An introduction to Halide

Jonathan Ragan-Kelley (Stanford)
Andrew Adams (Google)
Dillon Sharlet (Google)
Today’s agenda

Now: the big ideas in Halide

Later: writing & optimizing real code
   Hello world (brightness)
   Gaussian blur - 3x OpenCV
   Simple enhancement pipeline - 6x OpenCV
   MATLAB integration
   IIR filter
   CNN layers
   GPU scheduling

Finally: real-time HOG on a phone
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: 8 Mpixels
(96MB/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: 8 Mpixels
(96MB/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)
Local Laplacian Filters
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster than reference C
Local Laplacian Filters
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster than reference C
Local Laplacian Filters
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++
multi-threaded, SSE

3 months of work
10x faster than reference C
Local Laplacian Filters
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++
multi-threaded, SSE

3 months of work
10x faster than reference C

Just writing in C isn’t nearly enough!
Imaging is everywhere
A simple example: 3x3 blur

```cpp
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)*((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;
      }
      __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);
      }
    }
  }
}
Halide’s answer: *decouple* algorithm from schedule

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

Easy for programmers to build pipelines  
simplifies algorithm code  
improves modularity

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

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

Pipeline stages are *functions* from coordinates to values

Execution order and storage are unspecified
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:
Var x, y; Func blurx, blury;
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;
Domain scope of the programming model

All computation is over **regular grids**.

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

**Recursion** must have **bounded depth**.
Domain scope
of the programming model

All computation is over **regular grids**.

- **Not Turing complete**
- Only **feed-forward pipelines**
- Recursive/reduction computations are a (partial) escape hatch.
- **Recursion** must have **bounded depth**.
Halide’s answer: *decouple* algorithm from schedule

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

Easy for programmers to build pipelines  
simplifies algorithm code  
improves modularity

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

Easy for the compiler to generate fast code
Halide’s answer: *decouple* algorithm from schedule

**Algorithm:** *what* is computed

**Schedule:** *where* and *when* it’s computed

Easy for programmers to build pipelines
simplifies algorithm code
improves modularity

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

Easy for the compiler to generate fast code
The 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**

- NaCl, PNaCl (in-browser)
- x86 (with SSE/AVX)
- ARM (with NEON)
- CUDA, OpenCL, GL ES (host+kernel graph)
- C (source)
Local Laplacian Filters
prototype for Adobe Photoshop Camera Raw / Lightroom
Local Laplacian Filters
prototype for Adobe Photoshop Camera Raw / Lightroom

Reference: 300 lines C++
Adobe: 1500 lines
3 months of work
10x faster (vs. reference)
Local Laplacian Filters
prototype for Adobe Photoshop Camera Raw / Lightroom

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)
Local Laplacian Filters
prototype for Adobe Photoshop Camera Raw / Lightroom

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)
9x faster (vs. Adobe)
Local Laplacian Filters
prototype for Adobe Photoshop Camera Raw / Lightroom

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)
9x faster (vs. Adobe)
<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
<th>Factor shorter</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blur</td>
<td>1.2 ×</td>
<td>18 ×</td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>4.4 ×</td>
<td>4 ×</td>
</tr>
<tr>
<td>Camera pipeline</td>
<td>3.4 ×</td>
<td>2 ×</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>1.7 ×</td>
<td>7 ×</td>
</tr>
<tr>
<td>Local Laplacian</td>
<td>1.7 ×</td>
<td>5 ×</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 ×</td>
<td>11 ×</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>5.9* ×</td>
<td>7* ×</td>
</tr>
<tr>
<td>Local Laplacian</td>
<td>9* ×</td>
<td>7* ×</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 ×</td>
<td>3 ×</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 ×</td>
<td>18 ×</td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>4.4 ×</td>
<td>4 ×</td>
</tr>
<tr>
<td>Camera pipeline</td>
<td>3.4 ×</td>
<td>2 ×</td>
</tr>
<tr>
<td>“Healing brush”</td>
<td>1.7 ×</td>
<td>7 ×</td>
</tr>
<tr>
<td>Local Laplacian</td>
<td>1.7 ×</td>
<td>5 ×</td>
</tr>
<tr>
<td>Gaussian Blur</td>
<td>1.5 ×</td>
<td>5 ×</td>
</tr>
<tr>
<td>FFT (vs. FFTW)</td>
<td>1.5 ×</td>
<td>10s</td>
</tr>
<tr>
<td>BLAS (vs. Eigen)</td>
<td>1 ×</td>
<td>100s</td>
</tr>
</tbody>
</table>

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

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

*HDR+
- Glass
- Nexus devices

>Movidius
- >20 companies on Halide-Dev

*n secret/unannounced projects*
Today’s agenda

the big ideas in Halide

Now: writing & optimizing real code

Hello world (brightness)
Gaussian blur - 3x OpenCV
Simple enhancement pipeline - 6x OpenCV
MATLAB integration
IIR filter
CNN layers
GPU scheduling

Finally: real-time HOG on a phone
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

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

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

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

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

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

\[
brighten(x, y, c) = \ldots
\]
Domain order defines a **loop nest** for each function

```
brighten(x, y, c) = ...
```

**Default:**
- Serial c,
- Serial y,
- Serial x

```
for c:
  for y:
    for x:
      brighten(...)
```
**Parallel** marks a loop to be multithreaded.

