Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, Saman Amarasinghe

Group 1
Grace Ma, Nachiketa Gargi, Nathan Tseng

Motivation

- Image processing is everywhere
- Image processing pipelines are composed of large graphs with data parallelism
- The difference between a naive and optimized implementations are of orders of magnitude

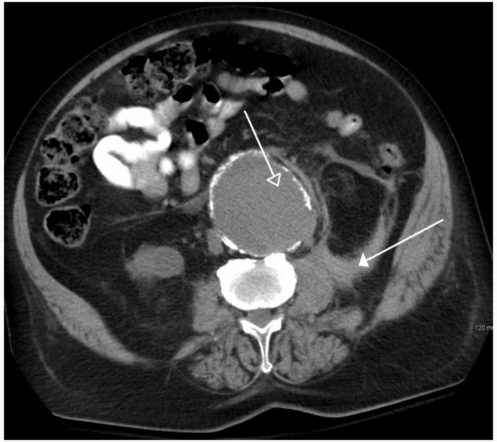


Image Processing Pipelines

`in` $\xrightarrow{\text{3x1 conv}}$ `blurx` $\xrightarrow{\text{1x3 conv}}$ `out`

Two Stage Pipeline

Scheduling Overview

Halide DSL

```
UniformImage in(UInt(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) +
in(x+1,y)
Func out(x,y) = blurx(x,y-1) + blurx(x,y) +
blurx(x,y+1)
```

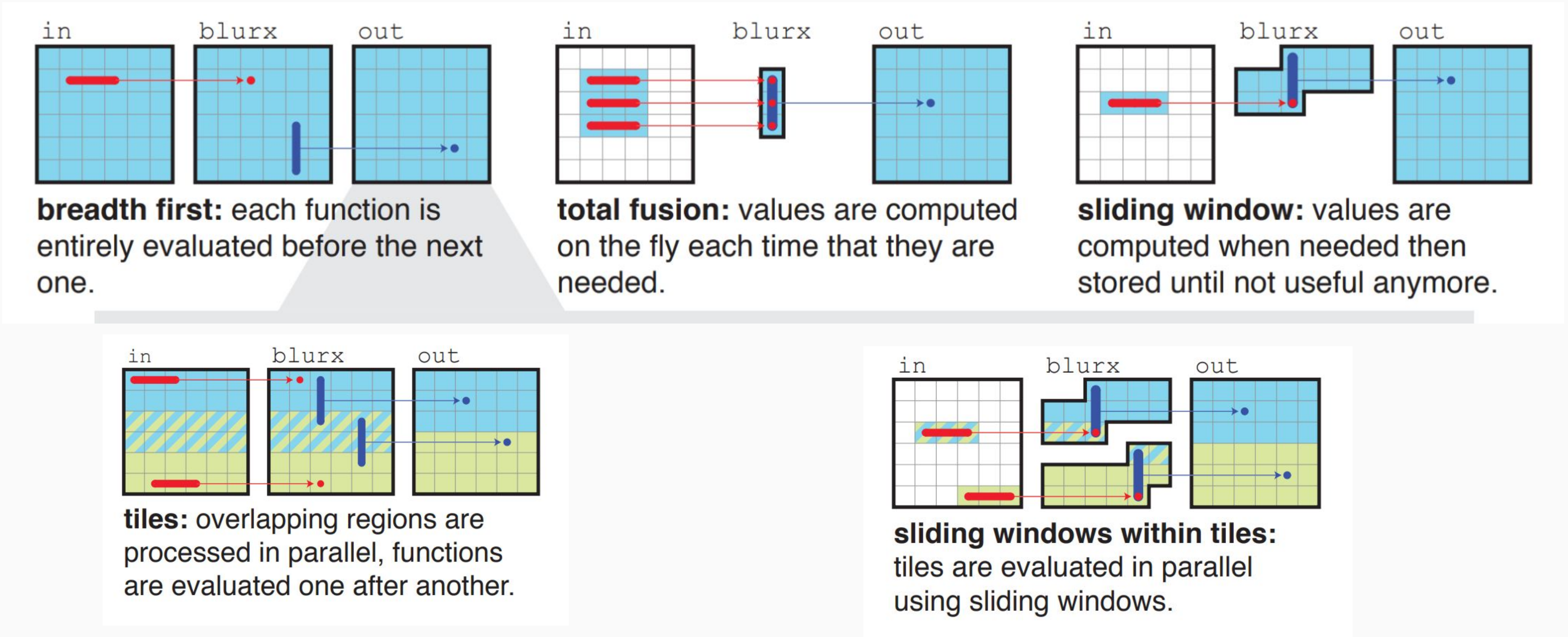
Valid Schedule #1

```
alloc blurx[2048][3072]
for each y in 0..2048:
  for each x in 0..3072:
    blurx[y][x] = in[y][x-1] + in[y][x] +
in[y][x+1]
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    out[y][x]=blurx[y-1][x] + blurx[y][x] +
blurx[y+1][x]
```

