implementation is valid for all parameter sizes. Generating an optimized implementation for all possible parameter values is often not feasible. The grouping algorithm uses the estimates to avoid considering functions of very small size for merging.

Initially, each function in the pipeline is placed in a separate group. At any point, the groups in G are disjoint i.e., a stage cannot be in two groups at the same time. The union of all the groups is the set of all stages, S. The grouping algorithm iteratively merges groups until no further merging is possible. For every iteration, it finds all groups that have only a single child or successor group, with respect to the pipeline graph (line 6). These candidate groups are sorted (line 7) in the decreasing order of their sizes determined from the parameter estimates. Next, the algorithm iterates over the sorted groups to check for merging opportunities. An iteration finishes when either a child is merged or there is no opportunity to merge, in which case the algorithm terminates.

Lines 10, 11, and 12 show the criteria used for a profitable merge. The first criterion is that it should be possible to make the dependence vector components constant by aligning and scaling the functions in child and parent groups. Otherwise, overlapped tiling cannot be performed on the group and merging the groups is not desirable. The second criterion is the amount of redundant computation that would be introduced. The algorithm merges groups only when the size of overlapping region, as a fraction of the tile size, is less than the overlap threshold. The size of the overlapping region along a dimension is independent of the tile size along that dimension, and is determined only by the slopes of the bounding hyperplanes and the group size. Recall that the slope itself was determined by dependences among functions in the group, as shown in Figure 3.7. The tile size in effect restricts group sizes. This is exploited by our auto-tuner to explore a range of implementations with a very small parameter space, as described later in Section 3.8.

Algorithm 2 is greedy, but fast and effective. A characteristic of the algorithm is that it groups maximally in a greedy fashion subject to the constraints on redundant computation, scaling and alignment. The pipeline graphs we consider have a single sink node (stage), which is the final output of the pipeline. Therefore, there exists at least one node with a single child group, i.e., the parent(s) of the sink node. When the sink node and its parents are grouped together, they form the new sink node. Now, the new sink node will be the only child of it's parents. By repeatedly merging the sink node with its parents, the entire DAG can be grouped together if the overlap and alignment criteria permit.

Hence, the algorithm, (1) *tends* to maximize reuse as it only groups stages connected by producer-consumer relationships and (2) prevents merging of groups *only* when the overlap threshold and alignment criteria do not permit such a merge. Once the groups are formed, overlapped tiling for each group is performed as described in Section 3.4. The algorithm is not provably optimal in minimizing the number of groups or maximizing any pre-defined notion of reuse. However, our experiments demonstrate that it is effective in practice when combined with auto-tuning in a restricted space. As an example, the grouping obtained for the Pyramid Blending pipeline is shown in Figure 3.9.

Validity A grouping is valid if there is no cycle between the groups with reference to the pipeline DAG. Since algorithm 2 merges a group with its single child into itself, it does not create cycles.

Termination Every iteration of the repeat-until loop that does not lead to termination, reduces the cardinality of G by one. The algorithm thus terminates in $|S|-1$ iterations in the worst case.

3.6 Storage Mapping

Functions that are outputs of the pipeline need to be stored in memory after they are computed, and we allocate arrays to store values of both output and intermediate functions. Array layout for the output functions is dictated by the domain order in their definitions and cannot be altered. However, the data layout for intermediate functions closely follows the schedule transformations applied to them, thus considering them in an integrated manner. For example, a function $f(x,y)$ whose schedule is given by $(x,y) \to (y,x)$ will be stored in a 2-dimensional array with y and x as the outer and inner dimensions respectively.

For a group of functions that are tiled, the values of the intermediate functions are used only within the tile. This can be seen in Figure 3.10, in the case of intermediate functions f, $f_{\downarrow 1}$, $f_{\downarrow 2}$ and $f_{\uparrow}$. These intermediate values can be discarded after computing the live-out values at the top of a tile. Therefore, the intermediate functions need not be allocated as

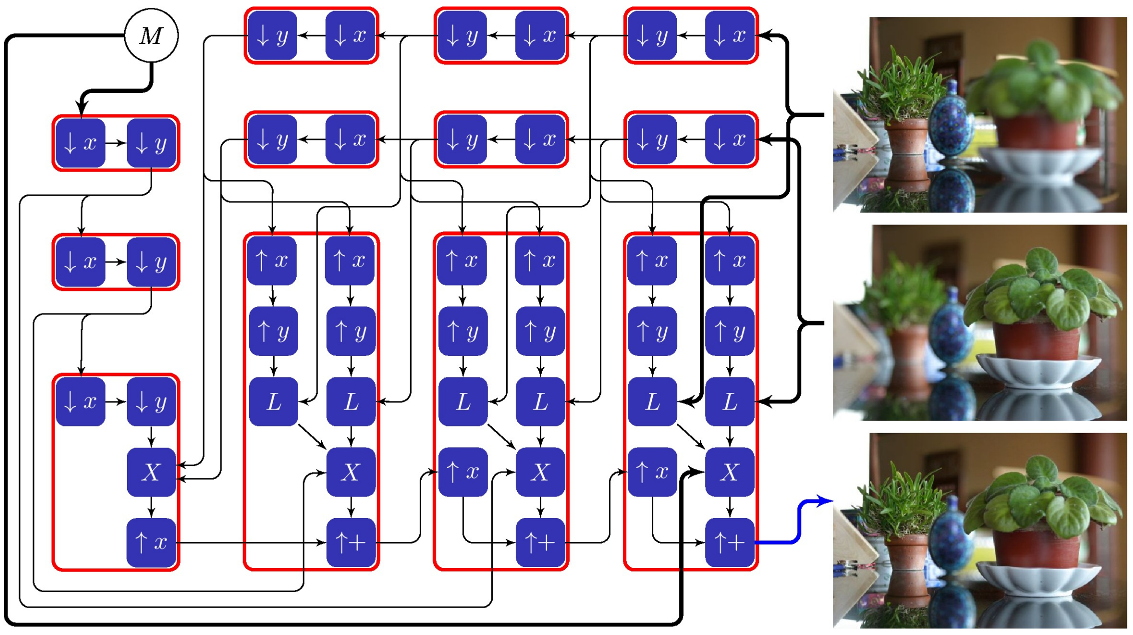


Figure 3.9: Pyramid Blending pipeline with four pyramid levels. The grouping generated by our compiler is shown by the enclosing boxes, all the stages in one group are enclosed by a box. Inputs to the pipeline are the two images on the top right, each with one of the halves out of focus, and a mask image *M*. The image on the bottom right is the blended output where both halves of the image are in focus. (Image courtesy Kyros Kutulakos)

full buffers, instead they can be stored in small scratchpads which are private to each tile. Horizontal boxes in Figure 3.10 indicate such scratchpad allocations. All the tiles which are executed sequentially by a single thread can reuse the same set of scratchpads. The only full allocations required are for the live-out functions in a group.

For tile sizes which are small relative to the size of the functions, the reduction in storage is quite significant, leading to better locality. For example, consider the storage requirement of the unoptimized C code for the Harris Corner Detection shown in Chapter 2, Figure 2.2 versus the optimized code generated by PolyMage in Figure 3.11. The naive version after inlining the Ixx, Iyy, Ixy, det and trace stages takes five $(R+2)*(C+2)$ size floating point buffers to store the intermediate results for Ix, Iy, Sxx, Syy and Sxy. For an image of size $2048*2048$, the intermediate buffers require approximately 16MB of storage. On the other had the optimized code only requires 187KB for scratchpad allocations when run on a single thread.