```plaintext
brighten(x, y, c) = ...  
for c:
  parallel y:
    for x:
      brighten(...) = ...

brighten.parallel(y)
```
**Parallel** marks a loop to be multithreaded

```python
brighten(x, y, c) = ...  

for c:
    parallel y:
        for x:
            vectorized x.v in [0,7]:
                brighten(...) = ...
```

```python
brighten.parallel(y) .vectorize(x, 8)
```
Parallel marks a loop to be multithreaded

\[
\text{for } c:\n\begin{align*}
\text{parallel } y: \\
\text{for } x: \\
\text{unrolled } x.v \text{ in } [0,3]: \\
\text{brighten}(\ldots) = \ldots
\end{align*}
\]

\[
\text{brighten}(x, y, c) = \ldots
\]

\[
\text{brighten}.\text{parallel}(y) \cdot \text{unroll}(x, 4)
\]
Parallel marks a loop to be multithreaded

`brighten(x, y, c) = ...` for `c`:
  `for y_o:`
    `for y_i in [0,63]:`
      `for x:`
        `brighten(...) = ...`

`brighten.split(y, y_o, y_i, 64)`
Parallel marks a loop to be multithreaded

brighten(x, y, c) = ...

for y₀:
  for c:
    for yᵢ in [0, 63]:
      for x:
        brighten(...)
          = ...

brighten.split(y, y₀, yᵢ, 64)
  .reorder(c, y₀)
Today's agenda

the big ideas in Halide
writing & optimizing real code

Now: Hello world (brightness)
Gaussian blur - 3x OpenCV
Simple enhancement pipeline - 6x OpenCV
MATLAB integration
IIR filter
CNN layers
GPU scheduling

Finally: real-time HOG on a phone
Today’s agenda

the big ideas in Halide
writing & optimizing real code
Hello world (brightness)

Now: Gaussian blur
Simple enhancement pipeline - 6x OpenCV
MATLAB integration
IIR filter
CNN layers
GPU scheduling

Finally: real-time HOG on a phone
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**?
Organizing the algorithm as a data-parallel pipeline & loops
Organizing the algorithm as a data-parallel pipeline & loops
Organizing the algorithm as a data-parallel pipeline & loops:

- **compute blury:**
  
  ```plaintext
  for ...
  blury(...) = ...
  ```

- **compute blurx:**
  
  ```plaintext
  for ...
  blurx(...) = ...
  ```
Inline maximizes locality, but also recomputes values.

```
compute blurx:
  for c:
    for y:
      for x:
        blurx(...) = ...
```

```
... ...
... ...
... ...
... ...
... ...
... ...
... ...
... ...
```
Inline maximizes locality, but also recomputes values

```plaintext
compute blurx:
  for c:
    for y:
      for x:
        blurx(...) = ...
```

 inline maxmizes locality, but also recomputes values
**Compute root** minimizes **recompute**, but also **locality**

```plaintext
compute blury:
  for c:
    for y:
      for x:
        blury(...) = ...

compute blurx:
  for c:
    for y:
      for x:
        blurx(...) = ...
```
Compute root minimizes recompute, but also locality.

compute blury:
  for c:
    for y:
      for x:
        blury(...) = ...

compute blurx:
  for c:
    for y:
      for x:
        blurx(...) = ...
Compute at $\text{blur}_{x.y}$ interleaves scanlines for better locality

compute $\text{blur}_x$:
  for $c$:
    for $y$:
      compute $\text{blur}_y$:
        for $x$:
          $\text{blur}_y(...) = \ldots$
      for $x$:
        $\text{blur}_x(...) = \ldots$
Compute at $\text{blur}_x.y$ interleaves scanlines for better locality.

\[
\text{compute blur}_x:
\]
\[
\text{for } c:
\]
\[
\text{for } y:
\]
\[
\text{compute blur}_y:
\]
\[
\text{for } x:
\]
\[
\text{blur}_y(...) = \ldots
\]
\[
\text{for } x:
\]
\[
\text{blur}_x(...) = \ldots
\]
Today’s agenda

the big ideas in Halide

writing & optimizing real code

Hello world (brightness)

Now: Gaussian blur - 3x OpenCV
Simple enhancement pipeline - 6x OpenCV
MATLAB integration
IIR filter
CNN layers
GPU scheduling

Finally: real-time HOG on a phone