Halide: a language and compiler for high-performance image processing
Image processing is everywhere
Internet-scale computer vision: orders of magnitude from “good enough”

YouTube: 400 hrs uploaded / min
[Brewer et al. 2016]
25K constant HD streams of the world
1.5 terapixels / sec

One object recognition CNN
250W GPU → 0.1 megapixels at video rates
Internet-scale computer vision: orders of magnitude from “good enough”

YouTube: 400 hrs uploaded / min
[Brewer et al. 2016]
25K constant HD streams of the world
1.5 terapixels / sec

250M security cameras installed / year (2014)
2.5B smartphones worldwide
Pervasive sensing: orders of magnitude from “good enough”

Sensor + Read out

5 Mpixels

~1 mJ/frame

Eulerian Video Magnification [Wu et al. 2012]
Pervasive sensing: orders of magnitude from “good enough”

Sensor + Read out
5 Mpxles
~1 mJ/frame

Eulerian Video Magnification [Wu et al. 2012]

LTE radio
50 Mbit/seg
1 W
~1 J/frame

transmission power costs 1,000x capture
Your data-intensive problem here...
What drives performance?

**Parallelism** (SIMD, multi-core, GPU)
“Moore’s law” scaling will require exponentially more parallelism.

**Locality** (caches, memory, networks)
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
Message #1: Performance requires complex tradeoffs.
Where does performance come from?

- Program
- Hardware
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

Diagram:
- Tradeoff
- Serial dependence (parallelism)
- Communication (locality)
- Amount of work
Message #2: organization of computation is a first-class issue

Program:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Organization of computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td></td>
</tr>
</tbody>
</table>

- Amount of work
- Communication (locality)
- Serial dependence (parallelism)

Tradeoff
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: 100x faster (vs. reference)
Organization is a problem across domains

Simulation, ML, vision: multiple representations
e.g., organize across sparse linear algebra & graph operations

Graphics: pipeline of computation from geometry to pixels

ASICs & FPGAs: co-optimize hardware & organization especially memory hierarchy & communication

Mobile + Cloud: organize across network
Halide
a language and compiler for image processing


Algorithm
Organization of computation
Hardware

redundant work
tradeoff
locality
parallelism

Algorithm
Organization of computation
Hardware
Algorithm vs. Organization: 3x3 blur

for all pixel coordinates \(x, y\) in input:
\[
\text{blurH}(x, y) = \frac{\text{input}(x-1, y) + \text{input}(x, y) + \text{input}(x+1, y)}{3};
\]

for all pixel coordinates \(x, y\) in input:
\[
\text{blurV}(x, y) = \frac{\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1)}{3};
\]
Algorithm vs. Organization: 3x3 blur

**for all** pixel coordinates $x,y$ in input:

$\text{blurH}(x, y) = (\text{input}(x-1, y) + \text{input}(x, y) + \text{input}(x+1, y))/3;$

**for all** pixel coordinates $x,y$ in input:

$\text{blurV}(x, y) = (\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1))/3;$
Algorithm vs. Organization: 3x3 blur

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

for (int x = 0; x < input.width(); x++)
    for (int y = 0; y < input.height(); y++)
        blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
Algorithm vs. Organization: 3x3 blur

```
for (int y = 0; y < input.height(); y++)
    for (int x = 0; x < input.width(); x++)
        blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int y = 0; y < input.height(); y++)
    for (int x = 0; x < input.width(); x++)
        blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```
Algorithm vs. Organization: 3x3 blur

\[
\begin{align*}
\text{blurH}(x, y) &= \left( \text{input}(x-1, y) + \text{input}(x, y) + \text{input}(x+1, y) \right) / 3; \\
\text{blurV}(x, y) &= \left( \text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1) \right) / 3;
\end{align*}
\]

Same algorithm, different organization

One of them is 15x faster
Algorithm vs. Organization: 3x3 blur

```cpp
for (int y = 0; y < input.height(); y++)
    for (int x = 0; x < input.width(); x++)
        blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int y = 0; y < input.height(); y++)
    for (int x = 0; x < input.width(); x++)
        blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*((32+2))]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      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(blurHPtr++, avg);
          inPtr += 8;
        }
        blurHPtr = blurH;
      }
      __m128i *outPtr = (__m128i *)(&blurV[yTile+y][xTile]));
      for (int x = 0; x < 256; x += 8) {
        a = _mm_load_si128(blurHPtr+((2*256)/8));
        b = _mm_load_si128(blurHPtr+256/8);
        c = _mm_load_si128(blurHPtr++);
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_si128(outPtr++, avg);
      }
    }
  }
}
(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’s answer: *decouple* algorithm from schedule