In order to perform scratchpad allocation for intermediate functions, their accesses have

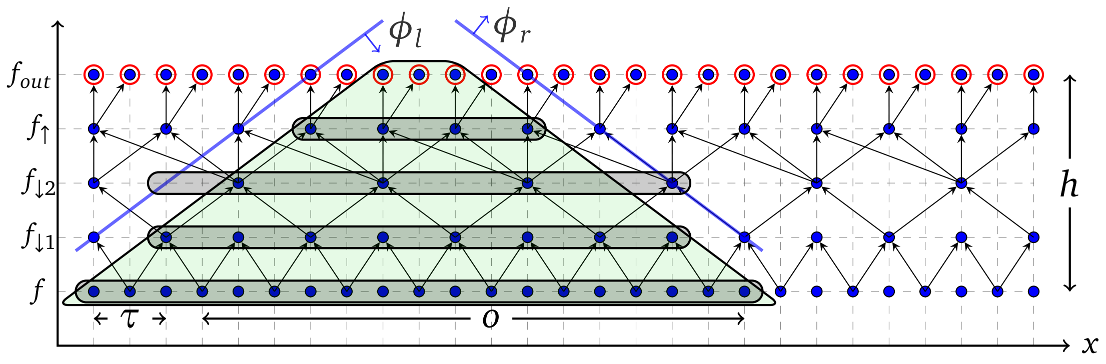


Figure 3.10: The horizontal boxes within the tile show scratchpad allocations used to compute the live-outs in the tile.

to be remapped to the scratchpads. The compiler generates index expressions into scratchpads relative to the origin of each tile: in Figure 3.10, the origin is the left bottom corner of the tile shown. Relative indexing generates simple indexing expressions for scratchpads allowing for easier code generation and vectorization. The generated code shown in Figure 3.11 shows the indexing expressions for scratchpad allocations. Without storage reduction, the tiling transformations are not very effective due to the streaming nature of image processing pipelines. The reduction of memory footprint coupled with a schedule optimized for parallelism and locality results in a dramatic improvement in performance as we demonstrate in our experimental evaluation.

3.7 Code Generation

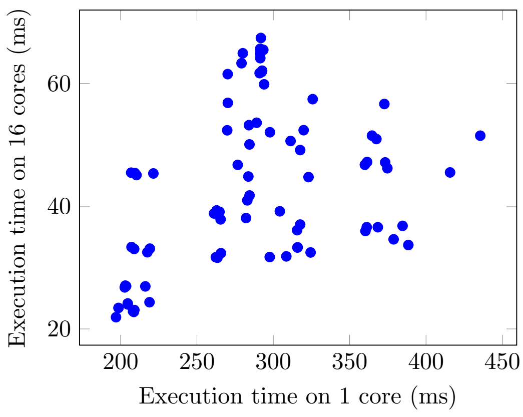
After partitioning the pipeline into groups, building overlapped tiled schedules and optimizing storage, the compiler generates a C++ function implementing the pipeline. Figure 3.11 shows the code generated for Harris corner detection specification (Figure 2.2). The integer set library (isl) [52] is used to generate loops to scan each group of functions as per the ordering implied by our schedules. The outermost parallel dimension for each group is marked parallel using OpenMP pragmas. Scratchpad allocations are placed at the start of the parallel loop's body. For the code in Figure 3.11, scratchpads are allocated in the T_i loop. Our alignment and scaling method always ensures that the innermost loop iterator has a unit stride. The compiler also avoids branching in the innermost loops by splitting function domains and unrolling loops. Unit stride loops are annotated using `ivdep` pragmas which inform the downstream C++ compiler of the absence of any vector dependences. From our experiments, we found the Intel C++ compiler's cost model for vectorization to be very effective, and we relied on it to decide which loops to vectorize and in what way.

3.8 Autotuning

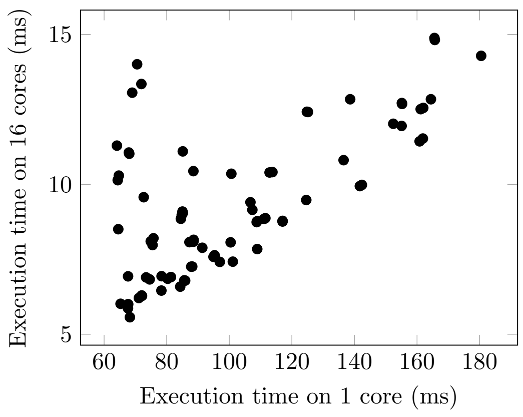
Our grouping heuristic (Section 3.5) and overlapped tiling (Section 3.4) use fixed tile sizes and overlap threshold to generate an implementation of the pipeline. It is tedious for a user to infer the right choice of parameters that lead to the best performance. Given that the solution space is narrowed down only to tile size choices, we use an autotuning mechanism to infer the right ones. The grouping heuristic we proposed takes a tile size configuration, and determines a grouping structure considering the overlap. This model-driven approach reduces the search space to one of a very tractable size. The parameter space we explore comprises seven tile sizes – 8, 16, 32, 64, 128, 256, 512, for each dimension, and three threshold values, 0.2, 0.4, 0.5, for o_{thresh}. Even for a pipeline that has four tilable dimensions, the size of the parameter space is $7^4 \times 3$ configurations. We however note that even the complex pipelines in our benchmarks have only 2 dimensions that can be tiled, and the parameter space we consider for them is thus $7^2 \times 3 = 147$. Figure 3.12 shows the single and 16-thread performance for various configurations explored by the auto-tuner for three of our benchmarks. For all the benchmarks we considered, the autotuner took under 30 minutes to explore the parameter space.

```
void pipe_harris(int C, int R, float* I, float*& harris)
{
   /* Live out allocation */
   harris = (float*) (malloc(sizeof(float)* (2+R)*(2+C)));
#pragma omp parallel for
   for (int $T_x$ = -1; $T_x$ <= R/32; $T_x$+=1){
      /* Scratchpads */
      float  Ix[36][260], Iy[36][260];
      float  Syy[36][260], Sxy[36][260], Sxx[36][260];
      for (int $T_y$ = -1; $T_y$ <= C/256; $T_y$+=1) {
          int $lb_x$ = max(1, 32*$T_x$);
          int $ub_x$ = min(R, 32*$T_x$ + 35);
          for (int x = $lb_x$; x <= $ub_x$; x+=1) {
             int $lb_y$ = max(1, 256*$T_y$);
             int $ub_y$ = min(C, 256*$T_y$ + 259);
#pragma ivdep
             for (int y = $lb_y$; y <= $ub_y$; y+=1) {
                Iy[-32*$T_x$+x][-256*$T_y$+y] = ...;
                Ix[-32*$T_x$+x][-256*$T_y$+y] = ...;
             }
          }
          $lb_x$ = max(2, 32*$T_x$ + 1);
          $ub_x$ = min(R - 1, 32*$T_x$ + 34);
          for (int x = $lb_x$; x <= $ub_x$; x+=1) {
             int $lb_y$ = max(2, 256*$T_y$ + 1);
             int $ub_y$ = min(C-1, 256*$T_y$ + 258);
#pragma ivdep
             for (int y = $lb_y$; y <= $ub_y$; y+=1) {
                Syy[-32*$T_x$+x][-256*$T_y$+y] = ...;
                Sxy[-32*$T_x$+x][-256*$T_y$+y] = ...;
                Sxx[-32*$T_x$+x][-256*$T_y$+y] = ...;
             }
          }
          if ($T_y$ >= 0 && $T_x$ >= 0) {
             $lb_x$ = 32 * $T_x$ + 2;
             $ub_x$ = min(R - 1, 32*$T_x$ + 33);
             for (int x = $lb_x$; x <= $ub_x$; x+=1) {
                int $lb_y$ = 256*$T_y$ + 2;
                int $ub_y$ = min(C-1, 256*$T_y$+257);
#pragma ivdep
                for (int y = $lb_y$; y <= $ub_y$; y+=1)
                   harris[x*(C+2)+y] = ...;
             }
          }
      }
   }
}
```

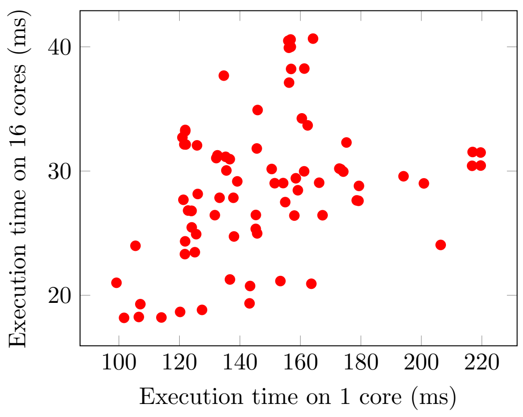
Figure 3.11: Generated code for Harris corner detection



(a) Pyramid Blending



(b) Camera Pipeline



(c) Multiscale Interpolation

Figure 3.12: Autotuning results (note: origin of the plots is not (0,0); it has been shifted for better illustration)

Chapter 4

Experimental Evaluation

This chapter describes the experimental methodology we used to evaluate the effectiveness of PolyMage relative to hand-tuned, auto-tuned and library implementations of image processing pipelines.

