Fast Image Processing using Halide

Andrew Adams (Google)
Dillon Sharlet (Google)
Jonathan Ragan-Kelley (Stanford/MIT)
Zalman Stern (Google)
Marc Levoy (Google/Stanford)
Frédéric Durand (MIT)
High demand for efficient image processing
Writing fast image processing pipelines is hard.

Halide is a language that makes it easier.

**Big idea: separate algorithm from optimization**
programmer defines both - no “*Sufficiently Smart Compiler*” needed
algorithm becomes simple, modular, portable
exploring optimizations is much easier
C/C++ is slow

void box_filter_3x3(const Image &in, Image &blury) {
    Image blurx(in.width(), in.height());  // allocate temporary array

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}

9.96 ms/megapixel
(quad core x86)
An optimized implementation is 11x faster than a naïve implementation.

```c
void box_filter_3x3(const Image &in, Image &blury) {
  __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 blurx((256/8)*(*(*(inPtr+1)-1)));
    // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32; y++) {
        const uint16_t *inPtr = &((in[yTile+y]-1));
        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(blurxPtr++, avg);
          inPtr += 8;
        }
        blurxPtr = blurx;
      }
      __m128i *outPtr = (_m128i*)&((blury[yTile+y]-1));
      for (int x = 0; x < 256; x += 8) {
        a = _mm_loadu_si128((__m128i*)((blurxPtr+(*(*(blurxPtr+1)-1)))/8))
        b = _mm_loadu_si128((__m128i*)((blurxPtr+256/8)));
        c = _mm_load_si128((__m128i*)((blurxPtr+256)));
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_si128(outPtr++, avg);
      }
    }
  }
}
```

0.9 ms/megapixel (quad core x86)
A Modern Multi-Core Processor
A Modern Multi-Core Processor

Scalar mode
(one instruction produces one result)

\[
\begin{align*}
a[j] + b[j] &= a + b
\end{align*}
\]

SIMD processing
(one instruction can produce multiple results)

\[
\begin{align*}
\end{align*}
\]

\[
\begin{align*}
\end{align*}
\]

\[
\begin{align*}
\end{align*}
\]
An optimized implementation is 11x faster

```c
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  __m128i a, b, c, sum, avg;

  __m128i *blurx = (256/8) * (xTile + 256); // allocate tile blurx array
  __m128i *blurxPtr = blurx;
  __m128i *outPtr = (256/8) * (yTile + y); // allocate tile blury array

  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    for (int xTile = 0; xTile < in.width(); xTile += 32) {
      sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
      avg = _mm_mulhi_epi16(sum, one_third);
      _mm_store_si128(blurxPtr++, avg);
    }
    blurxPtr = blurx;
  }

  for (int y = -1; y < 32; y++) {
    for (int x = 0; x < 256; x += 8) {
      a = __mm_load_si128((__m128i*)(inPtr - 1));
      b = __mm_load_si128((__m128i*)(inPtr));
      c = __mm_load_si128((__m128i*)(inPtr + 1));
      sum = __mm_add_epi16(__mm_add_epi16(a, b), c);
      avg = __mm_mulhi_epi16(sum, one_third);
      _mm_store_si128(outPtr++, avg);
    }
  }
}
```

0.9 ms/megapixel (quad core x86)
void box_filter_3x3(const Image &in, Image &blury) {
  __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 blurx[((32 / 8) + 1) * 2]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = (in[yTile+y][xTile]);
      for (int y = -1; y < 32; y++) {
        const uint16_t *inPtr = &in[yTile+y][xTile];
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)(inPtr-1));
          b = _mm_loadu_si128((__m128i*)(inPtr));
          c = _mm_load_si128((__m128i*)(inPtr+1));
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
          inPtr += 8;
        }
        blurxPtr = blurx;
      }
    }
  }
}
An optimized implementation is 11x faster

void box_filter_3x3(const Image &in, Image &blury)
{
    __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 blurx = _mm_loadu_si128((__m128i*)(inPtr - 1));
        __m128i b = _mm_loadu_si128((__m128i*)(inPtr + 1));
        __m128i 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(blurxPtr++, avg);
        inPtr += 8;
    }
    blurxPtr =blurx;
    for (int y = 0; y < 32; y++)
    {
        __m128i *outPtr = (__m128i*)(&blury[yTile + y][xTile]);
        for (int x = 0; x < 256; x += 8)
        
            a = _mm_loadu_si128((__m128i*)(blurxPtr + (2 * 256) / 8));
            b = _mm_loadu_si128(blurxPtr + 256 / 8);
            c = _mm_load_si128(blurxPtr++);
            sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
            avg = _mm_mulhi_epi16(sum, one_third);
            _mm_store_si128(outPtr++, avg);
        })
    
