Lecture 4:
Productive, high-performance image processing using Halide

Visual Computing Systems
CMU 15-769, Fall 2016
A Discussion of F-Cam
(last night’s reading)

[Adams 2010]
Key aspect in the design of any system: Choosing the “right” representations for the job
Frankencamera: some 2010 context

- Cameras: becoming increasingly cheap and ubiquitous
- Significant processing capability available on cameras
- Many 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

Flash-no-flash photography [Eisemann and Durand]
(use flash image for sharp, colored image, infer room lighting from no-flash image)
Frankencamera: some 2010 context

- Cameras are cheap and ubiquitous
- Significant processing capability available on cameras
- Many emerging techniques for combining multiple photos to overcome deficiencies in 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
- Influenced by physical button interface to a point-and-shoot camera:
  - `take_photograph(parameters, output_jpg_buffer)`
- Result: on most implementations, latency between two photos was high, mitigating utility of multi-shot techniques (large scene movement, camera shake, between shots)
Frankencamera goals

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

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

F2 Reference Implementation

Nokia N900 Smartphone Implementation

[Adams et al. 2010]
F-cam components

Sensor ** is really just a special case of a device
What are F-Cam’s key abstractions?
Key concept: a shot

■ A shot is a command
  - Actually it is a set of commands
  - Encapsulates both “set state” and “perform action(s)” commands

■ Defines state (configuration) for:
  - Sensor
  - Image processor
  - Relevant devices

■ Defines a timeline of actions
  - Exactly one sensor action: capture
  - Optional actions for devices
  - Note: timeline extends beyond length of exposure (“frame time”)
Key concept: a shot

- Interesting analogy (for graphics people)
  - An F-cam shot is similar to an OpenGL display list
  - A shot is really a series of commands (both action commands and state manipulation commands)
    - State manipulation commands specify the entire state of the system
    - But a shot defines precise timing of the commands in a shot (no OpenGL analogy for this)
Key concept: a frame

- A frame describes the **result** of a shot

- A frame contains:
  - Reference to corresponding image buffer
  - Statistics for image (computed by image processor)
  - Shot configuration data (what was specified by application)
  - Actual configuration data (configuration actually used when acquiring image)
    - This may be different than shot configuration data
Question

F-cam tries to address a problem in conventional camera interface designs: was this a problem of throughput or latency?
Aside: latency in modern camera systems

- Often in this class our focus will be on achieving high throughput
  - e.g., pixels per clock, images/sec, triangles/sec
- But low latency is critical in many visual computing domains
  - Camera metering, autofocus, etc.
  - Multi-shot photography
  - Optical flow, object tracking

Extreme example: CMU smart headlight project
[Tamburo et al. 2014]
F-cam “streaming” mode

- System repeats shot (or series of shots) in infinite loop
- F-cam only stops acquiring frames when told to stop streaming by the application
- Example use case: “live view” (digital viewfinder) or continuous metering
F-cam as an architecture

Application Commands ("Shots")

- Device (Flash)
- Device (Lens)

Sensor

- RAW Data

Image Processor

- Image Data
- Frames

Cmd Processor

- Stream Cmd Buffer

Completed Frames

Event Queue

Memory

Image Buffers
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.
Class design challenge 1

- Question: How is auto-focus expressed in F-cam?
  - Is autofocus part of F-cam?
  - Can you implement autofocus using F-cam?

- How might we extend the F-cam architecture to model a separate autofocus/metering sensor if the hardware platform contained them?
Class design challenge 2

- Should we add a face-detection unit to the architecture?
- How might we abstract a face-detection unit?
- Or a SIFT feature extractor?
Hypothetical F-cam extension: programmable image processing

Application Commands ("Shots")

Cmd Processor

Stream Cmd Buffer

Device (Flash)

Device (Lens)

Sensor

Programmable Image Processor

RAW Data

Image Data

Frames

Memory

Completed Frames

Event Queue

Image Buffers

...
Class design challenge 3

- If there was a programmable image processor, application would probably seek to use it for more than just on data coming off sensor

- E.g., HDR imaging app
Key aspect in the design of any system:
Choosing the “right” representations for the job
Choosing the “right” representation 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, conservation of quantities, type checking)
  - Performance (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)
Example task: sharpen an image

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

Input

Output
Four different representations of sharpen

Image input;
Image output = sharpen(input);

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

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

float input[\text{WIDTH+2} \times \text{HEIGHT+2}];
float output[\text{WIDTH} \times \text{HEIGHT}];

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

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

Image input;
Image output = sharpen(input);

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

float input[\text{WIDTH+2} \times \text{HEIGHT+2}];
float output[\text{WIDTH} \times \text{HEIGHT}];

