Decoupling Algorithms and Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley* Andrew Adams* Sylvain Paris†
Marc Levoy‡ Saman Amarasinghe* Frédo Durand*
* MIT CSAIL †Adobe ‡Stanford University

Presented by:
Sweta Yamini Pothukuchi
CS598 APK Class Presentation

Motivation

Naïve clean C++

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

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

Computation of a 3X3 box filter using a composition of a 1X3 and a 3X1 box filter on a quad-core x86 CPU

Hand-tuned 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);
     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_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);
}}}}}
```

Motivation

Naïve clean C++

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

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

9.94 ms per megapixel

Computation of a 3X3 box filter using a composition of a 1X3 and a 3X1 box filter on a quad-core x86 CPU

Optimizations performed:

- Multithreading
- Vectorization
- Tiling
- Fusion

Hand-tuned 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);
     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_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);
}}}}}
```

0.90 ms per megapixel

Motivation

- Main target – Image processing pipelines
- Wide and deep workloads
  - Many data-parallel stages
  - Benefit from parallelization across pixels
  - Memory bound as they have little work per memory access
  - Tiling and fusion across stages improves producer-consumer locality
- Best optimizations are machine dependent

Key Idea

- Separate the algorithm from the schedule
- Algorithm – specifies the computation to be performed
- Schedule – specifies the optimizations and transformations to determine when an actual computation occurs

Key Idea

- Separate the algorithm from the schedule
- Algorithm – specifies the computation to be performed
- Schedule – specifies the optimizations and transformations to determine when an actual computation occurs

HALIDE

- Programmer provides both algorithm and schedule
- Compiler(halide) combines them to generate efficient code

BLUR in 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;

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

 return blurred;
}
```

0.90 ms per megapixel

Halide

- Algorithm –
  - Purely functional specification of the value at each point
- Schedule –
  - Order of execution of points within the domain of a function, including parallelism and vectorization
  - Relative order of execution of points of one function to another, specifying fusion of functions
  - Specification of memory location to which an evaluated function is stored
  - Whether a value is recomputed or location from where it is to be loaded

Representing algorithms

- Each stage is a pure function defined over an infinite integer domain or a reduction over a bounded domain
- Expressions in functions include
  - Arithmetic and logical operations
  - Loads from external images
  - If-then-else expressions (semantically equivalent to ternary operator(? :) in C)
  - References to named values (other functions or expressions defined by functional *let* construct)
  - Calls to other scalar functions

Examples

Gradient

```
Func gradient("gradient");
gradient(x, y) = x + y;
```

Multistage

```
Func producer("producer"), consumer("consumer");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y)
                + producer(x, y+1)
                + producer(x+1, y)
                + producer(x+1, y+1))/4;
```

Boundary Condition

```
Buffer<uint8_t> input = load_image("images/rgb.png);
Func clamped("clamped”);
Expr clamped_x = clamp(x, 0, input.width()-1);
// clamp(x, a, b) is equivalent to max(min(x, b), a).
Expr clamped_y = clamp(y, 0, input.height()-1);
clamped(x, y, c) = input(clamped_x, clamped_y, c);
```

Reductions

- Reductions require
  - An initial value function which specifies a value for each value in the output domain
  - A recursive reduction function which redefines the value at points given by the output coordinate expression in terms of prior values of function
  - A reduction domain bounded by minimum and maximum values in each dimension
- Reduction meaning may change depending on the order of application of the reduction, so the order is specified by the reduction domain

Example algorithm – Histogram equalization

```
UniformImage in(UInt(8), 2);
Func histogram, cdf, out;
RDom r(0, in.width(), 0, in.height()), ri(0, 255);
Var x, y, i;

histogram(in(r.x, r.y))++;
cdf(i) = 0;
cdf(ri) = cdf(ri-1) + histogram(ri);
out(x, y) = cdf(in(x, y));
```

Example algorithm – Histogram equalization

```
UniformImage in(UInt(8), 2);
Func histogram, cdf, out;
RDom r(0, in.width(), 0, in.height()), ri(0, 255);
Var x, y, i;

histogram(in(r.x, r.y))++;
cdf(i) = 0;
cdf(ri) = cdf(ri-1) + histogram(ri);
out(x, y) = cdf(in(x, y));
```

Reduction domain

Reduction

Example algorithm – Histogram equalization

