Visual computing and spatiotemporal data are everywhere
Visual computing demands orders of magnitude more computation
The biggest data is visual

YouTube: 400 hrs uploaded / min
[Brewer 2016]
1.5 Terapixels/sec

250 M surveillance cameras,
2.5 B cell phone cameras, ...
Pervasive sensing: “the cloud” is not enough
data transfer ≫ capture

Sensor +
Read out
5 Mpixels
~1 mJ/frame

transmit
LTE radio
50 Mbit/sec
1 W
~1 J/frame

transmission power costs 1,000x capture

[Wu et al. 2012]
Visual data analysis is expensive

One object detection neural net:
250 Watt GPU $\rightarrow$ 0.25 megapixels at video rate

(yolo-v3 on Tesla V100)
Programmer productivity has exploded

1990s

ORACLE

2010s

React

Python

Firebase

Amazon Web Services
Building high-performance systems is harder than ever

Reference:
200 lines C++

Adobe: 1500 lines
3 months of work
10x faster
My research:
Compilers, systems, architectures, and algorithms for high-performance graphics & visual computing.

Reorganize computations & data.
- Simpler programs
- Order of magnitude faster
- Scalable on future architectures
How can we increase performance and efficiency?

Parallelism
“Moore’s law” scaling requires 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
Message #1: Performance requires complex tradeoffs

![Diagram showing tradeoff between amount of work, serial dependence, and communication (locality).]
Where does performance come from?

- Program
- Hardware

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

Tradeoff
Message #2: organization of computation is a first-class issue.

Program:

- Algorithm
- Organization of computation
- Hardware

Diagram:

- Tradeoff
- Amount of work
- Communication (locality)
- Serial dependence (parallelism)
Reorganizing computation is painful

Reference:
300 lines C++

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

Same algorithm,
Different organization
Global reorganization breaks modularity

The algorithm uses 8 pyramid levels
Algorithm vs. Organization: 3x3 blur

```cpp
#include "Image.h"

// 3x3 blur filter

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;
}
```
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 blrx[((256/8)*(32+2))]; // allocate tile blrx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blrx;
            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;
                }
            }
            blrxPtr = blrx;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i *)(blury[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurxPtr+(2*256)/8);
                    b = _mm_load_si128(blurxPtr+256/8);
                    c = _mm_load_si128(blurxPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
```

11x faster
(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?

*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

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

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 blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurx(x, y)

blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
blury.

blury.tile(x, y, xo, yo, xi, yi, 32, 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

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

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

compute here
The **Schedule** defines a **loop nest** to compute the pipeline

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

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

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

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

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

```c++
// for each tile
parallel for blury.yo:
  for blury.xo:
    // for pixel in required tile
    for blury.yi:
      vec for blury.x:
        compute blury<8>
    // for pixel in tile
    for blury.yi:
      for blury.xi:
        compute blury
```
Prior 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, Amarasinghe & Lam 1993]

**Parallel work scheduling**
- Cilk [Blumhoefe 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**
- [Shantzis 1994], [Levoy 1994]
- PixelBender, CoreImage

*a tiny sample. Thousands have come before us.*
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.

Not Turing complete
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
Simple loops execute **breadth-first** across stages.
Simple loops execute **breadth-first** across stages

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

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

write to memory

read from memory
Breadth-first execution sacrifices locality

locality is a function of reuse distance
Interleaved execution improves locality

(simplified)
Interleaved execution improves locality

input

blurx

blury

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

- **input**
  - **blurx** (fused)
  - **blury**

- **fusion globally interleaves computation**
Understanding dependencies

input

blurx

blury
Stencils have overlapping dependencies
Breaking dependencies introduces redundant work
Decoupled tiles optimize parallelism & locality

locality
(short reuse distance)

parallelism
(independence)
Organization requires global tradeoffs

**The algorithm uses 8 pyramid levels:**

**Subtract:** $O(x,y) \leftarrow I_1(x,y) - I_2(x,y)$

**Add:** $O(x,y) \leftarrow I_1(x,y) + I_2(x,y)$

**Data-dependent access:**

$k \leftarrow \text{floor}(I_1(x,y)) / \sigma$

$\alpha \leftarrow (I_1(x,y) / \sigma) - k$

$O(x,y) \leftarrow (1-\alpha) I_2(x,y,k) + \alpha I_2(x,y,k+1)$

**Upsample:** $T_1(2x,2y) \leftarrow I(x,y)$

$T_2 \leftarrow T_1 \ast [1 \ 3 \ 3 \ 1]$

$O \leftarrow T_2 \ast [1 \ 3 \ 3 \ 1]$
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, 10x faster on GPU
Message #1: performance requires tradeoffs

input

blurx

blury

... ... ...

redundant work

locality

parallelism

trade off with granularity of fusion

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

Order and interleaving radically alter performance of the same algorithm.
Message #3: **dependencies** limit choices of organization

![Diagram with nodes and arrows indicating dependencies and organization]
This is a general **task graph**

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

**Dependencies**

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

**Independent tasks**

- **reuse distance**

**Redundant work**

- **locality**

**Parallelism**

- **tradeoff**
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
  - blury

- task dependencies

- task schedule

- redundant work

- locality

- parallelism

- tradeoff
This is a general **task graph**

- **input**
- **blurx**
- **blurty**

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

- **redundant work**
- **locality**
- **parallelism**
- **tradeoff**
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16((21846));
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i *blurxPtr = blury;
        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));
                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;
            }
        }
    }
    for (int y = 0; y < 32; y++) {  
        __m128i *outPtr = (__m128i*)&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
            a = _mm_load128(blurxPtr); // allocate blurx array
            b = _mm_load128(blurxPtr+256/8);
            c = _mm_load128(blurxPtr++);
            sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
            avg = _mm_mulhi_epi16(sum, one_third);
            _mm_store128(outPtr++, avg);
        }}
    }
}

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

not readable
architecture-specific
hard to change organization or algorithm
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;
                }
            }
        }
    }
    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-(*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);
        }
    }
}}}

Optimized 3x3 blur in C++

parallelism
distribute across threads
SIMD parallel vectors
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]; // allocate tile blurx array
    }
    for (int xTile = 0; xTile < in.width(); xTile += 32) {
        __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)+256/8));
        b = _mm_loadu_si128((__m128i *)((inPtr+1)+256/8));
        c = _mm_load_u128((__m128i *)((inPtr)+256/8));
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_u128(blurxPtr++, avg);
        inPtr += 8;
    }
}
}

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+256/8);
        b = _mm_load_u128(blurxPtr+256/8);
        c = _mm_load_u128(blurxPtr++);
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_u128(outPtr++, avg);
    }
}
The effect of organization on performance

<table>
<thead>
<tr>
<th></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 &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __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+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;
                }
            }
        }
    }
}

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 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:

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

\[
\text{blury}(x, y) = \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)}{3};
\]
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>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>
<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>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</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

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 $y_{\text{outer}}$,
Serial $x_{\text{outer}}$,
Serial $y_{\text{inner}}$,
Serial $x_{\text{inner}}$
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

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

f.split(x, x_o, x_i, 4)
parfor (y_o : y_o_min..y_o_max)
    .split(y, y_o, y_i, 4)
    .reorder(y_o, x_o, y_i, x_i)
    .parallel(y_o)
    .vectorize(x_i, 4)
```

```
The schedule defines producer-consumer interleaving with redundant work and parallelism tradeoff.
Tradeoff space modeled by granularity of interleaving

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

Storage granularity

Valid schedules
Tradeoff space modeled by granularity of interleaving

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

- redundant computation
- no redundant computation

- breadth-first execution

