Halide
a language and compiler for high performance image processing

CS448h
Oct. 20, 2015
We are surrounded by computational cameras

Enormous opportunity, demands extreme optimization
parallelism & locality limit
performance and energy
We are surrounded by computational cameras

Enormous opportunity, demands extreme optimization
parallelism & locality limit
performance and energy

**Camera:** 12 Mpx
(144MB/frame as *float*)

**CPUs:** 15 GFLOP/sec

**GPU:** 115 GFLOP/sec
We are surrounded by computational cameras

Enormous opportunity, demands extreme optimization
parallelism & locality limit
performance and energy

Camera: 12 Mpx
(144MB/frame as float)

CPUs: 15 GFLOP/sec
GPU: 115 GFLOP/sec

Required arithmetic intensity > 40:1
Today’s methodology

C++ w/multithreading, SIMD
CUDA/OpenCL
OpenGL/RenderScript
Today’s methodology

C++ w/multithreading, SIMD
CUDA/OpenCL
OpenGL/RenderScript

Optimization requires manually transforming program & data structure for locality and parallelism.
Today’s methodology

C++ w/multithreading, SIMD
CUDA/OpenCL
OpenGL/RenderScript

Optimization requires manually transforming program & data structure for locality and parallelism.

libraries don’t solve this:
BLAS, IPP, MKL, OpenCV
optimized kernels compose into inefficient pipelines (no fusion)
Key challenge: reorganize computations & data

- Simpler programs
- Order of magnitude faster
- Scalable on future architectures
Simpler, Faster, Scalable

Reference: 300 lines C++
Adobe: 1500 lines
3 months of work
10x faster (vs. reference)

Halide: 60 lines
1 intern-day
20x faster (vs. reference)
2x faster (vs. Adobe)

GPU: 90x faster (vs. reference)
Simpler, Faster, Scalable

Reference: 300 lines C++

Adobe: 1500 lines
3 months of work
10x faster (vs. reference)

Halide: 60 lines
1 intern-day

20x faster (vs. reference)
2x faster (vs. Adobe)

GPU: 90x faster (vs. reference)
Imaging is everywhere
How can we scale image processing computation?
How can we scale image processing computation?

Parallelism

“Moore’s law” growth will require exponentially more parallelism.
How can we scale image processing computation?

Parallelism
“Moore’s law” growth will require exponentially more parallelism.

Locality
Data should move as little as possible.
How can we scale image processing computation?

Parallelism
“Moore’s law” growth will require exponentially more parallelism.

Locality
Data should move as little as possible.
Communication dominates computation in both energy and time

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (28 nm)</th>
<th>Cost (vs. ALU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU op</td>
<td>1 pJ</td>
<td>-</td>
</tr>
<tr>
<td>Load from SRAM</td>
<td>5 pJ</td>
<td>5x</td>
</tr>
<tr>
<td>Move 10mm on-chip</td>
<td>32 pJ</td>
<td>32x</td>
</tr>
<tr>
<td>Send off-chip</td>
<td>500 pJ</td>
<td>500x</td>
</tr>
<tr>
<td>Send to DRAM</td>
<td>1 nJ</td>
<td>1,000x</td>
</tr>
<tr>
<td>Send over LTE</td>
<td>&gt;50 μJ</td>
<td>50,000,000x</td>
</tr>
</tbody>
</table>

Data from John Brunhaver, Bill Dally, Mark Horowitz
Communication dominates computation in both energy and time

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (28 nm)</th>
<th>Cost (vs. ALU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU op</td>
<td>1 pJ</td>
<td>-</td>
</tr>
<tr>
<td>Load from SRAM</td>
<td>5 pJ</td>
<td>5x</td>
</tr>
<tr>
<td>Move 10mm on-chip</td>
<td>32 pJ</td>
<td>32x</td>
</tr>
<tr>
<td>Send off-chip</td>
<td>500 pJ</td>
<td>500x</td>
</tr>
<tr>
<td>Send to DRAM</td>
<td>1 nJ</td>
<td>1,000x</td>
</tr>
<tr>
<td>Send over LTE</td>
<td>&gt;50 µJ</td>
<td>50,000,000x</td>
</tr>
</tbody>
</table>

Data from John Brunhaver, Bill Dally, Mark Horowitz
Communication dominates computation in both energy and time.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (28 nm)</th>
<th>Cost (vs. ALU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU op</td>
<td>1 pJ</td>
<td>-</td>
</tr>
<tr>
<td>Load from SRAM</td>
<td>5 pJ</td>
<td>5x</td>
</tr>
<tr>
<td>Move 10mm on-chip</td>
<td>32 pJ</td>
<td>32x</td>
</tr>
<tr>
<td>Send off-chip</td>
<td>500 pJ</td>
<td>500x</td>
</tr>
<tr>
<td>Send to DRAM</td>
<td>1 nJ</td>
<td>1,000x</td>
</tr>
<tr>
<td>Send over LTE</td>
<td>&gt;50 µJ</td>
<td>50,000,000x</td>
</tr>
</tbody>
</table>

Data from John Brunhaver, Bill Dally, Mark Horowitz.
Communication dominates computation in both energy and time.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (28 nm)</th>
<th>Cost (vs. ALU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU op</td>
<td>1 pJ</td>
<td>-</td>
</tr>
<tr>
<td>Load from SRAM</td>
<td>5 pJ</td>
<td>5x</td>
</tr>
<tr>
<td>Move 10mm on-chip</td>
<td>32 pJ</td>
<td>32x</td>
</tr>
<tr>
<td>Send off-chip</td>
<td>500 pJ</td>
<td>500x</td>
</tr>
<tr>
<td>Send to DRAM</td>
<td>1 nJ</td>
<td>1,000x</td>
</tr>
<tr>
<td>Send over LTE</td>
<td>&gt;50 µJ</td>
<td>50,000,000x</td>
</tr>
</tbody>
</table>

data from John Brunhaver, Bill Dally, Mark Horowitz
Message #1: Performance requires complex tradeoffs
Where does performance come from?

- Redundant work
- Locality
- Parallelism

Tradeoff
Where does performance come from?

- Program
- Hardware

tradeoff

locality

parallelism

redundant work
Message #2: organization of computation is a first-class issue
Message #2: organization of computation is a first-class issue

Program:

Algorithm
Organization of computation
Hardware
Message #2: organization of computation is a first-class issue
Message #2: organization of computation is a first-class issue
Halide
a language and compiler for image processing

[SIGGRAPH 2012, PLDI 2013]
joint work with Andrew Adams, et al.