```
UniformImage in(UInt(8), 2);
Func histogram, cdf, out;
RDom r(0, in.width(), 0, in.height()), ri(0, 255);
Var x, y, i;

histogram(in(r.x, r.y))++;
cdf(i) = 0;
cdf(ri) = cdf(ri-1) + histogram(ri);
out(x, y) = cdf(in(x, y));
```

Reduction domain

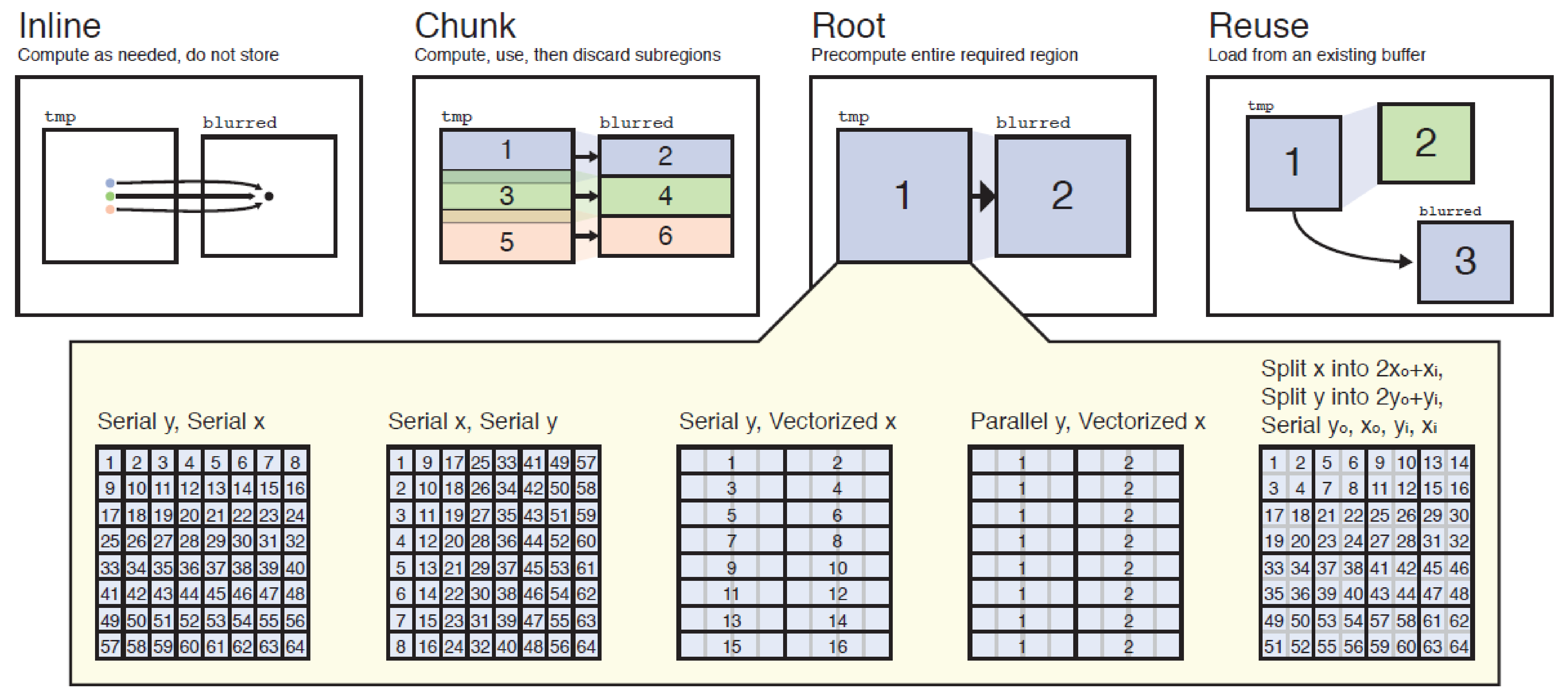
Initial value

Scan

Schedules

- For each pipeline stage, specify how its inputs are evaluated starting from the final output of the pipeline
- Caller-callee relationships:
  - Inline – compute as needed, do not store
  - Root – precompute entire required region
  - Chunk – compute, use and discard subregions
  - Reuse – load from an existing buffer
- Within a function:
  - Sequential, parallel, unroll, vectorize, transpose, split (tile), gpu
  - Can chain schedules, e.g., **im.root().vectorize(x, 4).parallel(x)**
- Reductions:
  - A schedule for initialization
  - A Schedule for the reduction (deduced from the reduction domain by default)

Schedules



Inline
Compute as needed, do not store
tmp
blurred
Chunk
Compute, use, then discard subregions
tmp
blurred
1
2
3
4
5
6
Root
Precompute entire required region
tmp
blurred
1
2
Reuse
Load from an existing buffer
tmp
1
2
blurred
3
Serial y, Serial x
Serial x, Serial y
Serial y, Vectorized x
Parallel y, Vectorized x
Split x into 2xo+xi,
Split y into 2yo+yi,
Serial yo, xo, yi, xi

Schedule demonstration

Multistage

