High-Performance Image Processing

Frédo Durand
most slides by Jonathan Ragan-Kelley
MIT CSAIL
4D lightfields: orders of magnitude from “good enough”

Current Lytro
10 Mrays
<1 Mpixels
5 secs (on desktop PC)

Scale to 4k video
100 Mrays
8 Mpixels
1 min/frame (on desktop PC)

[Ng 2005; Ng et al. 2006]
images by Ren Ng, Lytro
4D lightfields: orders of magnitude from “good enough”

Current Lytro
10 Mrays
<1 Mpixels
5 secs
(on desktop PC)

Scale to 4k video
100 Mrays
8 Mpixels
1 min/frame
(on desktop PC)

1 hour to process
1 second of video

[Ng 2005; Ng et al. 2006]
Images by Ren Ng, Lytro
Rendering: orders of magnitude from “good enough”

<table>
<thead>
<tr>
<th>Modern game: Team Fortress 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Mpixels</td>
</tr>
<tr>
<td>0.5 Mpolys</td>
</tr>
<tr>
<td>10 ms/frame</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CG movie: Tintin, Avatar</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Mpixels</td>
</tr>
<tr>
<td>5 Gpolys</td>
</tr>
<tr>
<td>5 hrs/frame</td>
</tr>
</tbody>
</table>

images by Valve, Weta
Rendering: orders of magnitude from “good enough”

Modern game:
Team Fortress 2
- 2 Mpixels
- 0.5 Mpolys
- 10 ms/frame

CG movie:
Tintin, Avatar
- 8 Mpixels
- 5 Gpolys
- 5 hrs/frame

6 orders of magnitude more computation

images by Valve, Weta
3D printing: orders of magnitude from “good enough”

1500 cm³ shoe,
10 µm detail,
16 materials

2500³ DPI
10¹² voxels
25 terabytes
3D printing: orders of magnitude from “good enough”

1500 cm$^3$ shoe, 10 µm detail, 16 materials
2500$^3$ DPI
10$^{12}$ voxels
25 terabytes

10 shoes/hour = 4B voxels/sec
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 Mpixels
~1 mJ/frame

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

<table>
<thead>
<tr>
<th>Sensor + Read out</th>
<th>LTE radio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Mpixles</td>
<td>50 Mbit/sec</td>
</tr>
<tr>
<td>~1 mJ/frame</td>
<td>1 W</td>
</tr>
</tbody>
</table>

![Eulean Video Magnification](Wu et al. 2012)

*transmission power costs 1,000x capture*
High throughput imaging: orders of magnitude from "good enough"

*most sensing is "imaging"*
Your data-intensive problem here...
Making image processing faster

- Faster algorithms
- Faster Hardware
- Parallelism
- Memory behavior
Algorithmic acceleration (not today’s topic though)

Sometimes exact, sometimes approximate

**e.g. Fast box blur**
Separable (exact)
Incremental (3 taps instead of 2*radius, exact)
\[(\text{box}(x+1)=\text{box}(x)+\text{input}(x-\text{radius})+\text{input}(x+\text{radius}+1))\]

**e.g. Bilateral Grid (approximate)**

**e.g. lookup tables (approximate)**

See e.g. Andrew Adams’ slides [http://www.stanford.edu/class/cs448f/lectures/2.2/Fast%20Filtering.pdf](http://www.stanford.edu/class/cs448f/lectures/2.2/Fast%20Filtering.pdf)
Algorithmic acceleration (not today’s topic though)

e.g. Fast Gaussian blur
Separable (exact)
Recursive (approximate)
Iterated Box (approximate)
FFT (exact up to wraparound)

\[ G(\mathbf{x} + \mathbf{w}) = a G(\mathbf{x}) + b G(\mathbf{x} + \mathbf{d}) \]

\[ + a' I(\mathbf{x}) + b' I(\mathbf{x} + \mathbf{d}) \]

See e.g. Andrew Adams’ slides
http://www.stanford.edu/class/cs448f/lectures/2.2/FastFiltering.pdf
Faster hardware

Faster CPU
More GHZ
More parallelism (multicore, SIMD vector-unit). But hard to program
Better memory bandwidth

Graphics Hardware
Lots of parallelism
Can be annoying to program and debug (CUDA)
Can we better exploit hardware?

