PolyMage: Automatic Optimization for Image Processing Pipelines

Ravi Teja Mullapudi
Department of Computer Science and Automation,
Indian Institute of Science
Bangalore 560012, INDIA
ravi.mullapudi@csa.iisc.ernet.in

Vinay Vasista
Department of Computer Science and Automation,
Indian Institute of Science
Bangalore 560012, INDIA
vinay.vasista@csa.iisc.ernet.in

Uday Bondhugula
Department of Computer Science and Automation,
Indian Institute of Science
Bangalore 560012, INDIA
uday@csa.iisc.ernet.in

Abstract

This paper presents the design and implementation of PolyMage, a domain-specific language and compiler for image processing pipelines. 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, 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. For applications that demand high performance, the traditional options are to use 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.

The focus of our system, PolyMage, is on automatically generating high-performance implementations of image processing pipelines expressed in a high-level declarative language. 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 automatically. Experimental results on a modern multicore system show that the performance achieved by our automatic approach is up to $1.81\times$ better than that achieved through manual tuning in Halide, a state-of-the-art language and compiler for image processing pipelines. For a camera raw image processing pipeline, our performance is comparable to that of a hand-tuned implementation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – compilers, optimization, code generation

Keywords Domain-specific language; image processing; polyhedral optimization; locality; parallelism; tiling; multicores; vectorization

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694364

1. Introduction

Image processing is pervasive, spanning several areas including computational photography, computer vision, medical imaging, and astronomy. Applications in these areas range from enhancing the capabilities of digital cameras and emerging devices like Google Glass [18], autonomous driving cars, to Magnetic Resonance Imaging (MRI) and analyzing astronomical data. 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 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. 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 for achieving high performance. Manually exploiting parallelism and locality on modern architectures for complex pipelines is a daunting task. Libraries such as OpenCV [32], CImg [12] and MATLAB image processing toolboxes only provide tuned implementations for a limited set of algorithms on specific architectures. Even when optimized implementations of the individual routines required for a task are available, the inability to optimize across them prevents achieving high performance.

A promising way to address the tension between ease of programming and high 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 [15, 28, 39]. For image processing, languages like CoreImage [35] and functional image synthesis [16] have focused on creating easy-to-use abstractions with minimal compiler optimization. Halide [36, 37] 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.

In this paper, we describe our framework, PolyMage, comprising a DSL, an optimizer, and an autotuner, for generating high performance implementations of image processing pipelines. We first briefly describe the input language in Section 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 Section 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.

Section 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 highly-tuned schedules manually specified using Halide and implementations using the OpenCV library. In cases where feasible, we show that the schedule determined by our system when specified using Halide provides improved performance. In Section 5 we discuss related work, and conclusions are presented in Section 6.

2. Language Specification

In this section, we give a brief overview of our DSL and also provide a description of the computation patterns that can be expressed with it.

The design of our language is inspired by Halide [36], and allows a user to intuitively express common computation patterns that emerge in image processing. These patterns include point-wise operations, stencils, upsampling and downsampling, histograms, and time-iterated methods. Table 1 shows the data access patterns corresponding to some of these operations. The language abstracts 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. Rather than building a standalone language, we chose to embed our language in Python.

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 1. Typical computation patterns in image processing