```
Func producer("producer"),
     consumer("consumer");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y)
                 + producer(x, y+1)
                 + producer(x+1, y)
                 + producer(x+1, y+1))/4;
```

`producer.compute_root();`

*image from http://halide-lang.org/tutorials/tutorial_lesson_08_scheduling_2.html

Schedule demonstration

Multistage

```
Func producer("producer"),
     consumer("consumer");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y)
                + producer(x, y+1)
                + producer(x+1, y)
                + producer(x+1, y+1))/4;
```

```
producer.compute_at(consumer, y);
```

*image from http://halide-lang.org/tutorials/tutorial_lesson_08_scheduling_2.html

Schedule demonstration

Multistage

```
Func producer("producer"),
     consumer("consumer");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y)
                + producer(x, y+1)
                + producer(x+1, y)
                + producer(x+1, y+1))/4;
```

```
producer.store_root();
producer.compute_at(consumer, y);
```

*image from http://halide-lang.org/tutorials/tutorial_lesson_08_scheduling_2.html

Schedule demonstration

Multistage

```
Func producer("producer"),
     consumer("consumer");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y)
                + producer(x, y+1)
                + producer(x+1, y)
                + producer(x+1, y+1))/4;
```

```
producer.store_root();
producer.compute_at(consumer, x);
```

*image from http://halide-lang.org/tutorials/tutorial_lesson_08_scheduling_2.html

Schedule demonstration

Multistage

```
Func producer("producer"),
     consumer("consumer");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y)
                + producer(x, y+1)
                + producer(x+1, y)
                + producer(x+1, y+1))/4;
```

```
Var x_outer, y_outer, x_inner, y_inner;
consumer.tile(x, y, x_outer, y_outer,
              x_inner, y_inner, 4, 4);

producer.compute_at(consumer, x_outer);
```

*image from http://halide-lang.org/tutorials/tutorial_lesson_08_scheduling_2.html

Schedule demonstration