Parallelism

Good cache coherence

Requires to reorganize computation!
e.g. Local Laplacian Filtering

Reference: 300 lines C++

Adobe: 1500 lines
3 months of work
10x faster (vs. reference)
Spoiler: e.g. Local Laplacian Filtering

Reference: 300 lines C++
Adobe: 1500 lines
3 months of work
10x faster (vs. reference)
Parallelize (multicore)
Parallelize (SIMD vectorization)
Organized into tiles to maximize locality
Other tricks
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: 70x faster (vs. reference)
Spoiler: 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: 70x faster (vs. reference)
How can we get there?
How can we get there?

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

- Frequency doesn’t increase much
- Pipeline parallelism has peaked
Parallelism
“Moore’s law” growth will require exponentially more parallelism.

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(32-bit operands)</td>
<td>(28 nm)</td>
<td>(vs. ALU)</td>
</tr>
<tr>
<td>ALU op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load from SRAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move 10mm on-chip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send off-chip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send to DRAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send over LTE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

data from John Brunhaver, Bill Dally, Mark Horowitz

<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>1-5 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;10 µJ</td>
<td>10,000,000x</td>
</tr>
</tbody>
</table>
Message #1: Performance requires complex tradeoffs
Where does performance come from?

- Program
- Algorithm
- Organization of computation
- Hardware

Redundant work
Locality
Parallelism
Tradeoff
Where does performance come from?

- Hardware
- Program

- Parallelism
- Locality
- Redundant work
- Tradeoff
Message #2: organization of computation is a first-class issue

Program:

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

Program:

- Algorithm
- Organization of computation
- Hardware
Message #2: organization of computation is a first-class issue

Program:

Algorithm
Organization of computation
Hardware
Message #2: organization of computation is a first-class issue

Program:

Algorithm

Organization of computation

Hardware

redundant work

locality

tradeoff

parallelism
Halide
a language and compiler for image processing

[SIGGRAPH 2012, PLDI 2013]

Algorithm
Organization of computation
Hardware

redundant work
tradeoff
locality
parallelism
Algorithm vs. Organization: 3x3 blur

void box_filter_3x3(const Image &in, Image &blury) {
    Image blurx(in.width(), in.height());  // allocate blurx array
    for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
            blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
            blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
Algorithm vs. Organization: 3x3 blur

