PolyMage: A Domain-Specific Language and Compiler for Image Processing Pipelines and Multigrid Methods

Vinay Vasista   Uday Kumar Reddy B   Ravi Teja Mullapudi
Indian Institute of Science

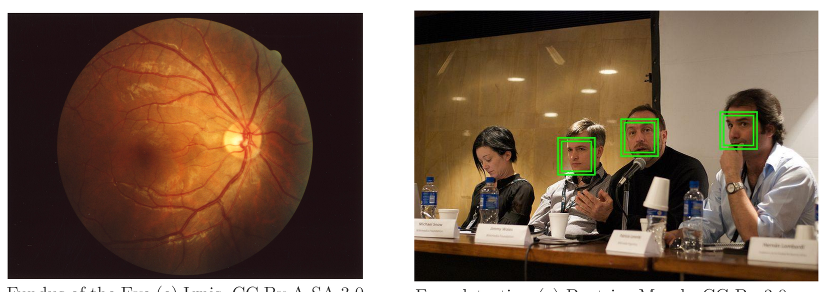
Image Processing Pipelines

Where are image processing pipelines used?

- Every image uploaded to social networks like Google and Facebook is processed by a pipeline
- Run on every camera-enabled device
- Important workloads both at data center and mobile device scale

Google+ Auto Enhance

- Not just limited to image enhancement Medical Imaging, Computer Vision ...



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

Manually optimizing pipelines for modern architectures is hard

Memory hierarchy, Parallelism

Goal: Performance levels of manual tuning in a fully automatic fashion

Approach: Domain-specific language and compiler to generate an optimized pipeline implementation

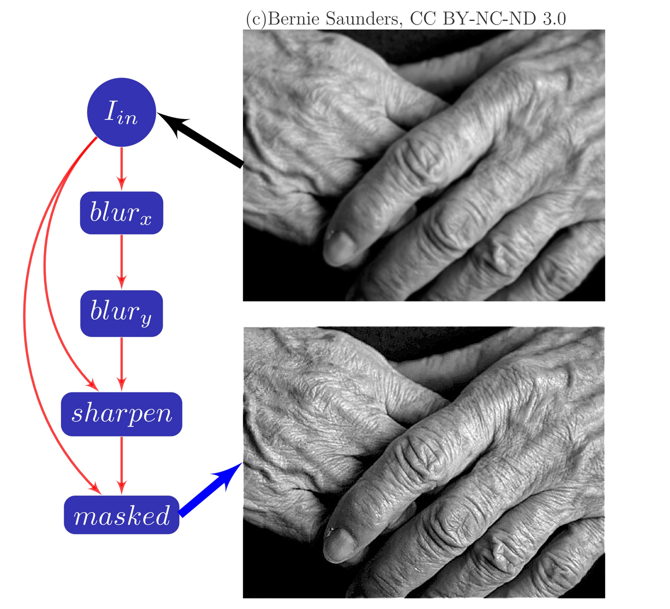
High-level Language to Describe Pipelines

Key abstractions: Image as a function on an integer grid;
Pipeline as a graph of interconnected stages

```
R, C = Parameter(Int), Parameter(Int)
thresh, w = Parameter(Float), Parameter(Float)

x, y, c = Variable(), Variable(), Variable()
I = Image(Float, [3, R+4, C+4])

cr = Interval(0, 2, 1)
xr, xc = Interval(2, R+1, 1), Interval(0, C+3, 1)
yr, yc = Interval(2, R+1, 1), Interval(2, C+1, 1)

blurx = Function(varDom = ([c, x, y], [cr, xr, xc]), Float)
blurx.defn = [ Stencil(I(c, x, y), 1.0/16,
               [[1, 4, 6, 4, 1]]) ]

blury = Function(varDom = ([c, x, y], [cr, yr, yc]), Float)
blury.defn = [ Stencil(blurx(c, x, y), 1.0/16,
               [[1], [4], [6], [4], [1]]]) ]

sharpen = Function(varDom = ([c, x, y], [cr, yr, yc]), Float)
sharpen.defn = [ I(c, x, y) * ( 1 + w ) - blury(c, x, y) * w ]

masked = Function(varDom = ([c, x, y], [cr, yr, yc]), Float)
diff = Abs((I(c, x, y) - blury(c, x, y)))
cond = Condition( diff, '<', thresh )
masked.definition = Select(cond, I(c, x, y), sharpen(c, x, y))
```

