Lecture 3:

Finishing up the Camera Pipeline + Frankencamera Discussion + Intro to Halide

Visual Computing Systems
Stanford CS348K, Spring 2026

Today

- Finish up description of algorithms for HDR+ pipeline (using slides from last lecture)
- Frankencamera discussion
- Modern AI-based camera pipeline features

Picking up from last time…
Finishing up the HDR+ pipeline and HDR+ paper

Frankencamera (Discussion)

Choosing the "right" representation for the job

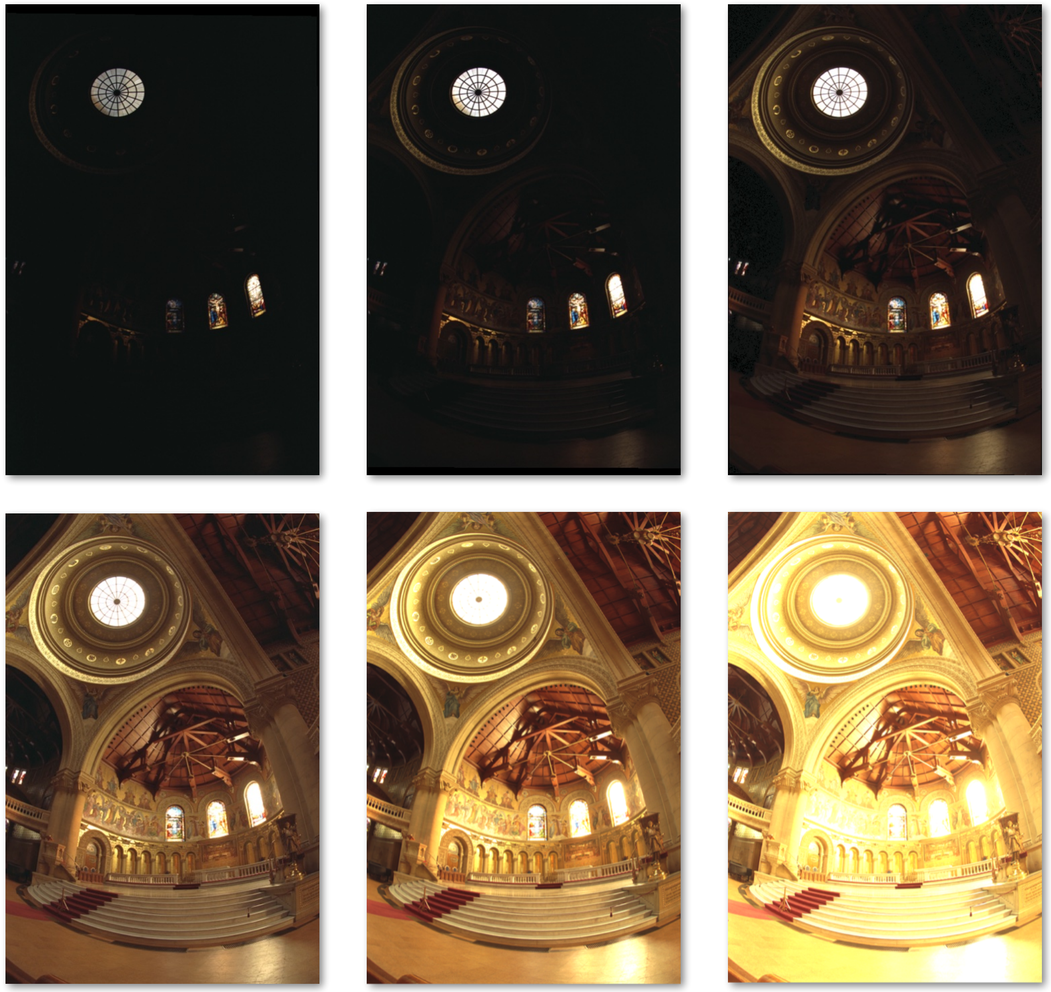
- Good representations are productive to use:
  - They embody the natural way of thinking about a problem

- 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 and rasterization in 3D graphics, auto-differentiation in ML frameworks)