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

Easy for programmers to build pipelines

Easy to specify & explore optimizations
for a human or the compiler

Easy for the compiler to generate fast code
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:

\[
\text{blurH}(x, y) = (\text{input}(x-1, y) + \text{input}(x, y) + \text{input}(x+1, y))/3; \\
\text{blurV}(x, y) = (\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1))/3;
\]
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 a **loop nest** to compute the pipeline.
The Schedule defines a loop nest to compute the pipeline

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

\[
\text{blurV}(x, y) = \frac{\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1)}{3};
\]
The Schedule defines a loop nest to compute the pipeline blurV.

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

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

\[
\text{blurV.tile}(x, y, xo, yo, xi, yi, 32, 32);
\]
The Schedule defines a loop nest to compute the pipeline

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

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

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

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

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

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

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

// for each tile
for blurV.yo:
  for blurV.xo:
    // for pixel in tile
    for blurV.yi:
      for blurV.xi:
        compute blurV
The Schedule defines a loop nest to compute the pipeline blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

blurV.tile(x, y, xo, yo, xi, yi, 256, 32);
blurH.compute_at(blurV, xo);

// for each tile
for blurV.yo:
    for blurV.xo:
        // for pixel in tile
        for blurV.yi:
            for blurV.xi:
                compute blurV
The **Schedule** defines a loop nest to compute the pipeline blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3; blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

blurV.tile(x, y, xo, yo, xi, yi, 256, 32);
blurH.compute_at(blurV, xo);

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

```
blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

blurV.tile(x, y, xo, yo, xi, yi, 256, 32);
blurH.compute_at(blurV, xo);

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

// for pixel in tile
for blurV.yi:
    for blurV.xi:
        compute blurV
The **Schedule** defines a **loop nest** to compute the pipeline blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

blurV.tile(x, y, xo, yo, xi, yi, 256, 32);
blurH.compute_at(blurV, xo);

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

\[
\begin{align*}
\text{blurH}(x, y) &= (\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y))/3; \\
\text{blurV}(x, y) &= (\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1))/3;
\end{align*}
\]

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

// for each tile
parallel for \text{blurV}.yo:
  for \text{blurV}.xo:
    // for pixel in required tile
    for \text{blurH}.y:
      vec for \text{blurH}.x:
        compute \text{blurH}<8>
    // for pixel in tile
    for \text{blurV}.yi:
      for \text{blurV}.xi:
        compute \text{blurV}
Related work*

**Streaming languages**
- Ptolemy [Buck et al. 1993]
- StreamIt [Thies et al. 2002]
- Brook [Buck et al. 2004]

**Loop optimization**
- Systolic arrays [Gross & Lam 1984]
- Polyhedral model [Ancourt & Irigoin 1991]
- Affine partitioning [Lim & Lam 1999]

**Parallel work scheduling**
- Cilk [Blumhofe et al. 1995]
- NESL [Blelloch et al. 1993]

**Region-based languages**
- ZPL [Chamberlain et al. 1998]
- Chapel [Callahan et al. 2004]

**Stencil optimization & DSLs**
- [Frigo & Strumpen 2005]
- [Krishnamoorthy et al. 2007]
- [Kamil et al. 2010]

**Mapping-based languages & DSLs**
- SPL/SPIRAL [Püschel et al. 2005]
- Sequoia [Fatahalian et al. 2006]

**Shading languages**
- RSL [Hanrahan & Lawson 1990]
- Cg, HLSL [Mark et al. 2003; Blythe 2006]

**Image processing systems**
- [Holzma 1988], [Shantzis 1994], [Eliot 2001]
- PixelBender, CoreImage

*a tiny sample.
Thousands have come before us.*
Domain scope of the programming model

All computation is over regular grids.

Only feed-forward pipelines
Iterative computations are a (partial) escape hatch.
Iteration 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
Roadmap

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

1. Fundamental transformations for stencil pipelines

2. Halide’s unified model of scheduling

3. Results on real image processing pipelines

4. Finding good schedules automatically
Organizing a data-parallel pipeline

- input
- blurH
- blurV

... [visual representation of data flow]...
Simple loops execute **breadth-first** across stages.
Simple loops execute **breadth-first** across stages.
Simple loops execute **breadth-first** across stages.
Breadth-first execution sacrifices locality

parallelism

locality

input

blurH

blurV
Breadth-first execution sacrifices locality

input

blurH

blurV

write to memory

locality

parallelism
Breadth-first execution sacrifices locality.
Breadth-first execution sacrifices locality

**locality** is a function of **reuse distance**

**parallelism**
Breadth-first execution sacrifices locality

locality is a function of reuse distance
Interleaved execution (fusion) improves locality
Interleaved execution (fusion) improves locality

input

blurH

blurV

reduce reuse distance from producer to consumer

locality

parallelism
Interleaved execution (fusion) improves locality

fusion globally interleaves computation

input
- blurH
- blurV

reduce reuse distance from producer to consumer

locality
parallelism
Understanding dependencies

input

blurH

blurV
Understanding dependencies

- input
- blurH
- blurV

Diagram:

- Input connected to blurH and blurV
- blurV has a connection to a series of elements
Understanding dependencies

input

→

blurH

→

blurV

...
Stencils have overlapping dependencies
Stencils have overlapping dependencies

input

blurH

blurV
Stencils have overlapping dependencies
Sliding window execution sacrifices parallelism

Input

BlurH

BlurV

Locality

Parallelism
Sliding window execution sacrifices parallelism.

