Lecture 14:
Scheduling Image Processing Pipelines

Visual Computing Systems
CMU 15-869, Fall 2014
Simple image processing kernel

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[WIDTH * HEIGHT];
float output[WIDTH * HEIGHT];

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        output[j*WIDTH + i] = 0.5f * input[j*WIDTH + i];
    }
}
```

Point-wise operation: one in, one out
3x3 box blur (Photoshop)
2D convolution using 3x3 filter

int WIDTH = 1024;
int HEIGHT = 1024;
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;
    }
}
2D convolution using 3x3 filter

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

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

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;
    }
}
```
Data-dependent filter

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

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float min_value = min(min(min(input[(j-1)*WIDTH + i], input[(j+1)*WIDTH + i]),
                                min(input[j*WIDTH + i-1], input[j*WIDTH + i+1])),
                            min(input[(j-1)*WIDTH + i], input[(j+1)*WIDTH + i]),
                            min(input[j*WIDTH + i-1], input[j*WIDTH + i+1]));
        float max_value = max(max(max(input[(j-1)*WIDTH + i], input[(j+1)*WIDTH + i]),
                                   max(input[j*WIDTH + i-1], input[j*WIDTH + i+1])),
                                max(input[(j-1)*WIDTH + i], input[(j+1)*WIDTH + i]),
                                max(input[j*WIDTH + i-1], input[j*WIDTH + i+1]));
        output[j*WIDTH + i] = clamp(min_value, max_value, input[j*WIDTH + i]);
    }
}
```
Image blur (convolution with 2D filter)

```
int WIDTH = 1024;
int HEIGHT = 1024;
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 image = 9 x WIDTH x HEIGHT
For NxN filter: \( N^2 \times WIDTH \times HEIGHT \)
Two-pass box 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./3, 1./3, 1./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
Separable filters

- A separable 2D filter is the outer product of two 1D filters

Implication:
- Can implement 2D convolution efficiently as two 1D convolutions

\[
F = \begin{bmatrix}
\frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\
\frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\
\frac{1}{9} & \frac{1}{9} & \frac{1}{9}
\end{bmatrix}
= \begin{bmatrix}
\frac{1}{3}
\end{bmatrix}
\begin{bmatrix}
\frac{1}{3} & \frac{1}{3} & \frac{1}{3}
\end{bmatrix}
\]

\[
F = \begin{bmatrix}
1 & 0 & -1 \\
2 & 0 & -2 \\
1 & 0 & -1
\end{bmatrix}
= \begin{bmatrix}
1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -1
\end{bmatrix}
\]
Two-pass box 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./3, 1./3, 1./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 N x N filter: 2N x WIDTH x HEIGHT

But implementation incurs bandwidth cost of writing and reading tmp_buf. (and memory footprint overhead of storing tmp_buf)
Two-pass image 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./3, 1./3, 1./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;
    }

Data from \texttt{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 \texttt{tmp} are overhead traffic (this memory traffic is an artifact of the two-pass implementation: it is not intrinsic to computation being performed)

Data from \texttt{tmp} 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)

Intrinsic bandwidth requirements of algorithm:
Application must read each element of input image and must write each element of output image.
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./3, 1./3, 1./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 needs to be allocated
- Produce 3 rows of tmp_buf
- Combine them together to get one row of output
- Total work per row of output:
  - step 1: $3 \times 3 \times WIDTH$ work
  - step 2: $3 \times WIDTH$ work

Total work per image = $12 \times WIDTH \times HEIGHT$
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./3, 1./3, 1./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 to fit in cache (capture all producer-consumer locality)

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

Produce CHUNK_SIZE rows of output

Total work per row 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

Trends to idea 6 x WIDTH x HEIGHT as CHUNK_SIZE is increased!
Conflicting goals (once again...)

- Want to be work efficient (perform fewer operations)
- Want to take advantage of locality when present
  - Ideal: bandwidth cost of implementation is very close to intrinsic cost of algorithm: data loaded from memory once and used in all instances it is needed prior to being discarded from processor’s cache
- Want to execute in parallel (multi-core, SIMD within core)
Optimized C++ code: 3x3 image 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_u128((__m128i*)inPtr);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_u128(tmpPtr++, avg);
                    inPtr += 8;
                }
            }
        }
    }
    tmpPtr = tmp;
    for (int y = 0; y < 32; y++) {
        _m128i *outPtr = (__m128i*)(&blurred(xTile, yTile+y));
        for (int x = 0; x < 256; x += 8) {
            a = _mm_load_u128(tmpPtr+(2*256)/8);
            b = _mm_load_u128(tmpPtr+256/8);
            c = _mm_load_u128(tmpPtr++);
            sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
            avg = _mm_mulhi_epi16(sum, one_third);
            _mm_store_u128(outPtr++, avg);
        }
    }
}
```
Halide image processing language
Halide blur

- Halide = two domain-specific co-languages
  1. A purely functional language for defining image processing algorithms
  2. A mini-language for defining “schedules” for how to map these algorithms to machines

Images are pure functions from integer coordinates (up to 4D domain) to values (color of corresponding pixels)

Algorithms are a series of functions
(Algorithms are image processing pipelines, and a function defines the logic of a pipeline stage)

Functions (side-effect-free) map coordinates to values
(in, tmp and blurred are functions)

```plaintext
Func halide_blur(Func in) {  
  Func tmp, blurred;  
  Var x, y, xi, yi;  

