Lecture 5:

Efficiently Scheduling Image Processing Pipelines

Visual Computing Systems

Stanford CS348K, Spring 2022

Today's themes

- **Techniques for efficiently mapping image processing applications to multi-core CPUs and GPUs**

- **The design of programming abstractions that facilitate efficient image processing applications**

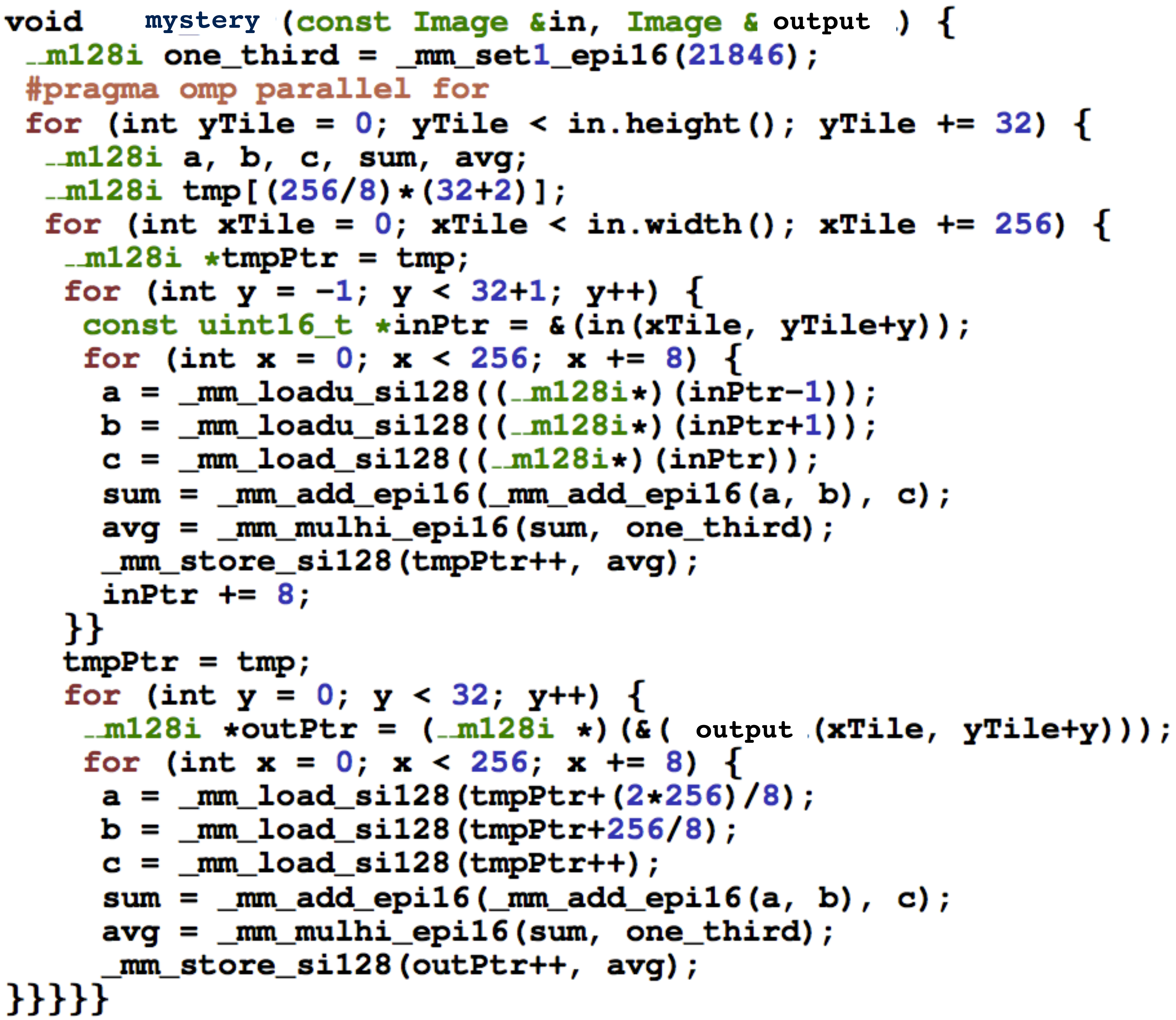
Reminder: key aspect in the design of any system:

Choosing the “right” representations for the job

- **Good representations are productive to use:**
  - Embody the natural way of thinking about a problem

- **Good representations enable the system to provide the application developer useful services:**
  - Validating/providing certain guarantees (correctness, resource bounds, conversation of quantities, type checking)
  - Performance optimizations (parallelization, vectorization, use of specialized hardware)
  - Implementations of common, difficult-to-implement functionality (texture mapping in 3D graphics, auto-differentiation)

What does this code do? 🤔😱😩😭

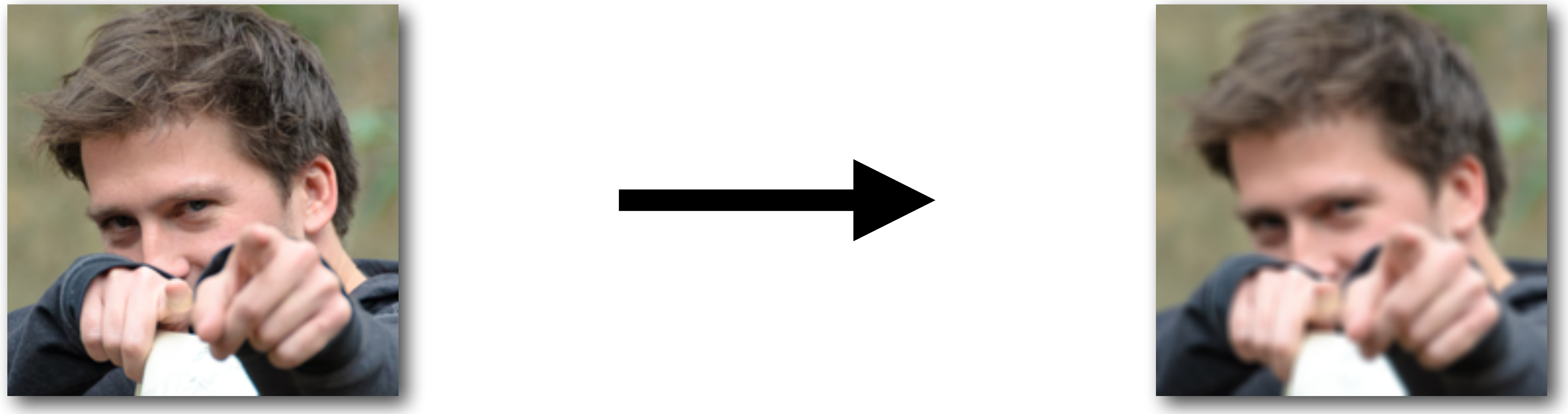


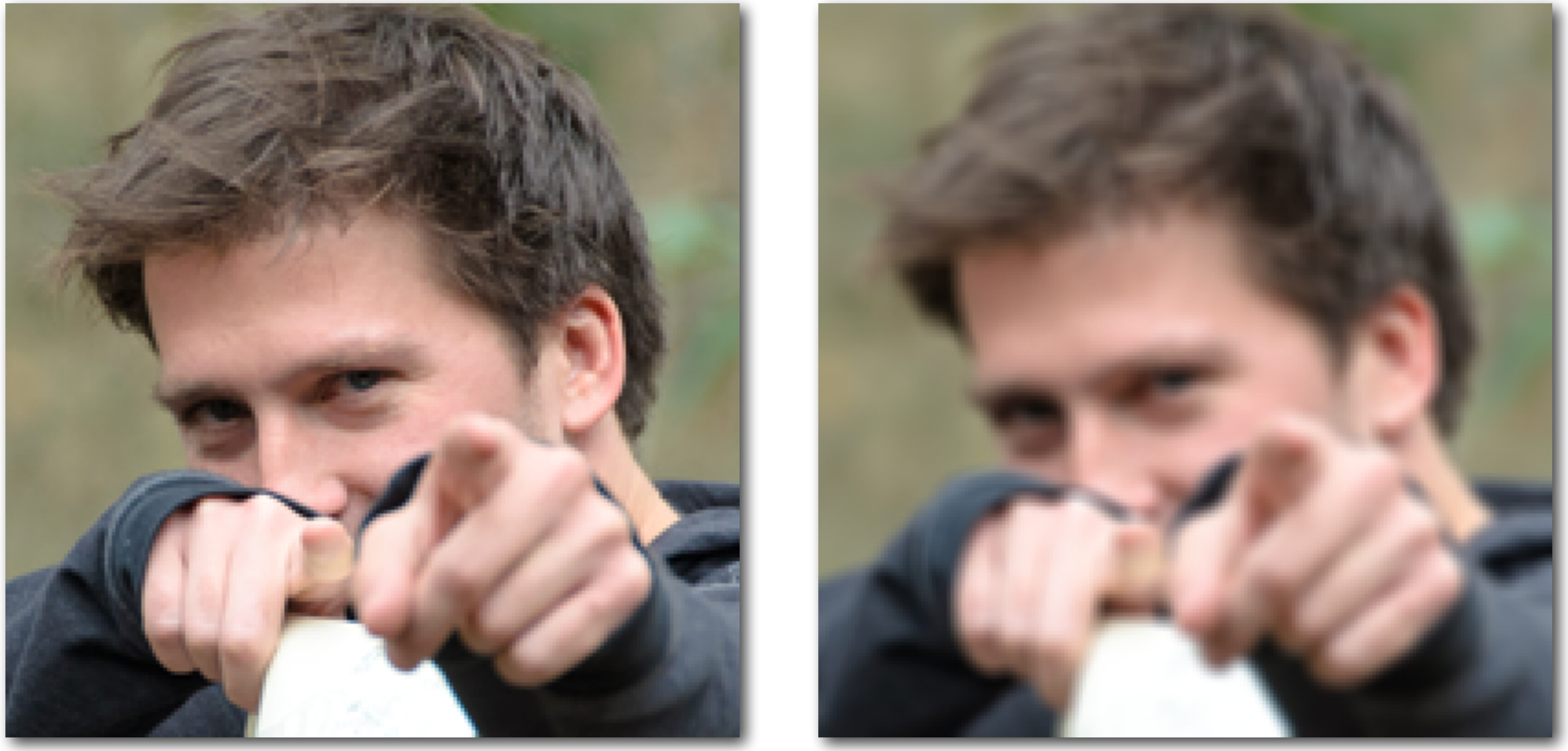
```
void    mystery (const Image &in, Image & output ) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&( output (xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

What does this C code do?

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9};

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
    output[j*WIDTH + i] = tmp;
  }
}
```

The code on the previous slide performed a 3x3 box blur



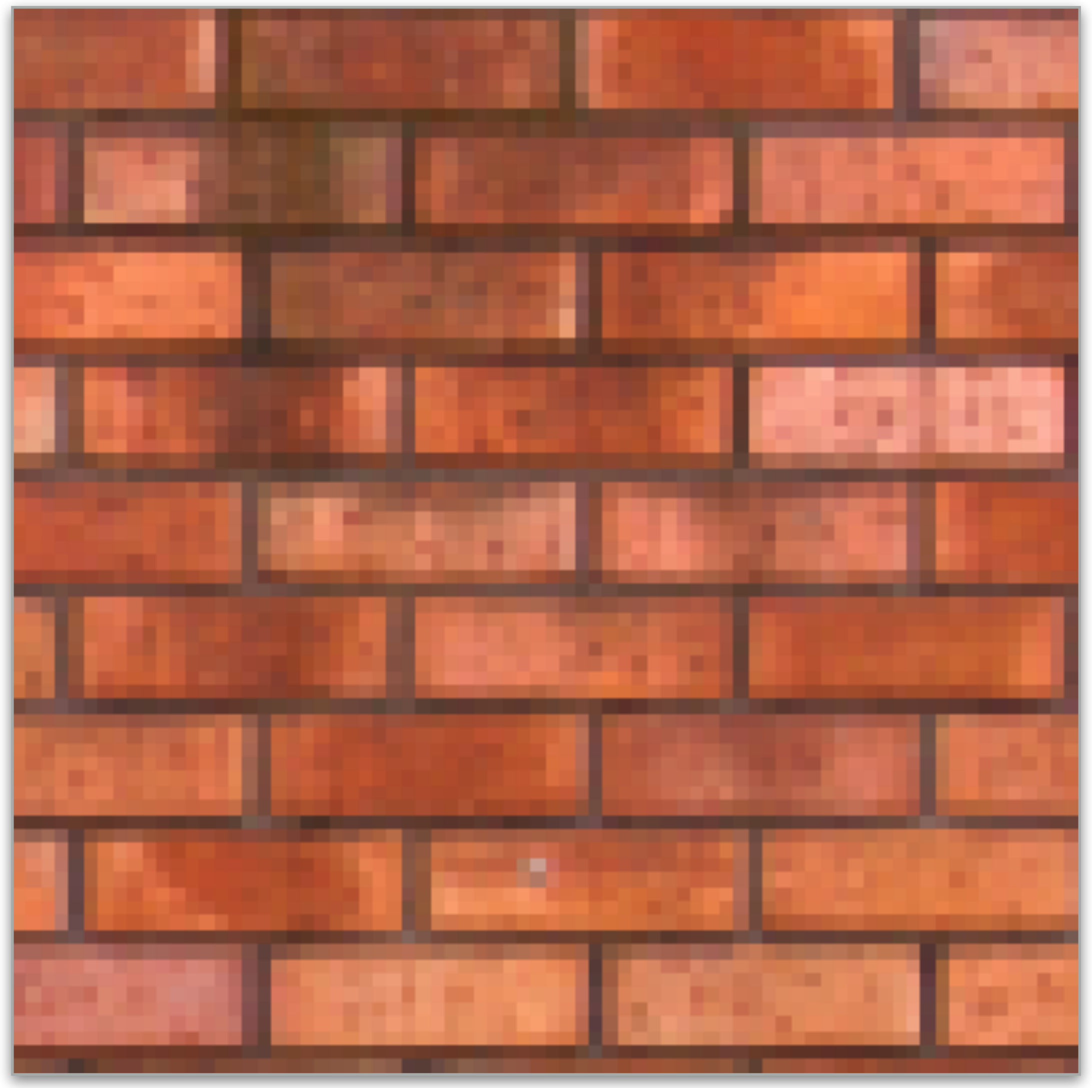


(Zoomed view)