The specification of Harris corner detection [25] algorithm in our PolyMage DSL is shown in Figure 1. Parameters like 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 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.

```
 1 R, C = Parameter(Int), Parameter(Int)
 2 I = Image(Float, [R+2, C+2])
 3
 4 x, y = Variable(), Variable()
 5 row, col = Interval(0,R+1,1), Interval(0,C+1,1)
 6
 7 c = Condition(x,'>=',1) & Condition(x,'<=',R) &
 8     Condition(y,'>=',1) & Condition(y,'<=',C)
 9
10 cb = Condition(x,'>=',2) & Condition(x,'<=',R-1) &
11      Condition(y,'>=',2) & Condition(y,'<=',C-1)
12
13 Iy = Function(varDom = ([x,y],[row,col]),Float)
14 Iy.defn = [ Case(c, Stencil(I(x,y), 1.0/12,
15                    [[-1, -2, -1],
16                     [ 0,  0,  0],
17                     [ 1,  2,  1]]) ]
18
19 Ix = Function(varDom = ([x,y],[row,col]),Float)
20 Ix.defn = [ Case(c, Stencil(I(x,y), 1.0/12,
21                    [[-1, 0, 1],
22                     [-2, 0, 2],
23                     [-1, 0, 1]]) ]
24
25 Ixx = Function(varDom = ([x,y],[row,col]),Float)
26 Ixx.defn = [ Case(c, Ix(x,y) * Ix(x,y)) ]
27
28 Iyy = Function(varDom = ([x,y],[row,col]),Float)
29 Iyy.defn = [ Case(c, Iy(x,y) * Iy(x,y)) ]
30
31 Ixy = Function(varDom = ([x,y],[row,col]),Float)
32 Ixy.defn = [ Case(c, Ix(x,y) * Iy(x,y)) ]
33
34 Sxx = Function(varDom = ([x,y],[row,col]),Float)
35 Syy = Function(varDom = ([x,y],[row,col]),Float)
36 Sxy = Function(varDom = ([x,y],[row,col]),Float)
37 for pair in [(Sxx, Ixx), (Syy, Iyy), (Sxy, Ixy)]:
38     pair[0].defn = [ Case(cb, Stencil(pair[1], 1,
39                                [[1, 1, 1],
40                                 [1, 1, 1],
41                                 [1, 1, 1]]) ]
42
43 det = Function(varDom = ([x,y],[row,col]),Float)
44 d =  Sxx(x,y) * Syy(x,y) - Sxy(x,y) * Sxy(x,y)
45 det.defn = [ Case(cb, d) ]
46
47 trace = Function(varDom = ([x,y],[row,col]),Float)
48 trace.defn = [ Case(cb, Sxx(x,y) + Syy(x,y)) ]
49
50 harris = Function(varDom = ([x,y],[row,col]),Float)
51 coarsity = det(x,y) - .04 * trace(x,y) * trace(x,y)
52 harris.defn = [ Case(cb, coarsity) ]
```

Figure 1. PolyMage specification for Harris corner detection

Lower and upper bounds are restricted to affine expressions involving constants and parameters. Lines 4 and 5 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 7 and 10. 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 37 and 38 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 [14].

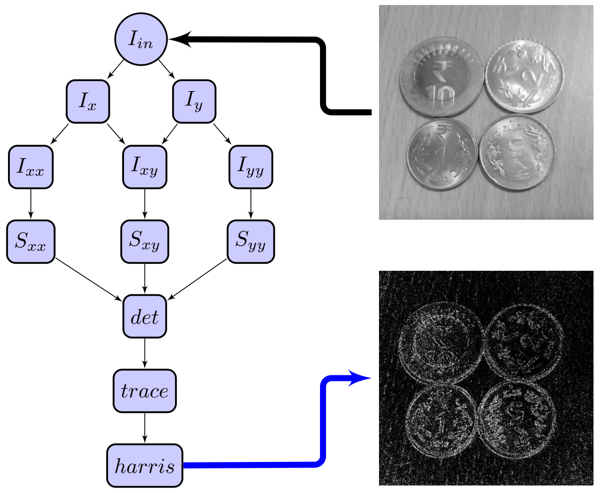


Figure 2. Harris corner detection pipeline as a graph of stages

Functions in our language do not have any state associated with them. Expressing histograms and other reduction operations however requires a mechanism which retains state. `Accumulator` is a function-like construct that serves this purpose. An accumulator has two domains, a reduction domain and a variable domain. Evaluation of the accumulator is done on the reduction domain while the accumulator is defined on the variable domain. Figure 3 shows an accumulator used to compute a histogram by counting the number of pixels of each intensity value, ranging from 0-255, in the image I.

```
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 = Accumulator(redDom = ([x,y],[row,col]),
                   varDom = ([x],bins), Int)
hist.defn = Accumulate(hist(I(x,y)), 1, Sum)
```

Figure 3. Grayscale histogram

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.

3. Pipeline Compiler

This section describes how our compiler translates pipelines specified in the PolyMage DSL into high-performance implementations. The sequence of compiler phases is shown in Figure 4. 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.