4.1 Setup

All experiments were conducted on an Intel Xeon E5-2680 based on the Sandybridge microarchitecture. The machine is a dual-socket NUMA with an 8-core Xeon E5 2680 processor in each socket and 64 GB of non-ECC RAM, running Linux 3.8.0-38 (64-bit). Each Xeon E5 2680 runs at 2.7 GHz and with a 32 KB L1 cache/core, 512 KB L2 cache/core, and a shared 20 MB L3 cache. The Sandybridge includes the 256-bit Advanced Vector Extensions (AVX). The experiments were conducted with hyperthreading disabled. All codes generated by PolyMage were compiled with Intel C/C++ compiler 14.0.1 with flags "-O3 -xhost". The Halide version [27] used for benchmarking uses LLVM 3.4 as its backend, and the OpenCV version used was 2.4.9. The PolyMage performance numbers were taken with 6 runs; the first warm up run was discarded, and the average of the other five is reported.

We use seven image processing application benchmarks which vary widely in structure and complexity. The number of stages and the lines of PolyMage code for each of these applications is shown in Table 4.1. We evaluate our results relative to other implementations

Benchmark	Stages	Lines	Image size	Computation Patterns
Unsharp Mask	4	16	$2048 \times 2048 \times 3$	P, ST
Bilateral Grid [14]	7	43	2560×1536	P, ST, R, SM
Harris Corner [29]	11	43	6400×6400	P, ST
Camera Pipeline	32	86	2528×1920	P, ST, SM, R
Pyramid Blending [12]	44	71	$2048 \times 2048 \times 3$	P, ST, SM
Multiscale Interpolate	49	41	$2560 \times 1536 \times 3$	P, SM
Local Laplacian [4]	99	107	$2560 \times 1536 \times 3$	P, ST, R, SM

Table 4.1: Benchmark characteristics. Columns from left to right: Application, number of pipeline stages, number of lines of PolyMage DSL code, input size, computation patterns (P - point-wise, ST - stencil, SM - sampling, R - reduction).

of the same benchmarks in the following ways.

- We compare with the highly tuned schedules available in the Halide repository for the applications evaluated by Ragan-Kelley et al. [46]. We tuned those schedules further for our target machine by varying tile sizes, vector lengths and unroll factors, and we call these *H-tuned*.
- We evaluate schedules generated by our compiler in conjunction with Halide. This is done by specifying a schedule, referred to as *H-matched*, that closely matches our best schedule for the benchmark. Coming up with a matching Halide schedule is not practically feasible for all the applications considered. It is too tedious in cases where the pipelines have a large number of stages – since schedules generated by our compiler are complex.
- We used the OpenTuner [3] framework and the associated Halide autotuner to generate schedules for all the benchmarks, by running the autotuner for 12 hours on each application.
- We also compare with OpenCV implementations for applications which could be written solely using optimized OpenCV library routines available.

An expert hand-tuned version is publicly available for *camera pipeline*, but other expert versions evaluated by Ragan-Kelley et al. [46] are either proprietary or not publicly available. In such cases, our comparison relative to *H-tuned* can be used to place PolyMage in relation to hand-tuned versions.

4.2 Benchmark Performance

Table 4.4 shows absolute execution times for the implementations generated by PolyMage that are fully optimized for 16 cores, their speedup over *H-tuned* and schedules generated by OpenTuner. Speedups on applications not from Halide's repository are marked with *. The table also provides the number of stages in each benchmark, number of lines of PolyMage DSL code, and execution times for OpenCV versions with optimized library routines. Table 4.2 shows some of the characteristics of the auto-tuned pipeline implementations ,i.e., the number of tiled groups, the size of the largest group, storage for intermediate buffers with and without the grouping and tiling transformations. Figure 4.1 shows performance comparing various configurations of PolyMage and Halide to provide insight into the benefits of grouping, tiling, and vectorization separately. The baseline is the sequential version generated by PolyMage without schedule transformations and vectorization: this is the same as *PolyMage (base)* for 1 thread.

4.2.1 Multiscale Interpolation

Multiscale Interpolation interpolates pixel values at multiple scales. The *H-tuned* schedule does loop reordering, vectorization, tiling, and parallelization but no fusion. The best schedule determined by the autotuner results in a non-trivial grouping of the pipeline stages. The grouping consists of 5 fused groups where the largest group is composed of 9 stages. This schedule outperforms the manually tuned schedule, *H-tuned*, by 2×. Specifying our best schedule using Halide (*H-matched*) completely bridged the 2× gap in performance between *H-tuned* and *PolyMage(opt+vec)*. Also, the *H-matched* schedule provided better vectorization gains when compared to the *H-tuned* one.

4.2.2 Harris Corner Detection

Harris Corner Detection [29] is a widely used method to detect interest points in an image. The feature or interest points are used in various computer vision tasks. A full description of the algorithm in our DSL is shown in Figure 2.2. The best schedule generated by PolyMage inlines all point-wise operations, and groups all stencil functions together. The speedup of the tiled and vectorized implementation is 46.78× over the baseline. An interesting point to note is that, without the tiling transformation, vectorization improves the single thread performance only by 1.12×. This shows the importance of locality transformations to effectively utilize vector parallelism. *H-tuned* schedule uses a different grouping and performs reasonably well. *H-matched* schedule uses the same grouping and inlining as our schedule, and performs much better than *H-tuned*. The performance gap between *H-matched (tuned+vec)* and *PolyMage (opt+vec)* is due to *icc* generating better vectorized code than Halide. This can be observed from the single thread vectorization speedups.

4.2.3 Pyramid Blending

Pyramid Blending [12] blends two images into one using a mask and constructing a Laplacian pyramid. The complex grouping performed by PolyMage is shown in Figure 3.9. Writing a similar schedule in Halide (*H-matched*) for such a complex grouping is a non-trivial task. The *H-tuned* schedule is provided by us along the lines of the tuned schedule available for the Local Laplacian Filter benchmark. The *H-matched* schedule provides a clear performance improvement over *H-tuned*.

4.2.4 Bilateral Grid

Bilateral Grid [14, 41] is a structure used for computing a fast approximation of the bilateral filter. The benchmark constructs a bilateral grid, and then uses it to perform edge-aware smoothing on the input image. The pipeline is a histogram operation followed by stencil and sampling operations. Our compiler fuses all the stencil and sampling stages into two groups, and the histogram into another. *H-tuned* schedule is quite different as it fuses the histogram computation with one of the stencil operations. Our current implementation does not attempt to fuse reduction operations. However, the schedule we generate is quite competitive to *H-tuned*.

4.2.5 Camera Pipeline

Camera Pipeline processes raw images captured by the camera into a color image. The camera sensor is overlaid with a color filter and the raw image captured only has a single channel data at each pixel location. Such raw sensor data is converted into a color image by interpolating the missing channels at each pixel location. As part of the processing, the pipeline also performs noise reduction, color translation and tone reproduction. The pipeline stages have stencil-like, interleaved, and data-dependent access patterns. Our best schedule fuses all stages except small lookup table computations into a single group. Performance of the PolyMage optimized code is slightly better than *H-tuned*, and matches that of an expert tuned version labeled 'FCam' [1] in Figure 4.1e.

4.2.6 Local Laplacian Filters

Local Laplacian Filters [40, 4] can be used in several applicaitons including detail enhancement and tone mapping. The benchmark we use computes a fast approximation [4] of the Local Laplacian Filters proposed in [40]. It is the most complex of our benchmarks, involving both sampling and data-dependent operations. The best schedule PolyMage generates is very complex and is tedious to manually express in Halide. We only compare with the *H-tuned* schedule which does not group any of the stages but exploits parallelism and vectorization.