Breadth-first

Scheduling Choice Space

Call Schedule



breadth first: each function is entirely evaluated before the next one.

total fusion: values are computed on the fly each time that they are needed.

sliding window: values are computed when needed then stored until not useful anymore.

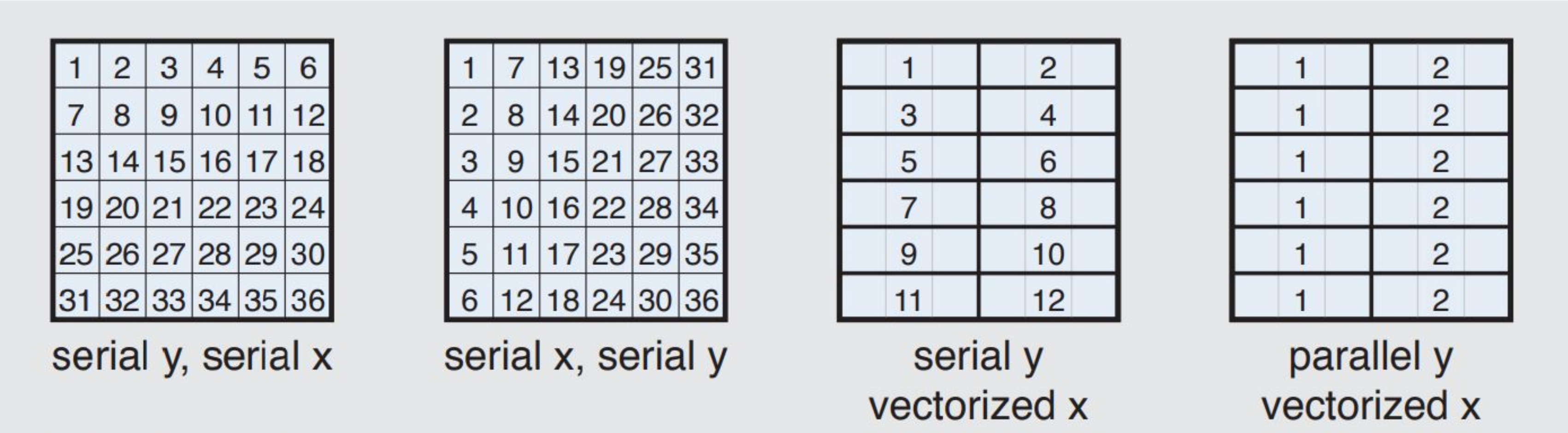
tiles: overlapping regions are processed in parallel, functions are evaluated one after another.

sliding windows within tiles: tiles are evaluated in parallel using sliding windows.