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 2 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 1, 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 1 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 5. 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 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

$$\phi(\vec{i_f}) = \vec{h} \cdot \vec{i_f} + h_0. \tag{1}$$

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

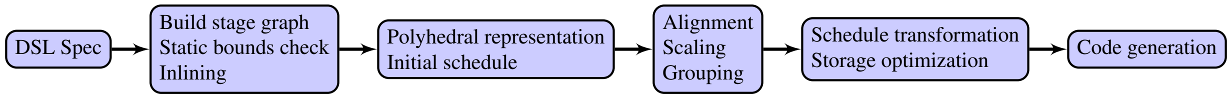


Figure 4. Phases of the PolyMage compiler

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 1 are as follows:

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

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 5 and 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. These factors 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 [8, 44], split [26], overlapped [27, 29], diamond [5], and hexagonal [21] tiling use the polyhedral model. Figure 5 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 5 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 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 5 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 [27, 29] 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.

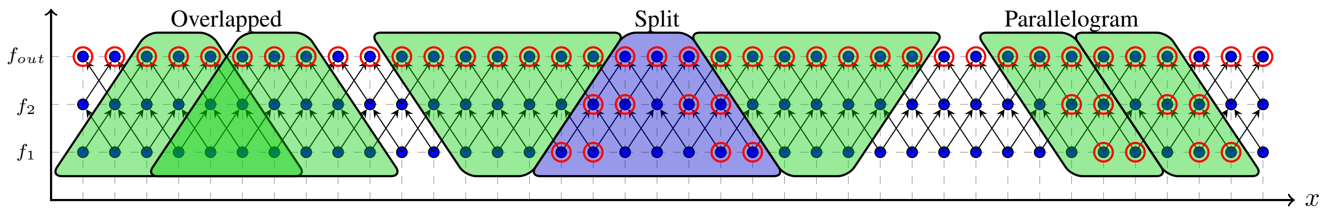
3.3 Alignment and Scaling of Functions

Constructing overlapped tiles for a group of functions is only possible when the dependences can be captured by constant vectors, as shown in Figures 5 and 6. In general, a group of heterogeneous functions can have different dimensions and complex access patterns, requiring alignment and scaling transformations to make the dependence vectors constant. 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 $(x,y)\to(1,x,y,0)$ and $(c,x,y)\to(0,c,x,y)$ 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 $(x,y)\to(1,0,x,y)$, the dependence vector becomes $(1,0,0,0)$.

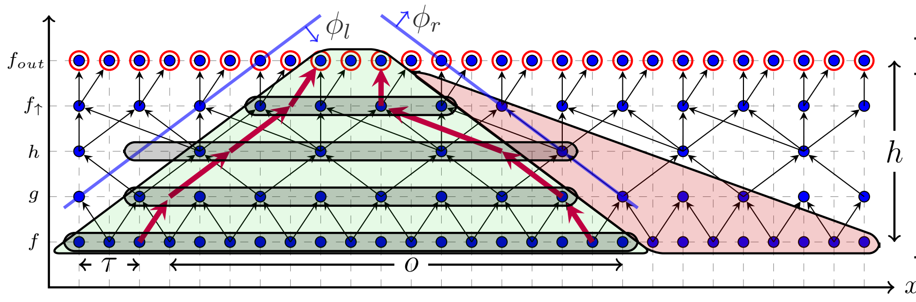
For the functions in Figure 6, value dependences are not near-neighbor like in stencils; evaluating $f_{out}(x)$ and $g(x)$ requires the values $f_{\uparrow}(x/2)$ and $f(2x-1)$ respectively. Under the initial schedule of these functions, both dependences



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

	Overlapped	Split	Parallelogram
Parallelism	✓	✓	×
Locality	✓	✓	✓
Redundancy	✓	×	×

Figure 5. Functions fused using overlapped, split and parallelogram tiling (left to right). Live-outs at tile boundaries are circled. Function definitions are on the bottom left. Characteristics of the tiling techniques are on the bottom right.



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

Figure 6. Anatomy of an overlapped tile for a group of heterogeneous functions. The functions and their scaled schedules are shown on the right. A tight tile shape is computed by analyzing dependence vectors between the stages. Extended region shows overlap with over-approximation. Horizontal boxes within the tile show scratchpad allocations.

cannot be captured by constant dependence vectors. Dependences of this form are characteristic of up-sampling and down-sampling operations. These dependences can be made near-neighbor by scaling the function schedules appropriately, as shown in Figure 6. The compiler determines the schedule alignment and scaling factors for each function in the group. This is done 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.