```
  input
    ↓
  blurH
    ↓
  blurV
```

Locality vs. parallelism.
Sliding window execution sacrifices parallelism
Sliding window execution sacrifices parallelism.
Sliding window execution sacrifices parallelism

short reuse distance

locality

parallelism

input

blurH

blurV
Sliding window execution sacrifices parallelism

fixed order, constrains parallelism

short reuse distance

input

blurH

blurV

locality

parallelism
Breaking dependencies with tiling

Input

blurH

blurV

locality

parallelism
Breaking dependencies with tiling

input

blurH

blurV

locality

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

- **input**
- **blurH**
- **blurV**

Diagram showing parallelism and locality.
Decoupled tiles optimize parallelism & locality

input

blurH

blurV

locality

parallelism
Decoupled tiles optimize parallelism & locality

input

blurH

blurV

locality

parallelism
Decoupled tiles optimize parallelism & locality

- input
- blurH
- blurV

short reuse distance

independence

locality

parallelism
Breaking dependencies introduces redundant work

```
input
\arrow{down}
\text{blurH}
\arrow{down}
\text{blurV}
```

locality

parallelism
Breaking dependencies introduces redundant work.
Breaking dependencies introduces redundant work.
Stencil pipelines require **tradeoffs** determined by **organization of computation**

- **trade off with granularity of fusion**
- **redundant work**
- **locality**
- **parallelism**

**tradeoff**

**trade off by constraining order**
Organization requires global tradeoffs

3x3 box filter
Organization requires global tradeoffs

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
\[ k \leftarrow \text{floor}(I_1(x,y) / \sigma) \]
\[ \alpha \leftarrow (I_1(x,y) / \sigma) - k \]
\[ O(x,y) \leftarrow (1-\alpha) I_2(x,y,k) + \alpha I_2(x,y,k+1) \]

UP: upsample
\[ T_1(2x,2y) \leftarrow I(x,y) \]
\[ T_2 \leftarrow T_1 \otimes [1 3 3 1] \]
\[ O \leftarrow T_2 \otimes [1 3 3 1] \]

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

The algorithm uses 8 pyramid levels

local Laplacian filters
[Paris et al. 2010, Aubry et al. 2011]
Local Laplacian Filters
prototype for Adobe Photoshop / Lightroom

Adobe: 1500 lines
expert-tuned 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,
10x faster on GPU
Message #1: performance requires tradeoffs
Message #1: performance requires tradeoffs

- input
  - blurH
  - blurV

redundant work → locality
Message #1: performance requires tradeoffs

input

blurH

blurV

... ... ...

redundant work

trade off with granularity of fusion

locality
Message #1: performance requires tradeoffs

input → blurH → blurV

... ... ... ...

trade off with granularity of fusion

redundant work

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

input

blurH

blurV

dependencies

task

task schedule

redundant work

tradeoff

locality

parallelism
This is a general task graph.
This is a general task graph

- **input**
  - **blurH**
  - **blurV**

- independent tasks
- redundant work
- locality
- tradeoff
- parallelism

- task
- task dependencies
- task schedule
This is a general task graph

- input
- blurH
- blurV

dependencies

- task
- task schedule

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

input
- blurH
- blurV

dependencies

task schedule

- task
- ↔ dependencies
- ↔ task schedule

tradeoff

parallelism

redundant work

locality
This is a general task graph

input
  ↓
blurH
  ↓
blurV

dependencies

↑
task schedule

task

-dependencies

redundant work

tradeoff

parallelism

locality
Traditional languages conflate algorithm & organization