```
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

Total fusion

compute granularity

storage granularity

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

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

- Storage granularity
  - Redundant computation: no reuse, high parallelism
  - No redundant computation: tight reuse, low parallelism

Capturing reuse constrains order (less parallelism)

Sliding window fusion

Parallelism

Example: 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

- tile-level fusion

- redundant work

```
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

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

- Storage granularity
  - Redundant computation
  - No redundant computation

- Parallelism
  - Parallel enlarged sliding windows
  - Enlarged sliding window: fine-grained data-parallelism within window
  - Sliding windows: coarse-grained parallelism across windows

- Data-parallelism within window
  - Enlarged sliding window
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+y)]; // 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);
            }
        }
    }
}

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;
}
Halide
0.9 ms/megapixel

```
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;
}
```

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 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;
                }
                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_loadu_si128(blurxPtr+(2*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);
                }
            }
        }
    }
}
```
More language features beyond the scope of this talk

Computed, data-dependent reads (gather)
\[ f(x) = g(floor(2.3*in(x))) \]

Computed, data-dependent *writes* (scatter)
\[ f(g(floor(2.3*in(x)))) = in(x) \]

Recursive functions (IIR convolution, scan)
\[ cdf(i) = cdf(i-1) + pdf(i) \]
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)

Regis 
Kelly
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)

automatic scheduler?
How can we automatically find good organizations?
We need:

1. A **model** of the organization space

2. An **objective function**

3. An **algorithm** to find good organizations in the space
**Autotuning:** stochastic search over organizations

**Model:** *Halide* schedules

**Objective:** measured benchmark runtime

**Algorithm:** hybrid heuristic search

---

![Graph showing speedup of Bilateral grid, Camera pipe, Interpolate, and Local Laplacian with Human and Autotuned labels.]  

- [Halide, PLDI 2013]  
- [PetaBricks, ASPLOS 2013]  
- [OpenTuner, PACT 2014]
**Autotuning:** stochastic search over organizations

**Model:** valid Halide schedules

**Objective:** measured benchmark runtime

**Algorithm:** hybrid heuristic search

**Requires:**
- Test harness + data (not just program code)
- Long runtime (hours or days)
- Domain-specific heuristics

- [Halide, PLDI 2013]
- [PetaBricks, ASPLOS 2013]
- [OpenTuner, PACT 2014]

---

**Speedup**
- Bilateral grid
- Camera pipe
- Interpolate
- Local Laplacian
Direct heuristic scheduling [SIGGRAPH 2016]

Model: restricted Halide schedules

Objective: simple cost model

\[ \text{cost(alg, org)} \]

Algorithm: greedy clustering,
heuristic within each group

Good schedules in **seconds**

**Scalable** to very large programs
Superhuman autoscheduling

Training a compiler to automatically optimize organization using learning & search.

Tree search on schedules

Learned cost model

Benchmark performance

Random Halide algorithm

Plausible schedules

Importance sample

Train

Input Halide algorithm

Plausible schedules

Importance sample

Fine tune

Search optimum

Benchmark performance

Fast schedule

Plausible schedules

direct

Fast schedule

with autotuning

Autoscheduling
Superhuman autoscheduling [in review]

![Graph showing runtime comparison between PyTorch, Caffe2, Ours (2 mins), and Ours (3 hours) for ResNet-50 (x86).]
The Halide language

Decouples **algorithm** from **organization**
structured as a scheduling language to navigate fundamental tradeoffs.

**Simpler** programs
**Faster** than hand-tuned code
**Scalable** across architectures
Real-world adoption

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

Google

> 1800 pipelines
10s of kLOC in production

Pixel HDR+

“Best smartphone camera ever”