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. The compiler constructs schedules for overlapped tiling of functions, one dimension after another. For each dimension, the shape of an overlapped tile is determined by analyzing the dependence vectors. 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 6 shows hyperplanes which define a valid tile shape. A naive approach to determine the tile shape is to assume that every dependence vector exists uniformly at every point in the space. This over-approximates the dependence cone increasing the redundant computation by a significant amount, as shown in Figure 6. It is desirable to minimize the redundant computation by determining the tightest *slope* possible for ϕ_l and ϕ_r.

Our compiler accounts for the heterogeneity of the functions, and determines the tile shape by examining the dependence vectors between two *level*s 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 6, 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. Using the dependences at the boundaries both ϕ_l and ϕ_r are determined. Then 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, 45] 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:

$$\begin{aligned} \tau \cdot T \leq \phi_l(\vec{s_k}) \leq \tau \cdot (T+1) + o - 1 \;\wedge \\ \tau \cdot T \leq \phi_r(\vec{s_k}) \leq \tau \cdot (T+1) + o - 1. \end{aligned} \qquad (2)$$

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 (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

Our algorithm to group stages is shown as Algorithm 1. The algorithm takes the directed acyclic graph, (S, E), where S is the set of stages or functions and E, the set of edges, as input. Initially, each function in the pipeline is in a separate group. The tile sizes, an overlap threshold and the approximate estimates of all pipeline parameters, are also part of the 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. At any point, the groups in G are disjoint and their union 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.

Algorithm 1: 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      $converge \leftarrow true$
6      $cand\_set \leftarrow$ getGroupsWithSingleChild($G$, $E$)
7      $ord\_list \leftarrow$ sortGroupsBySize($cand\_set$, $P$)
8      for each $g$ in $ord\_list$ do
9          $child$ = getChildGroup($g$, $E$)
10         if hasConstantDependenceVectors($g$, $child$) then
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 $G$
```

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 6. 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 1 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 8.

Validity A grouping is valid if there is no cycle between the groups with reference to the pipeline DAG. Since algorithm 1 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 $(x, y) \rightarrow (y, x)$ 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 6, in the case of intermediate functions f, g, h 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 full buffers, instead they can be stored in small scratchpads which are private to each tile. Horizontal boxes in Figure 6 indicate such scratchpad allocations. All 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