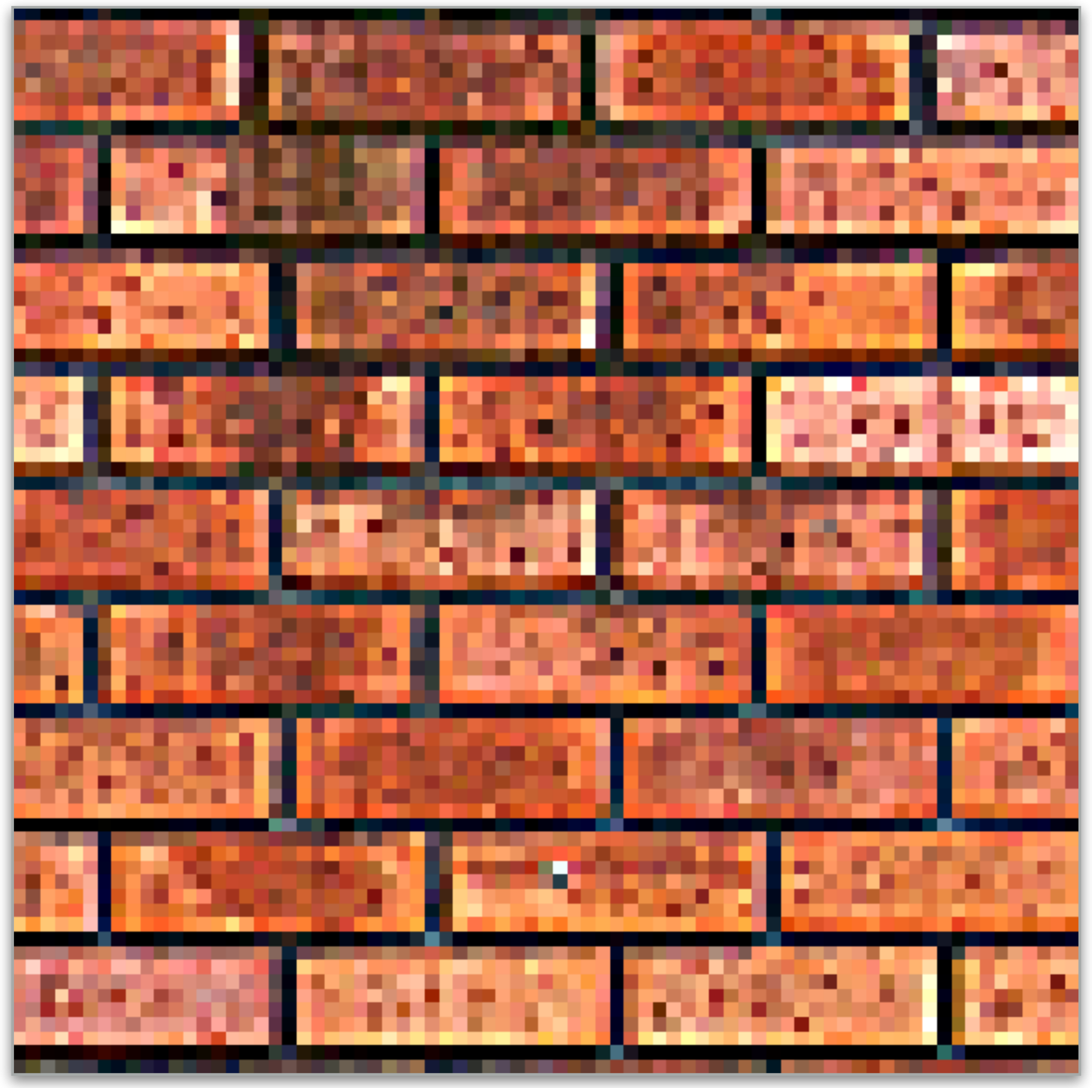
Consider a single task: sharpen an image

Example: sharpen an image

$$F = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



Input



Output

Four different representations of sharpen

1

```
Image input;
Image output = sharpen(input);
```

2

$$F=\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

```
Image input;
Image output = convolve(input, F);
```

3

```
Image input;
Image output;
output[i][j]
    = F[0][0] * input[i-1][j-1] +
      F[0][1] * input[i-1][j]   +
      F[0][2] * input[i-1][j+1] +
      F[1][0] * input[i][j-1]   +
      F[1][1] * input[i][j]     +
      ...
```

4

```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {0., -1., 0.,
                   -1., 5, -1.,
                   0., -1., 0.};

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+jj)*(WIDTH+2) + (i+ii)]
                   * weights[jj*3 + ii];
    output[j*WIDTH + i] = tmp;
  }
}
```

Image processing tasks from previous lectures

Sobel Edge Detection

$$\mathrm{G_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \mathrm{I}$$

$$\mathrm{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \mathrm{I}$$

$$\mathrm{G} = \sqrt{\mathrm{G_x}^2 + \mathrm{G_y}^2}$$

3x3 Gaussian blur

$$\mathbf{F=} \begin{bmatrix} .075 & .124 & .075 \\ .124 & .204 & .124 \\ .075 & .124 & .075 \end{bmatrix}$$

2x2 downsample (via averaging)

```
output[x][y] = (input[2x][2y]   + input[2x+1][2y] +
                input[2x][2y+1] + input[2x+1][2y+1]) / 4.f;
```

Gamma Correction

```
output[x][y] = pow(input[x][y], 0.5f);
```

LUT-based correction

```
output[x][y] = lookup_table[input[x][y]];
```

Local Pixel Clamp

```
float f(image input) {
   float min_value = min( min(input[x-1][y], input[x+1][y]),
                          min(input[x][y-1], input[x][y+1]) );
   float max_value = max( max(input[x-1][y], input[x+1][y]),
                          max(input[x][y-1], input[x][y+1]) );
output[x][y] = clamp(min_value, max_value, input[x][y]);
output[x][y] = f(input);
```

Histogram

```
bin[input[x][y]]++;
```

Let’s consider representations for authoring image processing applications

Image processing workload characteristics

- Structure: sequences (more precisely: DAGs) of operations on images
- Natural to think about algorithms in terms of their local behavior: e.g., output at pixel (x,y) is function of input pixels in neighborhood around (x,y)
- Common case: computing value of output pixel (x,y) depends on access to a bounded local “window” of input image pixels around (x,y)
- Some algorithms require data-dependent data access (e.g., data-dependent access to lookup tables)
- Upsampling/downsampling (e.g., to create image pyramids)
- Computations that reduce information across many pixels (e.g., building a histogram, computing maximum brightness pixel in an image)

Halide goals

- Expressive: facilitate intuitive expression of a broad class of image processing applications
  - e.g., all the components of a modern camera RAW pipeline

- High performance: want to generate code that efficiently utilizes the multi-core and SIMD processing resources of modern CPUs and GPUs, and is memory bandwidth efficient

Halide used in practice

- Halide used to implement camera processing pipelines on Google phones
  - HDR+, aspects of portrait mode, etc…
- Industry usage at Instagram, Adobe, etc.

Halide language

[Ragan-Kelley / Adams 2012]

Simple domain-specific language embedded in C++ for describing sequences of image processing operations

Functions map integer coordinates to values (e.g., colors of corresponding pixels)

```
Var x, y;
Func blurx, blury, bright, out;
Halide::Buffer<uint8_t> in = load_image(“myimage.jpg”);
Halide::Buffer<uint8_t> lookup = load_image(“s_curve.jpg”);  // 255-pixel 1D image

// perform 3x3 box blur in two-passes
blurx(x,y) = 1/3.f * (in(x-1,y)    + in(x,y)    + in(x+1,y));
blury(x,y) = 1/3.f * (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1));

// brighten blurred result by 25%, then clamp
bright(x,y) = min(blury(x,y) * 1.25f, 255);

// access lookup table to contrast enhance
out(x,y) = lookup(bright(x,y));

// execute pipeline to materialize values of out in range (0:800,0:600)
Halide::Buffer<uint8_t> result = out.realize(800, 600);
```

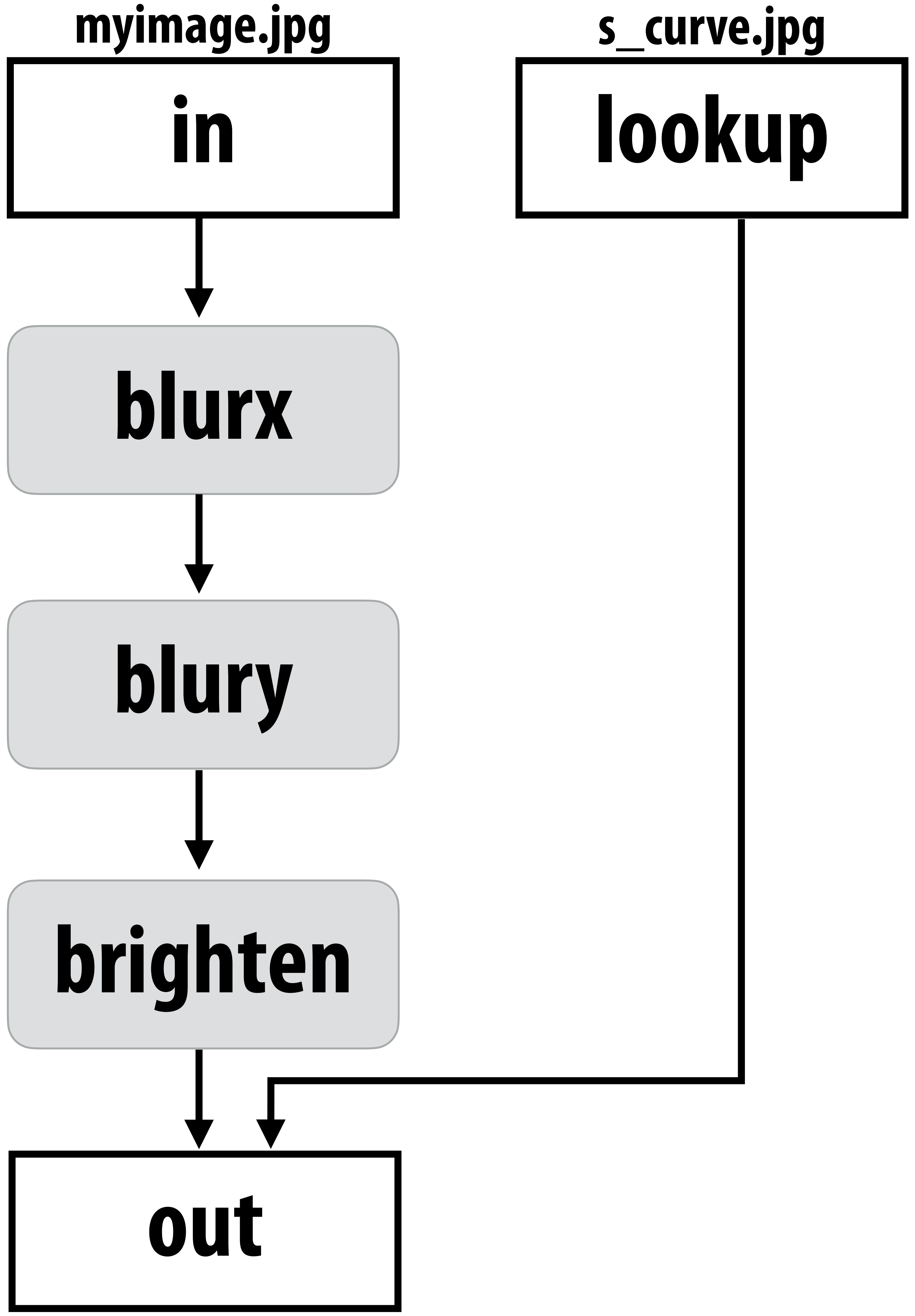
Value of `blurx` at coordinate (x,y) is given by expression accessing three values of `in`

Halide function: an infinite (but discrete) set of values defined on N-D domain

Halide expression: a side-effect free expression that describes how to compute a function’s value at a point in its domain in terms of the values of other functions.

Image processing as a DAG

Simple domain-specific language embedded in C++ for describing sequences of image processing operations



More Halide language (multi-stage functions)

```
Var x;
Func histogram, average;
Halide::buffer<uint8_t> in = load_image(“myimage.jpg”);

// declare “reduction domain” to be size of input image
RDom r(0, in.width(), 0, in.height());

/////////////////////////////////////////////////////////////////////
// compute avg of image pixels
/////////////////////////////////////////////////////////////////////

average(0) = 0;  // initialize average to 0

// “update definitions” on average: for all points in domain r do update
average(0) += in(r.x, r.y);
average(0) /= in.width() * in.height();
Halide::Buffer<uint8_t> avg_result = avg.realize(1);

/////////////////////////////////////////////////////////////////////
// Compute a histogram
/////////////////////////////////////////////////////////////////////

histogram(x) = 0;  // clear all bins of the histogram to 0

// “update definition” on histogram: for all points in domain r, increment
// appropriate histogram bin
histogram(in(r.x, r.y)) += 1;
Halide::Buffer<uint8_t> hist_result = histogram.realize(256);
```

Update definitions modify function values

Reduction domains provide the ability to iterate

Key aspects of representation

- Intuitive expression:
  - Adopts local “point wise” view of expressing algorithms
  - Halide language is declarative. It does not define order of iteration over elements in a domain, or even what values in domain are stored!
    - It only defines what operations are needed to compute these values.
    - Iteration over domain points is implicit (no explicit loops)

```
Var x, y;
Func blurx, out;
Halide::Buffer<uint8_t> in = load_image(“myimage.jpg”);

// perform 3x3 box blur in two-passes
blurx(x,y) = 1/3.f * (in(x-1,y)    + in(x,y)      + in(x+1,y));
out(x,y) =   1/3.f * (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1));

// execute pipeline on domain of size 800x600
Halide::Buffer<uint8_t> result = out.realize(800, 600);
```

Efficiently executing Halide programs

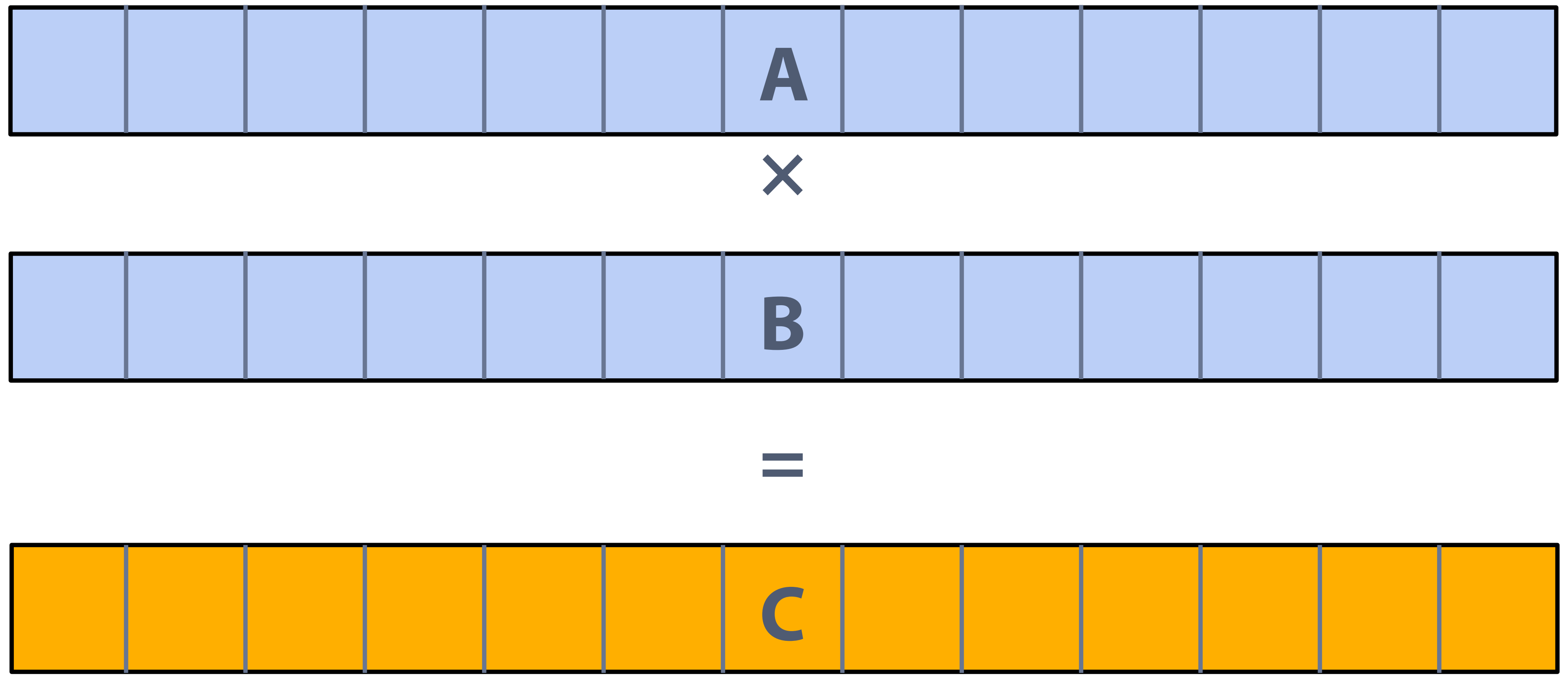
Quick review of memory bandwidth

Thought experiment

Task: element-wise multiplication of two vectors A and B

Assume vectors contain millions of elements

- Load input A[i]
- Load input B[i]
- Compute A[i] × B[i]
- Store result into C[i]



Three memory operations (12 bytes) for every MUL

NVIDIA V100 GPU can do 5120 fp32 MULs per clock (@ 1.6 GHz)

Need ~98 TB/sec of bandwidth to keep functional units busy

<1% GPU efficiency… but still 12x faster than eight-core CPU!