```c
void box_filter_3x3(const Image &in, Image &blurV) {
  __m128i one_third = _mm_set1_epi16(21846);
  Image blurH(in.width(), in.height()); // allocate blurH array
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i blurH[(256/8)*(yTile+8)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      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+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(blurHPtr++, avg);
          inPtr += 8;
        }
        blurHPtr = blurH;
      }
    }
    for (int y = 0; y < 32; y++) {
      __m128i *outPtr = (_m128i*)(&blurV[yTile+y][xTile]);
      for (int x = 0; x < 256; x += 8) {
        a = _mm_loadl_epi8(blurHPtr+(*256/8));
        b = _mm_loadl_epi8(blurHPtr+*256/8);
        c = _mm_loadl_epi8(blurHPtr++);
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_si128(outPtr++, avg);
      }
    }
  }
}
```

not readable
architecture-specific
hard to change organization
or algorithm
void box_filter_3x3(const Image &in, Image &blurV) {
    __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 *blurHPtr = (blurH + yTile * 32);
        for (int xTile = 0; xTile < in.width(); xTile += 32) {
            sum = _mm_add_epi16(_mm_add_epi16(_mm_loadu_si128((__m128i *)(&in[yTile+y][xTile]-1)),
                                        _mm_loadu_si128((__m128i *)(&in[yTile+y][xTile]+1))),
                            _mm_load_si128((__m128i *)(&in[yTile+y][xTile])));
            avg = _mm_mulhi_epi16(sum, one_third);
            _mm_store_si128(blurHPtr++, avg);
            inPtr += 8;
        }
        blurHPtr = blurH;
    }

    for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (blurV[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
            sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
            avg = _mm_mulhi_epi16(sum, one_third);
            _mm_store_si128(outPtr++, avg);
            outPtr += 8;
        }
    }
}}

// allocate tile blurH array
for (int x = 0; x < 32; x++) {
    __m128i *blurHPtr = (blurH + yTile * 32);
    for (int xTile = 0; xTile < 256; xTile += 8) {
        a = _mm_loadu_si128(blurHPtr + (*256/8));
        b = _mm_loadu_si128(blurHPtr+256/8);
        c = _mm_loadu_si128(blurHPtr++);
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_storeu_si128(outPtr++, avg);
    }
}}}

optimized 3x3 blur in C++

parallelism
distribute across threads
SIMD parallel vectors
void box_filter_3x3(const Image &in, Image &blurV) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurH[8 + 256/8]; // allocate tile blurH array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurHPtr = blurH;
            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(blurHPtr++, avg);
                    inPtr += 8;
                }
                blurHPtr = blurH;
            }
            __m128i *outPtr = (__m128i*)(blurV[yTile+y][xTile]);
            for (int x = 0; x < 256; x += 8) {
                a = _mm_load_si128(blurHPtr-(*256/8));
                b = _mm_load_si128(blurHPtr-256/8);
                c = _mm_load_si128(blurHPtr++);
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(outPtr++, avg);
            }
        }
    }
}
### The effect of organization on performance

<table>
<thead>
<tr>
<th>Approach</th>
<th>Performance (vs. root baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first</td>
<td>1 ×</td>
</tr>
<tr>
<td>Breadth-first + parallel</td>
<td>4 ×</td>
</tr>
<tr>
<td>Interleaving alone</td>
<td>0.8 ×</td>
</tr>
<tr>
<td>Interleaving + parallel</td>
<td>11.5 ×</td>
</tr>
</tbody>
</table>
void box_filter_3x3(const Image &in, Image &blurV) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurH[(256/8)*(yTile+2)]; // allocate tile blurH array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurHPtr = blurH;
            for (int y = -1; y < 32+1; y++) {
                __m128i *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(blurHPtr++, avg);
                    inPtr += 8;
                }
            }
            blurHPtr = blurH;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i*)&blurV[yTile+y][xTile];
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurHPtr+((256/8)-1));
                    b = _mm_load_si128(blurHPtr+256/8);
                    c = _mm_load_si128(blurHPtr++);
                    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 &blurV) {
    Image blurH(in.width(), in.height()); // allocate blurH array
    for (int y = 0; y < in.height(); y++) {
        for (int x = 0; x < in.width(); x++)
            blurH(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++)
            blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