```
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_i = -1; T_i <= R/32; T_i+=1){
      /* Scratchpads */
      float  Ix[36][260];
      float  Iy[36][260];
      float  Syy[36][260];
      float  Sxy[36][260];
      float  Sxx[36][260];
      for (int T_j = -1; T_j <= C/256; T_j+=1) {
         int lb_i = max(1, 32*T_i);
         int ub_i = min(R, 32*T_i + 35);
         for (int i = lb_i; i <= ub_i; i+=1) {
            int lb_j = max(1, 256*T_j);
            int ub_j = min(C, 256*T_j + 259);
#pragma ivdep
            for (int j=lb_j; j<=ub_j; j+=1) {
               Iy[-32*T_i+i][-256*T_j+j] = ...;
               Ix[-32*T_i+i][-256*T_j+j] = ...;
            }
         }
         lb_i = max(2, 32*T_i + 1);
         ub_i = min(R - 1, 32*T_i + 34);
         for (int i = lb_i; i <= ub_i; i+=1) {
            int lb_j = max(2, 256*T_j + 1);
            int ub_j = min(C-1, 256*T_j + 258);
#pragma ivdep
            for (int j=lb_j; j<=ub_j; j+=1) {
               Syy[-32*T_i+i][-256*T_j+j] = ...;
               Sxy[-32*T_i+i][-256*T_j+j] = ...;
               Sxx[-32*T_i+i][-256*T_j+j] = ...;
            }
         }
         if (T_j >= 0 && T_i >= 0) {
            lb_i = 32 * T_i + 2;
            ub_i = min(R - 1, 32*T_i + 33);
            for (int i = lb_i; i <= ub_i; i+=1) {
               int lb_j = 256*T_j + 2;
               int ub_j = min(C-1, 256*T_j+257);
#pragma ivdep
               for (int j=lb_j; j<=ub_j; j+=1)
                  harris[i*(R+2)+j] = ...;
             }
         }
      }
   }
}
```

Figure 7. Generated code for Harris corner detection

are small relative to the size of the functions, the reduction in storage is quite significant, leading to better locality.

In order to perform scratchpad allocation for intermediate functions, their accesses have to be remapped to the scratchpads. The compiler generates index expressions into scratchpads relative to the origin of each tile: in Figure 6, 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 7 shows the indexing ex-

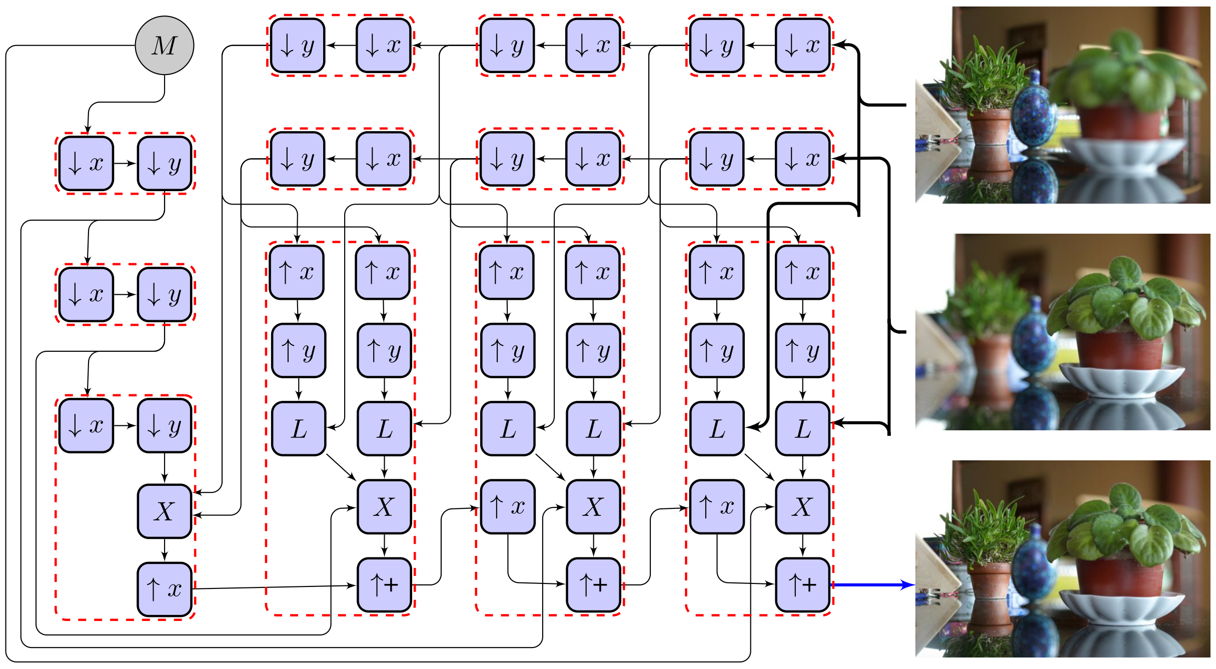


Figure 8. Pyramid Blending pipeline with four pyramid levels. The grouping generated by our compiler is shown by the dashed boxes, all the stages in one group are enclosed by a dashed box. Inputs to the pipeline are the two images on the top right, each with one of the halves out of focus, and a mask 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)

pressions 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.

3.7 Code Generation

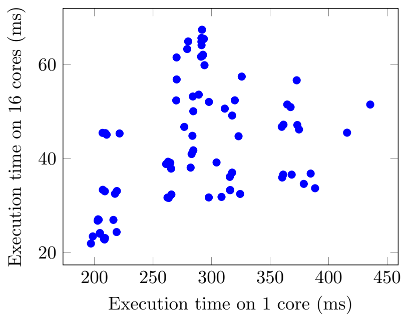