(3.2 GHz Xeon E5v4 eight-core CPU connected to 76 GB/sec memory bus: ~3% efficiency on this computation)

Bandwidth limited!

This computation is bandwidth limited!

If processors request data at too high a rate, the memory system cannot keep up.

Overcoming bandwidth limits is often the most important challenge facing software developers targeting modern throughput-optimized systems.

Which program performs better?

Program 1

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

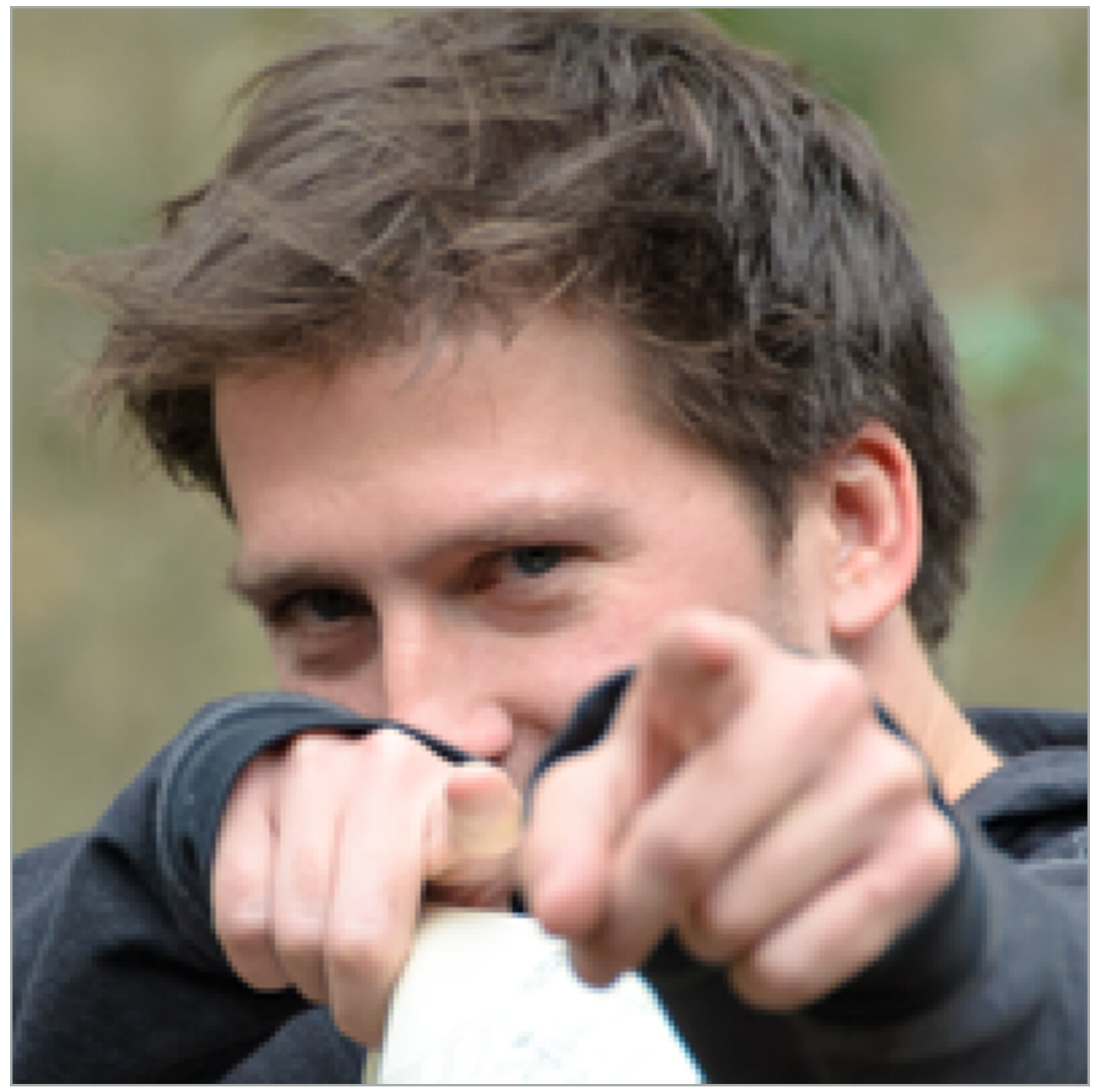
Program 2

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
       E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

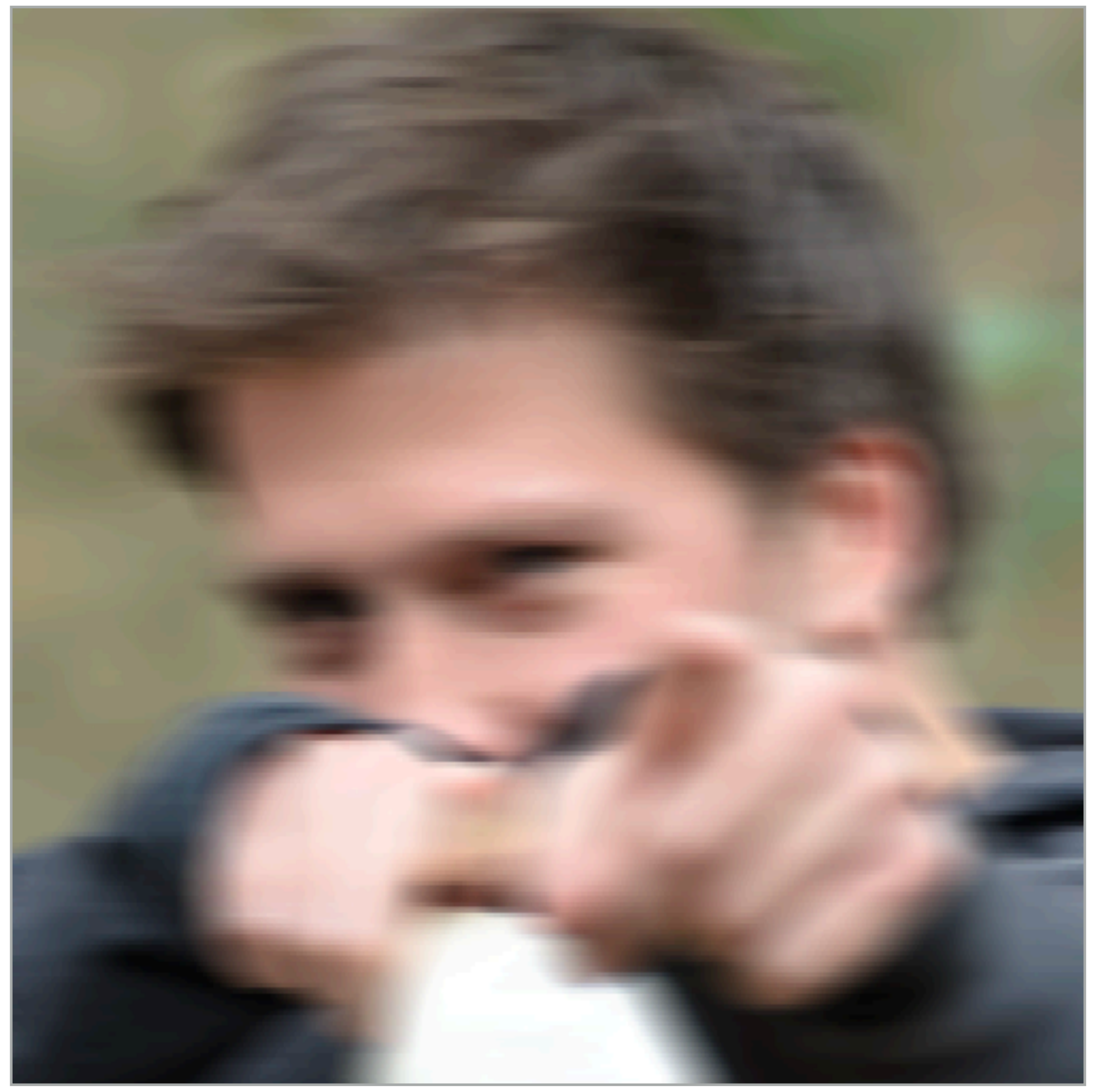
Which code structuring style would you rather write?

Two-pass blur

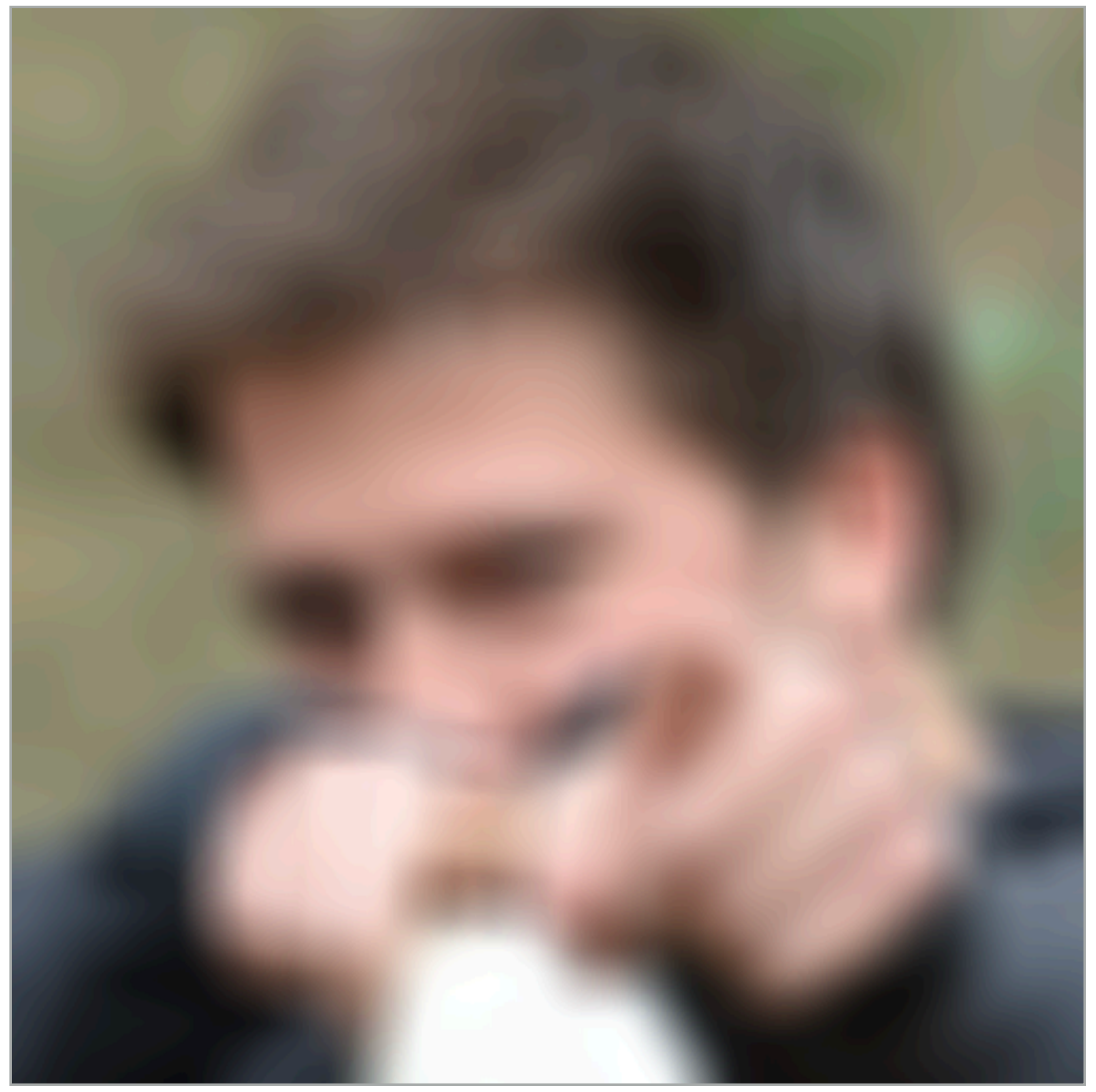
A 2D separable filter (such as a box filter) can be evaluated via two 1D filtering operations



Input



Horizontal Blur



Vertical Blur

Note: I’ve exaggerated the blur for illustration (the end result is actually a 30x30 blur, not 3x3)

Two-pass 3x3 blur

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
  }

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

1D horizontal blur

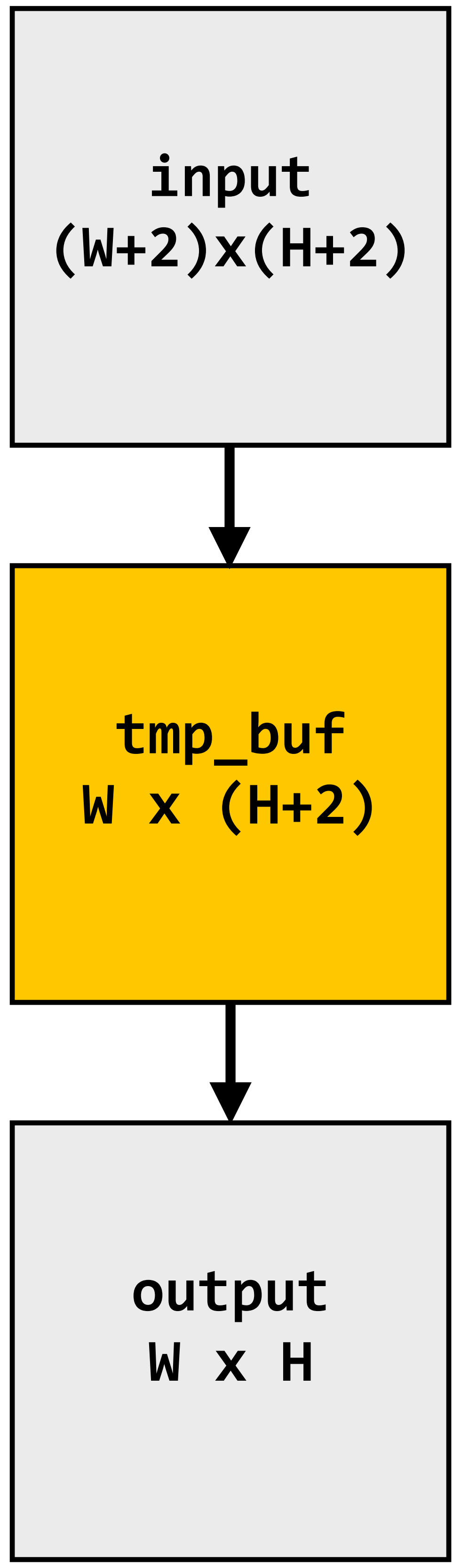
1D vertical blur

Total work per image = 6 x WIDTH x HEIGHT

For NxN filter: 2N x WIDTH x HEIGHT

WIDTH x HEIGHT extra storage

2x lower arithmetic intensity than 2D blur. Why?