Algorithm
Organization of computation
Hardware

redundant work
locality
parallelism
tradeoff
void box_filter_3x3(const Image &in, Image &blury) {
    Image blurx(in.width(), in.height()); // allocate blurx array

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

    for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
            blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
Algorithm vs. Organization: 3x3 blur

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

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

    for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
            blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```
Algorithm vs. Organization: 3x3 blur

void box_filter_3x3(const Image &in, Image &blury) {
    Image blurx(in.width(), in.height()); // allocate blurx 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;
}
Algorithm vs. Organization: 3x3 blur

```c
void box_filter_3x3(const Image &in, Image &blury) {
    Image blurx(in.width(), in.height()); // allocate blurx 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;
}
```

*Same algorithm, different organization*
Algorithm vs. Organization: 3x3 blur

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

Same algorithm, different organization
One of them is 15x faster
Hand-optimized C++

9.9 → 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_load_epi16((__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_u128(blurxPtr+((2*256)/8));
                    b = __mm_load_s128(blurxPtr+256/8);
                    c = __mm_load_s128(blurxPtr);
                    sum = __mm_add_epi16(__mm_add_epi16(a, b), c);
                    avg = __mm_mulhi_epi16(sum, one_third);
                    __mm_store_s128(outPtr++, avg);
                }
            }
        }
    }
}
```

11x faster

(quad core x86)

Tiled, fused
Vectorized
Multithreaded
Redundant computation
Near roof-line optimum
(Re)organizing computation is hard

Optimizing parallelism, locality requires transforming program & data structure.

What transformations are *legal*?

What transformations are *beneficial*?
(Re)organizing computation is hard

Optimizing parallelism, locality requires transforming program & data structure.

What transformations are legal?

What transformations are beneficial?

libraries don’t solve this:
BLAS, IPP, MKL, OpenCV, MATLAB
optimized kernels compose into inefficient pipelines (no fusion)
Halide

a new language & compiler for image processing
Halide
a new language & compiler for image processing

1. Decouple algorithm from schedule
   Algorithm: what is computed
   Schedule: where and when it’s computed
The algorithm defines pipelines as pure functions

Pipeline stages are functions from coordinates to values

Execution order and storage are unspecified

3x3 blur as a Halide algorithm:

\[
\begin{align*}
\text{blurx}(x, y) &= (\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y))/3; \\
\text{blury}(x, y) &= (\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1))/3;
\end{align*}
\]
Halide
a new language & compiler for image processing

1. Decouple *algorithm* from *schedule*
   Algorithm: *what* is computed
   Schedule: *where* and *when* it’s computed
Halide
a new language & compiler for image processing

1. Decouple *algorithm* from *schedule*
   Algorithm: *what* is computed
   Schedule: *where and when* it’s computed

2. Single, unified model for *all* schedules
Halide
a new language & compiler for image processing

1. Decouple *algorithm* from *schedule*
   Algorithm: *what* is computed
   Schedule: *where* and *when* it’s computed

2. Single, unified model for *all* schedules
   Simple enough to search, expose to user
   Powerful enough to beat expert-tuned code
The schedule defines intra-stage order, inter-stage interleaving.
The schedule defines intra-stage order, inter-stage interleaving.

For each stage:

1) In what order should we compute its values?
The schedule defines intra-stage order, inter-stage interleaving.

For each stage:

1) In what order should we compute its values?
2) When should we compute its inputs?
The schedule defines intra-stage order, inter-stage interleaving.

For each stage:

1) In what order should we compute its values?
2) When should we compute its inputs?

This is a co-language for scheduling choices.
The Schedule defines a loop nest to compute the pipeline
The Schedule defines a loop nest to compute the pipeline

\[
\text{blurx}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3};
\]

\[
\text{blury}(x, y) = \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)}{3};
\]
**The Schedule** defines a loop nest to compute the pipeline 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;
blury.tile(x, y, xo, yo, xi, yi, 32, 32);
The Schedule defines a loop nest to compute the pipeline.

\[
\begin{align*}
\text{blurx}(x, y) &= (\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y))/3; \\
\text{blury}(x, y) &= (\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1))/3; \\
\text{blury}.\text{tile}(x, y, xo, yo, xi, yi, 32, 32);
\end{align*}
\]

// for each tile
for blury.yo:
    for blury.xo:
        // for pixel in tile
        for blury.yi:
            for blury.xi:
                compute blury
The Schedule defines a loop nest to compute the pipeline:

\[
\text{blurx}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3}; \\
\text{blury}(x, y) = \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)}{3}; \\
\]

\text{blury}.tile(x, y, xo, yo, xi, yi, 256, 32);

// for each tile
for blury.yo:
  for blury.xo:
    // for pixel in tile
    for blury.yi:
      for blury.xi:
        compute blury
The **Schedule** defines a loop nest to compute the pipeline

\[
\text{blurx}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3};
\]

\[
\text{blury}(x, y) = \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)}{3};
\]

\[\text{blury}.tile(x, y, xo, yo, xi, yi, 256, 32);\]

\[\text{blurx}.compute\_at(\text{blury}, xo);\]

// for each tile
for blury.yo:
  for blury.xo:
    // for pixel in tile
    for blury.yi:
      for blury.xi:
        compute blury
The Schedule defines a loop nest to compute the pipeline 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;
blury.tile(x, y, xo, yo, xi, yi, 256, 32);
blurx.compute_at(blury, xo);

// for each tile
for blury.yo:
  for blury.xo:
    // for pixel in tile
    for blury.yi:
      for blury.xi:
        compute blury
The Schedule defines a loop nest to compute the pipeline

\[
\begin{align*}
\text{blurx}(x, y) &= \frac{(\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y))}{3}; \\
\text{blury}(x, y) &= \frac{(\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1))}{3};
\end{align*}
\]

\[
\begin{align*}
\text{blury}.\text{tile}(x, y, xo, yo, xi, yi, 256, 32); \\
\text{blurx}.\text{compute}_\text{at}(\text{blury}, xo);
\end{align*}
\]

// for each tile
for blury.yo:
  for blury.xo:
    compute here

// for pixel in tile
for blury.yi:
  for blury.xi:
    compute blury
The **Schedule** defines a loop nest to compute the pipeline blurx(x, y) = (\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y))/3;
blury(x, y) = (\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1))/3;

blury.tile(x, y, xo, yo, xi, yi, 256, 32);
blurx.compute_at(blury, xo);

// for each tile
for blury.yo:
   for blury.xo:
      // for pixel in required tile
      for blurx.y:
         for blurx.x:
            compute bluryx
      // for pixel in tile
      for blury.yi:
         for blury.xi:
            compute blury
The Schedule defines a loop nest to compute the pipeline blur.

\[
\text{blurx}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3};
\]

\[
\text{blury}(x, y) = \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)}{3};
\]

\text{blury}.tile(x, y, xo, yo, xi, yi, 256, 32).parallel(yo);
\text{blurx}.compute_at(blury, xo).vectorize(x, 8);

// for each tile
parallel for blury.yo:
  for blury.xo:
    // for pixel in required tile
    for blury.y:
      vec for blury.x:
        compute blury<8>
    // for pixel in tile
    for blury.yi:
      for blury.xi:
        compute blury

<table>
<thead>
<tr>
<th>Prior work*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Streaming languages</strong></td>
</tr>
<tr>
<td>Ptolemy [Buck et al. 1993]</td>
</tr>
<tr>
<td>StreamIt [Thies et al. 2002]</td>
</tr>
<tr>
<td>Brook [Buck et al. 2004]</td>
</tr>
<tr>
<td><strong>Loop optimization</strong></td>
</tr>
<tr>
<td>Systolic arrays [Gross &amp; Lam 1984]</td>
</tr>
<tr>
<td><strong>Parallel work scheduling</strong></td>
</tr>
<tr>
<td>Cilk [Blumhoef et al. 1995]</td>
</tr>
<tr>
<td>NESL [Blelloch et al. 1993]</td>
</tr>
<tr>
<td><strong>Region-based languages</strong></td>
</tr>
<tr>
<td>ZPL [Chamberlain et al. 1998]</td>
</tr>
<tr>
<td>Chapel [Callahan et al. 2004]</td>
</tr>
<tr>
<td><strong>Stencil optimization &amp; DSLs</strong></td>
</tr>
<tr>
<td>[Frigo &amp; Strumpen 2005]</td>
</tr>
<tr>
<td>[Krishnamoorthy et al. 2007]</td>
</tr>
<tr>
<td>[Kamil et al. 2010]</td>
</tr>
<tr>
<td><strong>Mapping-based languages &amp; DSLs</strong></td>
</tr>
<tr>
<td>SPL/SPIRAL [Püschel et al. 2005]</td>
</tr>
<tr>
<td>Sequoia [Fatahalian et al. 2006]</td>
</tr>
<tr>
<td><strong>Shading languages</strong></td>
</tr>
<tr>
<td>RSL [Hanrahan &amp; Lawson 1990]</td>
</tr>
<tr>
<td>Cg, HLSL [Mark et al. 2003; Blythe 2006]</td>
</tr>
<tr>
<td><strong>Image processing systems</strong></td>
</tr>
<tr>
<td>[Shantzis 1994], [Levoy 1994]</td>
</tr>
<tr>
<td>PixelBender, CoreImage</td>
</tr>
</tbody>
</table>
Domain scope
of the programming model

All computation is over regular grids (up to 4D).

Only feed-forward pipelines
Recursive/reduction computations are a (partial) escape hatch.

Recursion must have bounded depth.

Long, heterogeneous pipelines.
Complex graphs, deeper than traditional stencil computations.
All computation is over regular grids (up to 4D).

Only feed-forward pipelines

Recursive/reduction computations are a (partial) escape hatch.

Recursion must have bounded depth.

Long, heterogeneous pipelines.

Complex graphs, deeper than traditional stencil computations.
Roadmap

1. Fundamental transformations for stencil pipelines
2. Halide’s unified model of scheduling
3. Results on real image processing pipelines
Roadmap

1. Fundamental transformations for stencil pipelines
2. Halide’s unified model of scheduling
3. Results on real image processing pipelines
Organizing a data-parallel pipeline

input

blurx

blury
Simple loops execute **breadth-first** across stages.
Simple loops execute **breadth-first** across stages.
Simple loops execute **breadth-first** across stages.
Simple loops execute **breadth-first** across stages.
Simple loops execute *breadth-first* across stages.
Simple loops execute **breadth-first** across stages.