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

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

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

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

    for (int 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;
}
```

15x faster
because better memory coherence
Algorithm vs. Organization: 3x3 blur

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

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

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

Same algorithm, different organization
One of them is 15x faster
Why swapping loops make things faster/slower
In general

Reorganize computation to maximize parallelism & locality
- e.g. compute in tiles, merge pipeline stages
- e.g. compute blur_x and blur_y for a full tile, compute tiles in parallel,
  leverage SIMD vector units
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846); // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
        __m128i *blurxPtr = blury;
        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((__m128i *)(inPtr + y));
                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_storeu_si128(outPtr++, avg);
                inPtr += 8;
            }
            blurxPtr = blury;
        }
    }
}
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurx[32 * 32 * 32] = {0}; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = 0; y < in.height(); y++) {
                __m128i *inPtr = &in[yTile+y][xTile];
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(inPtr+0));
                    b = _mm_loadu_si128((__m128i*)(inPtr+1));
                    c = _mm_load_si128((__m128i*)(inPtr+2));
                    sum = _mm_add_epi16(_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 < in.width(); x += 8) {
                a = _mm_load_si128((blurxPtr+0)/8);
                b = _mm_load_si128((blurxPtr+256)/8);
                c = _mm_load_si128((blurxPtr+512)/8);
                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*?
(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: formula for desired value at pixel
no notion of loop

Schedule: organization of computation
when computed where stored
within a pipeline stage
across pipeline stages
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  
manual or automatic search

Easy for the compiler to generate fast code
Halide algorithm:

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

no loop, it's implicit
Halide algorithm:

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

Halide schedule:

\[
\begin{align*}
\text{blurx}.\text{tile}(x, y, xi, yi, 256, 32).\text{vectorize}(xi, 8).\text{parallel}(y); \\
\text{blurx}.\text{compute}_\text{at}(\text{blurx}, x).\text{store}_\text{at}(\text{blurx}, x).\text{vectorize}(x, 8);
\end{align*}
\]
**Prior work***

*Thousands have come before us.*

**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 [Blumhofs 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.*
```cpp
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);  // Allocate tile blurx array
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i blurx[16] = (in[yTile] + in[yTile + 1] + in[yTile + 2]) / 3;
        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];
                a = _mm_loadu_si128((__m128i *)(inPtr + 0));
                b = _mm_loadu_si128((__m128i *)(inPtr + 1));
                c = _mm_loadu_si128((__m128i *)(inPtr + 2));
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(blurxPtr++, avg);
                inPtr += 8;
            }
        }  // allocate tile blurx array
        __m128i *blurxPtr = blury;
        for (int y = 0; y < 32 + 1; y++) {
            const uint16_t *inPtr = &blury[yTile+y][xTile];
            a = _mm_loadu_si128((__m128i *)(inPtr));
            b = _mm_loadu_si128(blurxPtr + 16);
            c = _mm_loadu_si128(blurxPtr + 32);
            sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
            avg = _mm_mulhi_epi16(sum, one_third);
            _mm_store_si128(outPtr++, avg);
            blurxPtr += 8;
        }
    }
}
```
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 (*autotuning*)**
Pick your favorite high-dimensional search. (We used Petabricks’ genetic algorithm tuner [Ansel et al. 2009])
The Halide Language
Image<float> input = load<float>("images/rgb.png");

Var x, y;
Func blur_x;
Func blur_y;

blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;

Image<float> output = blur_y.realize(input.width()-2, input.height()-2);
Halide is an embedded language

```cpp
Image<float> input = load<float>("images/rgb.png");
Var x, y;
Func blur_x;
Func blur_y;
blur_x(x, y) = (input(x, y) + input(x+1, y) + input(x+2, y))/3.0;
blur_y(x, y) = (blur_x(x, y) + blur_x(x, y+1) + blur_x(x, y+2))/3.0;
Image<float> output = blur_y.realize(input.width()-2,
                                      input.height()-2);
```
Metaprogramming

Create C++ objects that describe a Halide program

Essentially algebraic trees (Abstract Syntax Tree, AST)
Metaprogramming

Image<float> input = load<float>("images/rgb.png");
Var x, y;
Func blur_x;
Func blur_y;
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;
Metaprogramming

Create C++ objects that describe a Halide program

Essentially algebraic trees (Abstract Syntax Tree, AST)

Once the representation is constructed, call .realize() to compile and execute

This calls the C++ Halide compiler, creates binary, executes it

Metaprogramming

Makes it easy to embed in an existing language and codebase
Avoids the need to parse
Syntax: Main types/keywords

*Func*: pure functions over an integer domain

*Var*: pure abstract variables for domain of Funcs

*Expr*: algebraic expressions of Funcs and Var

including standard operators and functions (+, -, &, /, **, sqrt, sin, cos...)

*Image*: arrays used as inputs and outputs
Basic Halide program (default schedule)

Image<float> input = load<float>("images/rgb.png");

Var x, y;
Func blur_x;
Func blur_y;

blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;
brm_x(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;

Image<float> output = blur_y.realize(input.width()-2,
input.height()-2);
Loops are implicit