Two-pass image blur: locality

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
  }

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

Intrinsic bandwidth requirements of blur algorithm: Application must read each element of input image and must write each element of output image.

Data from `input` reused three times. (immediately reused in next two i-loop iterations after first load, never loaded again.)

- **Perfect cache behavior: never load required data more than once**
- **Perfect use of cache lines (don't load unnecessary data into cache)**

Two pass: loads/stores to `tmp_buf` are overhead (this memory traffic is an artifact of the two-pass implementation: it is not intrinsic to computation being performed)

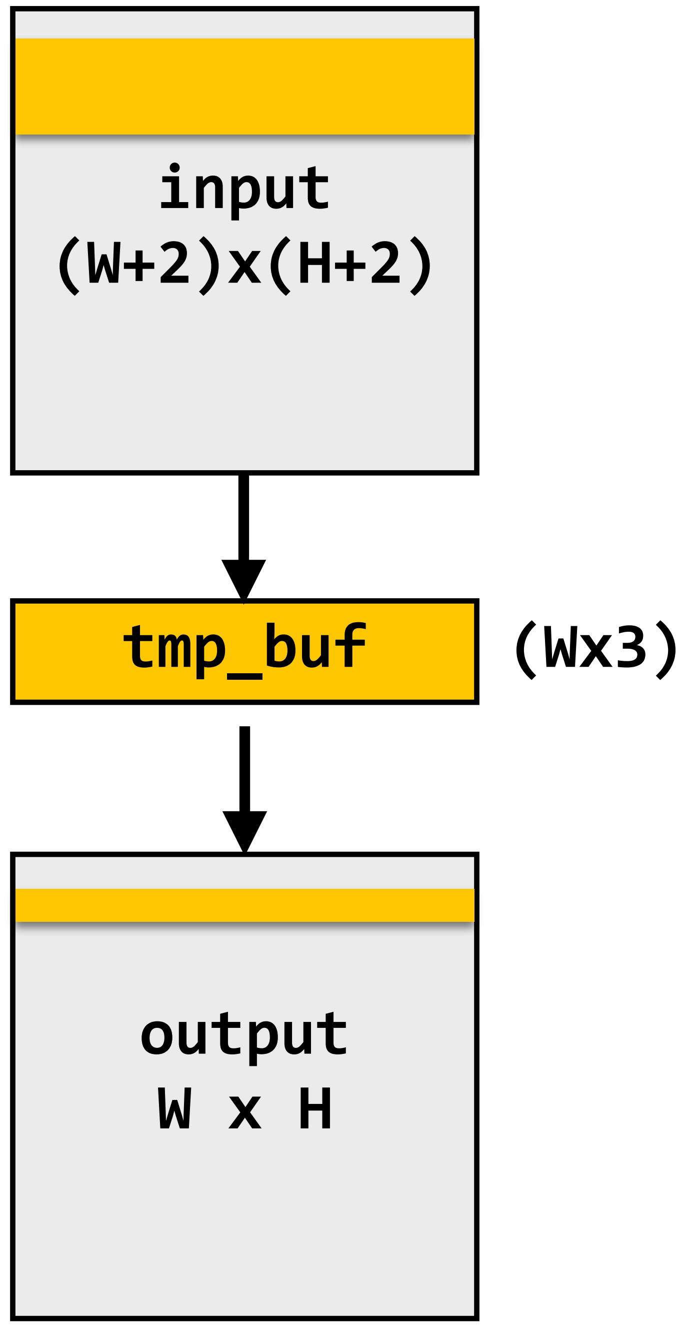
Data from `tmp_buf` reused three times (but three rows of image data are accessed in between)

- **Never load required data more than once… if cache has capacity for three rows of image**
- **Perfect use of cache lines (don't load unnecessary data into cache)**

Two-pass image blur, “chunked” (version 1)

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * 3];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<HEIGHT; j++) {

  for (int j2=0; j2<3; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
      tmp_buf[j2*WIDTH + i] = tmp;

  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[jj*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

Only 3 rows of intermediate buffer need to be allocated



Produce 3 rows of tmp_buf (only what’s needed for one row of output)

Combine them together to get one row of output

Total work per row of output:

- **step 1: 3 x 3 x WIDTH work**
- **step 2: 3 x WIDTH work**

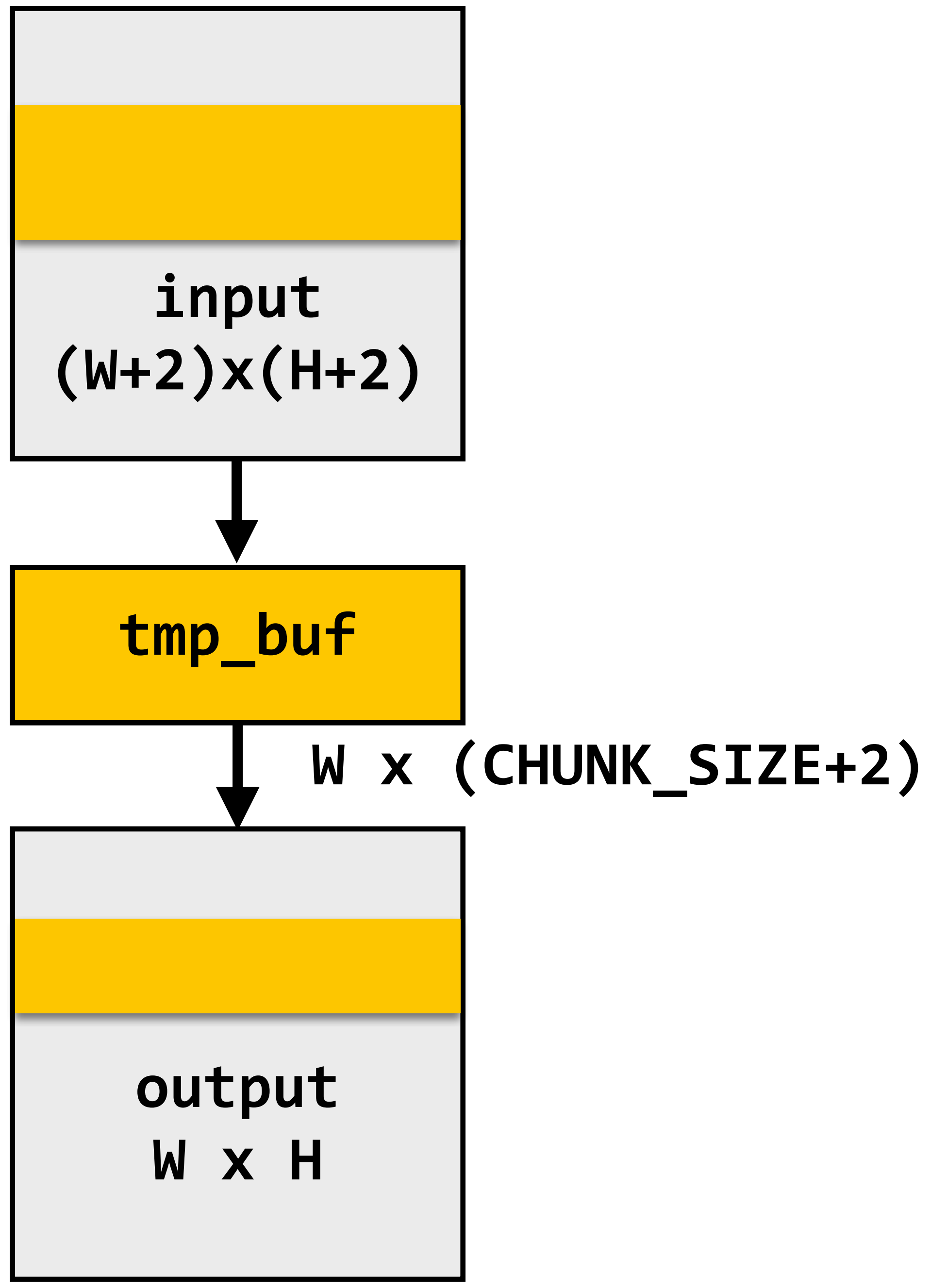
Total work per image = 12 x WIDTH x HEIGHT ????

Loads from tmp_buffer are cached (assuming tmp_buffer fits in cache)

Two-pass image blur, “chunked” (version 2)

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<HEIGHT; j+CHUNK_SIZE) {

  for (int j2=0; j2<CHUNK_SIZE+2; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
      tmp_buf[j2*WIDTH + i] = tmp;

  for (int j2=0; j2<CHUNK_SIZE; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<3; jj++)
        tmp += tmp_buf[(j2+jj)*WIDTH + i] * weights[jj];
      output[(j+j2)*WIDTH + i] = tmp;
    }
}
```

Sized so entire buffer fits in cache (capture all producer-consumer locality)



Produce enough rows of tmp_buf to produce a CHUNK_SIZE number of rows of output

Produce CHUNK_SIZE rows of output

Total work per chuck of output: (assume CHUNK_SIZE = 16)

- Step 1: 18 x 3 x WIDTH work
- Step 2: 16 x 3 x WIDTH work

Total work per image: (34/16) x 3 x WIDTH x HEIGHT

= 6.4 x WIDTH x HEIGHT

Trends to ideal value of 6 x WIDTH x HEIGHT as CHUNK_SIZE is increased!

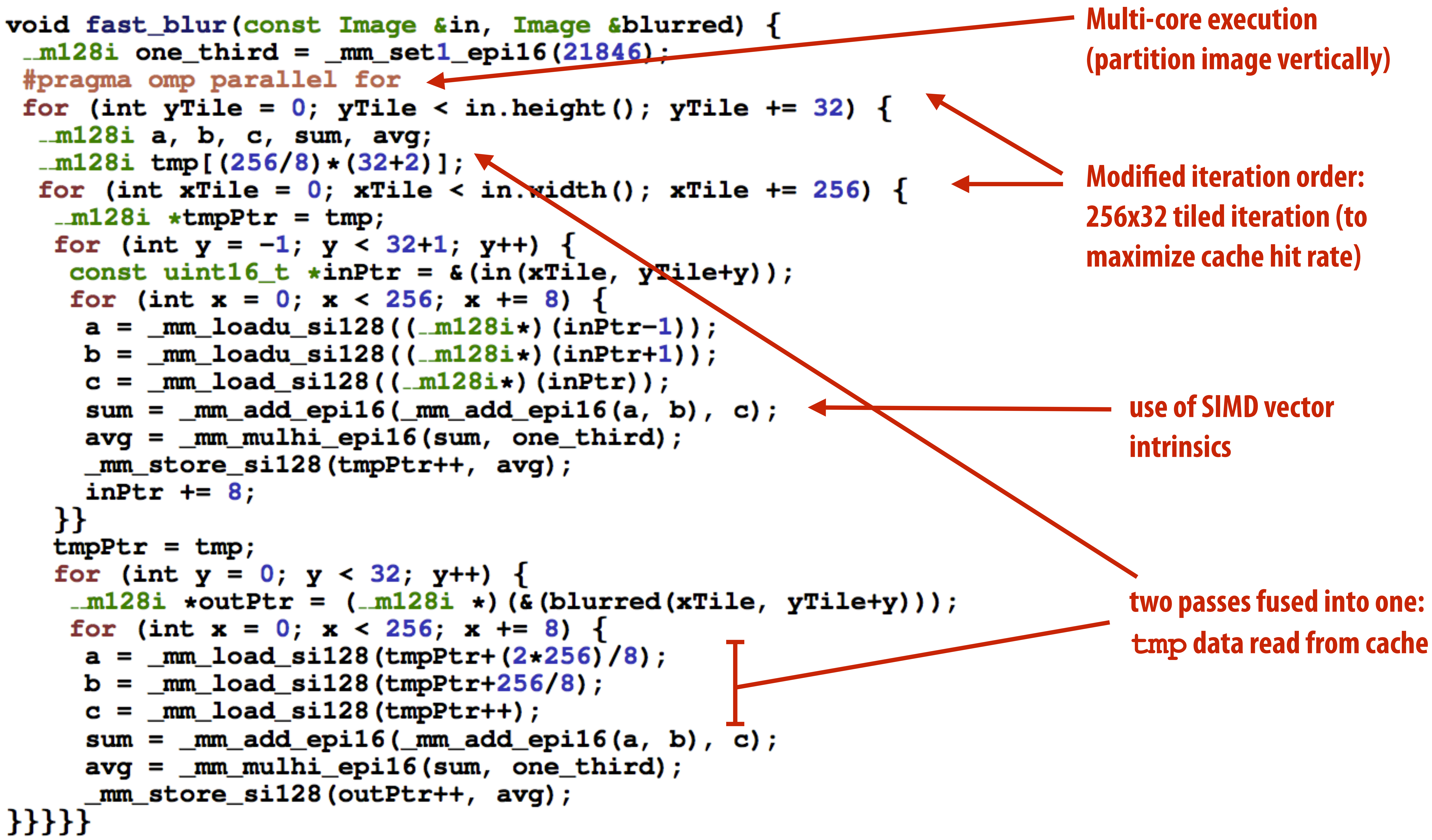
Still not done

- We have not parallelized loops for multi-core execution
- We have not used SIMD instructions to execute loops bodies
- Other basic optimizations: loop unrolling, etc…

Optimized x86 (SSE) implementation of 3x3 box blur

Good: ~10x faster on a quad-core CPU than my original two-pass code

Bad: specific to SSE (not AVX2), CPU-code only, hard to tell what is going on at all!