PolyMage code for Unsharp Mask pipeline

(c)Bernie Saunders, CC BY-NC-ND 3.0



- Capture common image processing operations: point-wise, stencil, sampling, histogram
- Enable compiler analysis and transformation

Compiler Phases

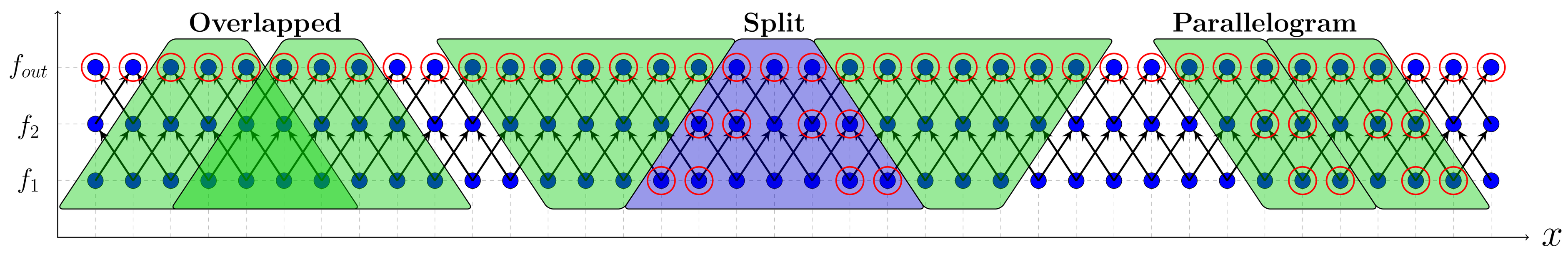
DSL Spec → Build stage graph / Static bounds check / Inlining → Alignment / Scaling / Grouping → Polyhedral representation (initial schedule) → Schedule transform - Fusion - Tiling - multidim parallelism → Storage Optimizations - scratchpad buffers - reuse allocs → Code generation ← Pooled memalloc

Optimizing for Parallelism, Locality and Storage

Polyhedral representation

- Geometric view of computation and schedules
- Effective representation for dependence analysis, schedule transformation and code generation

- Naive schedules for computing function values often yield sub-optimal performance
- Tiled execution to exploit *parallelism, locality,* and reduce *storage*



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

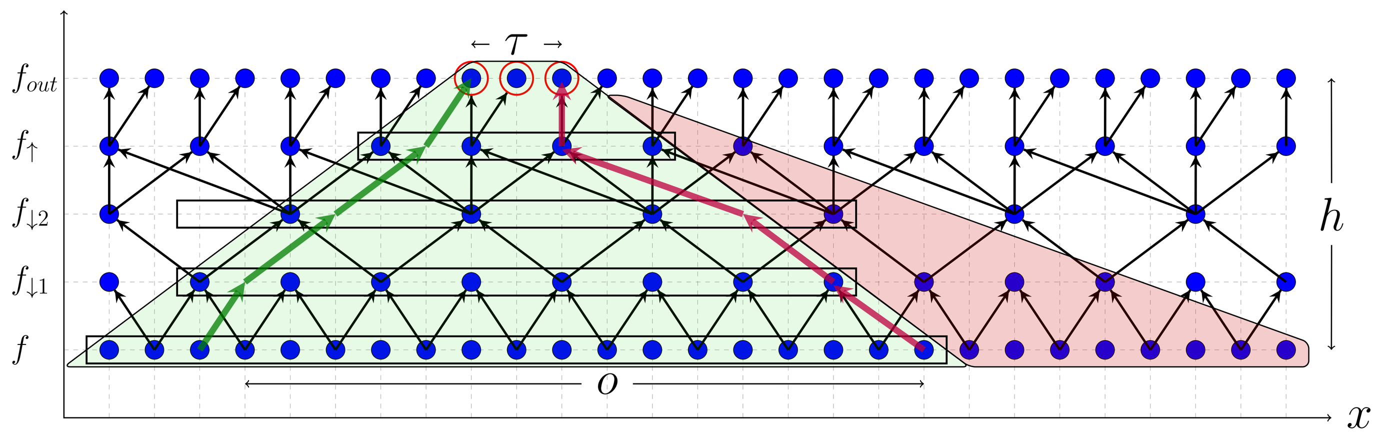
	Overlapped	Split	Parallelogram
Parallelism	✓	✓	×
Locality	✓	✓	✓
Redundancy	✓	×	×