After partitioning the pipeline into groups, building overlapped tiled schedules and optimizing storage, the compiler generates a C++ function implementing the pipeline. Figure 7 shows the code generated for Harris corner detection specification (Figure 1). The integer set library (isl) [42] is used 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 7, 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

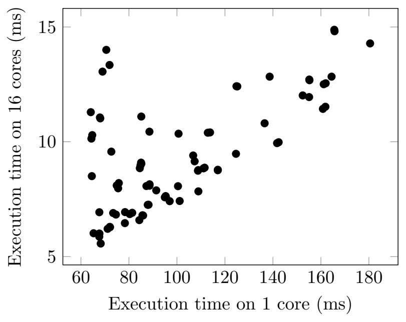
Our grouping heuristic (Section 3.5) and overlapped tiling (Section 3.4) use fixed tile sizes and overlap threshold to generate an implementation of the pipeline. It is tedious for a user to infer the right choice of parameters that lead to the best performance. Given that the solution space is narrowed 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 9 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.

4. Experimental Evaluation

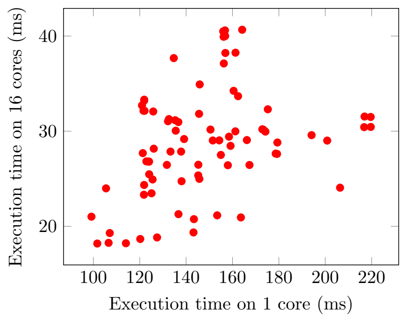
Setup: All experiments were conducted on an Intel Xeon E5-2680 based on the Sandybridge microarchitecture. The ma-



(a) Pyramid Blending



(b) Camera Pipeline



(c) Multiscale Interpolation

Figure 9. Autotuning results (note: origin of the plots is not (0,0); it has been shifted for better illustration)

chine 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 [23] 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 2. We evaluate our results relative to other implementations 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. [37]. 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. [37] 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.

Table 2 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, lines of code in PolyMage, and execution times for OpenCV versions with optimized library routines. Figure 10 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.

Multiscale Interpolation interpolates pixel values at multiple scales. The *H-tuned* schedule does loop reordering, vectorization, tiling, and parallelization but no fusion. Our best schedule performs non-trivial grouping of the pipeline stages and outperforms *H-tuned* by $2\times$. Specifying our best schedule using Halide (*H-matched*) completely bridged the $2\times$ gap in performance between *H-tuned* and *PolyMage (opt+vec)*. Also, the *H-matched* schedule provided better vectorization gains than the *H-tuned* one.

Harris Corner Detection [25] 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 1. The best schedule generated by PolyMage inlines all pointwise operations, and groups all stencil functions together. The speedup of the tiled and vectorized implementation is $46.78\times$ over the baseline. An interesting point to note is that, without the tiling transformation, vectorization improves the single thread performance only by $1.12\times$. This shows the importance of locality transformations to effectively utilize