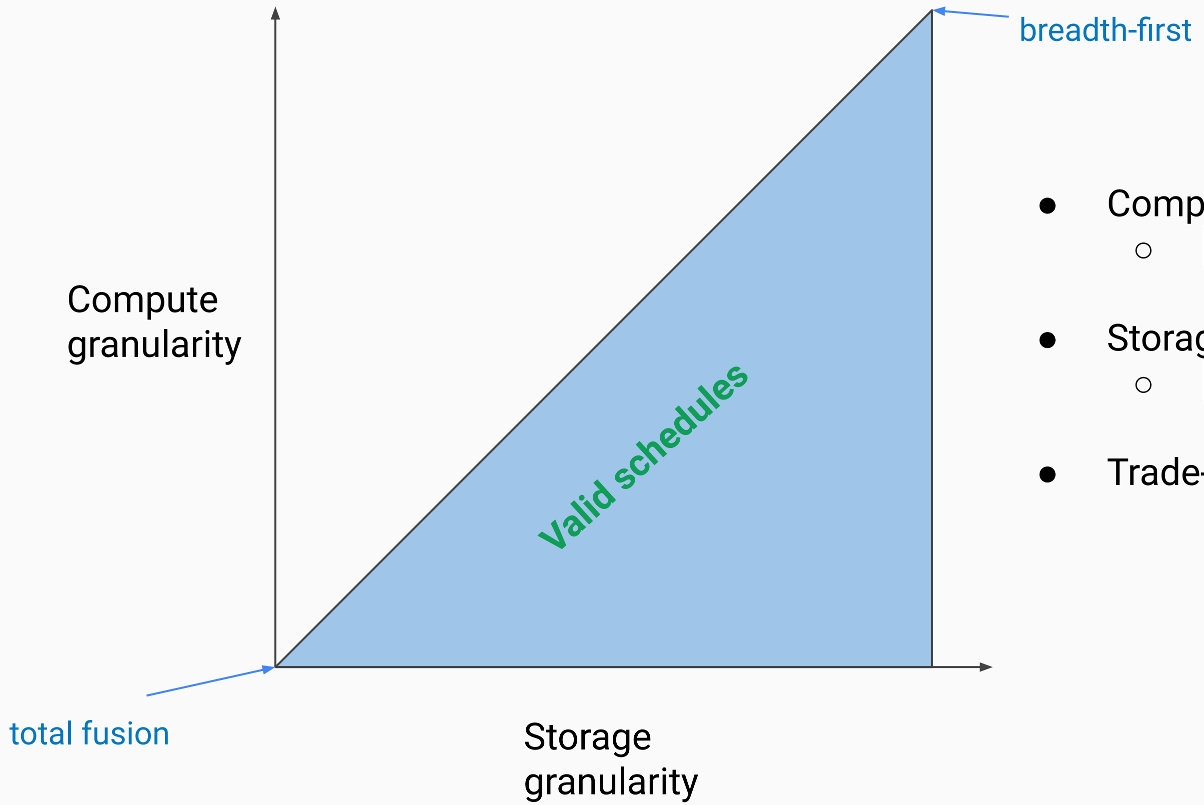
Scheduling Choice Space

Domain Order

- Order of traversal of image regions
- Traversal can be serial or parallel
- Constant size dimension → vectorized or unrolled
- Dimensions can be reordered



Scheduling Overview



- Compute Granularity
  - ↓ = more locality
- Storage Granularity
  - ↑ = more reuse
- Trade-off between the two

Autotuning Pipeline Schedules

Starting Point

1. Inline all functions with footprint of 1
2. Fully parallelized and tiled
    a. Tiled over x and y
    b. Vectorized within x
    c. Parallelized over y
3. Simply parallelized
    a. Only parallelized over y

Genetic Algorithm

- Population of 128
- Elitism, crossover, mutations, random individuals
- Schedules are validated