4.2.7 Unsharp Mask

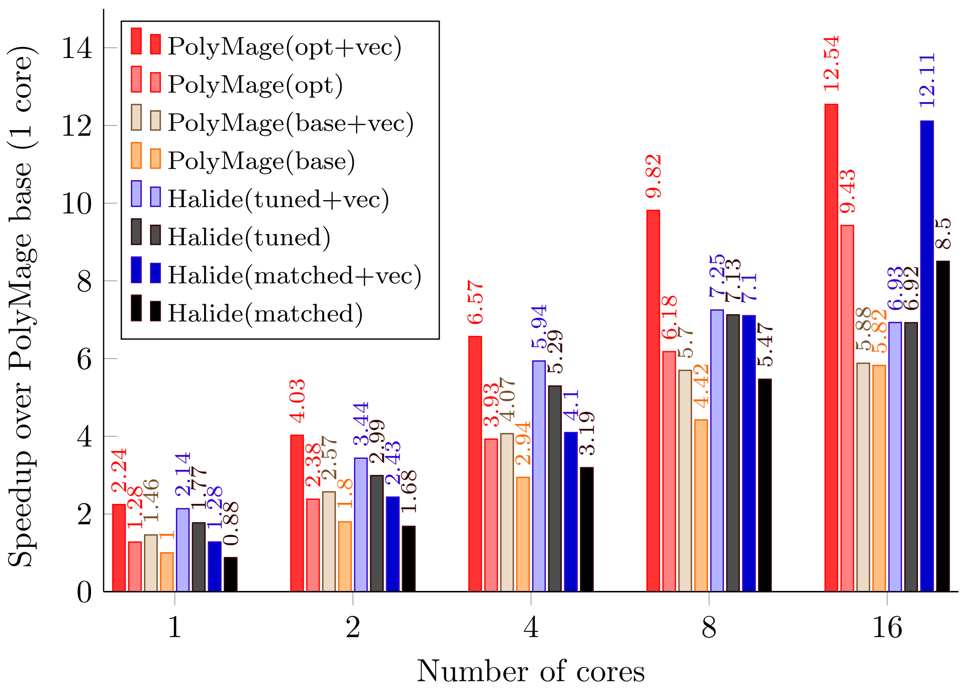
Unsharp Mask is a simple pipeline used to sharpen image edges. The pipeline separates the high frequency components like edges of an image and enhances them. The pipeline comprises a series of stencil operations. The *H-tuned* schedule we use is very similar to our best schedule. All the stages in the the pipeline are added to a single group.

4.3 Storage reduction

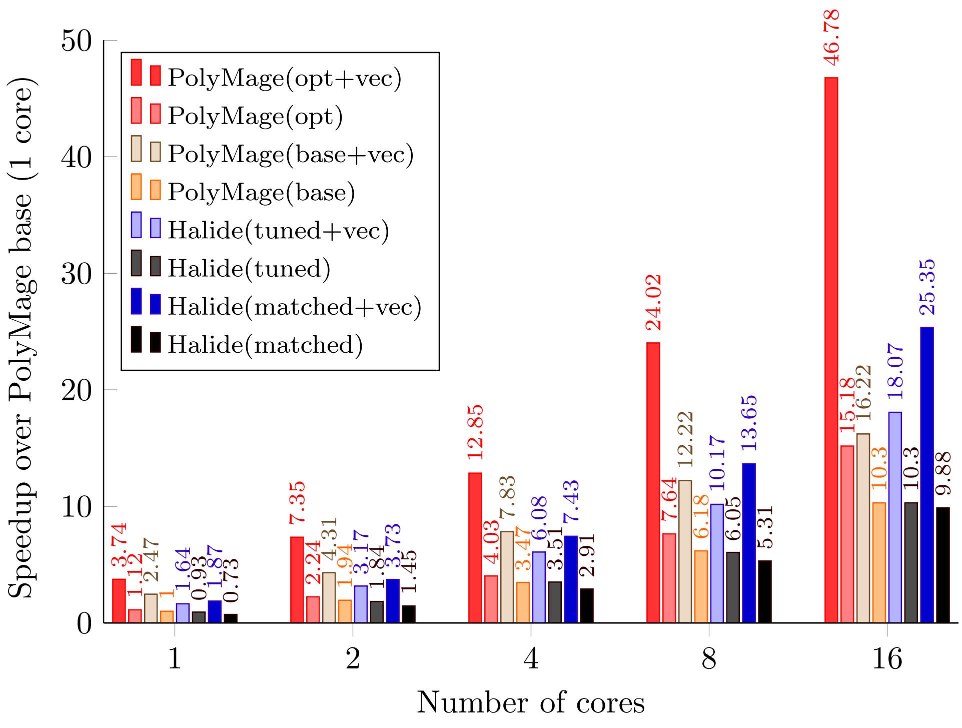
The reduction in the amount of storage by only using scratchpad allocations for intermediate function values can be seen Table 4.2. For example, in the case of the Camera Pipeline benchmark the storage is reduced from 168MB to a mere 0.67MB while executing on a single thread. Thus, enabling data to reside in faster levels of the memory hierarchy. However, the reduction in storage is not always desirable. For instance, the multi-scale applications like Local Laplacian Filters, Multiscale Interpolation and Pyramid Blending do not have large groups since the amount of redundant computation grows very quickly with the group size. Despite the gains due to improved locality and reduced storage, configurations with large group sizes end up performing worse than ones which have smaller groups. Large tile size configurations despite allowing for larger overlap sometimes do not generate enough tasks for all the threads resulting in sub-optimal performance. However, when auto-tuning finds the right balance the performance improvements can be quite dramatic as in the case of the Harris Corner Detection. The Table 4.3 gives the amount of scratchpad memory used by the largest group and the tile sizes for the best autotuned implementations.

Benchmark	Num of tiled groups	Group size (*max*)	Storage (MB) (*base*)	Storage (MB) (*opt*)
Unsharp Mask	1	4	150	0.15
Bilateral Grid [14]	2	2	55	44
Harris Corner [29]	1	11	819	0.18
Camera Pipeline	1	25	168	0.67
Pyramid Blending [12]	12	7	436	54
Multiscale Interpolate	5	9	252	85
Local Laplacian [4]	12	5	591	212

Table 4.2: Columns from left to right: Application, number of groups were fused and tiled, the number of stages in the largest group, amount of storage required to store intermediate function values in code without and with schedule transformations ,i.e., PolyMage(*base*) and PolyMage(*opt*). The intermediate storage is for input sizes shown in Table 4.1 when executed on a single thread.