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

for (int j=0; j<\text{HEIGHT}; j++) {
    for (int i=0; i<\text{WIDTH}; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+jj) \times \text{WIDTH+2} + (i+ii)] \times weights[jj*3 + ii];
        output[j*\text{WIDTH} + i] = tmp;
    }
}
More image processing tasks from last lecture

<table>
<thead>
<tr>
<th>Sobel Edge Detection</th>
<th>3x3 Gaussian blur</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_x = \begin{bmatrix} -1 &amp; 0 &amp; 1 \ -2 &amp; 0 &amp; 2 \ -1 &amp; 0 &amp; 1 \end{bmatrix} \ast I$</td>
<td>$F = \begin{bmatrix} .075 &amp; .124 &amp; .075 \ .124 &amp; .204 &amp; .124 \ .075 &amp; .124 &amp; .075 \end{bmatrix}$</td>
</tr>
<tr>
<td>$G_y = \begin{bmatrix} -1 &amp; -2 &amp; -1 \ 0 &amp; 0 &amp; 0 \ 1 &amp; 2 &amp; 1 \end{bmatrix} \ast I$</td>
<td></td>
</tr>
<tr>
<td>$G = \sqrt{G_x^2 + G_y^2}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local Pixel Clamp</th>
<th>2x2 downsample (via averaging)</th>
</tr>
</thead>
</table>
| 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);
} |
| output[x][y] = (input[2x][2y] + input[2x+1][2y] + input[2x][2y+1] + input[2x+1][2y+1]) / 4.f; |

<table>
<thead>
<tr>
<th>Gamma Correction</th>
<th>LUT-based correction</th>
</tr>
</thead>
<tbody>
<tr>
<td>output[x][y] = pow(input[x][y], 0.5f);</td>
<td></td>
</tr>
<tr>
<td>output[x][y] = lookup_table[input[x][y]];</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Histogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin[input[x][y]]++;</td>
</tr>
</tbody>
</table>
Image processing workload characteristics

- Sequences of operations on images
- Natural to think about algorithms in terms of their local behavior: “pointwise code” (output at pixel xy is function of input pixels in neighborhood around xy)
- Common case: access to local window around a point
- But some algorithms require data-dependent data access (e.g., data-dependent access to lookup-tables)
- Multiple rates of computation (upsampling/downsampling)
- Simple inter-pixel communication/reductions (e.g., building a histogram, computing maximum brightness pixel)
Halide language

Simple language embedded in C++ for describing sequences of image processing operations (image processing pipelines)

Var x, y;
Func blurx, blury, 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,y));
blury(x,y) = 1/3.f * (blurx(x,y-1) + blurx(x,y+1) + blurx(x,y+1));

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

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

- Function: an infinite (but discrete) set of values
- Expression: a side-effect free expression describes how to compute a function’s value at a point in it’s domain in terms of the values of other functions.
Halide language

Update definition modify function values

Reduction domains provide the ability to iterate

Var x;
Func histogram, modified;
Image<uint8_t> in = load_image(“myimage.jpg”);

modified(x,y) = in(x,y) + 10;
modified(x,3) *= 2;  // update definition, modifies 3rd row
modified(3,y) *= 2;  // update definition, modifies 3rd column

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

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

// update definition on histogram
// for all points in domain, increment appropriate bin
histogram(in(r.x, r.y)) += 1;

Image<int> result = histogram.realize(256);
Key observations about Halide’s design

- Adopts local “pointwise” view of expressing algorithms
- Language is highly constrained so that iteration over domain points is implicit (no explicit loops in Halide)
  - Halide language is declarative. It does not define order of iteration, or what values in domain or stored! *(It only defines what operations are needed to compute these values.)*

```c
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,y));
out(x,y) = 1/3.f * (blurx(x,y-1) + blurx(x,y+1) + blurx(x,y+1));

// execute pipeline on domain of size 800x600
Image<uint8_t> result = our.realize(800, 600);
```
Efficiently executing Halide programs
Example

Consider writing code for the 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,y));
out(x,y) = 1/3.f * (blurx(x,y-1) + blurx(x,y+1) + blurx(x,y+1));

// execute pipeline on domain of size 1024x1024
Image<uint8_t> result = out.realize(1024, 1024);
Two-pass 3x3 blur

```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[] = {1.0/3, 1.0/3, 1.0/3};

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;
    }
```

Total work per image = \(6 \times WIDTH \times HEIGHT\)

For \(N \times N\) filter: \(2N \times WIDTH \times HEIGHT\)

WIDTH \times HEIGHT extra storage

1D horizontal blur

1D vertical blur

input \((W+2)\times(H+2)\)

tmp_buf \(W \times (H+2)\)

output \(W \times H\)
Two-pass image blur: locality

```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[] = {1.0/3, 1.0/3, 1.0/3};

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 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[] = {1.0/3, 1.0/3, 1.0/3};

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)

```c
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[] = {1.0/3, 1.0/3, 1.0/3};

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

Trends to idea 6 x WIDTH x HEIGHT as CHUNK_SIZE is increased!