```
Var yo, yi;
consumer.split(y, yo, yi, 16);
consumer.parallel(yo);
consumer.vectorize(x, 4);
producer.store_at(consumer, yo);
producer.compute_at(consumer, yi);
producer.vectorize(x, 4);
```

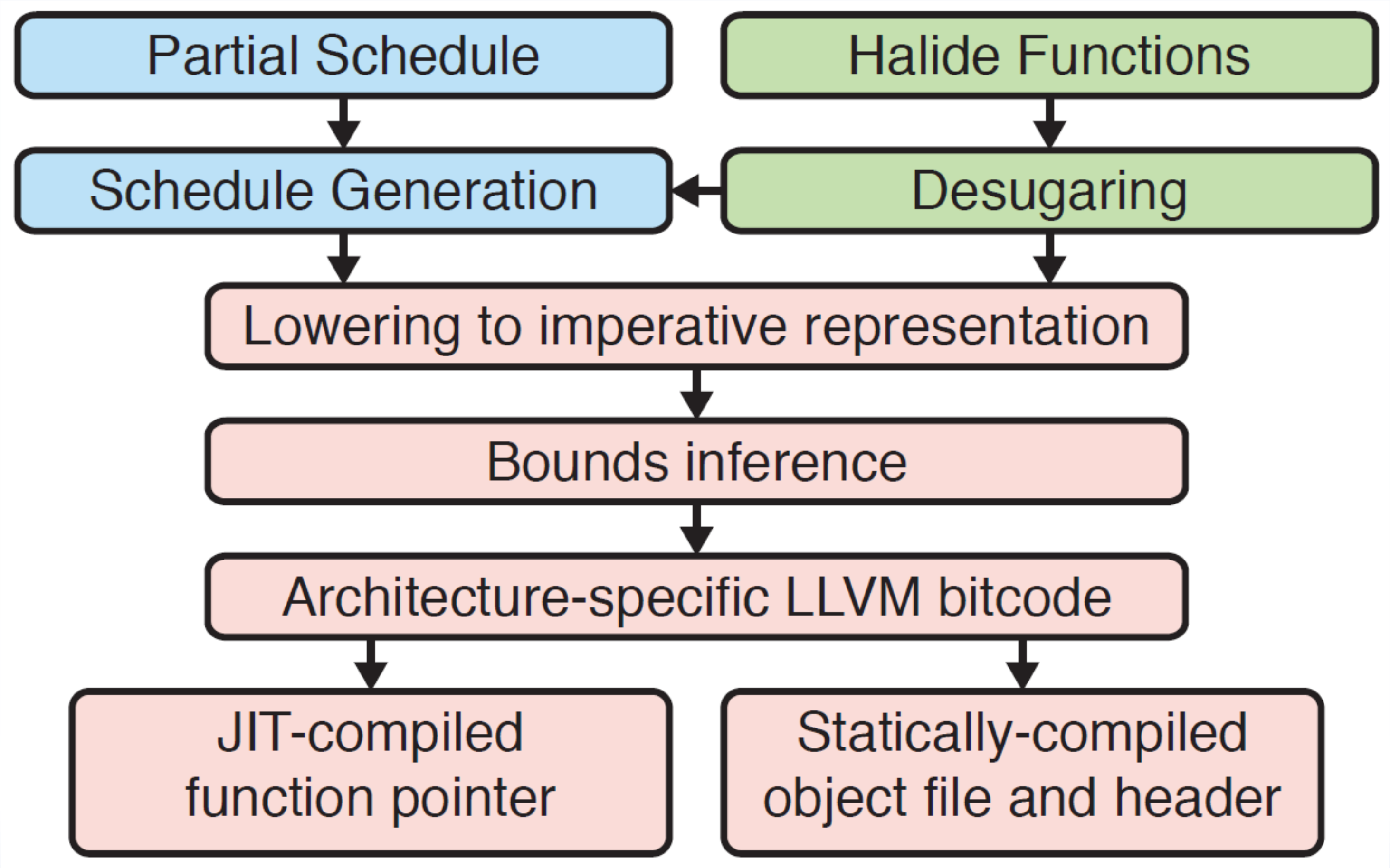
Multistage