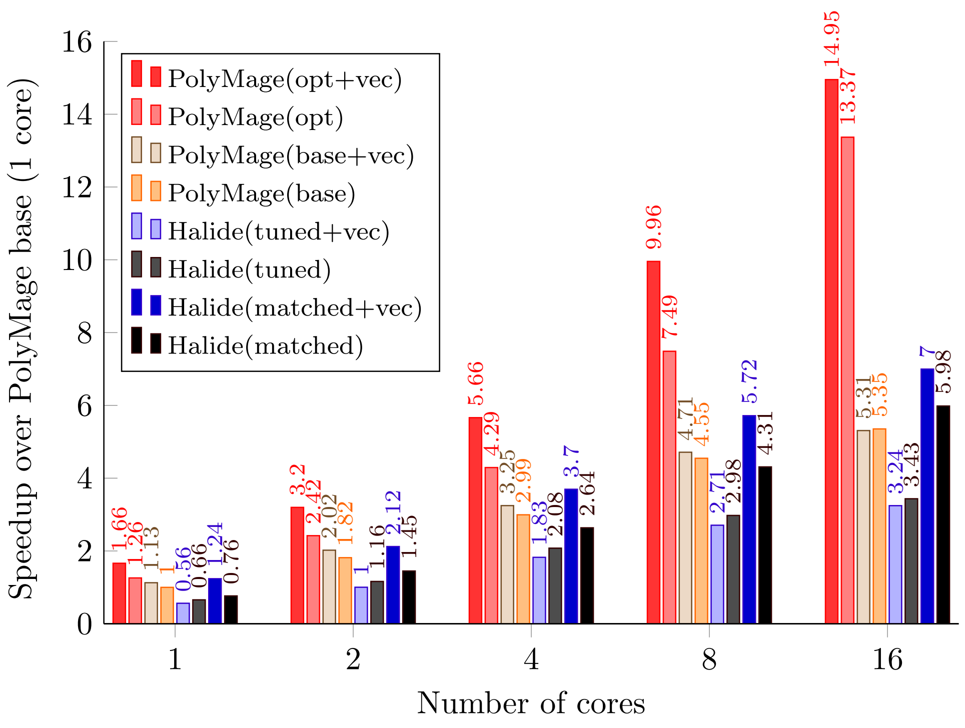


(a) Multiscale Interpolation

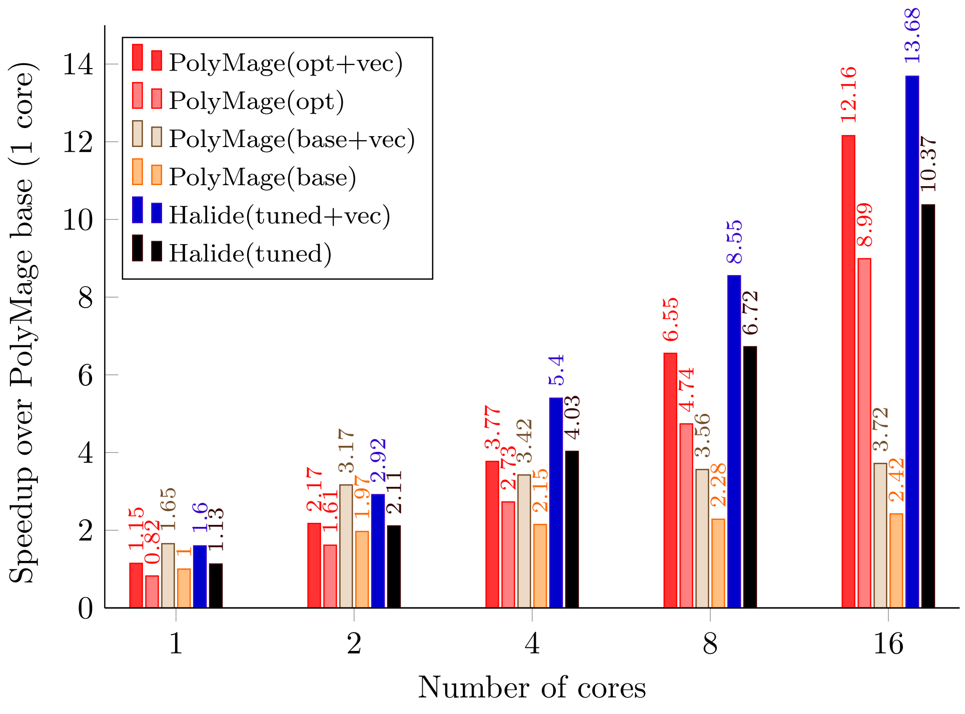


(b) Harris Corner Detection

Figure 4.1: Multiscale Interpolation and Harris Corner Detection

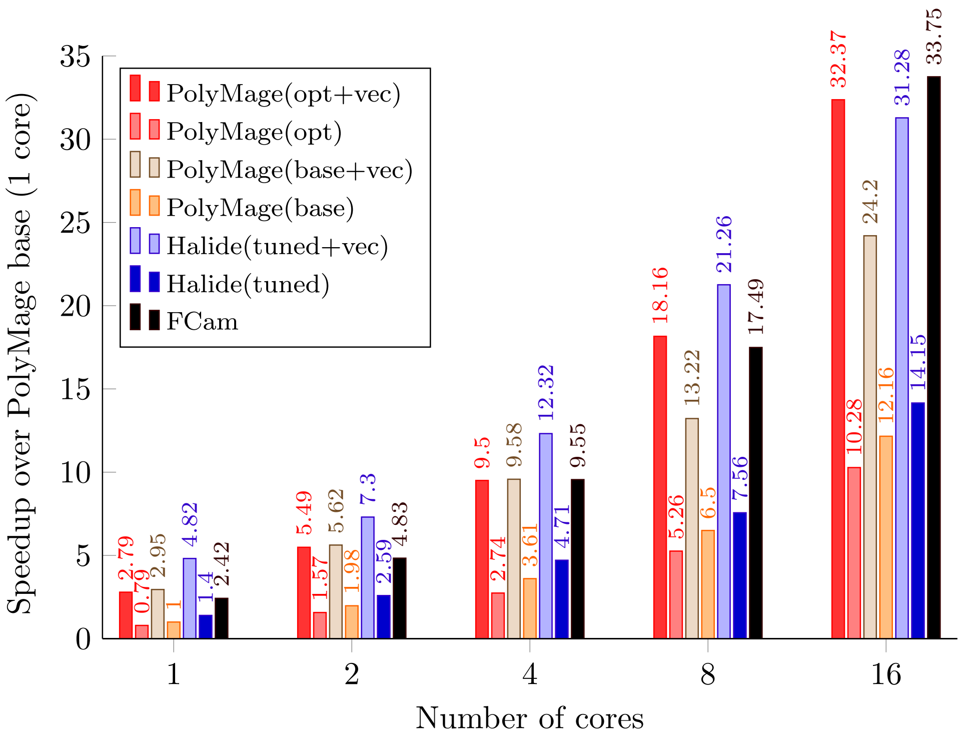


(c) Pyramid Blending

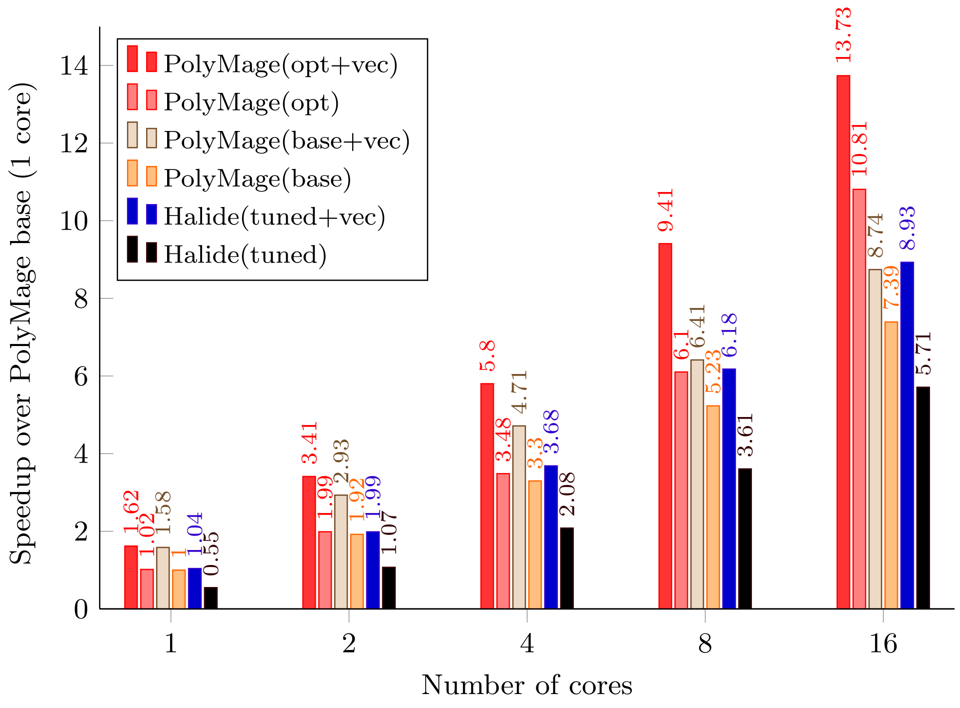


(d) Bilateral Grid

Figure 4.1: Pyramid Blending and Bilateral Grid



(e) Camera Pipeline



(f) Local Laplacian Filter

Figure 4.1: Speedups relative to PolyMage (base) on a single thread. For PolyMage, ‘opt’ includes all optimizations other than enabling *icc* auto-vectorization. ‘base’ implies all scalar optimizations including stage inlining, but not grouping, tiling, and storage optimizations. Absolute execution times can be determined in conjunction with Table 4.1.

Benchmark	Group size (*max*)	Scratch pad allocation (KB) (for largest group)	Tuned tile sizes
Unsharp Mask	4	150	8 × 512
Bilateral Grid [14]	2	135	8 × 128
Harris Corner [29]	11	180	32 × 256
Camera Pipeline	25	610	64 × 256
Pyramid Blending [12]	7	332	16 × 256
Multiscale Interpolate	9	286	32 × 128
Local Laplacian [4]	5	113	8 × 256

Table 4.3: Columns from left to right: Application, size of the largest group in the best implementation found after autotuning, the amount of scratch pad allocated for each tile in the largest group, and the best tile sizes (2-d tiling) determined by autotuning.

Benchmark	Execution times (ms)				PolyMage speedup	
	PolyMage (opt + vec)			OpenCV	(16 cores) over	
	1 core	4 cores	16 cores	(1 core)	OpenTuner	H-tuned
Unsharp Mask	42.21	11.43	3.95	84.44	1.39×	*1.63×
Bilateral Grid	89.76	27.30	8.47	-	1.09×	0.89×
Harris Corner	233.79	68.03	18.69	810.24	2.61×	*2.59×
Camera Pipeline	67.87	19.95	5.86	-	10.05×	1.04×
Pyramid Blending	196.99	57.84	21.91	197.28	27.61×	*4.61×
Multiscale Interpolate	101.70	34.73	18.18	-	12.72×	1.81×
Local Laplacian	274.50	76.60	32.35	-	9.41×	1.54×

Table 4.4: Columns from left to right: Application, execution times (16 threads and vectorization enabled) in milliseconds of PolyMage (opt+vec) and OpenCV, speedup of PolyMage (opt+vec) over auto-tuned (OpenTuner) and hand-tuned Halide schedules. Execution times for Halide can be derived from Figure 4.1 and this table.