Compiling Scheduled Pipelines

lowering → bound inference → Sliding window optimization & storage folding → flattening → Vectorization & unrolling → Code generation

```
par for out.y_0 in 0..out.y.extent/4
    for out.x_0 in 0..out.x.extent/4
        alloc blurx[blurx.y.extent][blurx.x.extent]
        for out.y_i in 0..4
            let blurx.y.min = 4*out.y_0.min + out.y_i.min -1
            for blurx.y in blurx.y.min..blurx.y.max
                for blurx.x_0 in blurx.x.min/4..blur.x.max/4
                    blurx[blurx.y.stride*blurx.y+4*blurx.x_0+ramp(4)]
                    = in[in.y.stride*(blurx.y.min+blur.y)+4*blurx.x_0+ramp(4)]

            vec for out.x in 0..4
                out[out.y.stride*(4*(out.y_0-out.y_0.min)+out.y_i) + 4*(out.x_0-out.x_0.min)+out.x_i]
                = blurx[blurx.y.stride*(out.y_i-1-blurx.y.min)+out.x_i-blurx.x.min]
                + blurx[blurx.y.stride*(out.y_i  -blurx.y.min)+out.x_i-blurx.x.min]
                = blurx[blurx.y.stride*(out.y_i+1-blurx.y.min)+out.x_i-blurx.x.min]
```

Experiments

- **Blur**
  - 3x1 and 1x3 Filter applied to neighboring pixels
- **Bilateral Grid**
  - Smooths pixels but leaves edges
- **Camera Pipeline**
  - Raw data from camera sensors to image pixels
- **Multi-scale Interpolation**
  - Interpolating pixel values for different resolutions
- **Local Laplacian**
  - Applies laplacian filters to enhance local color contrast

Increasing Complexity

Task Results

CPU

x86

	Halide tuned (ms)	Expert tuned (ms)	Speedup	Lines Halide	Lines expert	Factor shorter
Blur	11	13	1.2×	2	35	18×
Bilateral grid	36	158	4.4×	34	122	4×
Camera pipe	14	49	3.4×	123	306	2×
Interpolate	32	54	1.7×	21	152	7×
Local Laplacian	113	189	1.7×	52	262	5×

GPU

CUDA

	Halide tuned (ms)	Expert tuned (ms)	Speedup	Lines Halide	Lines expert	Factor shorter
Bilateral grid	8.1	18	2.3×	34	370	11×
Interpolate	9.1	*54**	*5.9*×	21	*152**	7×
Local Laplacian	21	*189**	*9*×	52	*262**	5×

Autotuning Results

		Source size (MP)	Target size (MP)	Cross-tested time (ms)	Autotuned on target (ms)	Slowdown (vs target autotuned)
Low-to-High	Blur	0.3	30	13	11	1.2×
	Bilateral grid	0.3	2	35	36	0.97×
	Interpolate	0.3	2	31	32	0.97×
High-to-Low	Blur	30	0.3	1.1	0.07	16×
	Bilateral grid	2	0.3	9.6	6.7	1.4×
	Interpolate	2	0.3	9.7	5.2	1.9×

Conclusion

Strengths

- Pragmatic option with simple code but highly optimized performance
- Human testing and hand-tuned limited in testing options
- Converges in reasonable amount of time

Weaknesses

- Input resolution dependent for optimal performance
- No dependence analysis = conservative scheduling

Questions?

Extra Slide: Scheduling Overview

Halide DSL

```
UniformImage in(UInt(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1) + blurx(x,y) +
blurx(x,y+1)
```

Valid Schedule #2

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blurx[-1..1]
    for each i in -1..1:
      blurx[i]=
in[y-1+i][x-1]+in[y-1+i][x]+in[y-1+i][x+1]
      out[y][x] = blurx[0] + blurx[1] + blurx[2]
```

Compute where it's needed
(total fusion)

Extra Slide: Scheduling Overview

Halide DSL

```
UniformImage in(UInt(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1) + blurx(x,y) +
blurx(x,y+1)
```

Valid Schedule #3

```
alloc out[2046][3072]
alloc blurx[3][3072]
for each y in -1..2047:
  for each x in 0..3072:
    blurx[(y+1)%3][x] =
      in[y+1][x-1]+in[y+1][x]+in[y+1][x+1]
    if y < 1: continue
    out[y][x] = blurx[(y-1)%3][x] + blurx[y % 3][x] +
      blurx[(y+1)%3][x]
```

Compute where needed, but reuse values
(sliding window)