    }
}

For a given algorithm, organize to optimize:

- redundant work
- locality
- parallelism

Same algorithm, different organization
Halide’s answer: decouple algorithm from organization

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:

\[
\begin{align*}
\text{blurH}(x, y) &= (\text{input}(x-1, y) + \text{input}(x, y) + \text{input}(x+1, y))/3; \\
\text{blurV}(x, y) &= (\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1))/3;
\end{align*}
\]
The schedule defines intra-stage order, inter-stage interleaving.

For each stage:

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

For each stage:

1) In what order should it compute its values?

2) When should it compute its inputs?

Diagram showing the stages and their order with labels "input", "blurH", and "blurV".
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>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<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

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

<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>2</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
</tr>
<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>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>21</td>
<td>22</td>
<td>25</td>
<td>26</td>
<td>29</td>
<td>30</td>
</tr>
<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>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>37</td>
<td>38</td>
<td>41</td>
<td>42</td>
<td>45</td>
<td>46</td>
</tr>
<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>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>53</td>
<td>54</td>
<td>57</td>
<td>58</td>
<td>61</td>
<td>62</td>
</tr>
<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>
</tr>
</tbody>
</table>

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

Serial y,
Serial x

for (y : y_{min}..y_{max})
  for (x : x_{min}..x_{max}) {
    eval[ f(x, y) ]
  }
Domain order defines a loop nest for each function

Serial y,
Serial x

Split x by 4,
Split y by 4.
Parallel y₀,
Serial x₀,
Serial yᵢ,
Vectorize xᵢ by 4

$$\text{for } (y : y_{\text{min}}..y_{\text{max}})$$
$$\quad \text{for } (x : x_{\text{min}}..x_{\text{max}}) \{$$
$$\quad \quad \text{eval}[ f(x, y) ]$$
$$\quad \}$$

parfor \( (y_o : y_{o\text{min}}..y_{o\text{max}}) \)
$$\quad \text{for } (x_o : x_{o\text{min}}..x_{o\text{max}})$$
$$\quad \quad \text{for } (y_i : y_{i\text{min}}..y_{i\text{max}})$$
$$\quad \quad \quad \text{simdfor}(x_i : x_{i\text{min}}..x_{i\text{max}} \text{ by } 4) \{$$
$$\quad \quad \quad \quad \text{eval<4>}[ f(x_o*4+x_i, y_o*4+y_i) ]$$
$$\quad \quad \quad \}$$
Domain order defines a loop nest for each function

Serial y,
Serial x

Split x by 4,
Split y by 4.
Parallel yo,
Serial xo,
Serial yi,
Vectorize xi by 4

\[
f.\text{split}(x, \ x_o, \ x_i, \ 4) \\
.\text{split}(y, \ y_o, \ y_i, \ 4) \\
.\text{reorder}(y_o, \ x_o, \ y_i, \ x_i) \\
.\text{parallel}(y_o) \\
.\text{vectorize}(x_i, \ 4)\]
The schedule defines producer-consumer interleaving
The schedule defines producer-consumer interleaving with redundant work, locality, and parallelism tradeoff.
Tradeoff space modeled by granularity of interleaving.
Tradeoff space modeled by granularity of interleaving.
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
- 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

Storage granularity
Tradeoff space modeled by granularity of interleaving.

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

- **storage granularity**
  - redundant computation
  - valid schedules

The diagram illustrates the tradeoff between compute and storage granularity, with different interleaving methods affecting locality and validity of schedules.
Tradeoff space modeled by granularity of interleaving

- **Coarse interleaving**: low locality
- **Fine interleaving**: high locality
- **Compute granularity**: fine interleaving for high locality
- **Storage granularity**: coarse interleaving for low locality
- **Valid schedules**: no redundant computation
- **No redundant computation**: valid schedules
Tradeoff space modeled by granularity of interleaving

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

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

- **coarse interleaving**
  - low locality
  - redundant computation
  - breadth-first execution

- **fine interleaving**
  - high locality
  - no redundant computation

**compute granularity**

**storage granularity**

Code example:
```
blurH.compute_at(root).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
- total fusion

```
blurH.compute_at(blurV, x)
.store_at(blurV, x)
```
Tradeoff space modeled by granularity of interleaving