The figure shown above depicts the producer-consumer relationships between the values of functions f_1, f_2 and f_{out} defined in the table. Live-out points are encircled in red. Characteristics of each of the tiling techniques are shown in the right table.

Despite the redundant computation introduced, overlapped tiling is beneficial for image processing pipelines since it improves locality and parallelism while allowing for excellent storage optimization.

Tiling for Heterogeneous Functions

- Prior approaches for overlapped tiling only target homogeneous time-iterated stencil computations. Stages in image processing pipelines exhibit heterogeneous dependence patterns and are not limited to simple stencils
- Function schedules are scaled and aligned to make dependences short. The overlapped tile shape is determined by analyzing dependence vectors between stages.



- O is the amount of overlap, h is the tile height, and τ is the tile size
- Extended region shows overlap with an over-approximation for tile shape
- Scratchpad allocations are shown as rectangles within the tile

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

Fusing Pipeline Stages for Tiling

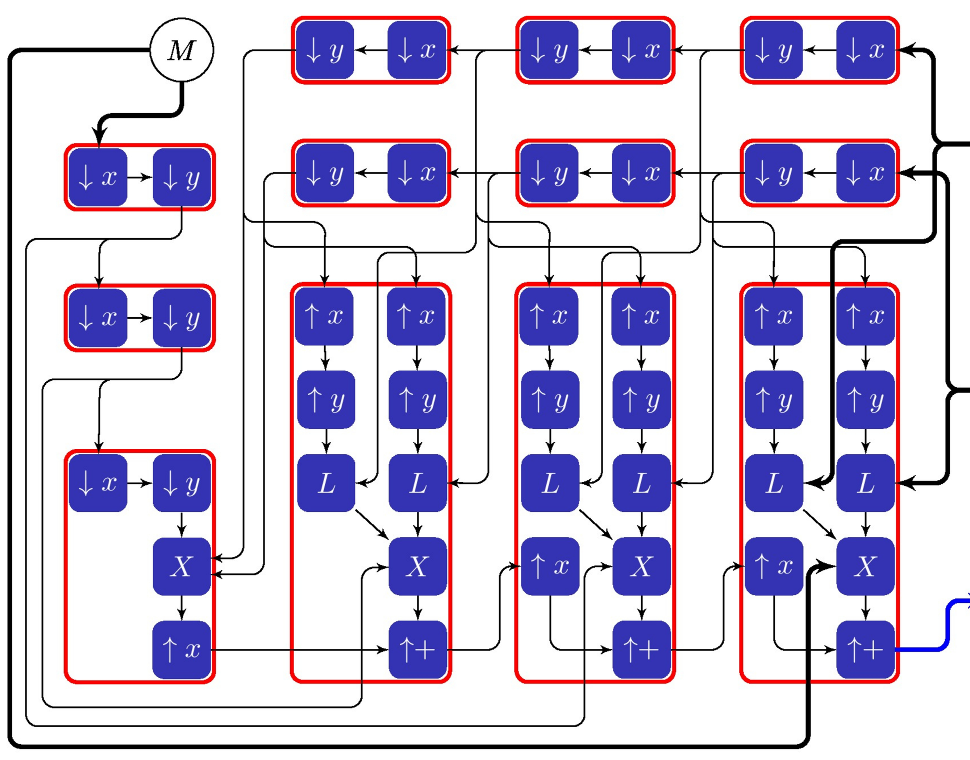


Image courtesy of Kyros Kutulakos

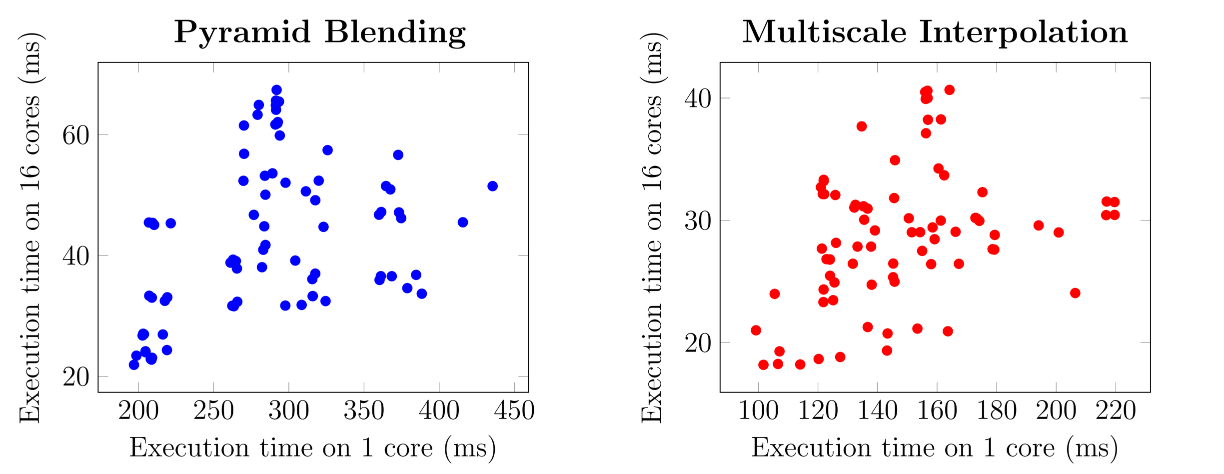
Grouping

- Greedy iterative algorithm to choose a grouping among an exponential number of valid groupings
- Groups only the stages which can be overlap tiled, i.e., stages whose schedules can be scaled and aligned while keeping dependences short
- Fuses stages till the overlap relative to input tile size is less than the specified overlap threshold

Figure shows **Laplacian pyramid blending** pipeline with four pyramid levels. Inputs to the pipeline are the top two images on the right, each with one of the halves out of focus, and a mask image M. The image at the bottom right is the blended output where both halves of the image are in focus.

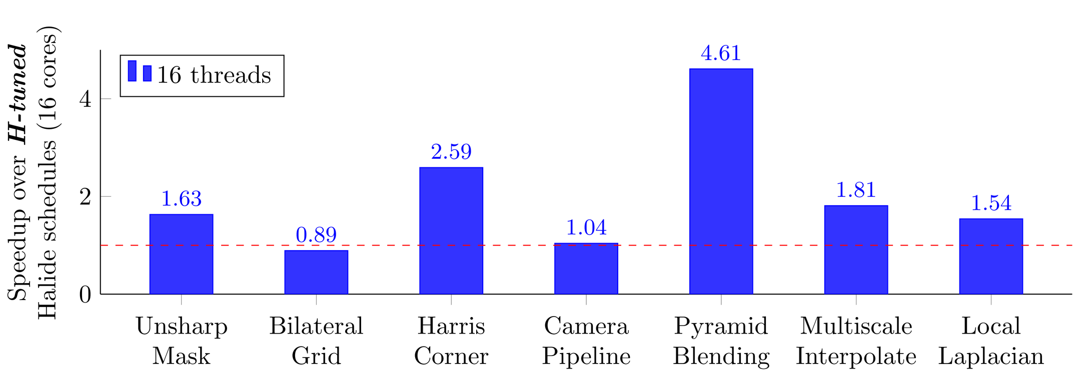
Autotuning for the Best Grouping

- Grouping heuristic takes tile sizes, overlap threshold as input, and determines a grouping structure
- The ideal grouping depends on pipeline characteristics and target machine
- Our model-driven approach narrows down the search space to a small set of tile size and overlap threshold parameters