Benchmark	Stages	Lines	Image size	Execution times (ms)				PolyMage speedup (16 cores) over	
				PolyMage (opt + vec)			OpenCV		
				1 core	4 cores	16 cores	(1 core)	OpenTuner	H-tuned
Unsharp Mask	4	16	2048×2048×3	42.21	11.43	3.95	84.44	1.39×	*1.63×
Bilateral Grid [11]	7	43	2560 × 1536	89.76	27.30	8.47	-	1.09×	0.89×
Harris Corner [25]	11	43	6400 × 6400	233.79	68.03	18.69	810.24	2.61×	*2.59×
Camera Pipeline	32	86	2528 × 1920	67.87	19.95	5.86	-	10.05×	1.04×
Pyramid Blending [10]	44	71	2048×2048×3	196.99	57.84	21.91	197.28	27.61×	*4.61×
Multiscale Interpolate	49	41	2560×1536×3	101.70	34.73	18.18	-	12.72×	1.81×
Local Laplacian [4]	99	107	2560×1536×3	274.50	76.60	32.35	-	9.41×	1.54×

Table 2. Columns from left to right: Application, number of pipeline stages, lines of code in PolyMage, input size, execution times in milliseconds of PolyMage (opt+vec) and OpenCV (those for Halide can be derived from Figure 10 and this table), speedup of PolyMage (opt+vec) over auto-tuned (OpenTuner) and hand-tuned Halide schedules (16 thread execution in all three cases).

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.

Pyramid Blending [10] blends two images into one using a mask and constructing a Laplacian pyramid. The complex grouping performed by PolyMage is shown in Figure 8. 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*.

Bilateral Grid [11, 34] 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 one group, 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*.

Camera Pipeline processes raw images captured by the camera into a color image. 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 10e.

Local Laplacian Filter [4, 33] enhances the local contrast in an image. 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.

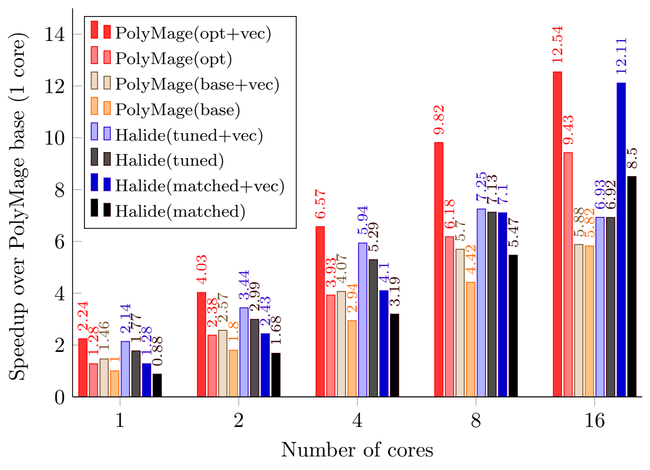
Unsharp Mask is a simple pipeline used to sharpen image edges, and comprises a series of stencil operations. The *H-tuned* schedule we use is very similar to our best schedule.

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 8). 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).

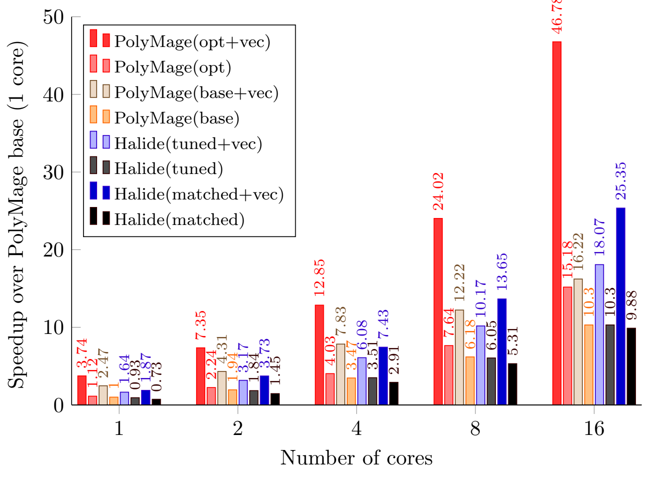
5. Related Work

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

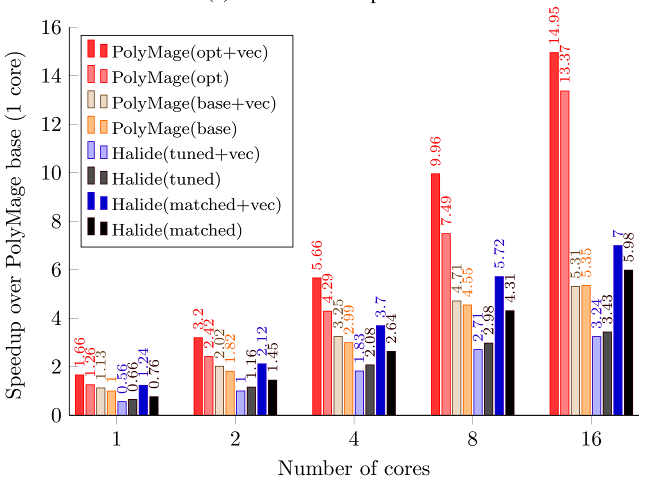
Halide is a recent domain-specific language for image processing pipelines [36] that decouples the algorithm and schedule specification. The Halide DSL allows the user to experiment with a wide variety of schedules without changing the algorithm specification, facilitating rapid experimen-



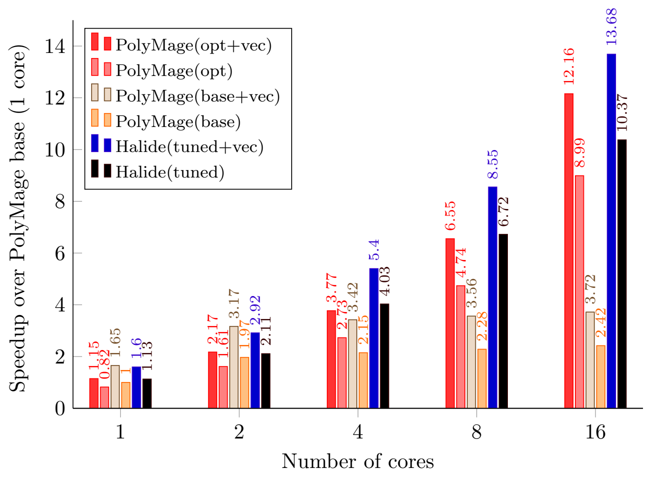
(a) Multiscale Interpolation



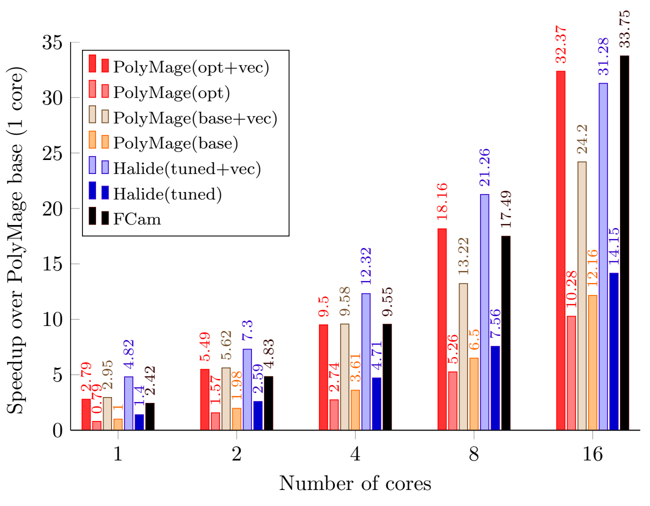
(b) Harris Corner Detection



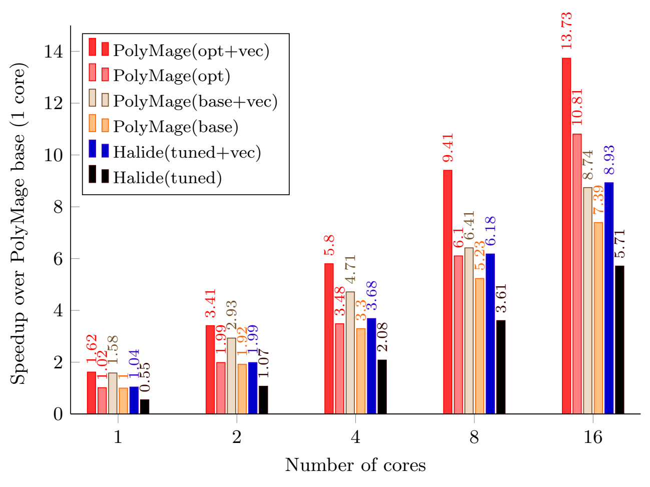