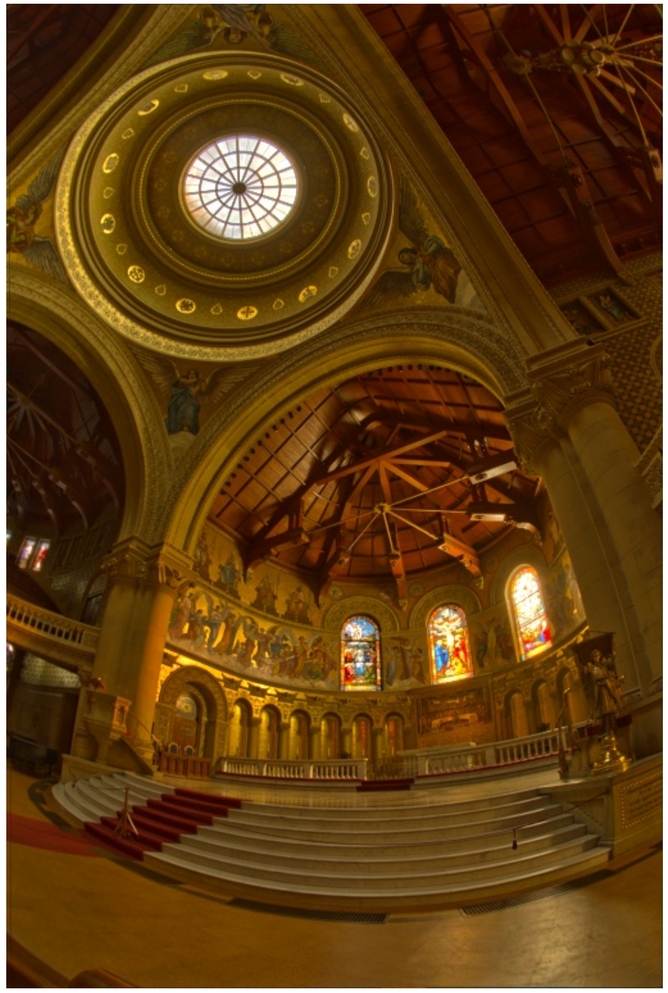
Frankencamera: some 2010 context

- Cameras were becoming increasingly cheap and ubiquitous
- Cameras featured increasing processing capability
- Large amount of computer graphics research focused on developing techniques for combining multiple photos to overcome deficiencies of traditional camera systems

Multi-shot photography example: high dynamic range (HDR) images



Source photographs: each photograph has different exposure



Tone mapped HDR image

Credit: Debevec and Malik

More multi-shot photography examples

"Lucky" imaging

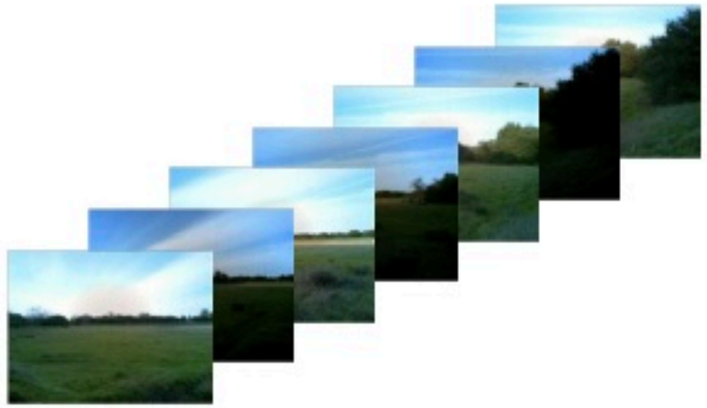
Take several photos in rapid succession:
likely to find one without camera shake

no-flash flash result

Flash-no-flash photography [Eisemann and Durand]
(use flash image for sharp, colored image, infer room lighting from no-flash image)

More multi-shot photography examples

Panorama capture



individual images

extended dynamic range panorama

Frankencamera: some 2010 context

- Cameras were becoming increasingly cheap and ubiquitous
- Cameras featured increasing processing capability
- Significant graphics research focus on developing techniques for combining multiple photos to overcome deficiencies of traditional camera systems

- Problem: the ability to implement multi-shot techniques on cameras was limited by camera system programming abstractions
  - Programmable interface to camera was very basic
  - Echoed physical button interface to a point-and-shoot camera:
    - `take_photograph(parameters, output_jpg_buffer)`
  - Result: on most camera implementations, latency between two photos was high, mitigating utility of multi-shot techniques (large scene movement or camera shake between shots)

Frankencamera (F-cam) goals

[Adams et al. 2010]

1. Create open, handheld computational camera platform for researchers

2. Define system architecture for computational photography applications
   - Motivated by impact of OpenGL on graphics application and graphics hardware development (portable apps despite highly optimized GPU implementations)
   - Motivated by proliferation of smart-phone apps

F2 Reference Implementation

Note: Apple was not involved in Frankencamera's industrial design. ;-)

Nokia N900 Smartphone Implementation

F-cam scope

- F-cam provides a set of abstractions that allow for manipulating configurable camera components
  - Timeline-based specification of actions
  - Feed-forward system: no feedback loops

- F-cam architecture performs image processing, but…
  - This functionality as presented by the architecture is not programmable
  - Hence, F-cam does not provide an image processing language (it's like fixed-function OpenGL)
  - Other than work performed by the image processing stage, F-cam applications perform their own image processing (e.g., on smartphone/camera's CPU or GPU resources)

Android Camera2 API

- Take a look at the documentation of the Android Camera2 API, and you'll see influence of F-Cam.