- **Compute granularity**
  - Fine interleaving: high locality
  - Coarse interleaving: low locality

- **Storage granularity**
  - Redundant computation
  - No redundant computation

**Total fusion**
- blurH.compute_at(blurV, x)
- .store_at(blurV, x)

Redundant work
Tradeoff space modeled by granularity of interleaving

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

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

compute granularity
storage granularity

sliding window fusion

blurH.compute_at(blurV, x)
.store_at(root)
Tradeoff space modeled by granularity of interleaving:

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

- **storage granularity**
  - redundant computation
  - no redundant computation

- **sliding window fusion**

```plaintext
blurH.compute_at(blurV, 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)
- sliding window fusion

```
blurH.compute_at(blurV, x) .store_at(root)
```
Tradeoff space modeled by granularity of interleaving

- **Coarse interleaving**: low locality, no redundant computation
- **Fine interleaving**: high locality, redundant computation

**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

- **tile-level fusion**

- **redundant work**

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

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

Compute granularity

Storage granularity
Tradeoff space modeled by granularity of interleaving

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

- compute granularity
- storage granularity

- redundant computation
- no redundant computation

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

- **Coarse interleaving**
  - Low locality
  - Redundant computation
  - Parallel sliding windows coarse-grained parallelism across windows

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

- **Compute granularity**

- **Storage granularity**

- **Enlarged sliding window**
  - Fine-grained data-parallelism within window
Tradeoff space modeled by granularity of interleaving

- **Coarse interleaving**: low locality, redundant computation
- **Fine interleaving**: high locality, no redundant computation

**Compute granularity**
- **Fine-grained**: parallel sliding windows, coarse-grained parallelism across windows
- **Coarse-grained**: enlarged sliding window, fine-grained data-parallelism within window
Schedule primitives compose to create many redundantly

```
blurH.compute_at_root()
```

```
blurH.compute_at(blurV, x)
  .store_at_root()
```

```
blurV.tile(x, y, xi, yi, 8, 8)
  .parallel(y)
  .vectorize(xi, 4)
```

```
blurV.split(x, x, xi, 8)
  .parallel(x)
  .vectorize(xi, 4)
```

```
blurH.compute_at(blurV, y)
  .store_at_root()
  .split(x, x, xi, 8)
  .vectorize(xi, 4)
  .parallel(x)
```

```
blurV.split(y, y, yi, 8)
  .parallel(y)
  .vectorize(x, 4)
```

```
blurH.compute_at(blurV, y)
  .store_at(blurV, yi)
  .vectorize(x, 4)
```

```
blurV.split(y, y, yi, 8)
  .parallel(y)
  .vectorize(x, 4)
```

```
Schedule primitives **compose** to create many redundant work.

Parallelism and locality are key concepts in this diagram.
Schedule primitives compose to create many redundant work
locality parallelism

void box_filter_3x3(const Image &in, Image &blurV)
{
    __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurHPtr = blurH;
            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(blurHPtr++, avg);
                    inPtr += 8;
                }
                blurHPtr = blurH;
            }
            __m128i *outPtr = ((__m128i*)(blurV[yTile+y][xTile]));
            for (int x = 0; x < 256; x += 8) {
                a = _mm_load_si128(blurHPtr+4);
                b = _mm_load_si128(blurHPtr+256);
                c = _mm_load_si128(blurHPtr++);
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(outPtr++, avg);
            }
        }
    }
}
Halide
0.9 ms/megapixel

Func box_filter_3x3(Func in) {
    Func blurH, blurV;
    Var x, y, xi, yi;

    // The algorithm - no storage, order
    blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

    // The schedule - defines order, locality; implies storage
    blurV.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    blurH.compute_at(blurV, x).store_at(blurV, x).vectorize(x, 8);

    return blurV;
}
Func box_filter_3x3(Func in) {
    Func blurH, blurV;
    Var x, y, xi, yi;

    // The algorithm - no storage, order
    blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

    // The schedule - defines order, locality; implies storage
    blurV.tile(x, y, xi, yi, 256, 32).
        .vectorize(xi, 8).parallel(y);
    blurH.compute_at(blurV, x).store_at(blurV, x).vectorize(x, 8);

    return blurV;
}
```cpp
Func box_filter_3x3(Func in) {
    Func blurH, blurV;
    Var x, y, xi, yi;

    // The algorithm - no storage, order
    blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

    // The schedule - defines order, locality; implies storage
    blurV.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    blurH.compute_at(blurV, x).store_at(blurV, x).vectorize(x, 8);

    return blurV;
}
```
void box_filter_3x3(const Image &in, Image &blurV) {
    __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 blurH[256/8*(32-2)]; // allocate tile blurH array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurHPtr = blurH;
            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)*256));
                    b = _mm_loadu_si128((__m128i*)((inPtr+2)*256));
                    c = _mm_loadu_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_storeu_si128(blurHPtr, avg);
                    inPtr += 8;
                }
                blurHPtr = blurH;
            }
        }
    }

    return blurV;
}
Results
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
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
“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
Local Laplacian Filters
[Paris et al. 2010, Aubry et al. 2014]