);blurxPtrحكم

parallelism
distribute across threads
SIMD parallel vectors

locality
compute in tiles
interleave tiles of blurx, blury
store blurx in local cache

0.9 ms/megapixel
(quad core x86)
Executing the pipeline

```c
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);

    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurx([256/8]*(*xTile)); // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &in[yTile+y][xTile];
                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(blurxPtr++, avg);
                    inPtr += 8;
                }
                blurxPtr = blurx;
            }
        }
    }
}
```
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurx[((256 / 8) * (yTile + 32))]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32; y++) {
                const uint16_t *inPtr = &in[yTile+y][xTile]);
                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(blurxPtr++, avg);
                    inPtr += 8;
                }
            }
            blurxPtr = blurx;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i *)&blury[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurxPtr+((256 / 8));
                    b = _mm_load_si128(blurxPtr+256 / 8);
                    c = _mm_load_si128(blurxPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
Executing the pipeline

```c
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurx[(256/8) * (256/8)]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = &blurx;
            for (int y = -1; y < 32; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                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(blurxPtr++, avg);
                    inPtr += 8;
                }
               blurxPtr = blurx;
            } for (int x = 0; x < 32; x++) {
                __m128i *outPtr = (_m128i*)(&blury[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurxPtr+256/8);
                    b = _mm_load_si128(blurxPtr+256/8);
                    c = _mm_load_si128(blurxPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a,b),c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }}
        }
    }
}
```
Fusing stages globally interleaves execution

```cpp
void box_filter_3x3(const Image &in, Image &blury) {
__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 blurx[((256/8)*(x++)+256/8)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
        __m128i *blurxPtr = blurx;
        for (int y = -1; y < 32+1; y++) {
            const uint16_t *inPtr = &in[yTile+y][xTile]);
            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(blurxPtr++,avg);
                inPtr += 8;
            }
            blurxPtr = blurx;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i *)&blury[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurxPtr@(256/8));
                    b = _mm_load_si128(blurxPtr+256/8);
                    c = _mm_load_si128(blurxPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a,b),c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
```
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        _m128i blurx[32];
        for (int y = -1; y < 32; y++) {
            const uint16_t *inPtr = &in[yTile+y][0];
            for (int x = 0; x < 256; x += 8) {
                a = _mm_load_u128((__m128i *)inPtr);
                b = _mm_load_u128((__m128i *)(inPtr+1));
                c = _mm_load_u128((__m128i *)(inPtr+2));
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mul_hi_epi16(sum, one_third);
                _mm_store_u128(outptr++, avg);
                inPtr += 8;
            }
            outptr = (uint16_t *)(&blury[yTile+y][0]);
            for (int x = 0; x < 256; x += 8) {
                a = _mm_load_u128(blurxptr);
                b = _mm_load_u128(blurxptr+256/8);
                c = _mm_load_u128(blurxptr+512/8);
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mul_hi_epi16(sum, one_third);
                _mm_store_u128(outptr++, avg);
            }
        }
    }
}
Fusion is a complex tradeoff

```c
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurx[8](256/8*(x+2)); // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
        __m128i *blurxPtr = blurx;
        for (int y = -1; y < 32; y++) {
            const uint16_t *inPtr = &in[yTile+y][xTile];
            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_mulhi_epi16(sum, one_third);
                _mm_storeu_si128(blurxPtr++, avg);
                inPtr += 8;
            }
        }
    for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)&blury[yTile+y][xTile];
        for (int x = 0; x < 256; x += 8) {
            a = _mm_loadu_si128(blurxPtr+256/8);
            b = _mm_loadu_si128(blurxPtr+256/8);
            c = _mm_loadu_si128(blurxPtr++);
            sum = _mm_add_epi16(_mm_add_epi16(a,b),c);
            avg = _mm_mulhi_epi16(sum, one_third);
            _mm_storeu_si128(outPtr++, avg);
        }
    }
    blurxPtr = blurx;
    for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)&blury[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
            a = _mm_loadu_si128(blurxPtr+256/8);
            b = _mm_loadu_si128(blurxPtr+256/8);
            c = _mm_loadu_si128(blurxPtr++);
            sum = _mm_add_epi16(_mm_add_epi16(a,b),c);
            avg = _mm_mulhi_epi16(sum, one_third);
            _mm_storeu_si128(outPtr++, avg);
        }
    }
}}}
```
Fusion is a complex tradeoff

```c
void box_filter_3x3(const Image &in, Image &blury) {
  __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 blurx[256/8*(x+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      m128i *blurxPtr = &blurx;
      for (int y = -1; y < 32; y++) {
        const uint16_t *inPtr = &in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i *)(inPtr+1));
          b = _mm_loadu_si128((__m128i *)(inPtr+2));
          c = _mm_loadu_si128((__m128i *)(inPtr));
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_storeu_si128(blurxPtr++, avg);
          inPtr += 8;
        }
        blurxPtr += 32;
      }
    }
}
```
The choice space

For each stage:

Question 1) In what order should it compute its values?
In what order should I compute my values?