Image processing workload characteristics

- "Pointwise" operations
  - output_pixel = f(input_pixel)
- "Stencil" computations (e.g., convolution, demosaic, etc.)
  - Output pixel (x,y) depends on fixed-size local region of input around (x,y)
- Lookup tables
  - e.g., contrast s-curve
- Multi-resolution operations (upsampling/downsampling)
- Fast-Fourier transforms
  - We didn't talk about Fourier domain techniques in class (but Hasinoff 16 reading has many examples)
- Long pipelines of these operations

Next class: efficiently mapping these workloads to modern processors

Abstractions for authoring image processing pipelines

Choosing the "right" representation for the job (again)

- This was the theme of our Frankencamera discussion

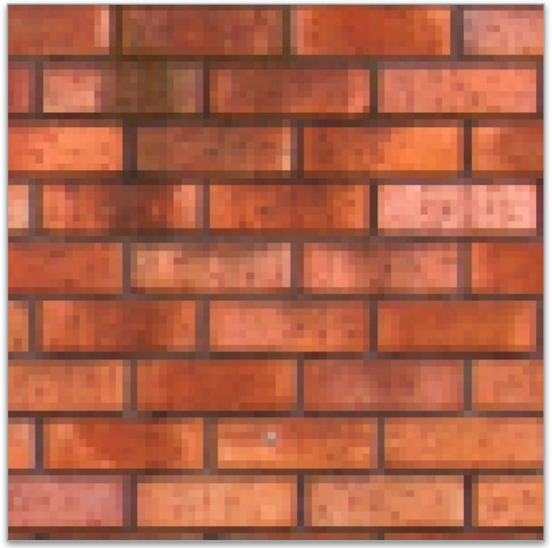
- Good representations are productive to use:
  - They embody the natural way of thinking about a problem

- 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 and rasterization in 3D graphics, auto-differentiation in ML frameworks)

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

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

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

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

3x3 Gaussian blur

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

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]]++;
```

New goals (setting up for next class)

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

[Ragan-Kelley / Adams 2012]

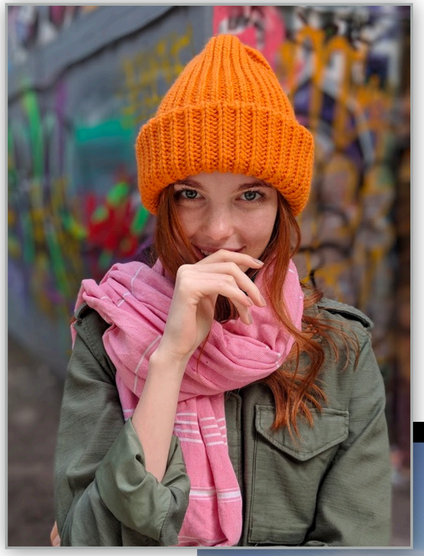
Halide goals

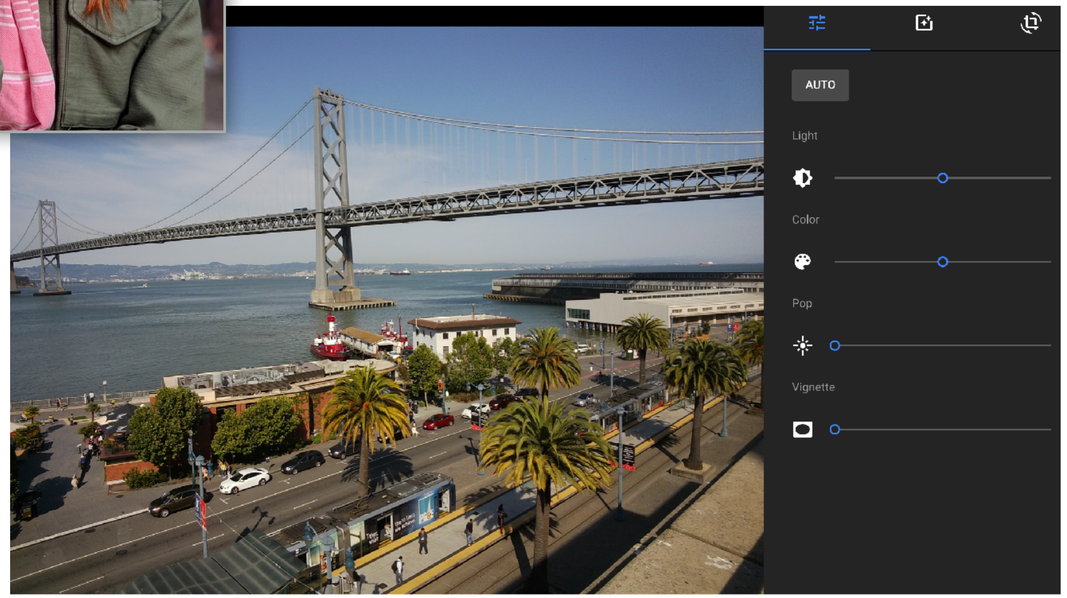
- Expressive: facilitate intuitive expression of a broad class of image processing applications
  - e.g., all the components of a modern camera RAW pipeline

- 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.





C++ code for a 3x3 “box blur”

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

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

Total work per output image =
9 x WIDTH x HEIGHT

For NxN filter: N^2 x WIDTH x HEIGHT

For now: ignore boundary pixels and assume output image is smaller than input (makes convolution loop bounds much simpler to write)

3x3 box blur in Halide

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

// expression for computing convolution result for one output pixel
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);
```

