Fast Image Processing using Halide

Andrew Adams (Google)
Jonathan Ragan-Kelley (Berkeley)
Zalman Stern (Google)
Steven Johnson (Google)
Dillon Sharlet (Google)
Patricia Suriana (Google)
Pixel smartphone camera review: At the top
By David Cardinal - Tuesday October 04 2016
Mobile Review

“The highest-rated smartphone camera we have ever tested”
Writing fast image processing pipelines is hard.

Halide is a language that makes it easier.

Big idea: separate algorithm from optimization
programmer defines both
algorithm becomes simple, modular, portable
exploring optimizations is much easier
C/C++ is slow

```c
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)
Scalar C code
An optimized implementation is 11x faster

```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; y++) {
            const uint16_t *inPtr = &(in[yTile+y][xTile]);
            for (int x = 0; x < 256; x += 8) {
                a = __mm_loadu_si128((__m128i*)inPtr);
                b = __mm_loadu_si128((__m128i*)(inPtr-1));
                c = __mm_loadu_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;
        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+2*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);
            }
        }
    }
}
```

11x faster than a naïve implementation

0.9 ms/megapixel (quad core x86)
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[(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-A));
                    b = _mm_loadu_si128((__m128i *)(inPtr-I));
                    c = _mm_loadu_si128((__m128i *)(inPtr+I));
                    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+(2*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);
                }}}}}}

parallelism
distribute across threads
SIMD parallel vectors

0.9 ms/megapixel
(quad core x86)
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[(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; 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+2*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);
            }}}}
An optimized implementation is 11x faster

```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; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)inPtr);
                    b = _mm_loadu_si128((__m128i*)(inPtr+1));
                    c = _mm_loadu_si128((__m128i*)(inPtr+2));
                    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 += 256;
            }
        }
    }
```
Scalar C code
Scalar C code + SIMD
Parallel C code + SIMD
Parallel C code + SIMD + fusion for locality
Parallel C code + SIMD + fusion for locality +
Executing the pipeline

```c
void __m128i #pragma omp parallel for for

  const uint16_t *inPtr = &(
      a = _mm_loadu_si128((
      b = _mm_loadu_si128((
      c = _mm_load_si128((
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_si128(blurxPtr++, avg);
      inPtr +=
  )
 .blurxPtr = blurx;

  a = _mm_load_si128(blurxPtr+{(
  b = _mm_load_si128(blurxPtr+{(
  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 __m128i _mm_fn(const uint16_t *inPtr)
{
    _mm_store_si128(outPtr++, avg);
}
```
void __m128i  
#pragma omp parallel for
for

const uint16_t *inPtr = &(
    a = _mm_loadu_si128((
    b = _mm_loadu_si128((
    c = _mm_load_u128((
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
    avg = _mm_mulhi_epi16(sum, one_third);
    _mm_store_si128(blurxPtr++, avg);
    inPtr +=
));
blurxPtr = blurx;

    a = _mm_load_u128(blurxPtr+;
    b = _mm_load_u128(blurxPtr+);
    c = _mm_load_u128(blurxPtr++);
    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
    avg = _mm_mulhi_epi16(sum, one_third);
    _mm_store_u128(outPtr++, avg);
));)))

Executing the pipeline

input

blurx

blury
Fusing stages globally interleaves execution

```c
void __m128i #pragma omp parallel for for

const uint16_t *inPtr = &(
    a = _mm_loadu_si128((
    b = _mm_loadu_si128((
    c = _mm_load_si128((
    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
    avg = _mm_mulhi_epi16(sum, one_third);
    _mm_store_si128(blurxPtr++, avg);
    inPtr +=
    )
    blurxPtr = blurx;
)
}

a = _mm_load_si128(blurxPtr+{
b = _mm_load_si128(blurxPtr+{
    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

```c
void
__m128i
#pragma omp parallel for
for
const uint16_t *inPtr = &(
    a = _mm_loadu_si128((
    b = _mm_loadu_si128((
    c = _mm_load_si128((
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_si128(blurxPtr++, avg);
        inPtr +=
    )
    blurxPtr = blurx;

    a = _mm_load_si128(blurxPtr+((
    b = _mm_load_si128(blurxPtr+((
    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);
    )))))
```
Fusion is a complex *tradeoff*

```c
void
__m128i
#pragma omp parallel for
for

const uint16_t *inPtr = &(
    a = _mm_loadu_si128((
    b = _mm_loadu_si128((
    c = _mm_load_si128((
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_si128(blurxPtr++, avg);
    inPtr +=
    blurxPtr = blurx;

    a = _mm_load_si128(blurxPtr+;
    b = _mm_load_si128(blurxPtr+;
    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);
    ))))))
```
Fusion is a complex *tradeoff*

```c
void __m128i #pragma omp parallel for
for

    
const uint16_t *inPtr = &(
    
a = __mm_loadu_si128((
   b = __mm_loadu_si128((
   c = __mm_load_si128((
   sum = __mm_add_epi16(__mm_add_epi16(a, b), c);
   avg = __mm_mulhi_epi16(sum, one_third);
   __mm_store_si128(blurxPtr++, avg);
   inPtr +=
))}
blurxPtr = blurx;

    
a = __mm_load_si128(blurxPtr+)(
   b = __mm_load_si128(blurxPtr+)(
   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);
))}))}
```
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?