```
Func producer("producer"),
     consumer("consumer");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y)
                + producer(x, y+1)
                + producer(x+1, y)
                + producer(x+1, y+1))/4;
```

(Image size of 160X160)

*image from http://halide-lang.org/tutorials/tutorial_lesson_08_scheduling_2.html

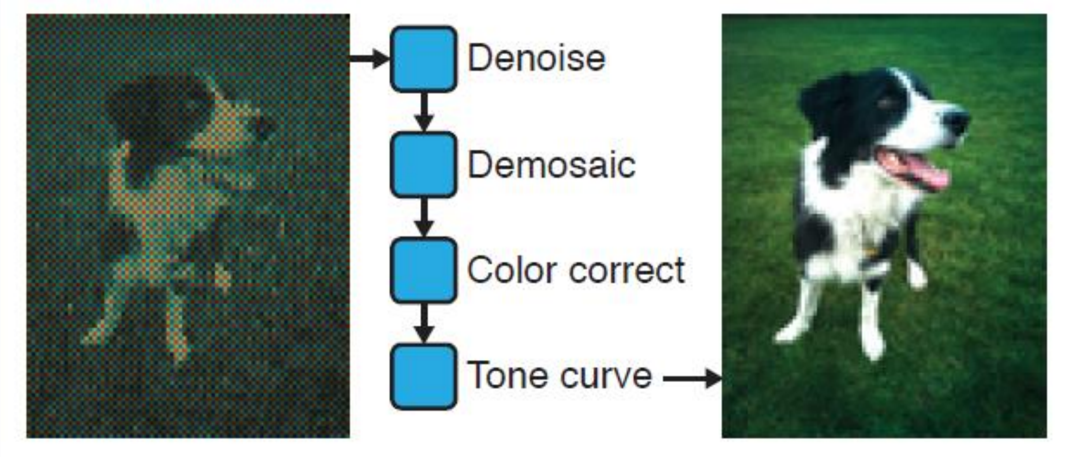
Compiler


Partial Schedule
Halide Functions
Schedule Generation
Desugaring
Lowering to imperative representation
Bounds inference
Architecture-specific LLVM bitcode
JIT-compiled function pointer
Statically-compiled object file and header

Code generation

- Generates machine code for ARM/NEON, x86/SSE and GPU/PTX
- Lowering to imperative form
  - Works on the pilepine from the output backwards and generates loopnests
- Bounds inference
  - Performed using symbolic interval arithmetic
  - Users can assist using min, max and clamp functions in schedule
  - Function realizations are added after bounds inference
- CPU code generation
  - LLVM IR code is generated from the imperative form
  - Parallelization is performed using a threadpool
  - Vectorization is performed using peephole optimizations to replace with architecture specific intrinsics
- GPU code generation
  - Generates both host and device code
  - Partitions and schedule are determined by the schedule

Results



Operations:
Denoise and demosaic are nearest neighbor stencils,
Color correct and tone curve are element-wise operations

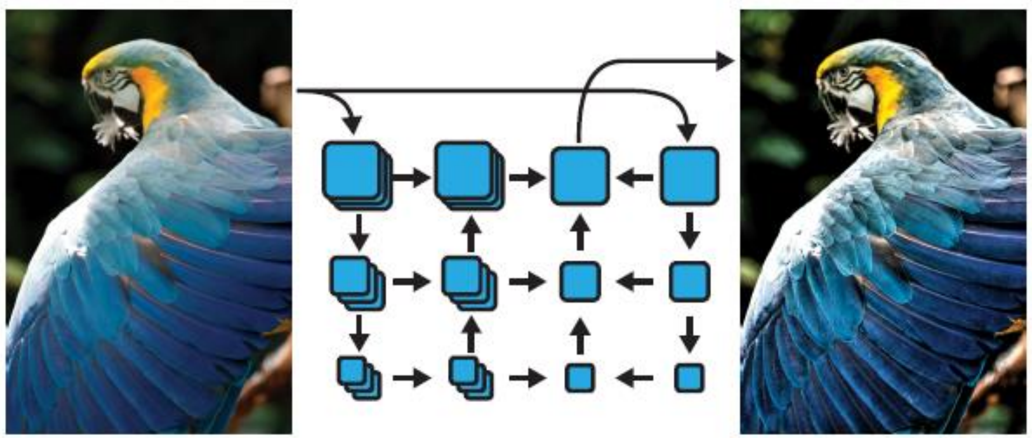
Camera Raw Pipeline

Optimized NEON ASM:	463 lines
Nokia N900:	772 ms
Halide algorithm:	145 lines
schedule:	23 lines
Nokia N900:	741 ms

2.75x shorter
5% *faster* than tuned assembly

Quad-core x86:	51 ms

Schedule:
Output is tiled,
each stage is computed in chunks within those tiles,
and then vectorized



Local Laplacian Filter

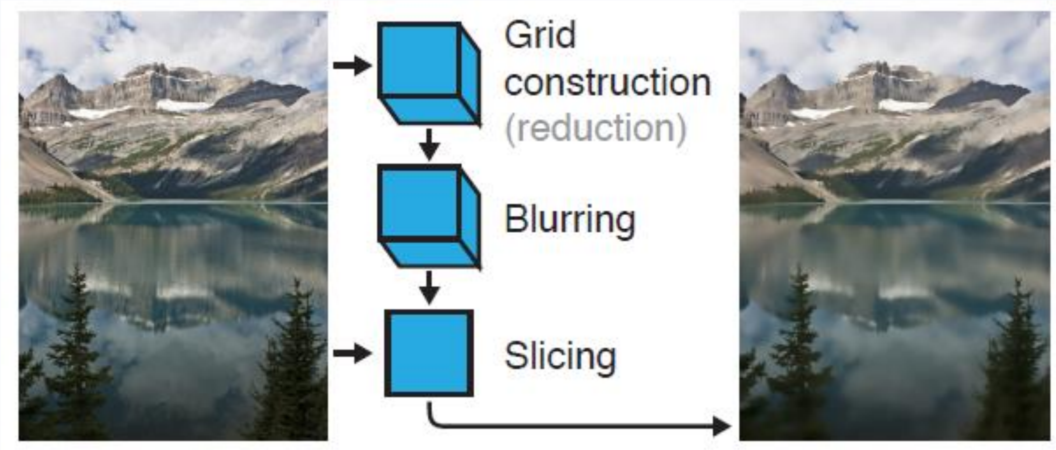
Operations:
Mixes images of different resolutions using gaussian and laplacian image pyramids

C++, OpenMP+IPP:	262 lines
Quad-core x86:	335 ms