4.4 Summary

For applications from the Halide repository, PolyMage obtains a mean (geometric) speedup of 1.27× over *H-tuned* while running on 16 cores. The corresponding speedup over manually tuned Halide schedules for all the seven applications is 1.75×. When compared with Halide schedules automatically tuned with OpenTuner, PolyMage is 5.39× faster on average. We believe that automatically obtaining this level of parallel performance while requiring the programmer to only provide a high-level specification of the computation is a significant result. Determining and applying a similar sequence of transformations manually is often either very tedious or infeasible (cf. Figure 3.9). For *camera pipeline*, our 86 line input code was transformed to 732 lines of C++ code, and performs only 10% slower than an expert-tuned version (FCam). The generated code for both PolyMage(*base*) and PolyMage(*opt*) is publicly available [43] for all the benchmarks.

Chapter 5

Related Work

In this chapter, we discuss related work from image processing pipeline compilation, stencil computation optimization and past polyhedral optimization efforts.

5.1 Domain Specific Languages for Image Processing

In this section, we discuss past work on image processing DSL's that aim to improve both performance and productivity.

5.1.1 Halide

Halide is a recent domain-specific language for image processing pipelines [45] that decouples the algorithm and schedule specification. The Halide DSL allows the user to experiment with a wide variety of schedules without changing the algorithm specification, facilitating rapid experimentation. However, providing a good schedule often requires a lot of effort, prior knowledge, and expertise in manual optimization. Autotuning based on genetic search [46] was used in conjunction with Halide to explore the vast space of schedules. However, this method converges on good schedules very slowly, taking hours to days, and requires seed schedules for fast convergence. This approach is no longer maintained or available with Halide ([3], section 4.2).

5.1.2 OpenTuner

A more recent approach for autotuning Halide programs is based on the OpenTuner [3] framework. Although more robust, the underlying approach still relies on combining several search techniques to stochastically explore the schedule space. This is only effective for small pipelines due to the exponential increase in the schedule space with pipeline size. Though the schedule space is vast, only a small subset of the space matters in practice. Our model-driven approach allows us to target such a subset and find schedules that outperform highly tuned schedules specified using Halide. Additionally, we employ a flexible transformation and code generation machinery that allows us to model a richer variety of transformations. For example, expressing parallelogram or split tiling [26] is currently not feasible with the Halide scheduling language.

5.1.3 Darkroom

Darkroom [30] is an image processing DSL which focuses on generating hardware descriptions of pipelines for FPGA or ASIC. The primary optimization strategy is line buffering which maps well to hardware implementations. The Darkroom compiler formulates an integer linear program to determine and minimize the line buffer sizes. To enable this, the Darkroom language only allows stencil and point-wise operations in a pipeline. Similar to scratch pad allocation, line buffering also results is reduced storage since it only stores the intermediate results in small local line buffers. However, it does not effectively extract task parallelism. For mapping to multicore architectures the image is tiled into chunks and each chunk is processed on a separate core using a line buffered implementation.

5.1.4 Forma

Forma [47] is another image processing DSL which maps high-level description of pipelines to multiple backends primarily focusing on GPU architectures. The compiler automatically handles memory management and effectively uses GPU hardware resources. However, the pipeline schedule optimizations are restricted to simple fusion. We believe our tiling, grouping and auto-tuning techniques can be leveraged by the Forma compiler.

5.2 Stencil Optimizers

Stencil optimization efforts have extensively focused on improving locality and parallelism for time-iterated stencil computations, resulting in parallelogram [53, 54, 10], diamond [5], split [26], and hybrid hexagonal [25] tiling techniques. The latter three techniques allow for concurrent start of tiles along a boundary, and are particularly effective in maximizing parallelism. These techniques exploit temporal locality across time steps without introducing any redundant computation. However, storage reduction and reuse using private scratchpads, a crucial optimization for image processing pipelines, is very difficult with these approaches due to the complex scratchpad indexing and management (and thus code generation) required. Overlapped tiling [35, 32, 56] is attractive in this context due to the dismissal of dependence between neighboring tiles – this greatly simplifies scratchpad allocation, indexing, and management.

In addition, dependences between stages of an image processing pipeline are of a heterogeneous nature, and more complex than those in time-iterated stencils. Our technique to construct overlapped tiles takes this heterogeneity into account, and minimizes overlap further in comparison to prior polyhedral approaches [35, 32]. As we have seen in Figures 3.6 and 3.7 prior approaches account for all dependences instead of analyzing the dependences between functions. This approach works well for time-iterated stencils since the dependence patterns across the time steps remain the same. However, when dealing with image processing pipelines our approach becomes necessary to avoid prohibitive amounts of redundant computation.

5.3 Polyhedral Compilers and Optimization

Polyhedral compilation frameworks, since the works of Bastoul [7], Cohen et al [16, 21], and Hall et al. [28, 51] have taken a decoupled view of computation (as a set of iteration domains) and schedules (as multi-dimensional affine functions). Schedules could be transformed and complex ones composed without worrying about domains. However, most subsequent works remained general-purpose, both in the techniques to determine schedules, and in the extraction of initial representation from input. Among existing fusion heuristics in the polyhedral framework [9, 36, 38], there is none suitable for image processing pipelines. The heuristics do not consider overlapped tiling of the fused groups as a possibility. In our context, we observe that the interactions between fusion, tile sizes and the overlap threshold are very important to capture for optimization. Using a domain-specific approach here is thus clearly the pragmatic one.

Other prior work on image processing languages [20, 42, 48] has focused more on the language, programmability and expressiveness aspects while proposing simple and limited optimization. There is a large body of work on compilation of stream languages [11, 23, 24, 50]. However, these works do not consider the space of optimizations that we do, in particular, the tradeoff between redundant computation and locality. Most work on stream programs dealt with one-dimensional streams while image processing pipelines involve two or higher dimensional data entities. The polyhedral framework makes it convenient to deal with such higher dimensional spaces.

Chapter 6

Conclusions

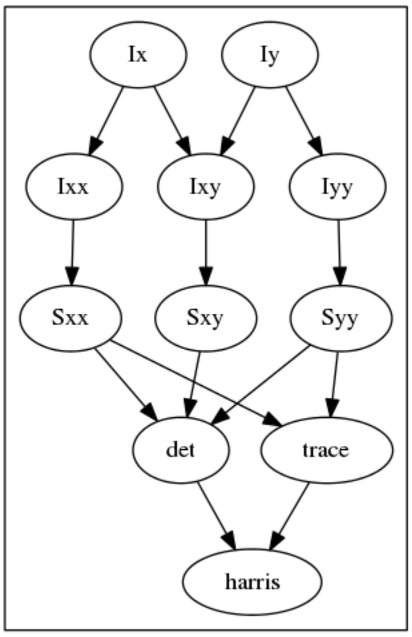
We presented the design and implementation of a domain-specific language along with its optimizing code generator, for a class of image processing pipelines. Our system, PolyMage, takes a high-level specification as input, and automatically transforms it into a high-performance parallel implementation. Such an automation was possible due to the effectiveness of our model-driven approach to fuse image processing stages, and our tiling strategy and memory optimizations for the fused stages. Experimental results on a modern multicore system with complex image processing pipelines show that the performance achieved by our automatic approach is up to 1.81× better than that achieved through tuned schedules with Halide, another state-of-the-art DSL and compiler for image processing pipelines. For a camera raw image processing pipeline, the performance of code generated by PolyMage is comparable to that of an expert-tuned version. We believe that our work is a significant advance in improving programmability while delivering high performance automatically for an important class of image processing computations.

The PolyMage [44] system can be extended in two orthogonal aspects, one in terms of the computation patterns that are supported and the other is targeting a wider range of architectures. For example, including support for dense linear algebra which will allow larger portions of computer vision pipelines to be expressed in the DSL. Thus, enabling more opportunities for optimization. Although the DSL's current focus is on image processing, applications in scientific computing domain like Geometric Multi Grid (GMG) methods can be readily expressed using the DSL. However, the grouping and tiling techniques may need to be further tailored for such applications. Supporting architectures like GPU and Xeon Phi will enable efficient usage of heterogeneous hardware. The tiling and grouping techniques developed in this thesis are architecture agnostic. However, developing effective code generation for different architectures requires further investigation. Overall, we believe that domain-specific approaches can deliver productivity and performance gains in the face of today's complex hardware.