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

Total work per output image =
9 x WIDTH x HEIGHT

For NxN filter: N^2 x WIDTH x HEIGHT

Value of `blurx` at coordinate (x,y) is given by expression that accesses 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.

An example application: 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 in C++

```
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

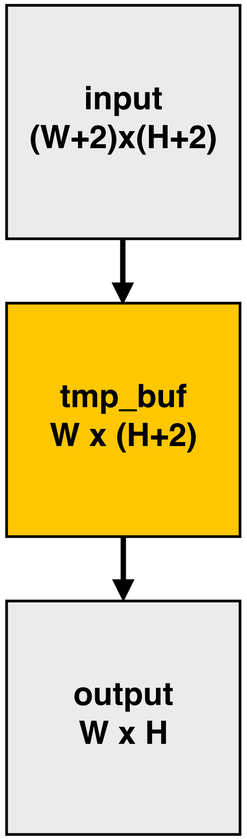
1D vertical blur

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 blur in Halide

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, 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);
```

Value of `blurx` at coordinate (x,y) is given by expression that accesses 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.

A more complicated Halide program

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

```
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);
```

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

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

Image processing as a DAG

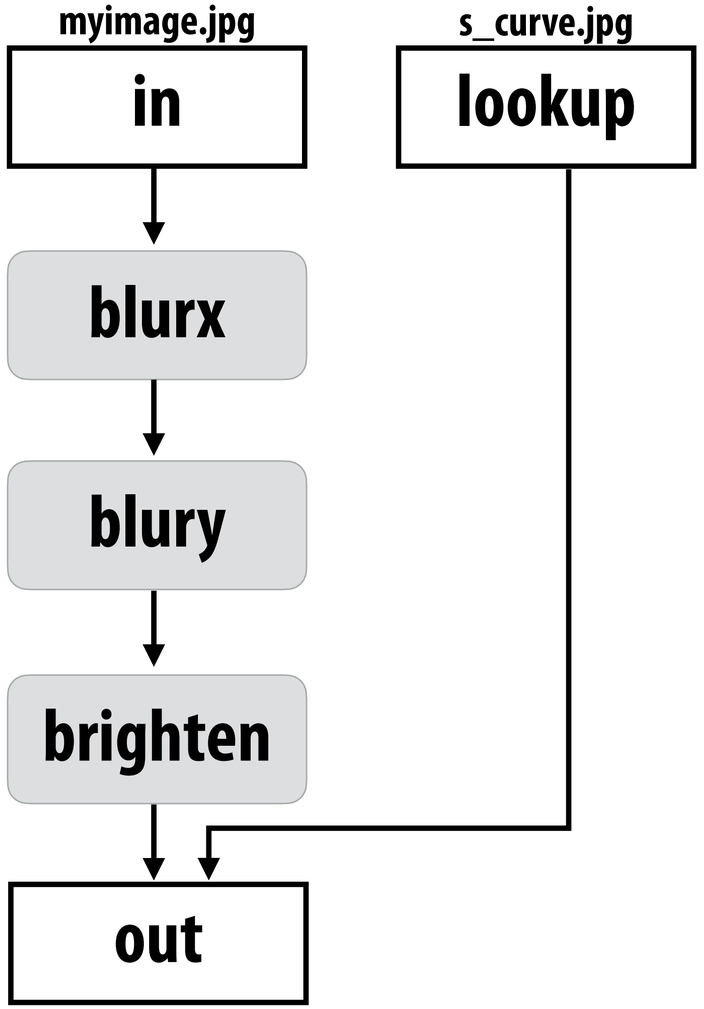


Image processing pipelines feature complex DAGs of functions

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.

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);
```

In tonight's readings, you'll learn about representations for efficiently executing Halide programs (the interesting part!)