```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Multi-core execution (partition image vertically)

Modified iteration order: 256x32 tiled iteration (to maximize cache hit rate)

use of SIMD vector intrinsics

two passes fused into one: tmp data read from cache

Halide blur example

Consider writing code for a basic 3x3 convolution

```
Var x, y;
Func blurx, out;
Image<uint8_t> in = load_image(“myimage.jpg”);

out(x,y) = 1/9.f * (in(x-1,y-1) + in(x,y-1) + in(x+1,y-1) +
                    in(x-1,y)   + in(x,y)   + in(x+1,y) +
                    in(x-1,y+1) + in(x,y+1) + in(x+1,y+1) );

// execute pipeline on domain of size 1024x1024
Image<uint8_t> result = out.realize(1024, 1024);
```

Total work per output image = 9 x WIDTH x HEIGHT

For NxN filter: N^2 x WIDTH x HEIGHT

Halide blur example

Consider writing code for a two-pass 3x3 image blur

```
Var x, y;
Func blurx, out;
Image<uint8_t> in = load_image(“myimage.jpg”);

// perform 3x3 box blur in two-passes (box blur is separable)
blurx(x,y) = 1/3.f * (in(x-1,y)    + in(x,y)      + in(x+1,y));
out(x,y)   = 1/3.f * (blurx(x,y-1) + blurx(x,y)   + blurx(x,y+1));

// execute pipeline on domain of size 1024x1024
Image<uint8_t> result = out.realize(1024, 1024);
```

Total work per output image = 6 x WIDTH x HEIGHT

Halide language

[Ragan-Kelley / Adams 2012]

Simple domain-specific language embedded in C++ for describing sequences of image processing operations

“Functions” map integer coordinates to values (e.g., colors of corresponding pixels)

```
Var x, y;
Func blurx, blury, bright, out;
Halide::Buffer<uint8_t> in = load_image(“myimage.jpg”);
Halide::Buffer<uint8_t> lookup = load_image(“s_curve.jpg”);  // 255-pixel 1D image

// perform 3x3 box blur in two-passes
blurx(x,y) = 1/3.f * (in(x-1,y)    + in(x,y)      + in(x+1,y));
blury(x,y) = 1/3.f * (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1));

// brighten blurred result by 25%, then clamp
bright(x,y) = min(blury(x,y) * 1.25f, 255);

// access lookup table to contrast enhance
out(x,y) = lookup(bright(x,y));

// execute pipeline to materialize values of out in range (0:1024,0:1024)
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

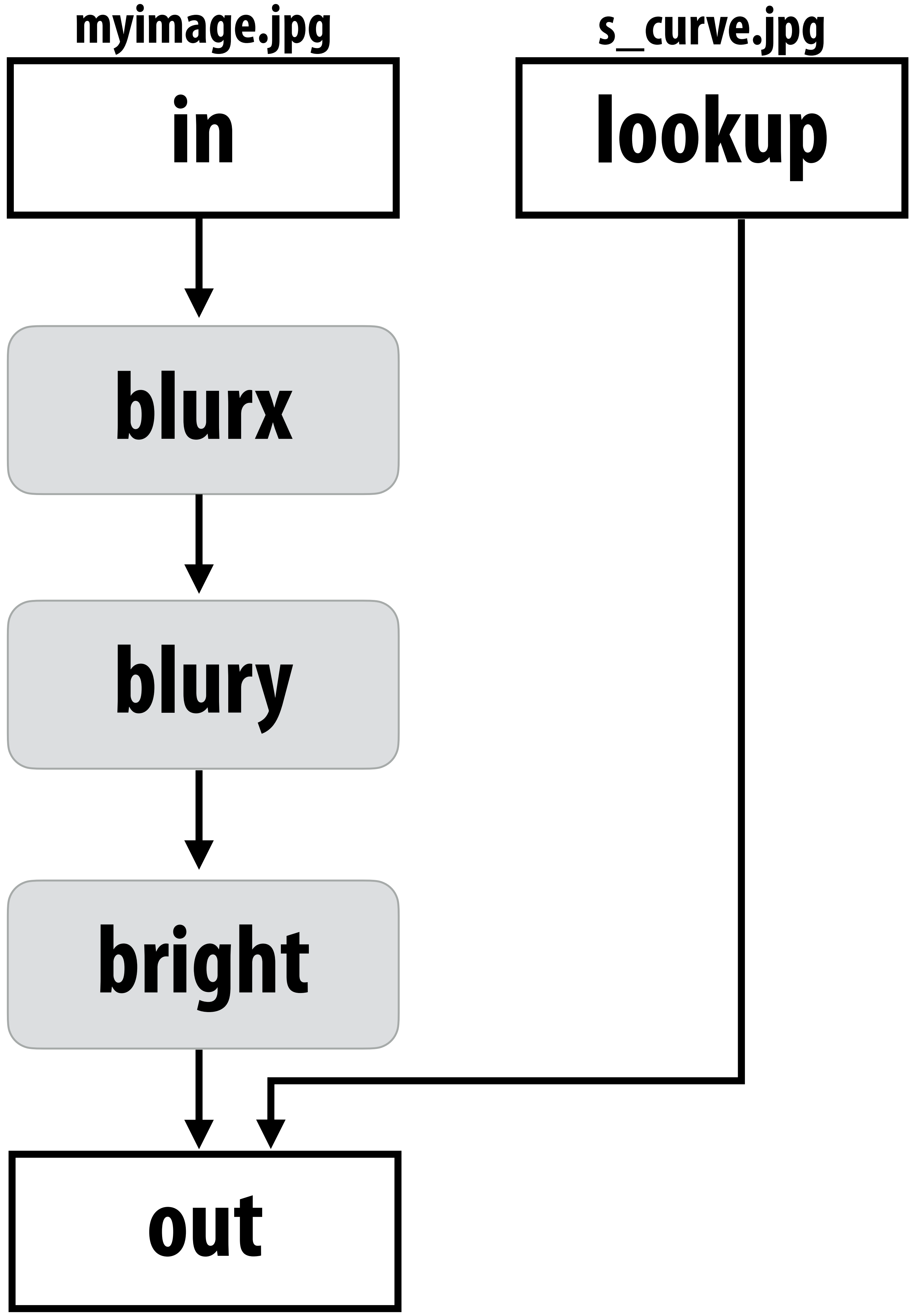
Value of `blurx` at coordinate (x,y) is given by expression accessing three values of `in`

Halide function: an infinite (but discrete) set of values defined on N-D domain

Halide expression: a side-effect free expression that describes how to compute a function’s value at a point in its domain in terms of the values of other functions.

Image processing application as a DAG

Simple domain-specific language embedded in C++ for describing sequences of image processing operations



Key aspects of representation

- Intuitive expression:
  - Adopts local "point wise" view of expressing algorithms
  - Halide language is declarative. It does not define order of iteration, or what values in domain are stored!
    - It only defines what is needed to compute these values.
    - Iteration over domain points is implicit (no explicit loops)

```
Var x, y;
Func blurx, out;
Halide::Buffer<uint8_t> in = load_image("myimage.jpg");

// perform 3x3 box blur in two-passes
blurx(x,y) = 1/3.f * (in(x-1,y)    + in(x,y)      + in(x+1,y));
out(x,y) =   1/3.f * (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1));

// execute pipeline on domain of size 1024x1024
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

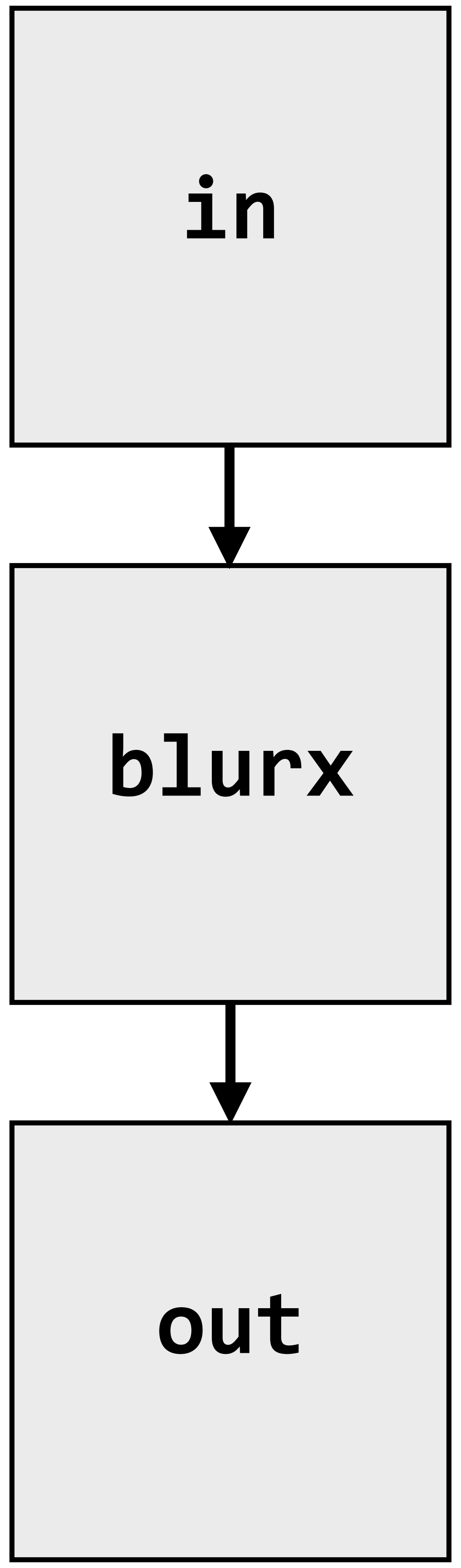


Image processing pipelines feature complex sequences of functions

Benchmark	Number of Halide functions
Two-pass blur	2
Unsharp mask	9
Harris Corner detection	13
Camera RAW processing	30
Non-local means denoising	13
Max-brightness filter	9
Multi-scale interpolation	52
Local-laplacian filter	103
Synthetic depth-of-field	74
Bilateral filter	8
Histogram equalization	7
VGG-16 deep network eval	64

Real-world production applications may features hundreds to thousands of functions!

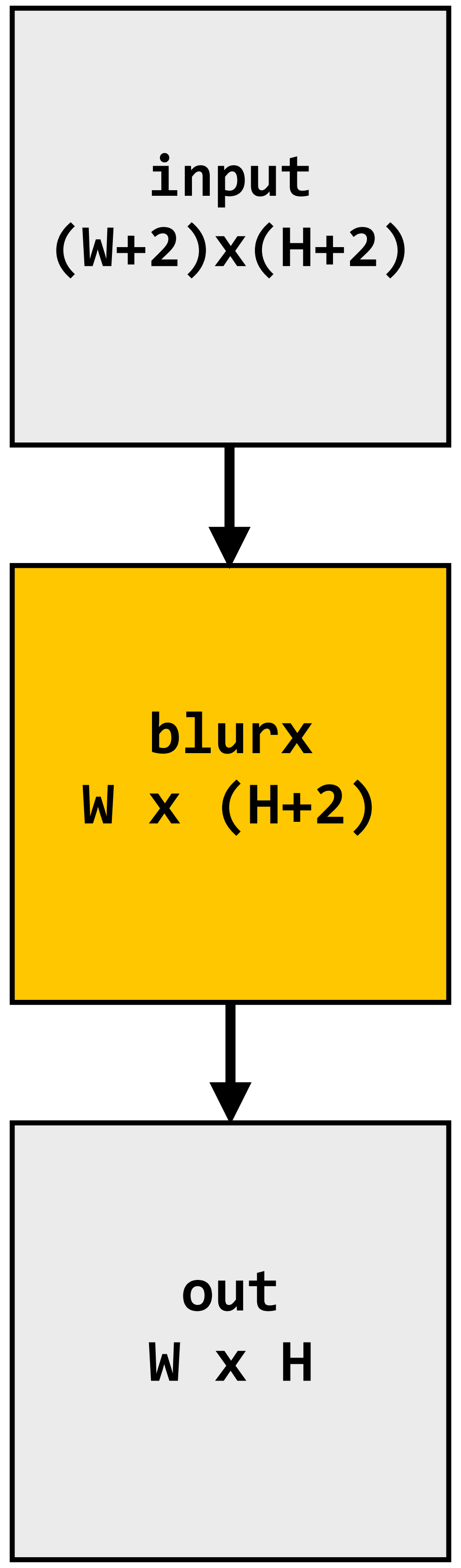
Google HDR+ pipeline: over 2000 Halide functions.

One (serial) implementation of Halide

```
Func blurx, out;
Var  x, y, xi, yi;
Halide::Buffer<uint8_t> in = load_image(“myimage.jpg”);

// the “algorithm description”  (declaration of what to do)
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

// execute pipeline on domain of size 1024x1024
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

Equivalent “C-style” loop nest:

```
allocate in(1024+2, 1024+2);   // (width,height)… initialize from image
allocate blurx(1024,1024+2);   // (width,height)
allocate out(1024,1024);       // (width,height)

for y=0 to 1024:
   for x=0 to 1024+2:
      blurx(x,y) = … compute from in

for y=0 to 1024:
   for x=0 to 1024:
      out(x,y) = … compute from blurx
```



Key aspect in the design of any system:

Choosing the “right” representations for the job

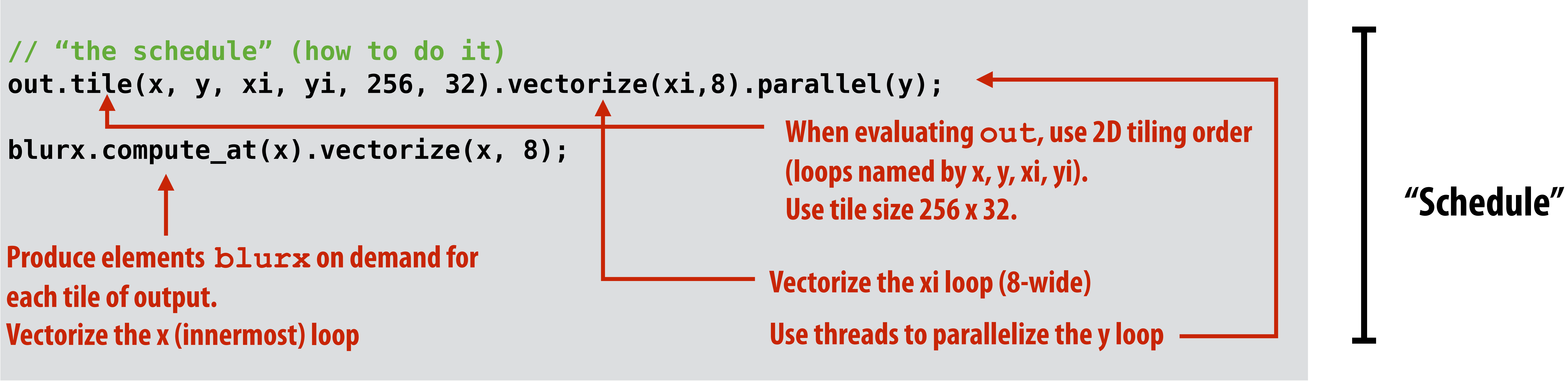
- **Good representations are productive to use:**
  - Embody the natural way of thinking about a problem

- **Good representations enable the system to provide the application useful services:**
  - Validating/providing certain guarantees (correctness, resource bounds, conversation of quantities, type checking)
  - Performance (parallelization, vectorization, use of specialized hardware)

Now the job is not expressing an image processing computation, but generating an efficient implementation of a specific Halide program.

A second set of representations for “scheduling”

```
Func blurx, out;
Var  x, y, xi, yi;
Halide::Buffer<uint8_t> in = load_image(“myimage.jpg”);

// the “algorithm description”  (declaration of what to do)
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
```



```
// “the schedule” (how to do it)
out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);

blurx.compute_at(x).vectorize(x, 8);
```

When evaluating out, use 2D tiling order (loops named by x, y, xi, yi). Use tile size 256 x 32.

Produce elements blurx on demand for each tile of output. Vectorize the x (innermost) loop

Vectorize the xi loop (8-wide)

Use threads to parallelize the y loop

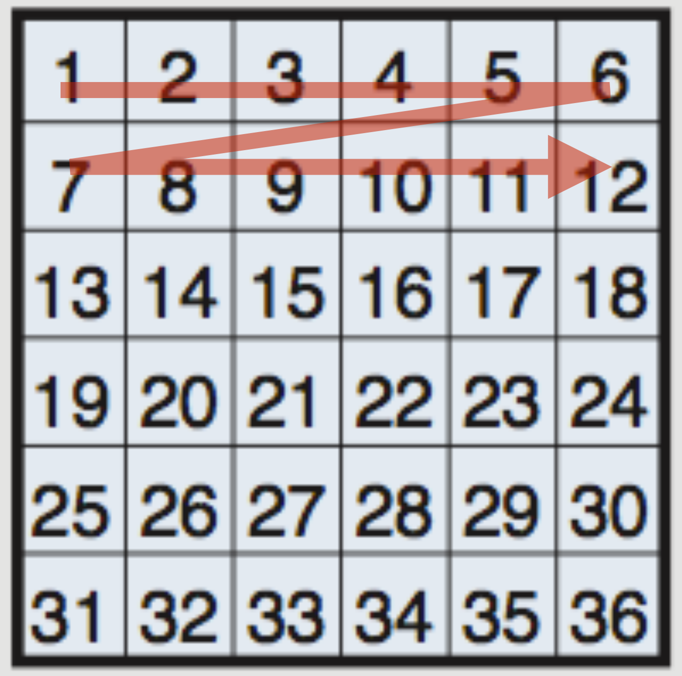
“Schedule”

```
// execute pipeline on domain of size 1024x1024
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