```
input → blurx → blury

... [rectangles] ...  ...

... [rectangles] ...  ...

... [rectangles] ...  ...

... [rectangles] ...  ...

```

read from memory
Breadth-first execution sacrifices locality
Breadth-first execution sacrifices locality
Breadth-first execution sacrifices locality
Breadth-first execution sacrifices locality

locality is a function of reuse distance
Interleaved execution improves locality
Interleaved execution improves locality

- Input
- Blurx
- Blury

(simplified)
Interleaved execution improves locality

(simplified)
Interleaved execution improves locality

(input) → (blurx) → (blury)

(simplified)
Interleaved execution improves locality

(input) → blurx → blury

(simplified)
Interleaved execution improves locality

(input) → blurx → blury

(simplified)
Interleaved execution improves locality

(simplified)
Interleaved execution improves locality
Interleaved execution improves locality

reduce reuse distance from producer to consumer
Fusion improves locality
Fusion improves locality
Fusion improves locality

input

blurx
(fused)
blury

fusion globally interleaves computation
Understanding dependencies

input

blurx

blury

...
Understanding dependencies

input

blurx

blury
Understanding dependencies

input

blurx

blury

...
Understanding dependencies

`input` → `blurx` → `blury`
Stencils have overlapping dependencies

- **input**
- **blurx**
- **blury**
Stencils have overlapping dependencies

input

blurx

blury

...
Stencils have overlapping dependencies
Stencils have overlapping dependencies

input

blurx

blury

...
Breaking dependencies introduces redundant work
Breaking dependencies introduces redundant work
Breaking dependencies introduces redundant work
Decoupled tiles optimize parallelism & locality
Decoupled tiles optimize parallelism & locality
Decoupled tiles optimize parallelism & locality
Decoupled tiles optimize parallelism & locality

input

blurx

blury
Decoupled tiles optimize parallelism & locality
Decoupled tiles optimize parallelism & locality

- Parallelism (independence)
- Locality (short reuse distance)
Organization requires global tradeoffs

3x3 box filter
Organization requires global tradeoffs.

**Local Laplacian filters**

[Paris et al. 2010, Aubry et al. 2011]

**Diagram:**

- **LUT:** look-up table
  
  \[ O(x,y,k) \leftarrow \text{lut}(I(x,y) - k\sigma) \]

- **ADD:** addition
  
  \[ O(x,y) \leftarrow I_1(x,y) + I_2(x,y) \]

- **DDA:** data-dependent access
  
  \[ k \leftarrow \text{floor}(I_1(x,y) / \sigma) \]
  
  \[ \alpha \leftarrow (I_1(x,y) / \sigma) - k \]
  
  \[ O(x,y) \leftarrow (1-\alpha) I_2(x,y,k) + \alpha I_2(x,y,k+1) \]

- **DOWN:** downsample
  
  \[ T_1 \leftarrow I \otimes [1 \ 3 \ 3 \ 1] \]
  
  \[ T_2 \leftarrow T_1 \otimes [1 \ 3 \ 3 \ 1] \]
  
  \[ O(x,y) \leftarrow T_2(2x,2y) \]

- **UP:** upsample
  
  \[ T_1 \leftarrow I \otimes [1 \ 3 \ 3 \ 1] \]
  
  \[ T_2 \leftarrow T_1 \otimes [1 \ 3 \ 3 \ 1] \]
  
  \[ O(x,y) \leftarrow T_2(2x,2y) \]
Local Laplacian Filters
prototype for Adobe Photoshop Camera Raw / Lightroom
Local Laplacian Filters
prototype for Adobe Photoshop Camera Raw / Lightroom

Adobe: 1500 lines of expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster than original C++
Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom

Adobe: 1500 lines of expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 intern-day

Halide vs. Adobe:
2x faster on same CPU,
9x faster on GPU
Message #1: performance requires tradeoffs
Message #1: performance requires tradeoffs

input

blurx

blury

... ... ...

redundant work

locality
Message #1: performance requires **tradeoffs**

- **input**
  - **blurx**
  - **blury**

- ... ... ... ... ...

- trade off with **granularity of fusion**

- **redundant work**

- **locality**
Message #1: performance requires **tradeoffs**

- **input**
  - **blurx**
  - **blury**

- redundant work
  - trade off with granularity of fusion

- locality
  - parallelism
    - trade off by constraining order
Message #2: algorithm vs. organization
Message #2: algorithm vs. organization

order and interleaving radically alter performance of the same algorithm
Message #2: algorithm vs. organization

Order and interleaving radically alter performance of the same algorithm.
Message #2: algorithm vs. organization

Order and interleaving radically alter performance of the same algorithm.
Message #3: **dependencies** limit choices of organization
This is a general task graph.
This is a general task graph

- **input**
  - blurx
  - blury

- **task**
- **dependencies**
- **task schedule**

- **redundant work**
- **locality**
- **parallelism**
- **tradeoff**
This is a general task graph

- input
  - blurx
  - blurry
This is a general task graph
This is a general task graph.
This is a general task graph

- input
- blurx
- blury

dependencies

Task schedule

- task

- independent tasks

- redundant work

- locality

- parallelism

Tradeoff

This is a general task graph

- **input**
  - **blurx**
  - **blury**

- **task**
- **dependencies**
- **task schedule**

- **tradeoff**
  - **redundant work**
  - **locality**
  - **parallelism**
This is a general task graph

input

blurx

blury

dependencies

task schedule

task

tradeoff

locality

parallelism

redundant work
This is a general task graph

input

blurx

blury

dependencies

task schedule

redundant work

locality

parallelism

tradeoff
This is a general task graph
Traditional languages conflate algorithm & organization

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) {
    _mm128i blurx[] = _m128i{256/8 * (32+i)}; // 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+0));
          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 x = 0; x < in.width(); x++) {
      __m128i *outPtr = &blury[yTile+y][xTile]);
      for (int y = 0; y < 32+1; y++) {
        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_store_si128(outPtr++, avg);
      }
    }
  }
}

void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height()); // allocate blurx 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;
    }
  }
}
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*3]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = &blurx[((256/8)*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+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[32/*((n+1)/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;
            }
            __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) {  // allocate tile blurx array
        m128i blurx[8*32];
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = &blurx[0];
            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[0]);
                    b = __mm_loadu_si128((__m128i *)&inPtr[1]);
                    c = __mm_load_si128((__m128i *)&inPtr[2]);
                    sum = __mm_add_epi16(a, b, c);
                    avg = __mm_mulhi_epi16(sum, one_third);
                    __mm_store_si128(blurxPtr++, avg);
                    inPtr += 8;
