Lecture 4:
High-performance image processing using Halide

Visual Computing Systems
Stanford CS348V, Winter 2018
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 developer 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)
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);
                }
                tmpPtr = tmp;
            }
        }
    }
}
Example 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. Image input;
   Image output = convolve(input, F);

Image input:
Image output = sharpen(input);

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

4. 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;
   }
}
More image processing tasks from last lecture

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)

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

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

\[ \text{bin}[\text{input}[x][y]]++; \]

Local Pixel Clamp

float \( f(\text{image input}) \) {
    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);
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 \((x,y)\) is function of input pixels in neighborhood around \((x,y)\)
- Common case: access to local "window" of pixels 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);

- Halide function: an infinite (but discrete) set of values
- Halide 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.

[Ragan-Kelley 2012]

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
Halide language

Update definitions 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 aspects of 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) = 1/3.0f * (in(x-1,y) + in(x,y) + in(x,y));
out(x,y) = 1/3.0f * (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

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

1D horizontal blur

input (W+2)x(H+2)

tmp_buf W x (H+2)

1D vertical blur

output W x H
Two-pass image blur: locality

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 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

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)

```cpp
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;
        }
}
```

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: $18 \times 3 \times Width$ work
- Step 2: $16 \times 3 \times Width$ work

Total work per image: $(34/16) \times 3 \times Width \times Height = 6.4 \times Width \times Height$

Trends to idea $6 \times Width \times Height$ 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 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 = 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;
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);

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

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

// 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 $y_o$, $x_o$, $y_i$, $x_i$
Specifying loop iteration order and parallelism

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

**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:
            for xi=0 to 256:  // vectorize this loop with SIMD instructions
                idx_x = x*256+xi;
                idx_y = y*32+yi
                out(idx_y, 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();

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

\text{allocate buffer for all of blur}(x,y)
\text{for } y=0 \text{ to HEIGHT:}
  \text{for } x=0 \text{ to WIDTH:}
    \text{blurx}(x,y) = …

\text{for } y=0 \text{ to num_tiles_y:}
  \text{for } x=0 \text{ to num_tiles_x:}
    \text{for } yi=0 \text{ to 32:}
      \text{for } xi=0 \text{ to 256:}
        \text{idx}_x = x*256+xi;
        \text{idx}_y = y*32+yi
        \text{out}([\text{idx}_y, \text{idx}_y]) = …

\text{all of blurx is computed here}

\text{values of blurx consumed here}
Primitives for how to interleave producer/consumer processing

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

\text{Compute necessary elements of blurx within out's xi loop nest}

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

allocate 3-element buffer for blurx

\[
\text{// compute 3 elements of blurx needed for out(idx}_x, \text{ idx}_y \text{) here}
\]
\[
\text{out(idx}_y, \text{ idx}_y) = ...$
\]
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(x);

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:
                \text{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
        \text{out}(idx_y, idx_y) = ...
Halide: two domain-specific co-languages

- Functional primitives for describing image processing operations
- Additional primitives for describing schedules
- Design principle: separate “algorithm specification” from schedule
  - Programmer’s responsibility: provide a high-performance schedule
  - Compiler’s responsibility: carry out mechanical process of generating threads, SIMD instructions, managing buffers, etc.
  - Result: enable programmer to rapidly exploration of space of schedules ("tile these loops", vectorize this loop", “parallelize this loop across cores”)

- Application domain scope:
  - All computation on regular N-D coordinate spaces
  - Only feed-forward pipelines (includes special support for reductions and fixed recursion depth)
  - All dependencies inferable by compiler
Initial 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 Google Pixel Photos app
- Halide code used to process images uploaded to Google Photos
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, such as a 15-418 student) 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 [Mullapudi 2016]
Problem definition

Input: DAG of image processing stages

Output: optimized schedule

for each 8x128 tile in parallel
compute required pixels of A
compute pixels in tile of B

for each 8x8 tile in parallel
compute required pixels of C
compute required pixels of D
compute pixels in tile of E
How to tile a group