Serial y,
Serial x
In what order should I compute my values?

Serial x, Serial y
In what order should I compute my values?

Serial y,
Vectorize x by 4
In what order should I compute my values?

Parallel y,
Vectorize x by 4
In what order should I compute my values?

Split $x$ by 4,
Split $y$ by 4.
Serial $y_{\text{outer}}$,
Serial $x_{\text{outer}}$,
Serial $y_{\text{inner}}$,
Serial $x_{\text{inner}}$
The choice space

For each stage:

Question 1) In what order should it compute its values?

Question 2) When should it compute its inputs?
When should I compute my inputs?

Poor locality

All at once, ahead of time
When should I compute my inputs?

Redundant recompute

As needed, discarding after use
When should I compute my inputs?

Poor parallelism

As needed, reusing old values
Some more points within the choice space
Some more points within the choice space

<table>
<thead>
<tr>
<th>input</th>
<th>blurred in x</th>
<th>output</th>
</tr>
</thead>
</table>


Some more points within the choice space
Scheduling is a complex tradeoff.

3x3 box filter

local Laplacian filters

[Paris et al. 2010, Aubry et al. 2011]
A typical mobile SoC (System on Chip) has a CPU, DSP and a GPU. Need ability to target different architectures and exploit them effectively.

Developing highly optimized low-level code for each architectures is not scalable.
Existing languages make optimizations hard

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>vectorization</td>
<td>fusion</td>
</tr>
<tr>
<td>multithreading</td>
<td>tiling</td>
</tr>
</tbody>
</table>

C - parallelism + tiling + fusion are hard to write or automate

CUDA, OpenCL, shaders - data parallelism is easy, fusion is hard

*libraries don’t help:*
BLAS, IPP, MKL, OpenCV, MATLAB
Halide: *decouple* algorithm from schedule

**Algorithm:** *what* is computed  
**Schedule:** *where* and *when* it’s computed

Easy for programmers to build pipelines  
simplifies algorithm code  
improves modularity

Easy for programmers to specify & explore optimizations  
fusion, tiling, parallelism, vectorization  
can’t break the algorithm

Easy for the compiler to generate fast code
The algorithm: pipelines as pure functions

Pipeline stages are functions from coordinates to values
no side effects
coordinates span an infinite domain
boundaries and required regions are inferred

Execution order and storage are unspecified
points can be evaluated (or reevaluated) in any order
results can be cached, duplicated, or recomputed anywhere

3x3 blur as a Halide algorithm:
Func blurx, blury;
Var x, y;
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
The schedule: producer-consumer interleaving

For each stage:

Question 1) In what order should it compute its output?

Question 2) When should it compute its inputs?
blur_x.compute_root();
blur_x.compute_at(blur_y, x);
blur_x.store_root().compute_at(blur_y, x);

blur_x.compute_at(blur_y, x)
  .vectorize(x, 4);
blur_y.tile(x, y, xi, yi, 8, 8)
  .parallel(y)
  .vectorize(xi, 4);
blur_y.split(x, x, xi, 8)
  .vectorize(xi, 4).parallel(x);
blur_y.split(y, y, yi, 8)
  .vectorize(x, 4)
  .parallel(y);

blur_x.store_root()
  .compute_at(blur_y, y)
  .split(x, x, xi, 8)
  .vectorize(xi, 4).parallel(x);
blur_y.split(x, x, xi, 8)
  .vectorize(xi, 4).parallel(x);
blur_x.store_at(blur_y, y)
  .compute_at(blur_y, yi)
  .vectorize(x, 4);
blur_y.split(y, y, yi, 8)
  .vectorize(x, 4)
  .parallel(y);
Halide

0.9 ms/megapixel

Func box_filter_3x3(Func in) {
    Func blurx, blury;
    Var x, y, xi, yi;

    // The algorithm
    blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

    // The schedule
    blury.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    blurx.compute_at(blur_y, x).vectorize(x, 8);

    return blury;
}
void box_filter_3x3(const Image &in, Image &blury) {
    __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 blurx[[(256/8)*(32-2)]]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32-1; y++) {
                const uint16_t *inPtr = (&(in[yTile+y][xTile]));
                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_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurxPtr++, avg);
                    inPtr += 8;
                }
            }
            blurxPtr = blurx;
            for (int y = 0; y < 32; y++) {
                _m128i *outPtr = (_m128i*)(blury[yTile+y][xTile]));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128(blurxPtr+(2*256)/8);
                    b = _mm_loadu_si128(blurxPtr+256/8);
                    c = _mm_loadu_si128(blurxPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}