                }
                blurxPtr = &blurx[0];
            }
            // reorganize computation: fuse two blurs, compute in tiles
        }
    }
}
The effect of organization on performance

<table>
<thead>
<tr>
<th>Organization Model</th>
<th>Performance (vs. root baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first</td>
<td>1 x</td>
</tr>
<tr>
<td>Breadth-first + parallel</td>
<td>4 x</td>
</tr>
<tr>
<td>Interleaving alone</td>
<td>0.8 x</td>
</tr>
<tr>
<td>Interleaving + parallel</td>
<td>11.5 x</td>
</tr>
</tbody>
</table>
Same algorithm, different organization

```c
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(1846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    _mm128i 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[x-1]);
          b = _mm_loadu_si128((__m128i *)&inPtr[x]);
          c = _mm_loadu_si128((__m128i *)&inPtr[x+1]);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
        }
        blurxPtr = blurx;
      }
    }
  }
}
```

```c
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height()); // allocate blurx 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;
    }
  }
```

```c
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;
  }
```
Same algorithm, different organization

void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(19846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurx = _mm_loadu_si128((__m128i*)(in+yTile+yTile));
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i blurxPtr = blury;
            for (int y = -1; y < 32; y++) {
                const uint16_t *inPtr = &in[y+yTile+yTile];
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(in+yTile+y+yTile));
                    b = _mm_loadu_si128((__m128i*)(in+y+yTile+y));
                    c = _mm_add_epi16(a, b);
                    sum = _mm_add_epi16(a, c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurxPtr++, avg);
                    inPtr += 8;
                }
            }
            blury = _mm_loadu_si128((__m128i*)(in+y+yTile+y));
            for (int x = 0; x < 256; x += 8) {
                a = _mm_add_epi16(a, b);
                sum = _mm_add_epi16(a, c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(outPtr++, avg);
            }
        }
    }
}

void box_filter_3x3(const Image &in, Image &blury) {
    Image blurx(in.width(), in.height()); // allocate blurx array
    for (int y = 0; y < in.height(); y++) {
        for (int x = 0; x < in.width(); x++) {
            blury(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
        }
    }
}

For a given algorithm, organize to optimize:
Halide’s answer: *decouple* algorithm from schedule

Algorithm: *what* is computed
Schedule: *where* and *when* it’s computed
The algorithm defines pipelines as pure functions.

Pipeline stages are functions from coordinates to values.

Execution order and storage are unspecified.

3x3 blur as a Halide algorithm:

\[
\text{blurx}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3};
\]

\[
\text{blury}(x, y) = \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)}{3};
\]
Domain scope of the programming model

All computation is over regular grids (up to 4D).

Only feed-forward pipelines
Recursive/reduction computations are a (partial) escape hatch.

Recursion must have bounded depth.

Dependence must be inferable.
User-defined clamping can impose tight bounds, when needed.

Long, heterogeneous pipelines.
Complex graphs, deeper than traditional stencil computations.
Domain scope of the programming model

All computation is over regular grids (up to 4D).

- Only feed-forward pipelines
- Recursive/reduction computations are a (partial) escape hatch.
- Recursion must have bounded depth.
- Dependence must be inferable.
  User-defined clamping can impose tight bounds, when needed.
- Long, heterogeneous pipelines.
  Complex graphs, deeper than traditional stencil computations.

not Turing complete
The schedule defines intra-stage order, inter-stage interleaving.
The schedule defines intra-stage order, inter-stage interleaving.

For each stage:

1) In what order should we compute its values?
The schedule defines intra-stage order, inter-stage interleaving.

For each stage:

1) In what order should we compute its values?

2) When should we compute its inputs?
The schedule defines order & parallelism within stages
The schedule defines order & parallelism within stages

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

Serial y, Serial x
The schedule defines order & parallelism within stages

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>7</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>

Serial y, Serial x
The schedule defines order & parallelism within stages

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Serial y, Vectorize x by 4
The schedule defines order & parallelism within stages

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Serial y,
Vectorize x by 4
The schedule defines order & parallelism within stages.

Parallel y,
Vectorize x by 4
The schedule defines order & parallelism within stages

Parallel y, Vectorize x by 4
The schedule defines order & parallelism within stages

Split x by 2,
Split y by 2.
The schedule defines order & parallelism within stages

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>6</th>
<th>9</th>
<th>10</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>17</th>
<th>18</th>
<th>21</th>
<th>22</th>
<th>25</th>
<th>26</th>
<th>29</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>20</td>
<td>23</td>
<td>24</td>
<td>27</td>
<td>28</td>
<td>31</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>33</th>
<th>34</th>
<th>37</th>
<th>38</th>
<th>41</th>
<th>42</th>
<th>45</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>36</td>
<td>39</td>
<td>40</td>
<td>43</td>
<td>44</td>
<td>47</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>49</th>
<th>50</th>
<th>53</th>
<th>54</th>
<th>57</th>
<th>58</th>
<th>61</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>52</td>
<td>55</td>
<td>56</td>
<td>59</td>
<td>60</td>
<td>63</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

Split x by 2,
Split y by 2.
Serial $y_{\text{outer}}$,
Serial $x_{\text{outer}}$,
Serial $y_{\text{inner}}$,
Serial $x_{\text{inner}}$
Domain order defines a **loop nest** for each function
Domain order defines a loop nest for each function

Serial \( y \),

Serial \( x \)

\[
\begin{align*}
\text{for } & (y : y_{\text{min}}..y_{\text{max}}) \\
\text{for } & (x : x_{\text{min}}..x_{\text{max}}) \\ 
& \text{eval[ } f(x, y) \text{ ]}
\end{align*}
\]
Domain order defines a **loop nest** for each function

Serial \( y \),
Serial \( x \)

Split \( x \) by 4,
Split \( y \) by 4.
Parallel \( y_\circ \),
Serial \( x_\circ \),
Serial \( y_i \),
Vectorize \( x_i \) by 4
Domain order defines a loop nest for each function

Serial y, Serial x

Split x by 4, Split y by 4.
Parallel y_o, Serial x_o,
Serial y_i, Vectorize x_i by 4