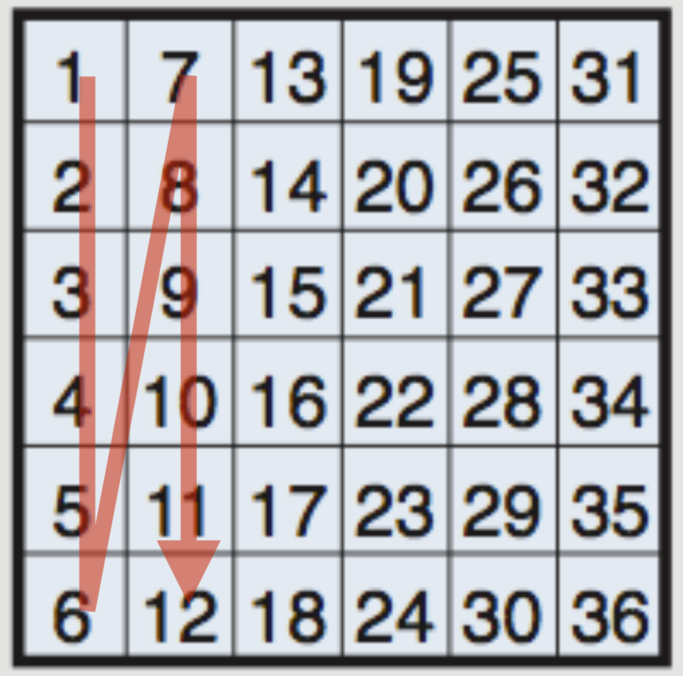
Scheduling primitives allow the programmer to specify a high-level “sketch” of how to schedule the algorithm onto a parallel machine, but leave the details of emitting the low-level platform-specific code to the Halide compiler

Primitives for iterating over N-D domains

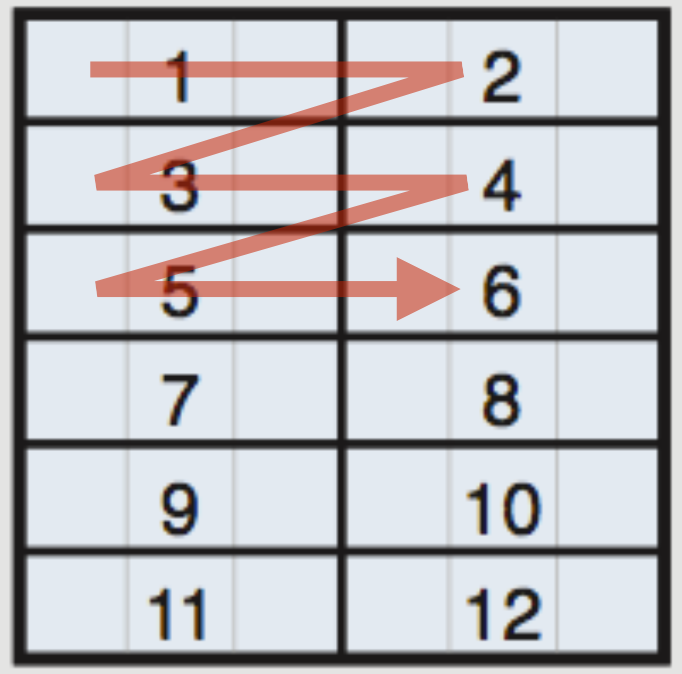
Specify both order and how to parallelize (multi-thread, vectorize via SIMD instr)



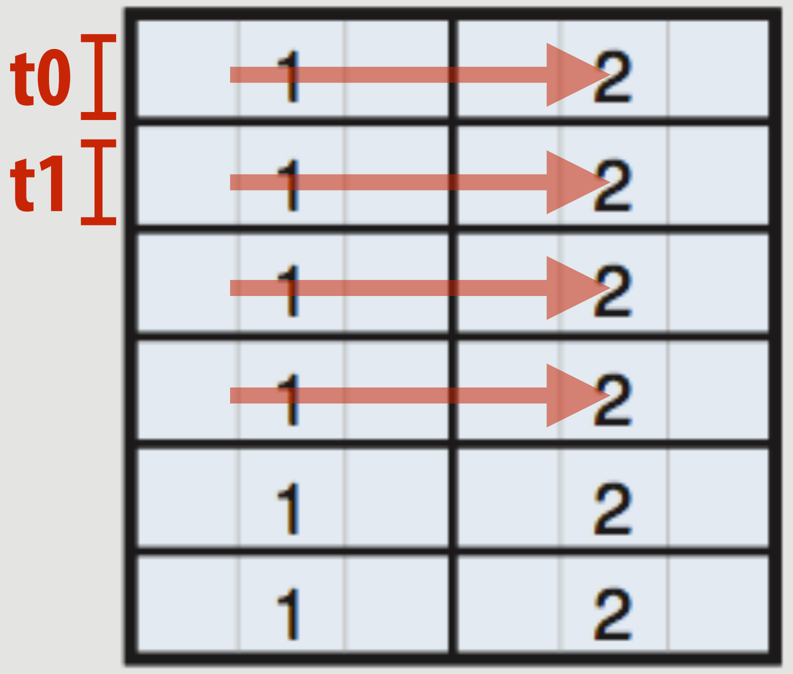
serial y, serial x



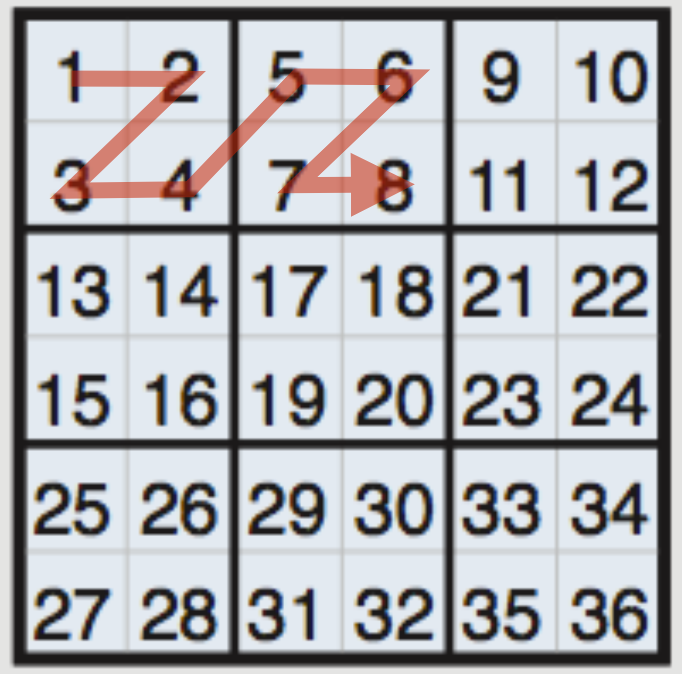
serial x, serial y



serial y
vectorized x



parallel y
vectorized x



split x into $2x_o+x_i$,
split y into $2y_o+y_i$,
serial y_o, x_o, y_i, x_i

2D blocked iteration order

(In diagram, numbers indicate sequential order of processing within a thread)

Primitives for how to interleave producer/consumer processing

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_root();
```

Do not compute blurx within out's loop nest.

Compute all of blurx, then all of out

```
allocate buffer for all of blurx(x,y)
for y=0 to HEIGHT:
  for x=0 to WIDTH:
     blurx(x,y) = …

for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:
      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_x, idx_y) = …
```

all of blurx is computed here

values of blurx consumed here

Primitives for how to interleave producer/consumer processing

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(out, xi);
```

Compute necessary elements of blurx within out's xi loop nest

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:
      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi

            allocate 3-element buffer for tmp_blurx

            // compute 3 elements of blurx needed for out(idx_x, idx_y) here
            for (blur_x=0 to 3)
                tmp_blurx(blur_x) = …

            out(idx_x, idx_y) = …
```

Note: Halide compiler performs analysis that the output of each iteration of the xi loop required 3 elements of blurx

Primitives for how to interleave producer/consumer processing

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(out, x);
```

Compute necessary elements of blurx within out's x loop nest (all necessary elements for one tile of out)

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:

      allocate 258x34 buffer for tile blurx
      for yi=0 to 32+2:
         for xi=0 to 256+2:
            tmp_blurx(xi,yi) = // compute blurx from in

      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_x, idx_y) = …
```

tile of blurx is computed here

tile of blurx is consumed here

An interesting Halide schedule

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.store_at(out, x)
blurx.compute_at(out, xi);
```

Compute necessary elements of blurx within out's xi loop nest, but fill in tile-sized buffer allocated at x loop nest.

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:

      allocate 258x34 buffer for tile tmp_blurx

      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi;

           // compute 3 elements of blurx needed for out(idx_x, idx_y) here
           for (blur_x=0 to 3)
              tmp_blurx(blur_x) = …

            out(idx_x, idx_y) = …
```

Can compiler be smarter?

"Sliding optimization" (reduces redundant computation)

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.store_at(out, x)
blurx.compute_at(out, xi);
```

Compute necessary elements of blurx within out's xi loop nest, but fill in tile-sized buffer allocated at x loop nest.

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:
      allocate 258x34 buffer for tile tmp_blurx

      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi;

            if (yi=0) {
               // compute 3 elements of blurx needed for out(idx_x, idx_y) here
               for (blur_x=0 to 3)
                 tmp_blurx(blur_x) = …
            } else
                 // only compute one additional element of tmp_blurx

            out(idx_x, idx_y) = …
```

Steady state: only one new element of tmp_blurx needs to be computed per output

“Folding optimization” (reduces intermediate storage)

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.store_at(out, x)
blurx.compute_at(out, xi);
```

Compute necessary elements of blurx within out’s xi loop nest, but fill in tile-sized buffer allocated at x loop nest.

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:
      allocate 3x256 buffer for tmp_blurx

      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi;

            if (yi=0) {
               // compute 3 elements of blurx needed for out(idx_x, idx_y) here
               for (blur_x=0 to 3)
                 tmp_blurx(blur_x) = …
            } else
                 // only compute one additional element of tmp_blurx

            out(idx_x, idx_y) = …
```

Circular buffer of 3 rows

Steady state: only one new element of tmp_blurx needs to be computed per output

Accesses to tmp_blurx modified to access appropriate row of circular buffer: e.g., ((idx_y+1)%3)

Summary of scheduling the 3x3 box blur

```
// the “algorithm description”  (declaration of what to do)
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

// “the schedule” (how to do it)
out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
blurx.compute_at(out, x).vectorize(x, 8);
```

Equivalent parallel loop nest:

```
for y=0 to num_tiles_y:   // iters of this loop are parallelized using threads
   for x=0 to num_tiles_x:
      allocate 258x34 buffer for tile blurx
      for yi=0 to 32+2:
         for xi=0 to 256+2 BY 8:
            tmp_blurx(xi,yi) = … // compute blurx from in using 8-wide
                                 // SIMD instructions here
                                 // compiler generates boundary conditions
                                 // since 256+2 isn’t evenly divided by 8
      for yi=0 to 32:
         for xi=0 to 256 BY 8:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_x, idx_y) = … // compute out from blurx using 8-wide
                                  // SIMD instructions here
```

What is the philosophy of Halide

- Programmer is responsible for describing an image processing algorithm
- Programmer has knowledge to schedule application efficiently on machine (but it's slow and tedious), so give programmer a language to express high-level scheduling decisions
  - Loop structure of code
  - Unrolling / vectorization / multi-core parallelization

- The system (Halide compiler) is not smart, it provides the service of mechanically carrying out the nitty gritty details of implementing the schedule using mechanisms available on the target machine (pthreads, AVX intrinsics, CUDA code, etc.)
  - There are deviations from this philosophy in Halide? What are they?

Constraints on language

(to enable compiler to provide desired services)

- Application domain scope: computation on regular N-D domains

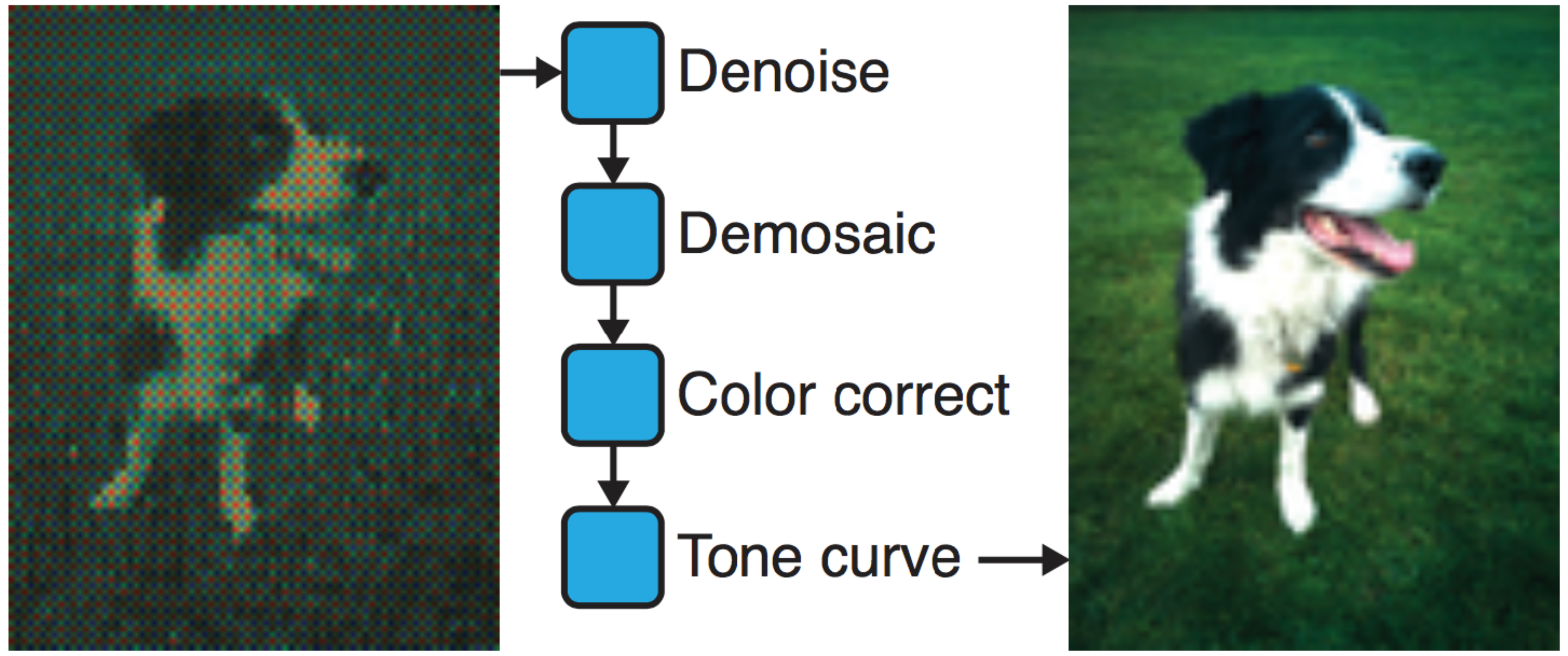
- Only feed-forward pipelines (includes special support for reductions and fixed recursion depth)

- All dependencies inferable by compiler

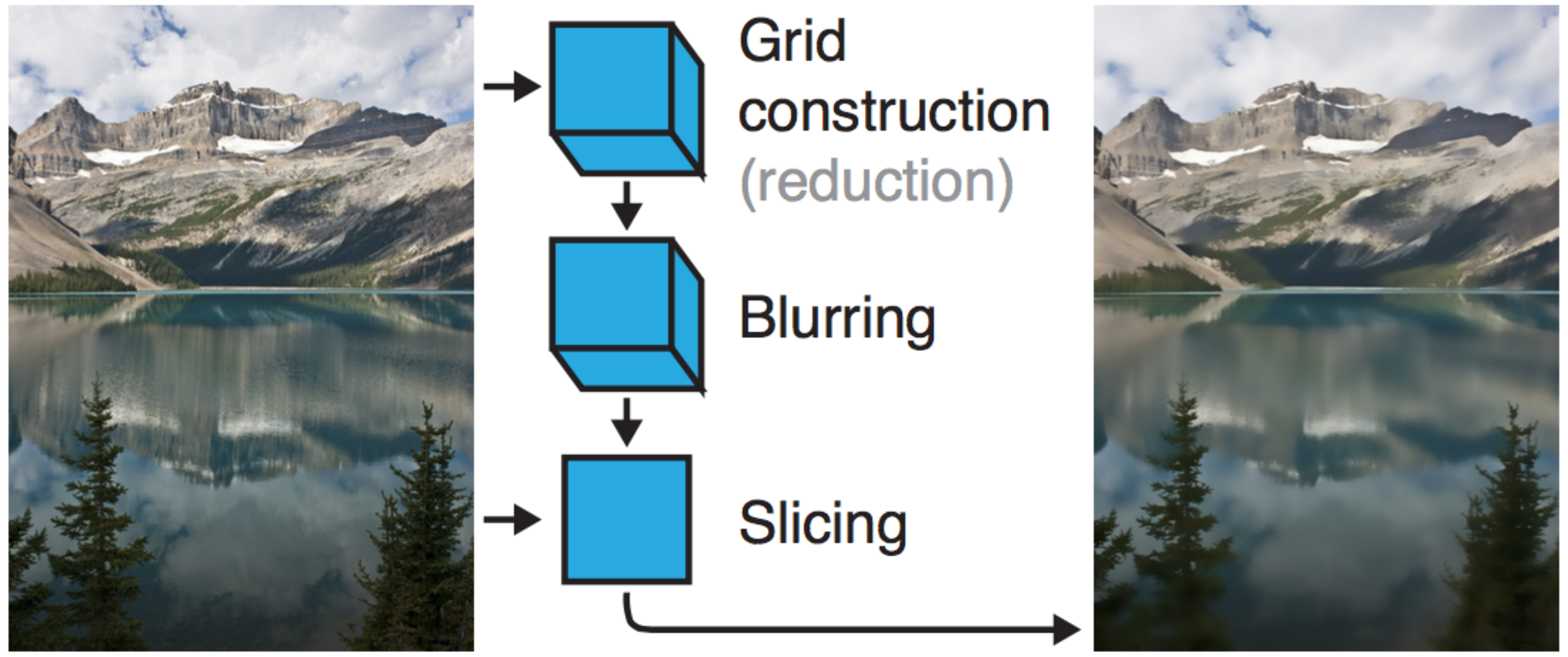
Initial academic Halide results

[Ragan-Kelley 2012]

- **Application 1: camera RAW processing pipeline**
  (Convert RAW sensor data to RGB image)
  - Original: 463 lines of hand-tuned ARM NEON assembly
  - Halide: 2.75x less code, 5% faster



- **Application 2: bilateral filter**
  (Common image filtering operation used in many applications)
  - Original 122 lines of C++
  - Halide: 34 lines algorithm + 6 lines schedule
    - CPU implementation: 5.9x faster
    - GPU implementation: 2x faster than hand-written CUDA



Stepping back: what is Halide?

- Halide is a DSL for helping expert developers optimize image processing code more rapidly
  - Halide does not decide how to optimize a program for a novice programmer
  - Halide provides a small number of primitives for a programmer (that has strong knowledge of code optimization) to rapidly express what optimizations the system should apply
    - parallel, vector, unroll, split, reorder, store_at, compute_at, etc…
  - Halide compiler carries out the mapping of that strategy to a machine

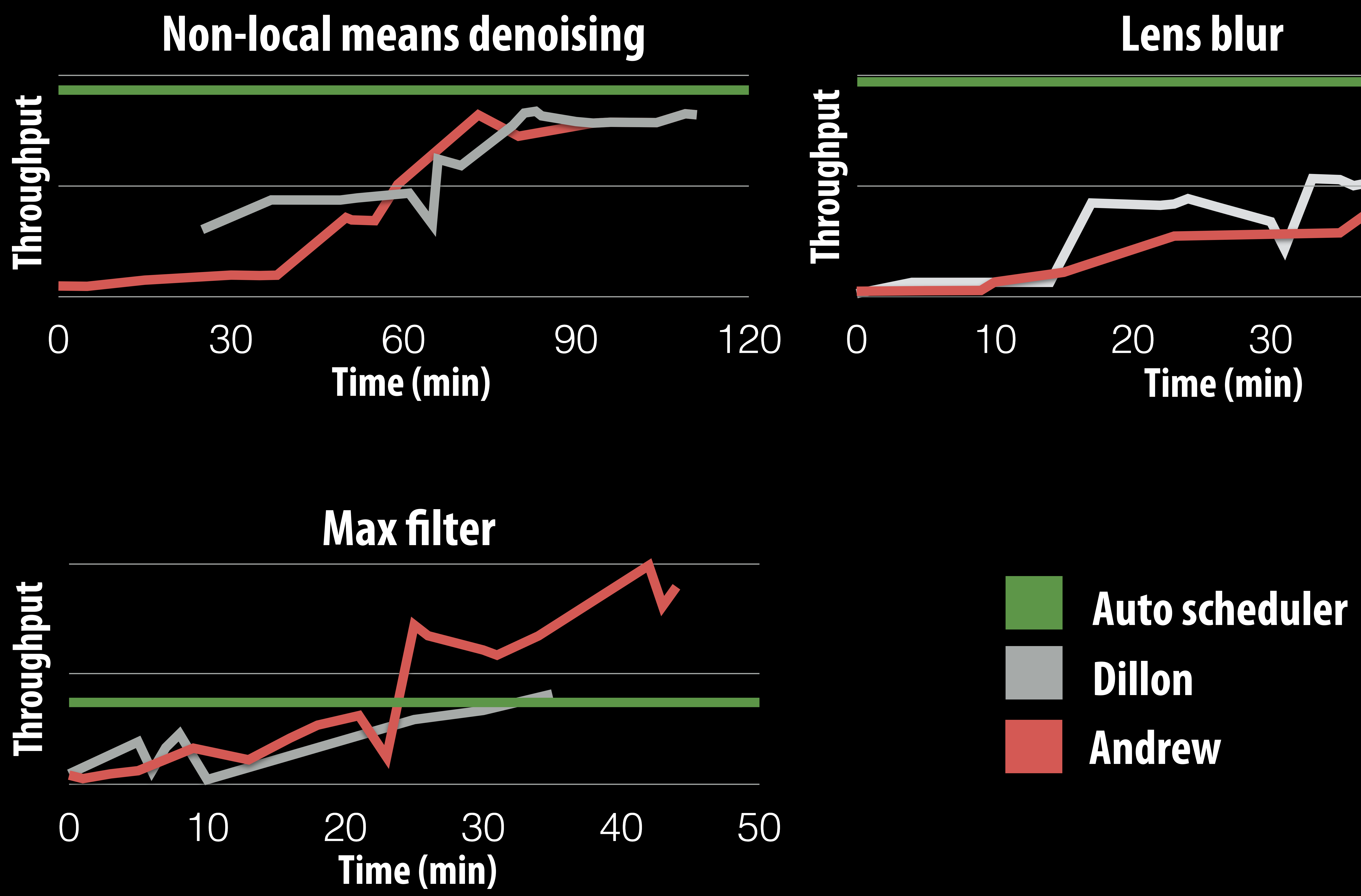
Automatically generating Halide schedules

- Problem: it turned out that very few programmers have the ability to write good Halide schedules
  - Circa 2017… 80+ programmers at Google write Halide
  - Very small number trusted to write schedules

- Recent work: compiler analyzes the Halide program to automatically generate efficient schedules for the programmer [optional reading: Mullapudi 2016, Adams 2019]
  - As of Adams 2019, you'd have to work pretty hard to manually author a schedule that is better than the schedule generated by the Halide autoscheduler

Autoscheduler saves time for experts

Early results from [Mullapudi 2016]



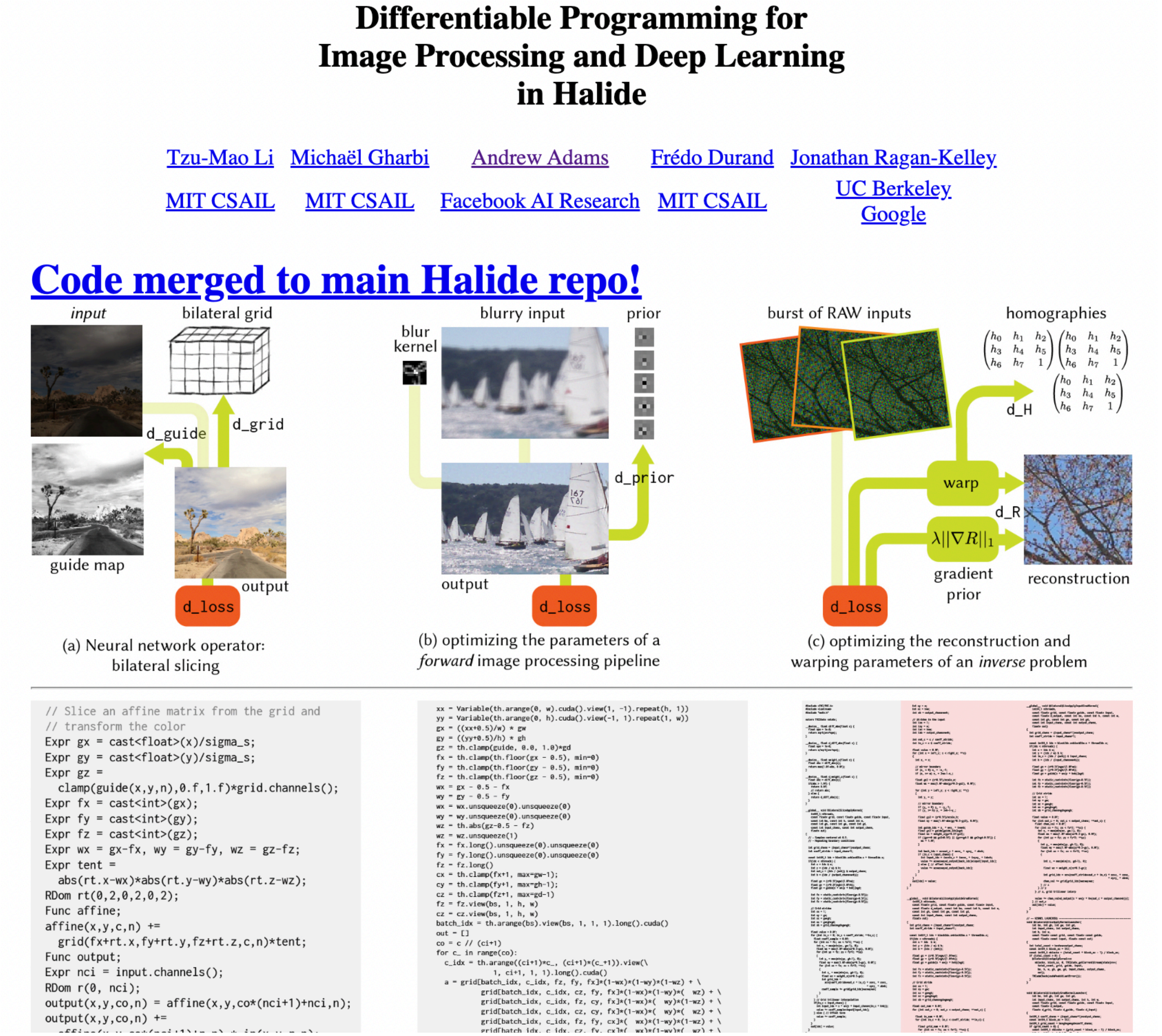
Non-local means denoising

Lens blur

Max filter

Halide extensions

[Li 2018]



Differentiable Programming for Image Processing and Deep Learning in Halide

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, Jonathan Ragan-Kelley

MIT CSAIL, MIT CSAIL, Facebook AI Research, MIT CSAIL, UC Berkeley Google

Code merged to main Halide repo!

(a) Neural network operator: bilateral slicing

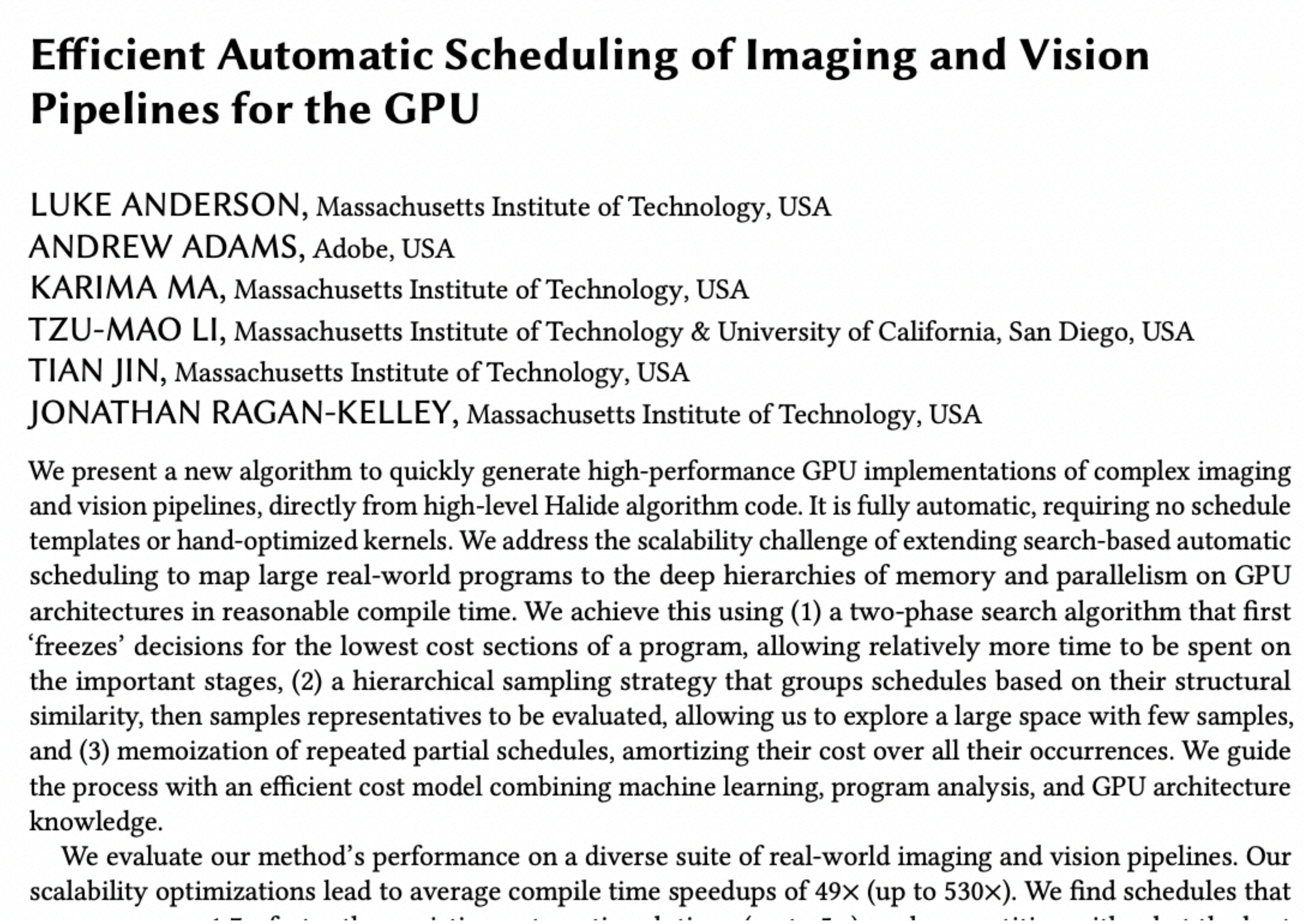
(b) optimizing the parameters of a *forward* image processing pipeline

(c) optimizing the reconstruction and warping parameters of an *inverse* problem