Tile size: 3x3
The autoscheduler fits programs to a single schedule template

For a single sub-DAG

for each tile_y: // multi-core parallel
for each tile_x: // multi-core parallel

allocate tmpC; // buffer for C for tile
allocate tmpD; // buffer for D for tile

for each y in required region of C
  for each x in required region of C
    tmpC = ... // DRAM accesses to B

for each y in required region of D
  for each x in required region of D
    tmpD = ... // DRAM accessed to B

for each y in tile:
  for each x in tile: // optionally SIMD vector parallel
    x' = tile_x * TILE_WIDTH + x;
    y' = tile_y * TILE_HEIGHT + y;
    E(x', y') = ...

Cost model to evaluate a tiling:
- cost of data access is proportional to buffer size
- cost arithmetic op = 1
Estimating costs via interval analysis

\[
\begin{align*}
A(x,y) &= \text{in}(x-1,y) + \text{in}(x,y) + \text{in}(x+1,y) \\
B(x,y) &= A(x,y-1) + A(x,y) + A(x,y+1)
\end{align*}
\]
Estimating costs via interval analysis

Tile size: 3x3

Cost = Number of arithmetic operations + Number of memory accesses x COST OF ELEMENT LOAD
Search for most efficient tile size
Finding efficient groupings

Given the ability to compute a good tiling of groups...

Which stages should be grouped together?
Greedy grouping strategy

Greedily form groups (take best merge)

But re-evaluate local optimization to determine tile size each step.

for each 8x128 tile
compute required pixels of A
compute required pixels of B
compute pixels in tile of D

for each 8x8 tile
compute required pixels of C
compute pixels in tile of E
Add standard compiler optimizations: multi-core parallelism, vectorization, loop unrolling, etc.

for each 8x128 tile in parallel
  vectorize compute required pixels of A
  unroll x by 4
  vectorize compute required pixels of B
  vectorize compute pixels in tile of D

for each 8x8 tile in parallel
  vectorize compute required pixels of C
  unroll y by 2
  vectorize compute pixels in tile of E
# Autoscheduler generates schedules in seconds

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Stages</th>
<th>Compile time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blur</td>
<td>3</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Unsharp mask</td>
<td>9</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Harris corner detection</td>
<td>13</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Camera RAW processing</td>
<td>30</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Non-local means denoising</td>
<td>13</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Max-brightness filter</td>
<td>9</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Multi-scale interpolation</td>
<td>52</td>
<td>2.6</td>
</tr>
<tr>
<td>Local-laplacian filter</td>
<td>103</td>
<td>3.9</td>
</tr>
<tr>
<td>Synthetic depth-of-field</td>
<td>74</td>
<td>55</td>
</tr>
<tr>
<td>Bilateral filter</td>
<td>8</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Histogram equalization</td>
<td>7</td>
<td>&lt;1</td>
</tr>
<tr>
<td>VGG-16 deep network eval</td>
<td>64</td>
<td>6.9</td>
</tr>
</tbody>
</table>
Autoscheduler performs comparably to experts

Performance relative to schedules authored by experts
(6 core Xeon CPU)

On 8 of the 14 benchmarks performance within 10% of experts or better

Baseline schedules exploit multi-core/vector parallelism and pointwise inlining but no global locality optimizations
Autoscheduler saves time for experts

Non-local means denoising

Lens blur

Max filter

Throughput

Time (min)

Auto scheduler
Dillon
Andrew
Auto scheduler now maintained by Google’s Halide team

- If you write a Halide program, you can now schedule it with `.autoschedule()`
- Works quite well for CPU targets (x86/ARM)
- Works okay for GPU targets
- Will generate better schedulers in <1 sec than I would be able to create manually in a few days
- May not beat top developers at a major company, but gives them a strong head start
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