```plaintext
for (y : y_min..y_max)
  for (x : x_min..x_max) {
    eval[ f(x, y) ]
  }

parfor (y_o : y_o_min..y_o_max)
  for (x_o : x_o_min..x_o_max)
    for (y_i : y_i_min..y_i_max)
      simdfor (x_i : x_i_min..x_i_max by 4) {
        eval<4>[ f(x_o*4+x_i, y_o*4+y_i) ]
      }
```
Domain order defines a loop nest for each function

Serial $y$, Serial $x$

Split $x$ by 4, Split $y$ by 4.
Parallel $y_o$, Serial $x_o$, Serial $y_i$, Vectorize $x_i$ by 4

```cpp
f.split(x, xo, xi, 4)
.split(y, yo, yi, 4)
.reorder(yo, xo, yi, xi)
.parallel(yo)
.vectorize(xi, 4)
```
The schedule defines producer-consumer interleaving between blurx and blury.
The schedule defines producer-consumer interleaving.
Tradeoff space modeled by granularity of interleaving
Tradeoff space modeled by granularity of interleaving
Tradeoff space modeled by granularity of interleaving

compute granularity

valid schedules
Tradeoff space modeled by granularity of interleaving

- Compute granularity
- Fine interleaving, high locality

Valid schedules
Tradeoff space modeled by granularity of interleaving

coarse interleaving low locality

compute granularity

fine interleaving high locality

valid schedules
Tradeoff space modeled by granularity of interleaving

- coarse interleaving
  - low locality
- fine interleaving
  - high locality

Valid schedules

Compute granularity vs. storage granularity
Tradeoff space modeled by granularity of interleaving.
Tradeoff space modeled by granularity of interleaving

- coarse interleaving
  - low locality
- fine interleaving
  - high locality
- redundant computation
  - no redundant computation

Valid schedules

Compute granularity

Storage granularity
Tradeoff space modeled by granularity of interleaving

- **Compute granularity**
  - coarse interleaving: low locality
  - fine interleaving: high locality

- **Storage granularity**
  - redundant computation
  - no redundant computation

```python
blur_x.compute_at(root)
.store_at(root)
```
Tradeoff space modeled by granularity of interleaving

- **Coarse interleaving**
  - Low locality
  - Redundant computation

- **Fine interleaving**
  - High locality
  - No redundant computation

**Compute granularity**

**Storage granularity**

**Breadth-first execution**

```
blur_x.compute_at(root) .store_at(root)
```
Tradeoff space modeled by granularity of interleaving

- coarse interleaving, low locality
  - compute granularity
  - fine interleaving, high locality
  - redundant computation
  - no redundant computation

- total fusion

```python
blur_x.compute_at(blury, x)
.store_at(blury, x)
```
Tradeoff space modeled by granularity of interleaving

- coarse interleaving: low locality
- fine interleaving: high locality

**compute granularity**

- fine interleaving: high locality
- coarse interleaving: low locality

**storage granularity**

- redundant computation
- no redundant computation

**total fusion**

- redundant work

Code snippet:
```
blur_x.compute_at(blury, x) .store_at(blury, x)
```
Tradeoff space modeled by granularity of interleaving

- coarse interleaving, low locality
- fine interleaving, high locality

compute granularity

storage granularity

- redundant computation
- no redundant computation

sliding window fusion

blur_x.compute_at(blury, x).store_at(root)
Tradeoff space modeled by granularity of interleaving

- coarse interleaving, low locality
- fine interleaving, high locality

Compute granularity:
- redundant computation
- no redundant computation

Storage granularity:
- sliding window fusion

blur_x.compute_at(blury, x).store_at(root)
Tradeoff space modeled by granularity of interleaving

- coarse interleaving: low locality
- fine interleaving: high locality

- compute granularity
- storage granularity

- redundant computation: no redundant computation

- capturing reuse
- constrains order
- less parallelism

- parallelism

- sliding window fusion

- blur_x.compute_at(blury, x).store_at(root)
Tradeoff space modeled by granularity of interleaving

- coarse interleaving, low locality
- fine interleaving, high locality

- redundant computation
- no redundant computation

compute granularity

storage granularity
Tradeoff space modeled by granularity of interleaving

- coarse interleaving: low locality
- fine interleaving: high locality
- redundant computation
- no redundant computation

---