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,
10x faster on GPU
Local Laplacian Filters

[Paris et al. 2010, Aubry et al. 2014]

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,
10x faster on GPU
Local Laplacian Filters

[Paris et al. 2010, Aubry et al. 2011]
<table>
<thead>
<tr>
<th>x86</th>
<th>Speedup</th>
<th>Factor shorter</th>
</tr>
</thead>
<tbody>
<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>1.2 x</td>
<td>100s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU</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>ARM</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>
<tr>
<td>VGG 16 layer</td>
<td>CAFFE/MKL</td>
<td>Halide</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------</td>
<td>---------</td>
</tr>
<tr>
<td>Forward time</td>
<td>4.4 secs</td>
<td>2.7 secs</td>
</tr>
<tr>
<td>(batch of 16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory footprint</td>
<td>3222 MB</td>
<td>576 MB</td>
</tr>
<tr>
<td>Code size</td>
<td>&gt;$10^4$ lines</td>
<td>700 lines</td>
</tr>
<tr>
<td>Tuning effort</td>
<td>Years</td>
<td>Hours</td>
</tr>
</tbody>
</table>

4x Xeon E5-4650  
(2.7 Ghz Sandybridge, 32 total cores)  

*extremely preliminary*
How can we determine *good* schedules?

Explicit programmer control
The compiler does exactly what you say.
Schedules cannot influence correctness.
Exploration is fast and easy.

Stochastic search + Empirical benchmarks (autotuning)
Pick your favorite high-dimensional search.
(We used Petabricks’ GA tuner [Ansel et al. 2009])
How can we determine *good* schedules?

Explicit programmer control
The compiler does exactly what you say.
Schedules cannot influence correctness.
Exploration is fast and easy.

Stochastic search + Empirical benchmarks (autotuning)
Brute-force seems easy, but challenging to encode and search since space of organizations is so large

Heuristics are necessary to get anywhere on real programs, but they’re complex and brittle
1. Simple cost model (not benchmarking)
parameters: fast mem. size, compute : mem. cost, min. parallelism

group block size set to fit fast memory
Automatic scheduling

1. Simple cost model (not benchmarking) parameters: fast mem. size, compute : mem. cost, min. parallelism
   group block size set to fit fast memory

2. Group functions to minimize cost
   group block size set to fit fast memory
   single fusion strategy within groups (tiling)
   agglomerative group merging, sorted by cost improvement of possible merges
Automatic scheduling

1. Simple cost model (not benchmarking)
   parameters: fast mem. size, compute : mem. cost, min. parallelism
   group block size set to fit fast memory

2. Group functions to minimize cost
   group block size set to fit fast memory
   single fusion strategy within groups (tiling)
   agglomerative group merging, sorted by cost improvement of possible merges

3. Schedule within groups
   sort dimensions by amount of reuse
   vectorize innermost, parallelize outermost until >threshold
Automatically Scheduling Halide Image Processing Pipelines

Abstract

In recent years, the Halide image processing language has proven to rapidly and automatically generate schedules for Halide pipelines that are performance competitive with expert-tuned Halide schedules across a wide range of applications. The system analyzes a wider range of image processing algorithms in Halide, but they rely on a much smaller set of image processing benchmarks that are competitive with widely-used CPU-based implementations on 6-cores, and up to 32-cores faster on a 32-core server.

The problem is that although Halide provides high-level abstractions enabling programmers to express schedules quickly by handling the mechanical details of platform-specific code optimizations for expressing schedules, and a separate, high-level description of how to globally optimize the algorithm's execution on a machine (called a compiler). The algorithm avoids costly and (often impractical) expert services already present in the Halide compiler to automatically generate high-performance schedules for Halide programs. Our approach is to leverage the function bounds-analysis and a wide set of image processing benchmarks that are competitive with the best expert-tuned Halide schedules across a wide range of applications. The system analyzes a wider range of image processing algorithms in Halide, but they rely on a much smaller number of image processing pipelines than prior automatic scheduling approaches, allowing the system to generate convolutional neural network code that is 30% faster than widely-used CPU-based implementations on 6-cores, and up to 32-cores faster on a 32-core server.