C++

void box_filter_3x3(const Image &in, Image &blury) {
    __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 blurx[[(256/8)*(32-2)]]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32-1; y++) {
                const uint16_t *inPtr = (&(in[yTile+y][xTile]));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128(blurxPtr-1);
                    b = _mm_loadu_si128(blurxPtr+1);
                    c = _mm_loadu_si128(blurxPtr);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurxPtr++, avg);
                    inPtr += 8;
                }
            }
            blurxPtr = blurx;
            for (int y = 0; y < 32; y++) {
                _m128i *outPtr = (_m128i*)(blury[yTile+y][xTile]));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128(blurxPtr+(2*256)/8);
                    b = _mm_loadu_si128(blurxPtr+256/8);
                    c = _mm_loadu_si128(blurxPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
The FCam Raw Pipeline
[Adams et al. 2010]

Converts raw image sensor data into an image

The original code is 463 lines of ARM assembly and intrinsics in one big function

Rewritten in Halide, it is 2.75x less code, and runs 5% faster

- Denoise
- Demosaic
- Color correct
- Tone curve
Local Laplacian Filters
[Paris et al. 2010, Aubry et al. 2011]

Pyramid-based algorithm for increasing local contrast

Original is 262 lines of optimized C++ using OpenMP and Intel Performance Primitives (IPP)

Rewritten in Halide: 62 lines of code for the algorithm, 7 lines of code for the schedule

2.1x faster on CPU, 7x faster on GPU
Local Laplacian Filters
[Paris et al. 2010, Aubry et al. 2011]

Pyramid-based algorithm for increasing local contrast

Original is 262 lines of optimized C++ using OpenMP and Intel Performance Primitives (IPP)

Rewritten in Halide: 62 lines of code for the algorithm, 7 lines of code for the schedule

2.1x faster on CPU, 7x faster on GPU
An accelerated bilateral filter

Original is 122 lines of clean C++

Halide version is 34 lines of algorithm, and 6 lines of schedule

On the CPU, 5.9x faster

On the GPU, 2x faster than Chen’s hand-written CUDA version
Halide is embedded in C++

Build Halide functions and expressions using C++

Evaluate Halide functions immediately
Just-in-time compile to produce and run a Halide pipeline

Or statically compile to an object file and header
One C++ program creates the Halide pipeline
When run, it produces an object file and header
You link this into your actual program
The Halide Compiler

- Halide Functions
- Halide Schedule

Imperative Blob

LLVM bitcode

- X86 (+SSE/AVX)
- ARM (+neon)
- CUDA
Example Pipelines

Unsharp Mask
More apps in the public repo

A faster FFT than FFTW

A faster Gaussian blur than OpenCV

A faster matrix multiply than Eigen

http://halide-lang.org/
Getting Started

Public website at http://halide-lang.org

Tutorials at http://halide-lang.org/tutorials

Some limitations
Only handles feed-forward pipelines
Only images - no trees or lists or hash tables
Schedule must be specified manually

We welcome contributions
http://github.com/Halide/halide
Automatically Scheduling Halide Image Processing Pipelines

Ravi Teja Mullapudi (CMU)
Andrew Adams (Google)
Dillon Sharlet (Google)
Jonathan Ragan-Kelley (Stanford)
Kayvon Fatahalian (CMU)
Few developers have the skill set to author highly optimized schedules.

Algorithm description:

```
Var x, y;
Func f, g;
g(x,y) = f(x,y) + ...
h(x) = g(x,y) + ...
```

Schedule (machine mapping):
- parallelize y loop
- tile output dims
- vectorize y loop

> 10x Faster Implementation
Automatic scheduling of image processing pipelines

Image processing algorithm developers

Algorithm description

Var x, y;
Func f, g;
g(x,y) = f(x,y) + ...
h(x) = g(x,y) + ...

Generates expert-quality schedules in seconds

> 10x Faster Implementation

Scheduling Algorithm
# Auto scheduler 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><strong>Synthetic depth-of-field</strong></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>
Auto scheduler performs comparably to experts

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

Baseline schedules exploit multi-core and vector parallelism but no grouping.

Performance relative to experts (6 core Xeon CPU)
Auto scheduler can save time for experts

Throughput vs Time (min) for:
- Non-local means
- Max filter
- Lens blur

Legend:
- Green: Auto scheduler
- Gray: Dillon
- Red: Andrew
Fast image processing is hard because you need to optimize for locality and parallelism.

Halide helps, by separating the algorithm from the optimizations (the schedule). Code becomes more modular, readable, and portable, making it easier to explore different optimizations.

Get the compiler at http://halide-lang.org