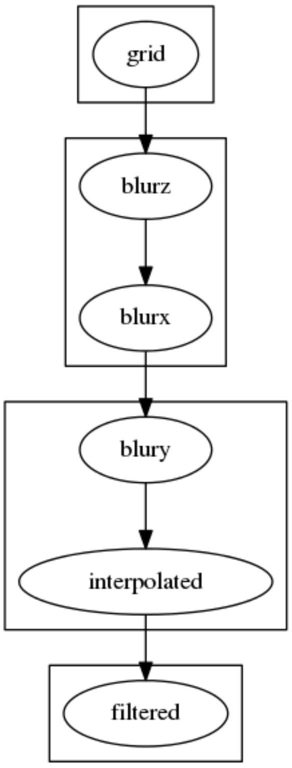
Appendix A

Benchmark Pipeline Graphs

The pipeline diagrams for all the benchmarks are listed here. These diagrams also show the grouping determined by the autotuner for the target multicore architecture. Each stage in the pipeline is show in an oval and the rectangles enclosing the stages show the grouping.

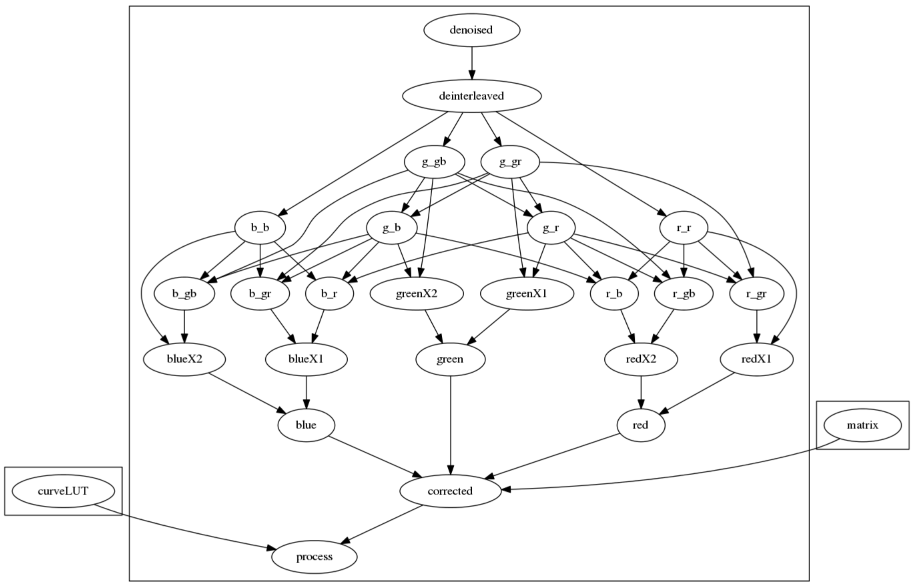


(a) Harris Corner Detection (all stages in a single group)