```
blur_y.tile(xo, yo, xi, yi, W, H)
blur_x.compute_at(blury, xo).store_at(blury, xo)
```
Tradeoff space modeled by granularity of interleaving

- coarse interleaving: low locality, redundant computation
- fine interleaving: high locality, no redundant computation

Compute granularity vs. Storage granularity
Tradeoff space modeled by granularity of interleaving

- coarse interleaving: low locality
- fine interleaving: high locality

- redundant computation
- no redundant computation

compute granularity

storage granularity

enlarged sliding window
fine-grained data-parallelism within window
Tradeoff space modeled by granularity of interleaving

- coarse interleaving: low locality
- fine interleaving: high locality

**Compute granularity**
- redundant computation
- no redundant computation

**Storage granularity**
- enlarged sliding window
- fine-grained data-parallelism within window

**Parallelism**
- coarse-grained
- fine-grained

- parallel sliding windows
- coarse-parallelism across windows
- within window
- parallelism across windows
Tradeoff space modeled by granularity of interleaving

- **Compute granularity**
  - coarse interleaving: low locality
  - fine interleaving: high locality

- **Storage granularity**
  - redundant computation
  - no redundant computation

- **Parallel sliding windows**
  - coarse-grained parallelism across windows

- **Enlarged sliding window**
  - fine-grained data-parallelism within window

- **Parallel enlarged sliding windows**
Schedule primitives **compose** to create many organizations.
Schedule primitives **compose** to create many organizations.
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_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);
                }
            }
        }
    }
}

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

    return blury;
}
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).store_at(blury, x).vectorize(x, 8);

    return blury;
}
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).store_at(blury, 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; 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;
                }
            }
        }
        for (int y = 0; y < 32; y++) {
            _mm_store_si128(outPtr++, avg);
        }
    }
    return blury;
}

C++

0.9 ms/megapixel

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_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;
                }
            }
        }
        for (int y = 0; y < 32; y++) {
            _mm_store_si128(outPtr++, avg);
        }
    }
}
More language features beyond the scope of this talk

Computed, data-dependent reads (gather)
\[ f(x) = g(\text{floor}(2.3 \times \text{in}(x))) \]

Computed, data-dependent \textit{writes} (scatter)
\[ f(g(\text{floor}(2.3 \times \text{in}(x)))) = \text{in}(x) \]

Recursive functions (IIR convolution, scan)
\[ \text{cdf}(i) = \text{cdf}(i-1) + \text{pdf}(i) \]
More language features beyond the scope of this talk

Computed, data-dependent reads (gather)
\[ f(x) = g(\text{floor}(2.3 \times \text{in}(x))) \]

Computed, data-dependent writes (scatter)
\[ f(g(\text{floor}(2.3 \times \text{in}(x)))) = \text{in}(x) \]

Recursive functions (IIR convolution, scan)
\[ \text{cdf}(i) = \text{cdf}(i-1) + \text{pdf}(i) \]
The Halide Compiler

- Halide Functions
- Halide Schedule
- Synthesized loop nest, allocations
- Vectorization & peephole optimization
- LLVM bitcode
- x86 (with SSE/AVX)
- ARM (with NEON)
- CUDA (host+kernel graph)
The Halide Compiler

- Halide Functions
- Halide Schedule

- Synthesized loop nest, allocations

- Vectorization & peephole optimization

- LLVM bitcode

- x86 (with SSE/AVX)
- ARM (with NEON)
- CUDA (host+kernel graph)
The Halide Compiler

- **Regis**
  - Halide Functions

- **Kelly**
  - Halide Schedule

---

- **Synthesized loop nest, allocations**

---

- **Vectorization & peephole optimization**

---

- **LLVM bitcode**
  - x86 (with SSE/AVX)
  - ARM (with NEON)
  - CUDA (host+kernel graph)
The Halide Compiler

Halide Functions → Halide Schedule

Synthesized loop nest, allocations →
Vectorization & peephole optimization →
LLVM bitcode

x86 (with SSE/AVX) → ARM (with NEON) → CUDA (host+kernel graph)
The Halide Compiler

- **Halide Functions**
- **Halide Schedule**
- **Synthesized loop nest, allocations**
- **Vectorization & peephole optimization**
- **LLVM bitcode**
- **x86 (with SSE/AVX)**
- **ARM (with NEON)**
- **CUDA (host+kernel graph)**

**Regis**

**autotuner**

*benchmark and iterate*
Local Laplacian Filters
[Paris et al. 2010, Aubry et al. 2011]

Adobe: 1500 lines of expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 intern-day

Halide vs. Adobe:
2x faster on same CPU,
9x faster on GPU
Local Laplacian Filters
[Paris et al. 2010, Aubry et al. 2011]

Adobe: 1500 lines of expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 intern-day

Halide vs. Adobe:
2x faster on same CPU,
9x faster on GPU
Local Laplacian Filters

[Paris et al. 2010, Aubry et al. 2011]
“Snake” Image Segmentation

[Li et al. 2010]

Segments objects in an image using level-sets

Original: 67 lines of matlab
“Snake” Image Segmentation  
[Li et al. 2010]

Segments objects in an image using level-sets

Original: 67 lines of matlab

Halide: 148 lines of algorithm, 7 lines of schedule
“Snake” Image Segmentation
[Li et al. 2010]

Segments objects in an image using level-sets

Original: 67 lines of matlab

Halide: 148 lines of algorithm, 7 lines of schedule

On the CPU, 70x faster
MATLAB is memory-bandwidth limited
“Snake” Image Segmentation

[Li et al. 2010]

Segments objects in an image using level-sets

Original: 67 lines of matlab

Halide: 148 lines of algorithm, 7 lines of schedule

On the CPU, 70x faster

MATLAB is memory-bandwidth limited
“Snake” Image Segmentation

[Li et al. 2010]

Segments objects in an image using level-sets

Original: 67 lines of matlab

Halide: 148 lines of algorithm, 7 lines of schedule

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

On the GPU, 1250x faster
The Bilateral Grid
[Chen et al. 2007]

An accelerated bilateral filter

Original: 122 lines of (clean) C++
The Bilateral Grid
[Chen et al. 2007]

An accelerated bilateral filter

Original: 122 lines of (clean) C++