(c) Pyramid Blending



(d) Bilateral Grid



(e) Camera Pipeline



(f) Local Laplacian Filter

Figure 10. 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 2.

tation. However, providing a good schedule often requires a lot of effort, prior knowledge, and expertise in manual optimization. Autotuning based on genetic search [37] 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).

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 [22] is currently not feasible with the Halide scheduling language.

Stencil optimization efforts have extensively focused on improving locality and parallelism for time-iterated stencil computations, resulting in parallelogram [8, 43, 44], diamond [5], split [22], and hybrid hexagonal [21] 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 [27, 29, 46] 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 [27, 29].

Polyhedral compilation frameworks, since the works of Bastoul [6], Cohen et al [13, 17], and Hall et al. [24, 41] 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 [7, 30, 31], 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 [16, 35, 38] 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 [9, 19, 20, 40]. 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.

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.

Acknowledgments

We gratefully acknowledge the authors of Halide for developing and actively maintaining Halide as an open-source project. We are very thankful to the anonymous reviewers of ASPLOS for their constructive comments. We thank Intel Labs, Bangalore for donation of software and equipment with which the experiments presented in this paper were conducted. We also thank Google India for supporting Ravi Teja Mullapudi's travel to ASPLOS'15.

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] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[15] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, 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.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] CoreImage. Apple Core Image programming guide.

[36] 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.

[37] 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.

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

[39] 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.

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

[41] 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.

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

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

[44] 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.

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

[46] 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.