Lecture 4:
Efficiently Scheduling Image Processing Pipelines

Visual Computing Systems
Stanford CS348K, Fall 2018
Today’s lecture: two themes

- Techniques for efficiently mapping image processing applications to multi-core CPUs/GPU
  - (Writing high performance image processing code)

- The design of programming abstractions that facilitate development of efficient image processing applications
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:
  - 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)
What does this code do? 😳😭😩🤔

I’ll tell you later in class.
Consider a single task: sharpen an image

\[ F = \begin{bmatrix}
0 & -1 & 0 \\
-1 & 5 & -1 \\
0 & -1 & 0 \\
\end{bmatrix} \]
Four different representations of sharpen

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

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

3. 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 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);

Image input;
Image output = sharpen(input);

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

Image input;
Image output = sharpen(input);

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

Image input;
Image output = sharpen(input);

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

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] +
  ...

F=
\[
\begin{bmatrix}
0 & -1 & 0 \\
-1 & 5 & -1 \\
0 & -1 & 0
\end{bmatrix}
\]
# Image processing tasks from previous lectures

## Sobel Edge Detection

\[
G_x = \begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1 \\
\end{bmatrix} \ast I
\]

\[
G_y = \begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1 \\
\end{bmatrix} \ast I
\]

\[
G = \sqrt{G_x^2 + G_y^2}
\]

## 3x3 Gaussian blur

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

## 2x2 downsample (via averaging)

\[
\text{output}[x][y] = \frac{\text{input}[2x][2y] + \text{input}[2x+1][2y] + \text{input}[2x][2y+1] + \text{input}[2x+1][2y+1]}{4.f};
\]

## Gamma Correction

\[
\text{output}[x][y] = \text{pow}(\text{input}[x][y], 0.5f);
\]

## LUT-based correction

\[
\text{output}[x][y] = \text{lookup_table}[\text{input}[x][y]];
\]

## Histogram

\[
\text{bin}[\text{input}[x][y]]++;
\]
Let’s consider representations for authoring image processing applications
Image processing workload characteristics

- Conceptual 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: access to bounded local “window” of pixels around a point

- 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)
Goals

- Expressive: facilitate intuitive expression of a broad class of image processing applications
  - e.g., consider 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
Halide language

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

```cpp
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);
```

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.

[Ragan-Kelley / Adams 2012]
More Halide language (multi-stage functions)

Var x;
Func histogram, average;
Halide::buffer<
\texttt{uint8}\_t> in = load\_image(\texttt{"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<
\texttt{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<
\texttt{uint8}\_t> hist\_result = histogram.realize(256);
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 operations are needed to compute these values.
    - Iteration over domain points is implicit (no explicit loops)

```cpp
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
Recall this example from lecture 1

Program 1

```c
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];
}


// 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

```c
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);
```
Halide blur 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+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);
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[] = {1.0/3, 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 x WIDTH x HEIGHT
For NxN filter: 2N x WIDTH x HEIGHT
WIDTH x HEIGHT extra storage
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[] = {1.0/3, 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)

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

```c
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

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

Produce `CHUNK_SIZE` rows of output

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

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 idea 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!

```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 = 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 Halide 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;
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);

// 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 domains

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

2D blocked iteration order

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

serial y, vectorized x

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

parallel y, vectorized x

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

split x into $2x_o + x_i$, split y into $2y_o + y_i$, serial $y_o$, $x_o$, $y_i$, $x_i$
Specifying loop iteration order and parallelism

```
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;
```

Given this schedule for the function “out”...

```
out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
```

Halide compiler will generate this parallel, vectorized loop nest for computing elements of out...

```
for y=0 to num_tiles_y:   // parallelize this loop over multiple threads
    for x=0 to num_tiles_x:
        for yi=0 to 32:
            // vectorize body of this loop with SIMD instructions
            for xi=0 to 256 by 8:
                idx_x = x*256+xi;
                idx_y = y*32+yi
                out(idx_x, idx_y) = ...
```
Primitives for how to interleave producer/consumer processing

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

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

\text{blurx.compute_root();}

Do not compute blurx within out's loop nest. Compute all of blurx, then all of 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) = \ldots

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 \( 256 \):
        \( \text{idx\_x} = x \times 256 + xi; \)
        \( \text{idx\_y} = y \times 32 + yi \)
        \text{out}(\text{idx\_x}, \text{idx\_y}) = \ldots

all of blurx is computed here

values of blurx consumed here
Primitives for how to interleave producer/consumer processing

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

\text{out.tile}(x, y, x_i, y_i, 256, 32);

\text{blurx.compute_at}(\text{out}, x_i);

for \( y=0 \) to num\_tiles\_y:
  for \( x=0 \) to num\_tiles\_x:
    for \( y_i=0 \) to 32:
      for \( x_i=0 \) to 256:
        idx\_x = x*256+x_i;
        idx\_y = y*32+y_i