Halide: 34 lines of algorithm, 6 lines of schedule
An accelerated bilateral filter

Original: 122 lines of (clean) C++

Halide: 34 lines of algorithm, 6 lines of schedule

On the CPU, 5.9x faster

The Bilateral Grid

[Chen et al. 2007]
An accelerated bilateral filter

Original: 122 lines of (clean) C++

Halide: 34 lines of algorithm, 6 lines of schedule

On the CPU, 5.9x faster

On the GPU, 2x faster than Chen’s hand-written CUDA version (and equivalent Halide schedule)
The Bilateral Grid

[Chen et al. 2007]

An accelerated bilateral filter

Original: 122 lines of (clean) C++

Halide: 34 lines of algorithm, 6 lines of schedule

On the CPU, 5.9x faster

On the GPU, 2x faster than Chen’s hand-written CUDA version *and equivalent Halide schedule*
The Frankencamera Raw Pipeline
[Adams et al. 2010]

Converts raw image sensor data into an image

Original: 463 lines of ARM assembly and intrinsics in one big function
The Frankencamera Raw Pipeline
[Adams et al. 2010]

Converts raw image sensor data into an image

Original: 463 lines of ARM assembly and intrinsics in one big function

Rewritten in Halide, it is 2.75x less code, and runs 5% faster
The Frankencamera Raw Pipeline
[Adams et al. 2010]

Converts raw image sensor data into an image

Original: 463 lines of ARM assembly and intrinsics in one big function

Rewritten in Halide, it is 2.75x less code, and runs 5% faster
<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
<th>Factor shorter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x86</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blur</td>
<td>1.2 x</td>
<td>18 x</td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>4.4 x</td>
<td>4 x</td>
</tr>
<tr>
<td>Camera pipeline</td>
<td>3.4 x</td>
<td>2 x</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>1.7 x</td>
<td>7 x</td>
</tr>
<tr>
<td>Local Laplacian</td>
<td>1.7 x</td>
<td>5 x</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>2.3 x</td>
<td>11 x</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>5.9* x</td>
<td>7* x</td>
</tr>
<tr>
<td>Local Laplacian</td>
<td>9* x</td>
<td>7* x</td>
</tr>
<tr>
<td><strong>ARM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camera pipeline</td>
<td>1.1 x</td>
<td>3 x</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>Factor shorter</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>----------------</td>
</tr>
<tr>
<td><strong>x86</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blur</td>
<td>1.2 x</td>
<td>18 x</td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>4.4 x</td>
<td>4 x</td>
</tr>
<tr>
<td>Camera pipeline</td>
<td>3.4 x</td>
<td>2 x</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>1.7 x</td>
<td>7 x</td>
</tr>
<tr>
<td>Local Laplacian</td>
<td>1.7 x</td>
<td>5 x</td>
</tr>
<tr>
<td>Gaussian Blur</td>
<td>1.5 x</td>
<td>5 x</td>
</tr>
<tr>
<td>FFT (vs. FFTW)</td>
<td>1.5 x</td>
<td>10s</td>
</tr>
<tr>
<td>BLAS (vs. Eigen)</td>
<td>1 x</td>
<td>100s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>GPU</strong></th>
<th>Speedup</th>
<th>Factor shorter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bilateral Grid</td>
<td>2.3 x</td>
<td>11 x</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>5.9* x</td>
<td>7* x</td>
</tr>
<tr>
<td>Local Laplacian</td>
<td>9* x</td>
<td>7* x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ARM</strong></th>
<th>Speedup</th>
<th>Factor shorter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera pipeline</td>
<td>1.1 x</td>
<td>3 x</td>
</tr>
</tbody>
</table>
Summary
Summary

Decouples algorithm from organization through a scheduling co-language to navigate fundamental tradeoffs.
Summary

Decouples algorithm from organization through a scheduling co-language to navigate fundamental tradeoffs.

Schedule:
Order within stage
Interleaving between stages

→ Loop nest & storage allocations
Summary

Decouples algorithm from organization through a scheduling co-language to navigate fundamental tradeoffs.

Schedule:
Order *within* stage
Interleaving *between* stages

Simpler programs

Faster than hand-tuned code

Scalable across architectures
Summary

Decouples algorithm from organization through a scheduling co-language to navigate fundamental tradeoffs.

Schedule:
- Order within stage
- Interleaving between stages

Simpler programs
Faster than hand-tuned code
Scalable across architectures

open source at http://halide-lang.org
Discussion

Why *reductions*, rather than loops?

What is *chunking*?

Why the four caller-callee relationships? what do these *not* represent?
Discussion

How did we choose our example apps?

Why are schedules user-controlled rather than compiler-controlled?

Are there other Halide-like systems, in other domains?
DARKROOM Compiling High-Level Image Processing Code into Hardware Pipelines

James Hegarty
Jonathan Ragan-Kelley
Artem Vasilyev

John Brunhaver
Noy Cohen
Mark Horowitz

Zachary DeVito
Steven Bell
Pat Hanrahan
Programmable processors are too inefficient for real-time image processing!

- x86
- Camera ISP
- Tesla

500x Improvement from Specialized Datapaths

Hours of battery life (iPhone battery, 1 Terop/sec)
convolve = \( \text{im}(x, y) \ (1*I(x-1, y) + 2*I(x, y) + 1*I(x +1, y))/4 \) end

blur.t
Stencil Engine Generator

1. Parameterized Linebuffer
Stencil Engine Generator

1. Parameterized Linebuffer
2. Stencil Shift Register
Stencil Engine Generator

1. Parameterized Linebuffer
2. Stencil Shift Register
3. Synopsys Designware
Stencil Engine Generator

1. Parameterized Linebuffer
2. Stencil Shift Register
3. Synopsys Designware
4. Pipeline Generator
FPGA

Xilinx Zynq XC7Z045
Darkroom pipelines deliver real-time performance

- CAMERA ISP
- EDGE DETECTION
- CORNER DETECTION
- DEBLUR

FPS (FPGA, 1080p)
Darkroom pipelines are energy efficient

- Camera ISP: 3.75
- Edge Detection: 5
- Corner Detection: 5

Est. Hours of battery life (including memory, iPhone battery)
Darkroom pipelines are energy efficient

Est. Hours of battery life (including memory, iPhone battery)

Aptina ISP
Darkroom pipelines are energy efficient

CAMERA ISP
EDGE DETECTION
CORNER DETECTION
DEBLUR

Est. Hours of battery life (including memory, iPhone battery)
Darkroom pipelines are energy efficient
Helium: decompiling x86 binaries