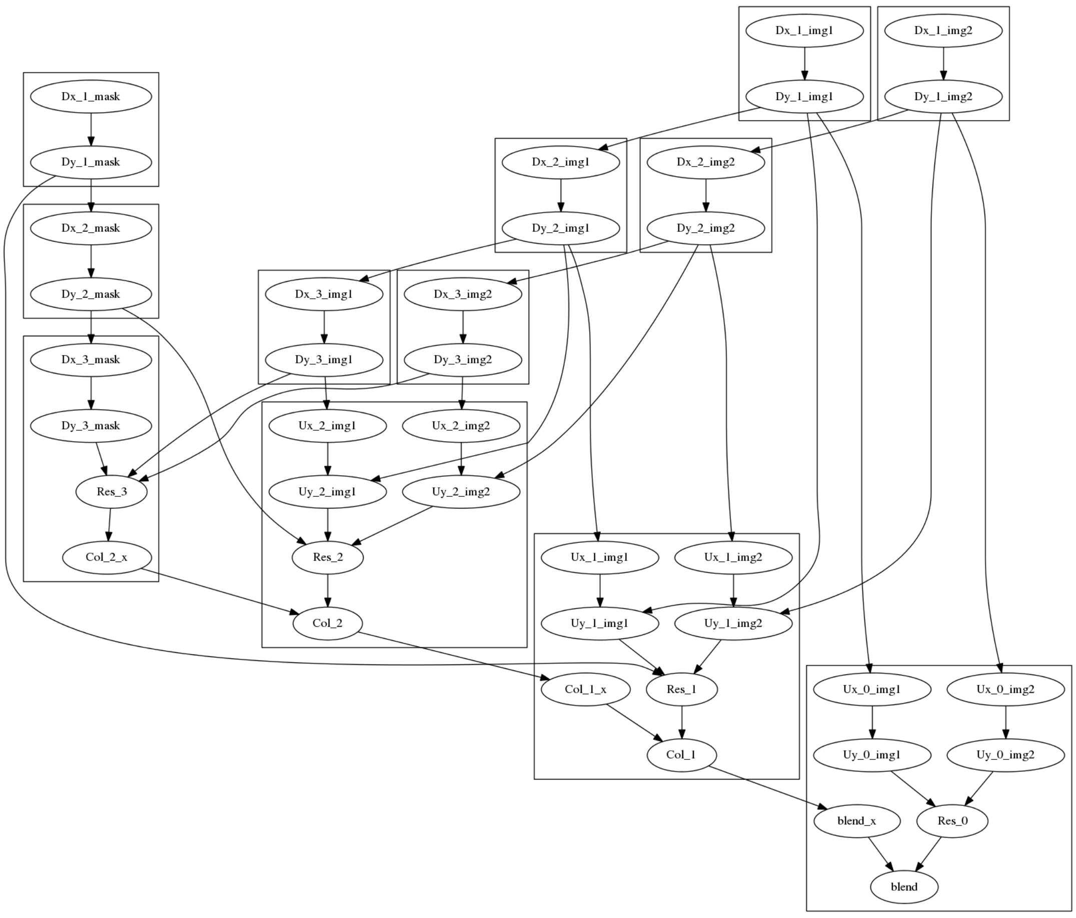


(b) Bilateral Grid

Figure A.1: Harris Corner Detection and Bilateral Grid

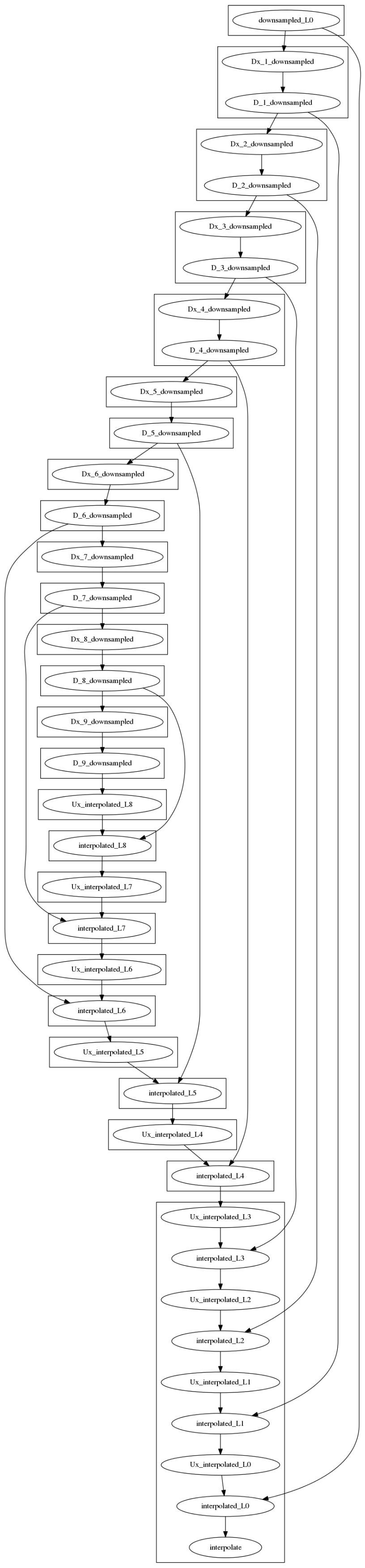


(c) Camera Pipeline



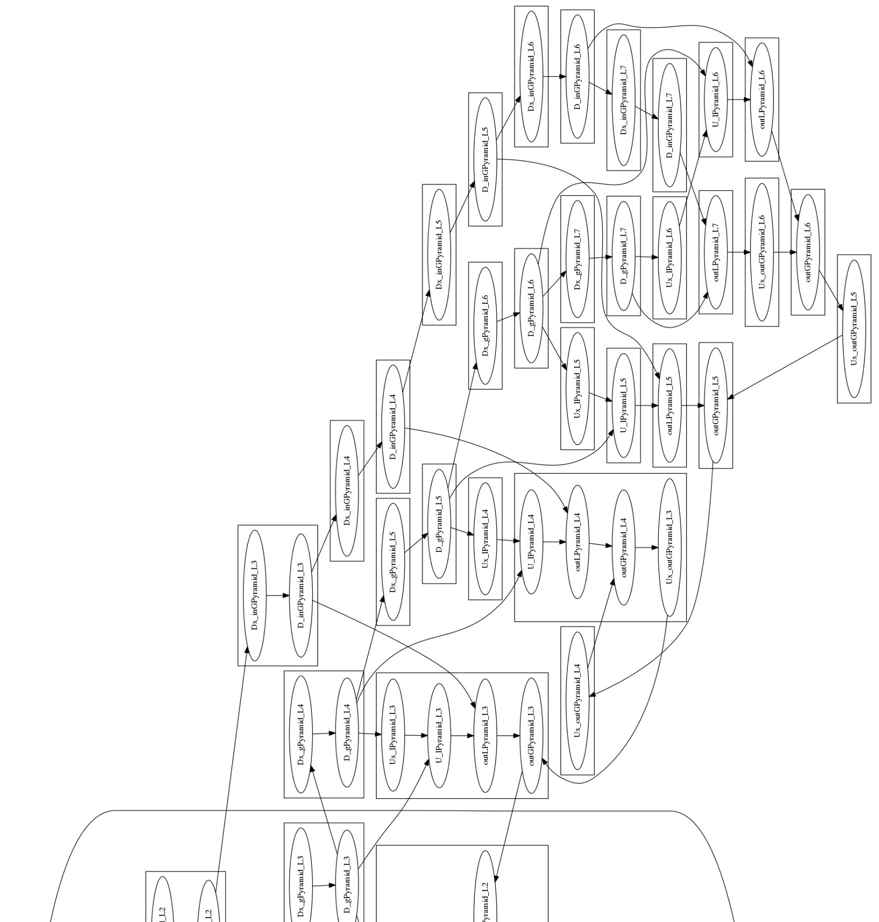
(d) Pyramid Blending

Figure A.1: Camera Pipeline and Pyramid Belnding



(e) Multiscale Interpolation

Figure A.1: Multiscale Interpolate



(f) Local Laplacian Filter

Figure A.1: The diagram shows grouping for each of the benchmark pipelines after auto-tuning for the target multicore architecture. Each stage in the pipeline is shown in an oval. All the stages in a group are enclosed by a rectangular box.

References

[1] Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. The Frankencamera: An Experimental Platform for Computational Photography. In *ACM Transactions on Graphics*, pages 29:1–29:12, 2010.

[2] Corinne Ancourt and Francois Irigoin. Scanning polyhedra with do loops. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 39–50, 1991.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International conference on Parallel Architectures and Compilation Techniques*, pages 303–316, 2014.

[4] M. Aubry, S. Paris, S. Hasinoff, J. Kautz, and F. Durand. Fast local laplacian filters: Theory and applications. *ACM Transactions on Graphics*, 2014.

[5] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *International conference for High Performance Computing, Networking, Storage, and Analysis*, pages 40:1–40:11, 2012.

[6] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan Goldman. Patchmatch: a randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics-TOG*, 28(3):24, 2009.

[7] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *International conference on Parallel Architectures and Compilation Techniques*, pages 7–16, 2004.

[8] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer vision–ECCV 2006*, pages 404–417. Springer, 2006.

[9] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *International conference on Parallel Architectures and Compilation Techniques*, pages 343–352, 2010.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, pages 101–113, 2008.

[11] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics*, 2004.

[12] Peter J. Burt and Edward H. Adelson. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics*, 2(4):217–236, 1983.

[13] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[14] Jiawen Chen, Sylvain Paris, and Frédo Durand. Real-time edge-aware image processing with the bilateral grid. In *ACM Transactions on Graphics*, 2007.

[15] The CImg Library: C++ Template Image Processing Toolkit. http://cimg.sourceforge.net/.

[16] Albert Cohen, Sylvain Girbal, David Parello, M. Sigler, Olivier Temam, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *International conference on Supercomputing*, pages 151–160, 2005.

[17] Franklin C. Crow. Summed-area tables for texture mapping. In *Annual conference on Computer Graphics and Interactive Techniques*, pages 207–212, 1984.

[18] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893 vol. 1, June 2005.

[19] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Doarve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *International conference for High Performance Computing, Networking, Storage, and Analysis*, pages 9:1–9:12, 2011.

[20] Conal Elliott. Functional image synthesis. In *Proceedings of Bridges*, 2001.

[21] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[22] Google Glass. http://www.google.com/glass.

[23] Michael I. Gordon, William Thies, and Saman P. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006.

[24] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman P. Amarasinghe. A stream compiler for communication-exposed architectures. In *International conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[25] Tobias Grosser, Albert Cohen, Justin Holewinski, P Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *International symposium on Code Generation and Optimization*, page 66, 2014.

[26] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 24–31, 2013.

[27] Halide git version. `https://github.com/halide/Halide`
Commit: 8a9a0f7153a6701b6d76a706dc08bbd12ba41396.

[28] Mary W. Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *International workshop on Languages and Compilers for Parallel Computing*, pages 50–64, 2009.

[29] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Fourth Alvey Vision Conference*, pages 147–151, 1988.

[30] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[31] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *International conference on Supercomputing*, pages 13–24, 2013.

[32] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *International conference on Supercomputing*, pages 311–320, 2012.

[33] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *International conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–362, 2012.

[34] Berthold K Horn and Brian G Schunck. Determining optical flow. In *1981 Technical Symposium East*, pages 319–331. International Society for Optics and Photonics, 1981.

[35] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, 2007.

[36] A. Leung, N.T. Vasilache, B. Meister, and R.A. Lethin. Methods and apparatus for joint parallelism and locality optimization in source code compilation, June 3 2010. WO Patent App. PCT/US2009/057,194.

[37] D.G. Lowe. Object recognition from local scale-invariant features. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1150–1157 vol.2, 1999.

[38] Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. Revisiting loop fusion in the polyhedral framework. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 233–246, 2014.

[39] OpenCV: Open Source Computer Vision. http://opencv.org.

[40] Sylvain Paris, Samuel W. Hasinoff, and Jan Kautz. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. In *ACM Transactions on Graphics*, pages 68:1–68:12, 2011.

[41] Sylvain Paris, Pierre Kornprobst, JackTumblin Tumblin, and Frédo Durand. Bilateral filtering: Theory and applications. *Foundations and Trends® in Computer Graphics and Vision*, 4(1):1–75, 2009.

[42] CoreImage. Apple Core Image programming guide.

[43] PolyMage benchmarks. `https://github.com/bondhugula/polymage-benchmarks`.

[44] PolyMage project page. `http://mcl.csa.iisc.ernet.in/polymage.html`.

[45] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, 2012.

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, pages 519–530, 2013.

[47] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. Forma: A dsl for image processing applications to target gpus and multi-core cpus. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU 2015, pages 109–120, New York, NY, USA, 2015. ACM.

[48] Michael A. Shantzis. A model for efficient and flexible image computing. In *ACM Transactions on Graphics*, pages 147–154, 1994.

[49] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing*, 13(4s):134:1–134:25, 2014.

[50] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *International conference on Compiler Construction*, pages 179–196, 2002.

[51] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *International Parallel and Distributed Processing Symposium*, pages 1–12, 2009.

[52] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress Conference on Mathematical Software*, volume 6327, pages 299–302. 2010.

[53] M. Wolf. More iteration space tiling. In *International conference for High Performance Computing, Networking, Storage, and Analysis*, pages 655–664, 1989.

[54] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *International Parallel and Distributed Processing Symposium*, pages 171 –180, 2000.

[55] Jingling Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[56] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *International symposium on Code Generation and Optimization*, pages 207–218, 2012.