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

CMU 15-769, Fall 2016
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 implementation

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!

```c
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);
                    tmpPtr += 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);
            }
        }
    }
}
```
Image processing pipelines feature complex sequences of functions!

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

Real-world production applications may feature hundreds to thousands of functions! Google HDR+ pipeline: over 2000 Halide functions.
Key aspect in the design of any system:
Choosing the “right” representations for the job

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;
Image<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.
Vectorize the xi loop (8-wide)
Use threads to parallelize the y loop

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

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

Scheduling primitives allow the programmer to specify a global “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 domains

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

2D blocked iteration order

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 $y_o$, $x_o$, $y_i$, $x_i$
Primitives for how to “fuse” adjacent stages

Describe where to compute producer function within the loop nest of the consumer.

\[
\text{blurx}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3.0f};
\]

\[
\text{out}(x, y) = \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)}{3.0f};
\]

\text{out}.tile(x, y, xi, yi, 256, 32);

\text{blurx}.compute\_root();

Do not compute \text{blurx} within \text{out}'s loop nest. Compute all of \text{blurx}, then all of \text{out}

allocate buffer for all of \text{blur}(x,y)

for \(y=0\) to \(\text{HEIGHT}\):
  for \(x=0\) to \(\text{WIDTH}\):
    \text{blurx}(x, y) = …

for \(y=0\) to \(\text{num\_tiles\_y}\):
  for \(x=0\) to \(\text{num\_tiles\_x}\):
    for \(yi=0\) to \(32\):
      for \(xi=0\) to \(256\):
        \text{idx\_x} = x*256+xi;
        \text{idx\_y} = y*32+yi
        \text{out}(\text{idx\_y}, \text{idx\_y}) = …
Primitives for how to “fuse” adjacent stages

Describe where to compute producer function within the loop nest of the consumer.

\[
\text{blurx}(x,y) = (\text{in}(x-1, y) + \text{in}(x,y) + \text{in}(x+1,y)) / 3.0f;
\]

\[
\text{out}(x,y) = (\text{blurx}(x,y-1) + \text{blurx}(x,y) + \text{blurx}(x,y+1)) / 3.0f;
\]

\[
\text{out.tile}(x, y, xi, yi, 256, 32);
\]

```
blurx.compute_at(x_i);
```

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
                // compute 3 elements of blurx needed for out(idx_x, idx_y) here
                out(idx_y, idx_y) = ... 
```
Primitives for how to “fuse” adjacent stages

Describe where to compute producer function within the loop nest of the consumer.

\[
\text{blurx}(x,y) = \frac{\text{in}(x-1, y) + \text{in}(x,y) + \text{in}(x+1,y)}{3.0f};
\]
\[
\text{out}(x,y) = \frac{\text{blurx}(x,y-1) + \text{blurx}(x,y) + \text{blurx}(x,y+1)}{3.0f};
\]
\[
\text{out}.tile(x, y, xi, yi, 256, 32);
\]

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

\[\text{Compute necessary elements of blurx within out’s x loop nest (all necessary elements for one tile of out)}\]

```
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:
        blur(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_y, idx_y) = …
```
Early Halide results

- **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

- **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

[Ragan-Kelley 2012]
What is Halide?

- Halide is a simple (highly constrained) declarative language for describing sequences of image processing operations

- Coupled with an additional declarative language for describing how to map the operations in these pipelines to a parallel machine (the “schedule”)
  - Primitives for describing producer-consumer locality optimizations
  - Domain traversal order
  - And basic multi-core and SIMD parallelization
  - Powerful primitives: composition of these primitives enables expression of a diverse set of schedules

- Two languages designed so that space of allowed schedules is enumerable and manifest in the program’s definition
Automatically generating schedules

- Halide’s design seeks to provide representations for developers with strong code optimization experience to do their job faster
  - Programmer still needs to have code optimization skill to specify a good schedule

- Recent work has demonstrated the ability to analyze the Halide program to automatically generate efficient schedules for the user
  - See tonight’s reading [Mullapudi 2016]
Tonight’s Halide readings

- What is the key intellectual idea of the Halide system?
  - Hint: it’s not the declarative language syntax

- What **services** does Halide provide its users?

- What aspects of the design of Halide allow it to provide those services?

- Keep in mind: the key aspect in the design of any system usually is choosing the “right” representations for the job