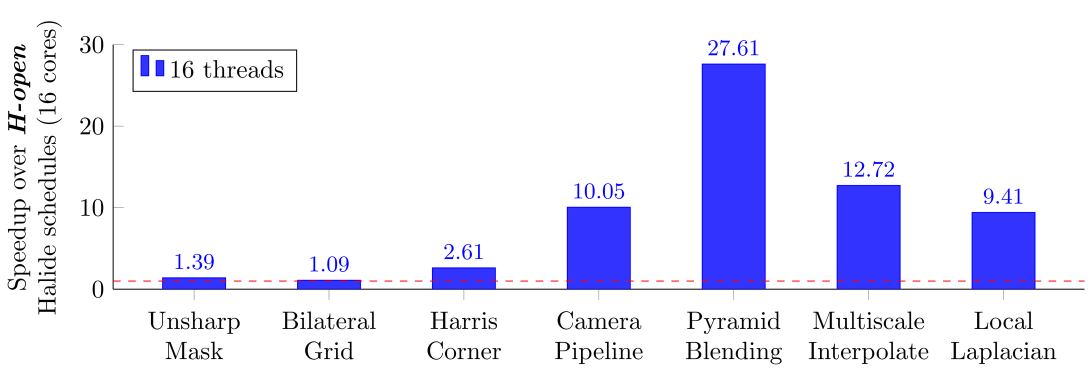


The scatter plots show the execution times (y-axis: on 16 cores, x-axis: on 1 core) in milliseconds for configurations explored by the autotuner for two benchmarks.

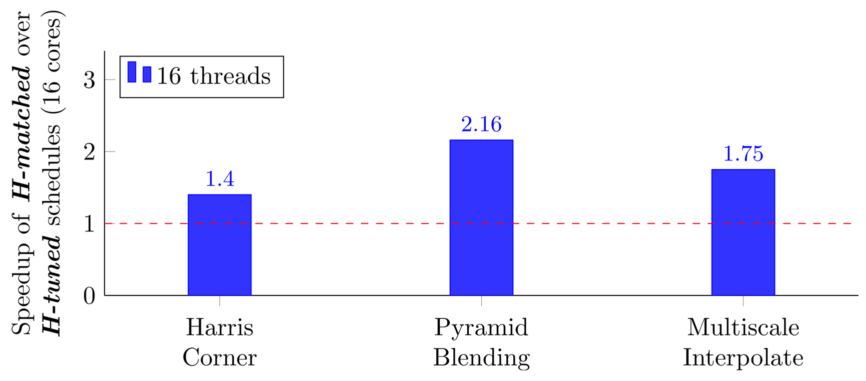
Experimental Results



Mean (geometric) speedup = 1.75×



Mean (geometric) speedup = 5.39×



PolyMage matched schedules outperform Halide tuned schedules

- Seven image processing application benchmarks, which vary widely in structure and complexity
- Comparison with Halide, a domain-specific language for image processing pipelines
  - *H-tuned*: schedule manually tuned for the target machine
  - *H-open*: best schedule found by OpenTuner after 12 hours of autotuning
  - *H-matched*: expressed to closely match PolyMage generated schedules

Target system: Intel Xeon E5-2680, dual socket NUMA (8 cores each), @2.7GHz, 32KB L1 and 512KB L2 cache/core, 20MB L3 shared cache, 64 GB non-ECC RAM. Intel C/C++ compiler v14.0.1

Conclusions

- Automatically generating image processing pipelines equaling or surpassing manual optimization is feasible
  - Choosing the right abstractions
  - Using a combination of model-driven approach and autotuning
- What can be improved?
  - Grouping heuristic can be more sophisticated
  - Vectorization for multiscale applications
- Going forward
  - Tackle more patterns and domains
  - Targeting GPUs and multicores in embedded systems

Acknowledgments

- Halide team, isl, islpy and cgen
- Intel Labs, Bangalore for donation of hardware and software
- Google, ACM SIGPLAN, IARCS for travel support