```cpp
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main()
{
    Var x_0;
    Var x_1;
    ImageParam input_1(UInt(8),2);
    Func output_1;
    output_1(x_0,x_1) =
        cast<uint8_t>(((((
            2
        +
            (2*cast<uint32_t>(input_1(x_0+1,x_1+1)))
        +
            cast<uint32_t>(input_1(x_0, x_1+1))
        +
            cast<uint32_t>(input_1(x_0+2,x_1+1))
        )
            >> cast<uint32_t>(2))) & 255));

    vector<Argument> args;
    args.push_back(input_1);
    output_1.compile_to_file("halide_out_0",args);
    return 0;
}
```
```cpp
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main()
{
    Var x_0;
    Var x_1;
    ImageParam input_1(UInt(8),2);
    Func output_1;
    output_1(x_0,x_1) =
        cast<uint8_t>((((
            2 +
            (2*cast<uint32_t>(input_1(x_0+1,x_1+1))) +
            cast<uint32_t>(input_1(x_0, x_1+1)) +
            cast<uint32_t>(input_1(x_0+2,x_1+1)))
        >> cast<uint32_t>(2))) & 255));
    vector<Argument> args;
    args.push_back(input_1);
    output_1.compile_to_file("halide_out_0",args);
    return 0;
}
```

Generated Halide DSL code
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main()
{
  Var x_0;
  Var x_1;
  ImageParam input_1(UInt(8),2);
  Func output_1;
  output_1(x_0,x_1) = cast<uint8_t>((((
      2 +
      (2*cast<uint32_t>(input_1(x_0+1,x_1+1))) +
      cast<uint32_t>(input_1(x_0, x_1+1)) +
      cast<uint32_t>(input_1(x_0+2,x_1+1))))
    >> cast<uint32_t>(2))) & 255));
  vector<Argument> args;
  args.push_back(input_1);
  output_1.compile_to_file("halide_out_0",args);
  return 0;
}
Performance Results

Lifted Halide Source → Autotune schedule
Performance Results

- Lifted Halide Source
- Autotune schedule

<table>
<thead>
<tr>
<th>Task</th>
<th>Relative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invert</td>
<td>1.4</td>
</tr>
<tr>
<td>Blur</td>
<td>2.8</td>
</tr>
<tr>
<td>Blur More</td>
<td>1.4</td>
</tr>
<tr>
<td>Sharpen</td>
<td>1.4</td>
</tr>
<tr>
<td>Sharpen More</td>
<td>2.1</td>
</tr>
<tr>
<td>Threshold</td>
<td>1.4</td>
</tr>
<tr>
<td>Box Blur</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Photoshop

96
Next time: final projects

*Come to class with a potential project idea*

1. Build a new DSL
2. Extend an existing DSL
3. Build or extend DSL-creation infrastructure
<table>
<thead>
<tr>
<th></th>
<th>x86</th>
<th></th>
<th>GPU</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Halide autotuned</td>
<td>Expert tuned</td>
<td>Speedup</td>
<td>Halide source</td>
</tr>
<tr>
<td>Blur</td>
<td>11 ms</td>
<td>13 ms</td>
<td>1.2 ×</td>
<td>2 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>35 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18 ×</td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>36 ms</td>
<td>158 ms</td>
<td>4.4 ×</td>
<td>34 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>122 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 ×</td>
</tr>
<tr>
<td>Camera pipeline</td>
<td>14 ms</td>
<td>39 ms</td>
<td>3.4 ×</td>
<td>123 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>306 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 ×</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>32 ms</td>
<td>54 ms</td>
<td>1.7 ×</td>
<td>21 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>152 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 ×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Halide autotuned</td>
<td>Expert tuned</td>
<td>Speedup</td>
<td>Halide source</td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>8.1 ms</td>
<td>18 ms</td>
<td>2.3 ×</td>
<td>34 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>370 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 ×</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>9.1 ms</td>
<td>54* ms</td>
<td>5.9* ×</td>
<td>21 lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>152* lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7* ×</td>
</tr>
<tr>
<td></td>
<td>Halide autotuned</td>
<td>Expert tuned</td>
<td>Speedup</td>
<td>Halide source</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------</td>
<td>--------------</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td><strong>x86</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blur</td>
<td>11 ms</td>
<td>13 ms</td>
<td>1.2 x</td>
<td>2 lines</td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>36 ms</td>
<td>158 ms</td>
<td>4.4 x</td>
<td>34 lines</td>
</tr>
<tr>
<td>Camera pipeline</td>
<td>14 ms</td>
<td>39 ms</td>
<td>3.4 x</td>
<td>123 lines</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>32 ms</td>
<td>54 ms</td>
<td>1.7 x</td>
<td>21 lines</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>8.1 ms</td>
<td>18 ms</td>
<td>2.3 x</td>
<td>34 lines</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>9.1 ms</td>
<td>54* ms</td>
<td>5.9* x</td>
<td>21 lines</td>
</tr>
<tr>
<td></td>
<td>Halide autotuned</td>
<td>Expert tuned</td>
<td>Speedup</td>
<td>Halide source</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------</td>
<td>--------------</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>Blur</td>
<td>11 ms</td>
<td>13 ms</td>
<td>1.2 x</td>
<td>2 lines</td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>36 ms</td>
<td>158 ms</td>
<td>4.4 x</td>
<td>34 lines</td>
</tr>
<tr>
<td>Camera pipeline</td>
<td>14 ms</td>
<td>39 ms</td>
<td>3.4 x</td>
<td>123 lines</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>32 ms</td>
<td>54 ms</td>
<td>1.7 x</td>
<td>21 lines</td>
</tr>
<tr>
<td></td>
<td><strong>x86</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Halide autotuned</th>
<th>Expert tuned</th>
<th>Speedup</th>
<th>Halide source</th>
<th>Expert source</th>
<th>Factor shorter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bilateral Grid</td>
<td>8.1 ms</td>
<td>18 ms</td>
<td>2.3 x</td>
<td>34 lines</td>
<td>370 lines</td>
<td>11 x</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>9.1 ms</td>
<td>54* ms</td>
<td>5.9* x</td>
<td>21 lines</td>
<td>152* lines</td>
<td>7* x</td>
</tr>
</tbody>
</table>

**Autotuning time:** 2 hrs to 2 days (single node) 85% within < 24 hrs
High-efficiency image processing

Automatic inference of good schedules beyond brute force autotuning

Synthesis of specialized hardware

“OpenGL” and “programmable GPU” for image processing pipelines

current ISP pipelines are hard-wired, inflexible
High-performance computer vision
programming model, language, compiler support

Detection & retrieval
e.g. HOG + LDA, Viola-Jones

Markov Random Fields

Photosynth
Automatic panoramas as a starting proxy
Computational fabrication software systems

Scalable synthesis pipeline for huge data size architecture
programming model
compiler

New representations for authoring, output

Authoring tools
beyond CAD for classical mass production
Compilers & DSLs

Optimization as data-driven search
Composition of DSLs
Domain-specific programming tools
Richer authoring, debugging, and tuning
Ongoing work

Synthesizing hardware imaging pipelines from a Halide-like description
with James Hegarty, John Brunhaver, Pat Hanrahan, Mark Horowitz

Halide schedule visualization, debugging
with Jovana Knezevic

Producer-consumer parallelism, schedule-controlled memory layouts
with Nick Chornay
Ongoing work

Static schedule inference based on the task dependence graph

Irregular data in Halide

Level-of-detail for imaging pipelines

Scaling the OpenFab 3D printing pipeline
Stable Fluids [Stam 1999]

~200 stages

Complex dependence

Iterative linear solvers

Multi-phase computation
Visual sensor networks