  // The algorithm  
  tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;  
  blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

  return blurred;  
}
```

NOTE: Neither execution order of processing nor image storage format is specified by the functional abstraction. The Halide implementation can evaluate, reevaluate, cache individual points as desired!
Halide program as a pipeline

```c
Func halide_blur(Func in) {
    Func tmp, blurred;
    Var x, y, xi, yi;

    // The algorithm
    tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

    return blurred;
}
```
Halide blur

- Halide = two domain-specific co-languages
  1. A purely functional language for defining image processing algorithms
  2. A mini-language for defining “schedules” for how to map these algorithms to machines

```c
Func halide_blur(Func in) {
    Func tmp, blurred;
    Var x, y, xi, yi;

    // The algorithm
    tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

    // The schedule
    blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    tmp.chunk(x).vectorize(x, 8);
    return blurred;
}
```

- When evaluating `blurred`, 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 only chunks of `tmp` at a time. Vectorize the x (innermost) loop.
Separation of algorithm from schedule

Key idea of Halide: a small algebra of composable scheduling operations can describe optimization decisions for a broad class of image processing operations

- Programmers “optimize” an algorithm by quickly describing a schedule in a domain-specific scheduling co-language.

Given algorithm + schedule, Halide system generates high-quality code for a target machine

- Powerful optimizations enabled by limiting scope of application domain:
  - All computation over regular (up to 4D) grids
  - Only feed-forward pipelines (includes special support for reductions)
  - Language constrained so that all dependencies can be inferred by compiler
Halide schedule: producer/consumer scheduling

- Four basic scheduling primitives shown below
- Fifth primitive: “reuse” not shown

**breadth first:** each function is entirely evaluated before the next one.

**sliding window:** values are computed when needed then stored until not useful anymore.

**total fusion:** values are computed on the fly each time that they are needed.

**tiles:** overlapping regions are processed in parallel, functions are evaluated one after another.
Halide schedule: domain iteration

Schedule specifies both iteration order and how to parallelize independent iterations (multi-threading, SIMD vector)

2D blocked iteration order
Halide results

- **Camera RAW processing pipeline**
  (Convert RAW sensor data to RGB image)
  - Original: 463 lines of hand-tuned ARM 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

*Takeaway: Halide abstractions allow rapid exploration of optimization space, allowing programmer to reach optimal points quickly*
Lesson: establishing good abstractions is extremely valuable

- Halide is one attempt to raise the level of abstraction for authoring image-processing applications
  - Focus: make it easier for humans to iterate through possible scheduling choices
  - Auto-tuning was explored (see readings) but very long compilation times
  - Other emerging systems with similar goals: See “Darkroom” from readings

- Halide developed at MIT/Stanford circa 2010-2012

- Halide now in use at Google for Android camera processing pipeline
Summary: image processing basics

- Same trade-offs arise as in previous discussions of graphics pipelines
  - Need to maintain work efficiency
  - Need to achieve bandwidth efficiency (exploit locality when it exists)
  - Need to maintain parallelism

- A well-tuned piece of C code can be an order of magnitude faster than a basic C implementation
  - However, a few lines of code turns into many (difficult to understand and maintain)
  - Scheduling decisions not portable across machines (different compute/BW trade-offs, different cache sizes, different instruction sets, specialized HW, ...)
Darkroom: efficient programmable image processing from a hardware perspective
Image signal processor

Fixed-function ASIC in modern cameras: responsible for processing RAW pixels to produce output image

Point-wise operation: one input pixel → one output pixel

brighten_image img(x,y) I(x,y) * 1.1

How much buffering is needed in the ISP to implement pointwise operations?
Stencil operations

Stencil-operation: local region of input pixels → one output pixel

How much buffering is needed in the ISP to implement this stencil operation?

\[
\text{convolve\_image\_1d}\; \text{img}(x,y) = \frac{(I(x-2,y) + I(x-1,y) + I(x, y))}{3.0}
\]
Line-buffered execution

Dependency graph:

ISP “line-buffer” (shift-register) contents:

t=0: \quad \text{in}(0), \quad \_ \quad \_ \quad \_ \\
t=1: \quad \text{in}(1), \quad \text{in}(0), \quad \_ \\
t=2: \quad \text{in}(2), \quad \text{in}(1), \quad \text{in}(0) \\
t=3: \quad \text{in}(3), \quad \text{in}(2), \quad \text{in}(1) \\
t=4: \quad \text{in}(4), \quad \text{in}(3), \quad \text{in}(2) \\

Each clock, ISP performs:

\[ \text{out} = \frac{\text{line}_\text{buffer}[0] + \text{line}_\text{buffer}[1] + \text{line}_\text{buffer}[2]}{3.0}; \]
Stencil operations

How much buffering is needed in the ISP to implement this stencil operation?

```
convolve_image_2d img(x,y) ((I(x-2,y-2) + I(x-1,y-2) + I(x, y-2) +
I(x-2,y-1) + I(x-1,y-1) + I(x, y-1)) +
I(x-2,y) + I(x-1,y) + I(x, y)) / 9.0;
```

Conversion to 1D problem (assuming image width W):

```
I(x + c1, y + c2) = I(x' + c1 + W*c2)
```

where \( x' = y*W + x \)

Figure credit: Hegarty 2014
Readings


- Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. Hegarty et al. SIGGRAPH 2014