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

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

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

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

\[
\text{blurx.compute_at}(\text{out}, x);
\]

\[
\text{for } y=0 \text{ to num_tiles_y:}
\]
\[
\text{for } x=0 \text{ to num_tiles_x:}
\]

allocate 258x34 buffer for tile blurx

for yi=0 to 32+2:
    for xi=0 to 256+2:
        \[
        \text{tmp_blurx}(xi,yi) = // compute blurx from in
        \]

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) = …

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

tile of blurx is computed here

tile of blurx is consumed here
An interesting Halide schedule

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

\[
\begin{align*}
\text{blurx}.\text{store}_\text{at}(x) \\
\text{blurx}.\text{compute}_\text{at}(\text{out}, xi);
\end{align*}
\]

\[
\begin{align*}
\text{for } y=0 \text{ to } \text{num}_\text{tiles}_y: \\
\quad \text{for } x=0 \text{ to } \text{num}_\text{tiles}_x:
\end{align*}
\]

\[
\begin{align*}
\text{allocate 258x34 buffer for tile tmp}_\text{blurx}
\end{align*}
\]

\[
\begin{align*}
\text{for } yi=0 \text{ to } 32: \\
\quad \text{for } xi=0 \text{ to } 256:
\end{align*}
\]

\[
\begin{align*}
\quad \text{id}_x &= x*256+xi; \\
\quad \text{id}_y &= y*32+yi;
\end{align*}
\]

\[
\begin{align*}
\quad \text{// compute 3 elements of blurx needed for out(id}_x, id}_y \text{ here}
\end{align*}
\]

\[
\begin{align*}
\text{for (blur}_x=0 \text{ to } 3)
\quad \text{tmp}_\text{blurx}(\text{blur}_x) &= \ldots
\end{align*}
\]

\[
\begin{align*}
\text{out(id}_x, id}_y &= \ldots
\end{align*}
\]
“Sliding optimization” (reduces redundant computation)

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

\[
\text{blurx}.\text{store}\_\text{at}(x) \\
\text{blurx}.\text{compute}\_\text{at}(\text{out}, xi);
\]

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

\[
\text{for } y=0 \text{ to num\_tiles\_y:} \\
\quad \text{for } x=0 \text{ to num\_tiles\_x:} \\
\quad \quad \text{allocate 258x34 buffer for tile tmp\_blurx}
\]

\[
\text{for } yi=0 \text{ to 32:} \\
\quad \text{for } xi=0 \text{ to 256:} \\
\quad \quad \text{idx}\_x = x*256+xi; \\
\quad \quad \text{idx}\_y = y*32+yi;
\]

\[
\quad \text{if } (yi=0) \{ \\
\quad \quad \text{// compute 3 elements of blurx needed for out(idx\_x, idx\_y) here} \\
\quad \quad \text{for (blur\_x=0 to 3)} \\
\quad \quad \quad \text{tmp\_blurx(blur\_x) = …} \\
\quad \} \text{ else} \\
\quad \quad \text{// only compute one additional element of tmp\_blurx}
\]

\[
\quad \quad \text{out(idx\_x, idx\_y) = …}
\]

\[
\text{Steady state: only one new element of tmp\_blurx needs to be computed per output}
\]
“Folding optimization” (reduces intermediate storage)

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

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

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

\[
\text{blurx}.\text{store}\_\text{at}(x)
\]

\[
\text{blurx}.\text{compute}\_\text{at}(\text{out}, xi);
\]

for \(y=0\) to \(\text{num\_tiles\_y}\):

\[
\text{for } x=0 \text{ to } \text{num\_tiles\_x}:
\]

allocate 3x256 buffer for tmp_blurx

\[
\text{for } yi=0 \text{ to } 32:
\]

\[
\text{for } xi=0 \text{ to } 256:
\]

\[
\text{id}_{x} = x*256 + xi;
\]

\[
\text{id}_{y} = y*32 + yi;
\]

if \((yi=0)\) {

// compute 3 elements of blurx needed for \(\text{out}(\text{id}_{x}, \text{id}_{y})\) here

\[
\text{for } (\text{blur\_x}=0 \text{ to } 3)
\]

\[
\text{tmp\_blurx}(\text{blur\_x}) = \ldots
\]

} else

// only compute one additional element of tmp_blurx

\[
\text{out}(\text{id}_{x}, \text{id}_{y}) = \ldots
\]

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., \(((\text{id}_{y}+1)\%3)\)
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(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 details of the schedule in terms of mechanisms available on the target machine (phthreads, AVX intrinsics, etc.)
  - There are two major examples of deviation from this philosophy. 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

- **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]
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.
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 primitives for a programmer (that has strong knowledge of code optimization) to rapidly express what optimizations the system should apply
  - Halide compiler carries out the nitty-gritty of mapping 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
  - 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]
Tonight’s Halide readings

- What is the key intellectual idea of the Halide system?
  - Hint: it is 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