However, although Halide enables programmers to work more quickly by handling the mechanical details of platform-specific code optimizations, it quickly by handling the mechanical details of platform-specific code optimizations, it requires expert knowledge of modern code-optimization techniques and hardware architecture. Even for experts, finding an efficient schedule for a complex image processing pipeline remains difficult and time consuming. In this paper we provide an algorithm for automatically generating high-performance schedules for Halide programs. Our approach is to leverage the function bounds-analysis and a wide set of image processing benchmarks that are competitive with widely-used CPU-based implementations on 6-cores, and up to 32-cores faster on a 32-core server.

We present an algorithm that rapidly and automatically generates schedules for Halide pipelines that are performance competitive with expert-tuned code. (The above pipelines where scheduled in 55, 0.015, 4, 0.05, and 7 seconds, respectively.) Our algorithm requires no about efficient optimization choices. We demonstrate performance than is often equivalent (always within a factor of two above or below) with expert-tuned code. (The above pipelines where scheduled in 55, 0.015, 4, 0.05, and 7 seconds, respectively.) Our algorithm requires no about efficient optimization choices. We demonstrate performance than is often equivalent (always within a factor of two above or below) with expert-tuned code.
Although not shown in the Figure, in addition to comparing the auto-scheduler's output for DNN workloads (e.g., VGG, CONV, and BLUR) to tens of applications, we also ran the same full benchmark suite on ARM, for which the auto-scheduler's output outperforms the manually-optimized schedules. On a number of the more complex benchmarks (e.g., BILATERAL and LENSBLUR), the auto-scheduler’s output performs significantly better than the manually-optimized schedule.

We find that, in most cases, our auto-scheduling approach generalizes well for different target architectures, we also ran the same full benchmark suite extended our cost analysis to consider effects of input storage layout on this benchmark would further exceed that of Caffe/MKL if we deployed Caffe uses Intel Math Kernel library (MKL 11.3.1) for all configurations. The Halide runtime introduces little overhead, so new Halide programmers are able to write.

On the other hand, the manually-optimized schedule unrolls some loops to simplify the resulting performance comparison is shown in Figure 8. This variation is not dissimilar to that seen between the auto-scheduled variant was derived with parallelism in mind, and so the auto-scheduler generates a sub-optimal schedule for an affine transformation of the manually-optimized schedules. Disabling vectorization in our auto-scheduler revealed a performance problem created by vector code generation forms the auto-scheduler's output by an additional 30%. We found that the manually-optimized Halide schedule for CAMERA outperformed Caffe/MKL by 1.3 times the best known values. (Note the performance is that manually-optimized schedule unrolls some loops to simplify the resulting performance comparison is shown in Figure 8. This variation is not dissimilar to that seen between the auto-scheduled variant was derived with parallelism in mind, and so the auto-scheduler generates a sub-optimal schedule for an affine transformation of the manually-optimized schedules. Disabling vectorization in our auto-scheduler revealed a performance problem created by vector code generation forms the auto-scheduler's output by an additional 30%. We found that the manually-optimized Halide schedule for CAMERA outperformed Caffe/MKL by 1.3 times the best known values.

The resulting performance comparison is shown in Figure 8. This variation is not dissimilar to that seen between the auto-scheduled variant was derived with parallelism in mind, and so the auto-scheduler generates a sub-optimal schedule for an affine transformation of the manually-optimized schedules. Disabling vectorization in our auto-scheduler revealed a performance problem created by vector code generation forms the auto-scheduler's output by an additional 30%. We found that the manually-optimized Halide schedule for CAMERA outperformed Caffe/MKL by 1.3 times the best known values.

The auto-scheduled implementation is close to both the reference and auto-scheduler performance because performance is dominated by the bilateral grid construction and vectorization similarly. The manually-optimized implementation is close to both the reference and auto-scheduler performance because performance is dominated by the bilateral grid construction and vectorization similarly. The manually-optimized implementation is close to both the reference and auto-scheduler performance because performance is dominated by the bilateral grid construction and vectorization similarly.
Auto scheduler vs. expert humans

Optimization of Manually Authored Schedules

Throughput (1/ms)

- LENSBLUR
- MAXFILTER
- NL MEANS

Schedule development time (minutes)

- Orange = Auto-scheduler
- Light Blue = Programmer 1
- Dark Blue = Programmer 2
Current status

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

Google
- 65 active developers
- > 200 pipelines
- 10s of kLOC in production

G Photos **auto-enhance**
Data center
Android
Chrome (PNaCl)

YouTube: **65B frames/day**

HDR+
Glass
Nexus devices

dozens of companies on Halide-Dev