Halide algorithm:	62 lines
schedule:	7 lines
Quad-core x86:	158 ms

3.7x shorter
2.1x faster

CUDA GPU:	48 ms (7x)

Schedule:
Inlining some stages, computing the rest as root,
With parallelization and vectorization



Bilateral Grid

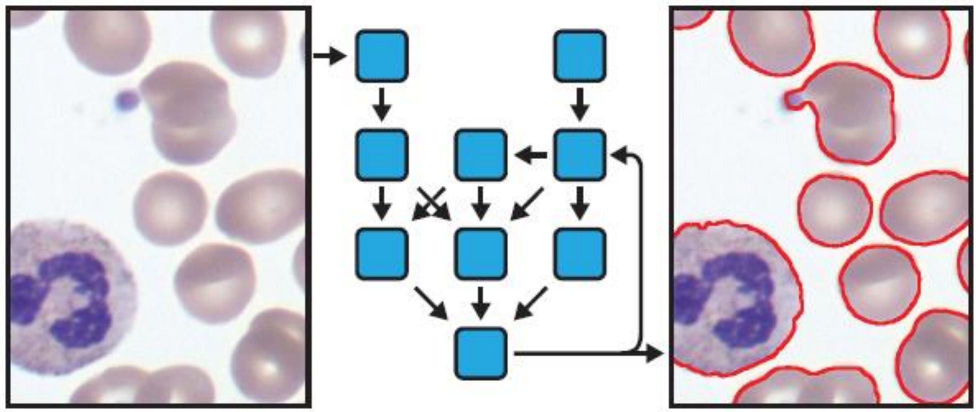
Tuned C++:	122 lines
Quad-core x86:	472ms
Halide algorithm:	34 lines
schedule:	6 lines
Quad-core x86:	80 ms

3x shorter
5.9x faster

CUDA GPU:	11 ms (42x)
Hand-written CUDA: [Chen et al. 2007]	23 ms

Operations:
Weighted histogram, blurred with a stencil, Trilinear interpolations at irregular data-driven locations

Schedule:
Parallelizing each stage



Operations:
Iterative computation composed of simple 3X1 and 1X3 filters and nonlinear point-wise operations

***Snake* Image Segmentation**

Vectorized MATLAB:	67 lines
Quad-core x86:	3800 ms
Halide algorithm:	148 lines
schedule:	7 lines
Quad-core x86:	55 ms

2.2x longer
70x faster

CUDA GPU:	3 ms (1250x)

Schedule:
Three pipelines
Two initialization ones,
One performing one iteration of the iterative process

Fully fused iteration steps

Conclusions

- Halide provides a system to specify complex code transforms in simple terms keeping the code readable and manageable
- It provides a platform for easy exploration of optimizations
- It provides a framework that is amenable to both user interaction and to automate the process of efficient code generation

What’s next?

- This paper was published in 2012
- Automatic generation of tuned Halide schedules
  - Autotuning using genetic search

    Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. PLDI '13.
  - Autotuning using opentuner

    Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. PACT ‘14.
  - Analytically

    Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*
- Halide for distributed memory systems

  Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed Halide. PPoPP ‘16.

Thank you!

*All images and code in this presentation are picked from the paper -

Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, Frédo Durand.

SIGGRAPH 2012