Lecture 15: Scheduling Image Processing Pipelines

Visual Computing Systems
CMU 15-869, Fall 2013
Simple image processing kernel

\[
\begin{align*}
\text{int WIDTH} & = 1024; \\
\text{int HEIGHT} & = 1024; \\
\text{float input[WIDTH } \times \text{ HEIGHT];} \\
\text{float output[WIDTH } \times \text{ HEIGHT];} \\
\end{align*}
\]

\[
\begin{align*}
\text{for (int } j=0; j<\text{HEIGHT; } j++) \{ \\
\quad \text{for (int } i=0; i<\text{WIDTH; } i++) \{ \\
\quad\quad \text{output[j*WIDTH } + \text{ i] = 0.5f } \times \text{ input[j*WIDTH } + \text{ i];} \quad \\
\quad \} \\
\} \\
\end{align*}
\]

Point-wise operation: one in, one out
2D convolution with 3x3 filter

```c
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;
    }
}
```
3x3 box blur (Photoshop)
Convolution with 3x3 filter

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(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(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)

```c
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 = \( 9 \times WIDTH \times HEIGHT \)
For NxN filter: \( N^2 \times WIDTH \times HEIGHT \)
Two-pass image 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+11] * 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 = 6 x WIDTH x HEIGHT
For NxN filter: 2N x WIDTH x HEIGHT
Separable filter

- Separable 2D filter is outer product of two 1D filters
- Implication:
  - Can implement 2D convolution as two 1D convolutions

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

\[ F = \begin{bmatrix}
1 & 0 & -1 \\
2 & 0 & -2 \\
1 & 0 & -1
\end{bmatrix} = \begin{bmatrix}
1 \\
2 \\
1
\end{bmatrix} \begin{bmatrix}
1 & 0 & -1
\end{bmatrix} \]
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+11] * 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 = 6 x WIDTH x HEIGHT

For NxN filter: 2N x WIDTH x HEIGHT

But ... incurs bandwidth cost of writing and reading tmp_buf. (and footprint overhead of storing tmp_buf)
Two-pass image 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+11] * 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:
Must read each element of input and must write each element of output.

Data from `input` reused 3 times. (immediately reused 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)

Data from `tmp_buf` reused 3 times (but 3 rows of image data are accessed in between)
- Never load required data more than once if cache can contain three rows of image
- Perfect use of cache lines (don’t load unnecessary data)

Two pass: loads/stores to `tmp_buf` are overhead (artifact of 2-pass implementation: not intrinsic)
Two-pass image blur, chunked

```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+11] * 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;
    }
}
```

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

`tmp_buf` (Wx3)

output WxH

Produce 3 rows of `tmp_buf`

Combine them together to get one row of output

Total work per row of output:
step 1: 3 rows = 3 x 3W work
step 2: 1 row = 3W work
total: 12 x W x H work
Two-pass image blur, chunked

```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+11] * 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:**
- Let CHUNK_SIZE = 16
- step 1: 18 x 3W work
- step 2: 16 x 3W work
- total: \((2 + 2/16) \times 3 \times W \times H\) work
Conflicting goals (again...)  

- Want to be work efficient  
- Want to be parallel (multi-core, SIMD within core)  
- Want to take advantage of locality when present
  - Ideal: bandwidth cost of implementation is 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
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_loadu_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_interleaved_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_loadu_si128(tmpPtr+(2*256)/8);
                        b = _mm_loadu_si128(tmpPtr+256/8);
                        c = _mm_loadu_si128(tmpPtr++);
                        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                        avg = _mm_interleaved_epi16(sum, one_third);
                        _mm_store_si128(outPtr++, avg);
                    }
                }
            }
        }
    }
}
```
Example: Halide 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

```
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;
}
```

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

Algorithms are a series of functions (think: a function is a pipeline stage)

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

NOTE: execution order and storage are unspecified by functional abstraction! Implementation can evaluate, reevaluate, cache individual points as desired!
Halide program as a pipeline

```halide
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: separate specification of image processing algorithm (machine independent) from specification of schedule (machine dependent mapping)

- Given algorithm and schedule description, Halide attempts to generate high quality code for a target machine
  - Domain scope:
    - 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.

**“Root”**

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

**“Sliding Window”**

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

**“Inline”**

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

**“Chunked”**
Halide schedule: domain iteration

Specify both order and how to parallelize (multi-thread, SIMD vector)

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
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 is not magic, but its abstractions allow rapid exploration of optimization space, allowing programmer to reach optimal points quickly
Summary: image processing basics

- Same trade-offs arise
  - Maintain work efficiency
  - Maintain parallelism
  - Achieve bandwidth efficiency (exploit locality when it exists)

- A well-tuned piece of C code can be an order of magnitude faster than a basic C implementation
  - A few lines of code turns into many
  - Scheduling decisions not portable across machines (different compute/BW trade-offs, different cache sizes, different instruction sets, specialized HW, ...)

- Halide is one attempt to raise the level of abstraction