```
// Slice an affine matrix from the grid and
// transform the color
Expr gx = cast<float>(x)/sigma_s;
Expr gy = cast<float>(y)/sigma_s;
Expr gz =
  clamp(guide(x,y,n),0.f,1.f)*grid.channels();
Expr fx = cast<int>(gx);
Expr fy = cast<int>(gy);
Expr fz = cast<int>(gz);
Expr wx = gx-fx, wy = gy-fy, wz = gz-fz;
Expr tent =
  abs(rt.x-wx)*abs(rt.y-wy)*abs(rt.z-wz);
RDom rt(0,2,0,2,0,2);
Func affine;
affine(x,y,c,n) +=
  grid(fx+rt.x,fy+rt.y,fz+rt.z,c,n)*tent;
Func output;
Expr nci = input.channels();
RDom r(0, nci);
output(x,y,co,n) = affine(x,y,co*(nci+1)+nci,n);
output(x,y,co,n) +=
```

```
xx = Variable(th.arange(0, w).cuda().view(1, -1).repeat(h, 1))
yy = Variable(th.arange(0, h).cuda().view(-1, 1).repeat(1, w))
gx = ((xx+0.5)/w) * gw
gy = ((yy+0.5)/h) * gh
gz = th.clamp(guide, 0.0, 1.0)*gd
fx = th.clamp(th.floor(gx - 0.5), min=0)
fy = th.clamp(th.floor(gy - 0.5), min=0)
fz = th.clamp(th.floor(gz - 0.5), min=0)
wx = gx - 0.5 - fx
wy = gy - 0.5 - fy
wx = wx.unsqueeze(0).unsqueeze(0)
wy = wy.unsqueeze(0).unsqueeze(0)
wz = th.abs(gz-0.5 - fz)
wz = wz.unsqueeze(1)
fx = fx.long().unsqueeze(0).unsqueeze(0)
fy = fy.long().unsqueeze(0).unsqueeze(0)
fz = fz.long()
cx = th.clamp(fx+1, max=gw-1);
cy = th.clamp(fy+1, max=gh-1);
cz = th.clamp(fz+1, max=gd-1)
fz = fz.view(bs, 1, h, w)
cz = cz.view(bs, 1, h, w)
batch_idx = th.arange(bs).view(bs, 1, 1, 1).long().cuda()
out = []
co = c // (ci+1)
for c_ in range(co):
  c_idx = th.arange((ci+1)*c_, (ci+1)*(c_+1)).view(\
              1, ci+1, 1, 1).long().cuda()
  a = grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, fy, fx]*(1-wx)*(1-wy)*(  wz) + \
           grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, cy, fx]*(1-wx)*(  wy)*(  wz) + \
           grid[batch_idx, c_idx, fz, fy, cx]*(  wx)*(1-wy)*(1-wz) + \
```

[Anderson 2021]
Better GPU support



Efficient Automatic Scheduling of Imaging and Vision Pipelines for the GPU

LUKE ANDERSON, Massachusetts Institute of Technology, USA
ANDREW ADAMS, Adobe, USA
KARIMA MA, Massachusetts Institute of Technology, USA
TZU-MAO LI, Massachusetts Institute of Technology & University of California, San Diego, USA
TIAN JIN, Massachusetts Institute of Technology, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA

We present a new algorithm to quickly generate high-performance GPU implementations of complex imaging and vision pipelines, directly from high-level Halide algorithm code. It is fully automatic, requiring no schedule templates or hand-optimized kernels. We address the scalability challenge of extending search-based automatic scheduling to map large real-world programs to the deep hierarchies of memory and parallelism on GPU architectures in reasonable compile time. We achieve this using (1) a two-phase search algorithm that first 'freezes' decisions for the lowest cost sections of a program, allowing relatively more time to be spent on the important stages, (2) a hierarchical sampling strategy that groups schedules based on their structural similarity, then samples representatives to be evaluated, allowing us to explore a large space with few samples, and (3) memoization of repeated partial schedules, amortizing their cost over all their occurrences. We guide the process with an efficient cost model combining machine learning, program analysis, and GPU architecture knowledge.

We evaluate our method's performance on a diverse suite of real-world imaging and vision pipelines. Our scalability optimizations lead to average compile time speedups of 49× (up to 530×). We find schedules that

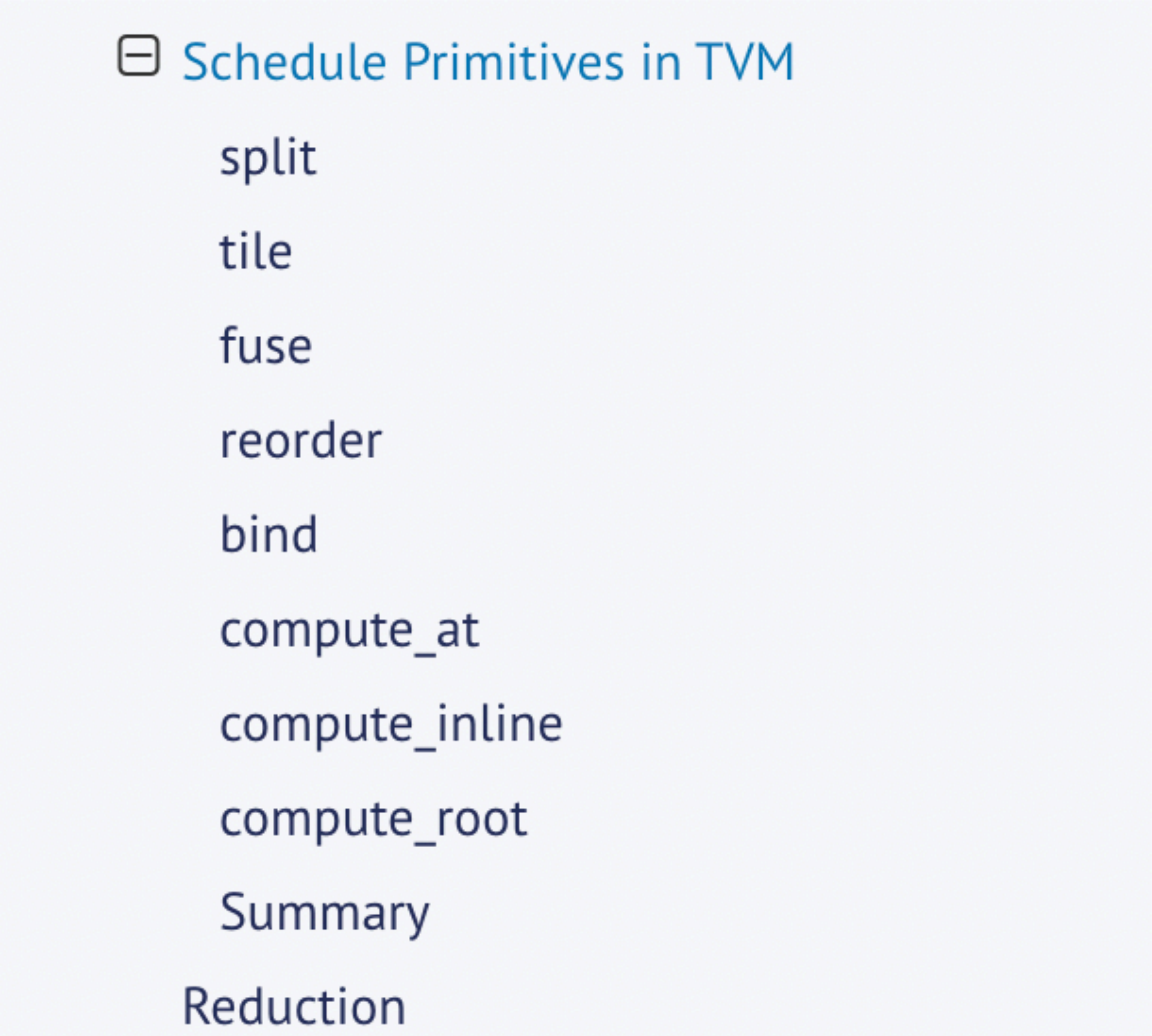
Influence on code generation for ML applications

- **Example: Apache TVM**

tvm

Apache TVM

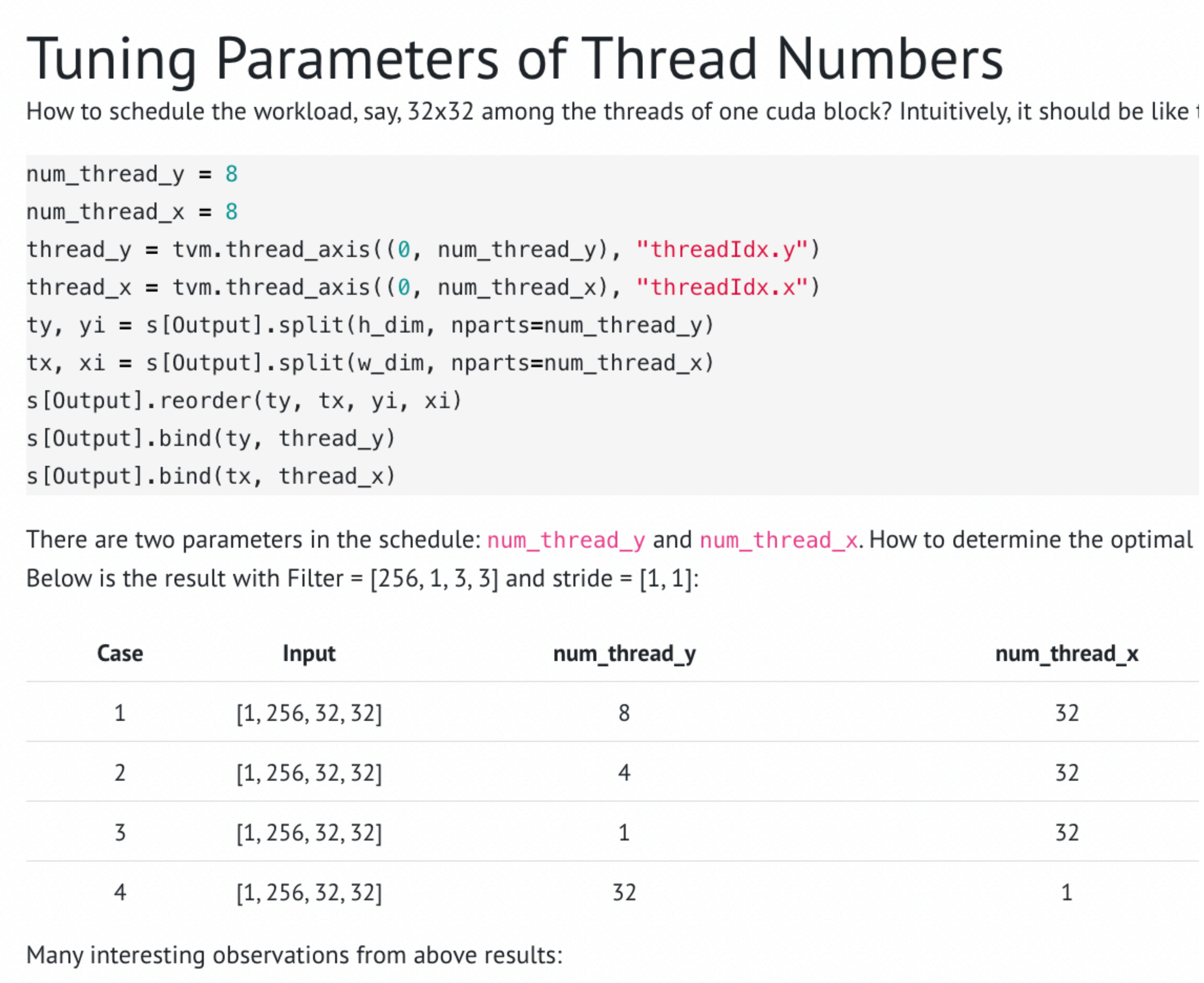
An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators



⊟ Schedule Primitives in TVM

- split
- tile
- fuse
- reorder
- bind
- compute_at
- compute_inline
- compute_root
- Summary

Reduction



Tuning Parameters of Thread Numbers

How to schedule the workload, say, 32x32 among the threads of one cuda block? Intuitively, it should be like

```
num_thread_y = 8
num_thread_x = 8
thread_y = tvm.thread_axis((0, num_thread_y), "threadIdx.y")
thread_x = tvm.thread_axis((0, num_thread_x), "threadIdx.x")
ty, yi = s[Output].split(h_dim, nparts=num_thread_y)
tx, xi = s[Output].split(w_dim, nparts=num_thread_x)
s[Output].reorder(ty, tx, yi, xi)
s[Output].bind(ty, thread_y)
s[Output].bind(tx, thread_x)
```

There are two parameters in the schedule: `num_thread_y` and `num_thread_x`. How to determine the optimal
Below is the result with Filter = [256, 1, 3, 3] and stride = [1, 1]:

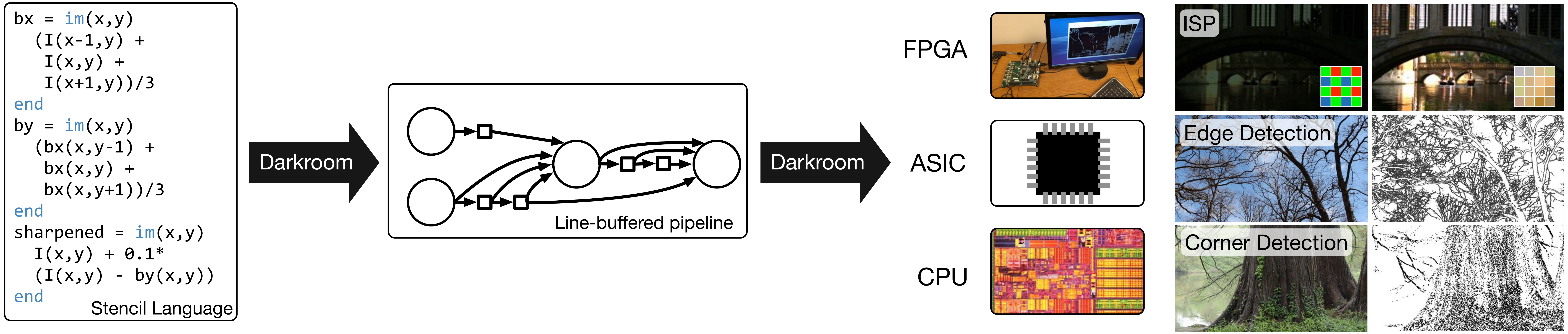
Case	Input	num_thread_y	num_thread_x
1	[1, 256, 32, 32]	8	32
2	[1, 256, 32, 32]	4	32
3	[1, 256, 32, 32]	1	32
4	[1, 256, 32, 32]	32	1

Many interesting observations from above results:

Darkroom/Rigel/Aetherling

[Hegarty 2014, Hegarty 2016, Durst 2020]

Goal: directly synthesize ASIC or FGPA implementation of image processing pipelines from a high-level algorithm description
(a constrained "Halide-like" language)



```
bx = im(x,y)
  (I(x-1,y) +
   I(x,y) +
   I(x+1,y))/3
end
by = im(x,y)
  (bx(x,y-1) +
   bx(x,y) +
   bx(x,y+1))/3
end
sharpened = im(x,y)
  I(x,y) + 0.1*
  (I(x,y) - by(x,y))
end
```

Goal: very-high efficiency image processing