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

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

As needed, reusing old values
Some more points within the choice space
Some more points within the choice space
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]
Local Laplacian Filters

[Paris et al. 2010, Aubry et al. 2011]
Existing languages make optimizations hard

Parallelism
- vectorization
- multithreading

Locality
- fusion
- tiling

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
optimized kernels compose into inefficient pipelines (no fusion)
**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_x.store_root()
  .compute_at(blur_y, y)
  .split(x, x, xi, 8)
  .vectorize(xi, 4).parallel(x);
bray_y.split(y, y, yi, 8)
  .vectorize(x, 4)
  .parallel(y);
blur_y.split(x, x, xi, 8)
  .vectorize(xi, 4).parallel(x);
blur_x.store_root().compute_at(blur_y, y)
  .compute_at(blur_y, yi)
  .vectorize(x, 4);
blur_y.split(y, y, yi, 8)
  .vectorize(x, 4)
  .parallel(y);
blur_x.store_at(blur_y, y)
Func box_filter_3x3(Func in) {
    Func blur_x, blur_y;
    Var x, y, xi, yi;

    // The algorithm
    blur_x(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

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

    return blur_y;
}
**Halide**

0.9 ms/megapixel

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

    // The algorithm - no storage, order
    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 - defines order, locality; implies storage
    blury.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    blurx.compute_at(blury, x).vectorize(x, 8);

    return blury;
}
```

**C++**

0.9 ms/megapixel

```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((__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_load_si128(blurxPtr+(2*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);
                }
            }
        }
    }

```
Break
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
Halide Functions

Halide Schedule

Lowering passes

LLVM bitcode
Halide Functions

Halide Schedule

Lowering passes
Transforms according to schedule
Optimizes
Lowers level of abstraction

LLVM bitcode
Halide Functions

Halide Schedule

Lowering passes

Transforms according to schedule
Optimizes
Lowers level of abstraction

Halide IR

LLVM bitcode
Halide Functions

Halide Schedule

Lowering passes

Transforms according to schedule
Optimizes
Lowers level of abstraction

Halide IR

Must span many levels of abstraction
Starts as functional
Ends as loop nests

LLVM bitcode
Applications
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
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
The Bilateral Grid  
[Chen et al. 2007]

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
“Snake” Image Segmentation
[Li et al. 2010]

Segments objects in an image using level-sets

Original is 67 lines of matlab

Halide version is 148 lines of algorithm and 7 lines of schedule

On the CPU, 70x faster
matlab is memory-bandwidth limited

On the GPU, 1250x faster
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/
Problems with Halide
Domains are amorphous

- Linear Algebra
- Machine Learning
- Computer Vision
- Image processing
- Scientific Computation
- Video Processing
Domains are amorphous
Domains are amorphous

Linear Algebra

Scientific Computation

Machine Learning

Image processing

Halide

Video Processing

Computer Vision
Metaprogramming is weird

Halide is embedded in C++. This means:

You write a C++ program

which you must compile

when you run it, it builds a Halide pipeline in memory

then compiles it to a .o file

which you then link into another C++ program

which you deploy
Aspect-oriented programming is weird

The schedule and the algorithm are two aspects of the Halide program

The schedule refers to the Funcs and Vars in the algorithm

This means those names must be in the scope that contains the scheduling code

Awkward but workable, just put the algorithm and schedule together in one scope
Aspect-oriented programming is weird

Consider a long pipeline like HDR+ that you want to break into individually-testable components:

The algorithm for each component refers to earlier stages
    A Func must refer to its inputs

The schedules for each component refer to later stages
    e.g. A Func should be computed per tile of the next stage

But the algorithm and schedule for each component have to live in the same scope due to the naming issue...
Scheduling is hard

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

    // The algorithm - no storage, order
    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 - defines order, locality; implies storage
    blury.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    blurx.compute_at(blury, x).vectorize(x, 8);

    return blury;
}
```

Easy

Hard
Scheduling is hard

All programming is done to an abstract machine.

To write code, you must be able to mentally emulate that abstract machine.
  What does “x = 5;” mean in C?

A Halide algorithm is a program for an abstract machine that does simple arithmetic on scalar types.
  Easy to mentally emulate.

A Halide schedule is a program for an abstract machine that builds and manipulates loop nests.
  Hard to hold a loop nest in your head.
Halide at Google
Conclusion

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
Fast image processing is hard because you need to optimize for locality \textit{and} parallelism

Halide helps, by separating the algorithm from the optimizations (the \textit{schedule})
code becomes more modular, readable, and portable
makes it easier to explore different optimizations

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