High-Performance Image Processing

Frédo Durand
most slides by Jonathan Ragan-Kelley
MIT CSAIL
Our code is slow
Often minutes of computation for small images
Simple box blur

```python
def box_x(im):
    w, h = im.shape[1], im.shape[0]
    out = numpy.empty([h, w])
    for y in xrange(1, h-1):
        for x in xrange(1, w-1):
            out[y, x] = (im[y, x-1] + im[y, x] + im[y, x+1]) / 3.0
    return out

def box_y(im):
    w, h = im.shape[1], im.shape[0]
    out = numpy.empty([h, w])
    for y in xrange(1, h-1):
        for x in xrange(1, w-1):
            out[y, x] = (im[y-1, x] + im[y, x] + im[y+1, x]) / 3.0
    return out

def blur(im):
    return box_y(box_x(im))
```

6.4 seconds per megapixel
Speed

Our code is slow
Often minutes of computation for small images

Professional code is very fast
e.g. Lightroom processes megapixels images in real time

Why?
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”

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

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”

<table>
<thead>
<tr>
<th>Sensor + Read out</th>
<th>5 Mixels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>~1 mJ/frame</td>
</tr>
</tbody>
</table>

*Eulerian Video Magnification [Wu et al. 2012]*
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 programming language
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)

See e.g. Andrew Adams’ slides
http://www.stanford.edu/class/cs448f/lectures/2.2/Fast%20Filtering.pdf
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)
Making image processing faster

- Faster algorithms
- Faster programming language
- Faster Hardware
- Parallelism
- Memory behavior
Faster programming language

Python is slow because it’s interpreted:

**Parse (syntax)**
- verify syntax
- translate into internal representation

**Execute (semantics)**

All this takes time!
for us, mostly execution
def eval(x, env=global_env):
    "Evaluate an expression in an environment."
    if isa(x, Symbol):       # variable reference
        return env.find(x)[x]
    elif isa(x, list):       # constant literal
        return x
    elif x[0] == 'quote':    # (quote exp)
        (_, exp) = x
        return exp
    elif x[0] == 'if':       # (if test conseq alt)
        (_, test, conseq, alt) = x
        return eval(conseq if eval(test, env) else alt, env)
    elif x[0] == 'set!':     # (set! var exp)
        (_, var, exp) = x
        env.find(var)[var] = eval(exp, env)
    elif x[0] == 'define':   # (define var exp)
        (_, var, exp) = x
        env[var] = eval(exp, env)
    elif x[0] == 'lambda':   # (lambda (var*) exp)
        (_, vars, exp) = x
        return lambda *args: eval(exp, Env(vars, args, env))
    elif x[0] == 'begin':    # (begin exp*)
        for exp in x[1:]:
            val = eval(exp, env)
        return val
    else:                    # (proc exp*)
        exps = [eval(exp, env) for exp in x]
        proc = exps.pop(0)
        return proc('exp*')

isa = isinstance
Symbol = str
Some interpreters

http://norvig.com/lispy.html


http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-26.html#%e5%8f%b7%e6%b5%81%e8%af%a2%e5%9f%b9%e6%8e%a8%e5%bc%8f%e4%b8%8b%e5%8d%81
Faster programming language

Python is slow because it’s interpreted

Switch to a compiled language
typically C, C++
Java is intermediate (just-in-time compilation)
There are compilers for Python, e.g. pypy

BTW, numpy code is compiled.
void blur(const Image &in, Image &blurred) {
    Image tmp(in.width(), in.height());
    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            tmp(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++)
            blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
void blur(const Image &in, Image &blurred) {
    Image tmp(in.width(), in.height());
    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            tmp(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++)
            blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}

Recall Python: 6.4 s/ megapixel
void blur(const Image &in, Image &blurred) {
    Image tmp(in.width(), in.height());
    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            tmp(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++)
            blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}

Recall Python: 6.4 s/ megapixel

C: 0.015s/megapixel
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)

The free lunch is over

Same Instruction Multiple Data 8
as opposed to MIMD
Can we do better?

Parallelism
Good cache coherence
Requires to reorganize computation!
Can we do better?

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)
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
Decoupling Algorithms from the Organization of Computation for High-Performance Graphics & Imaging

Jonathan Ragan-Kelley
MIT CSAIL
Graphics & Imaging are orders of magnitude from “good enough”
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?

Parallelism
“Moore’s law” growth will require exponentially more parallelism.
How can we get there?

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (28 nm)</th>
<th>Cost (vs. ALU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU op</td>
<td>1 pJ</td>
<td>-</td>
</tr>
<tr>
<td>Load from SRAM</td>
<td>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>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (28 nm)</th>
<th>Cost (vs. ALU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU op</td>
<td>1 pJ</td>
<td>-</td>
</tr>
<tr>
<td>Load from SRAM</td>
<td>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>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (28 nm)</th>
<th>Cost (vs. ALU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU op</td>
<td>1 pJ</td>
<td>-</td>
</tr>
<tr>
<td>Load from SRAM</td>
<td>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>

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

- Redundant work
- Locality
- Parallelism
- Tradeoff
Where does performance come from?

- Program
- Hardware

- Redundant work
- Tradeoff
- Locality
- Parallelism
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

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

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

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

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

```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;
}
```

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

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

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

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

Same algorithm, different organization
One of them is 15x faster
Why does swapping loops make things slower?
Why does swapping loops make things slower?

Memory behavior

Images are layed out linearly in memory.
More coherence along scanlines
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    __m128i* blurx = _mm_malloc((256/8)*(32+2)); // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
        for (int y = 1; y < 32+1; y++) {
            __m128i* outPtr = (&(blury[yTile+y][xTile]));
            for (int x = 0; x < 256; x += 8) {
                a = _mm_loadu_si128((__m128i*)(inPtr+x));
                b = _mm_loadu_si128((__m128i*)(inPtr+x+256/8));
                c = _mm_loadu_si128((__m128i*)(inPtr+x+256/8));
                sum = _mm_add_epi16(_mm_add_epi16(a,b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_storeu_si128(outPtr++, avg);
                inPtr += 8;
            }
        }
    }
}
void box_filter_3x3(const Image &in, Image &blury) {

  __m128i one_third = _mm_set1_epi16(21846);
  
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)(inPtr-1));
          b = _mm_loadu_si128((__m128i*)(inPtr+1));
          c = _mm_load_si128((__m128i*)(inPtr));
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
          inPtr += 8;
        }
      }
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i*)(&blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
        }
      }
    }
  }
}
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 blury[(256/8)*(yTile+1)]; // allocate tile blury array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blury;
            for (int y = 0; 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(outPtr++, avg);
                    inPtr += 8;
                }
            }
            bluryPtr = 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(bluryPtr+((2*256)/8));
                    b = _mm_loadu_si128(bluryPtr+256/8);
                    c = _mm_loadu_si128(bluryPtr);
                    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?
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 bluryx[(256/8) * (yTile + 2)]; // allocate tile bluryx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = bluryx;
      for (int y = -1; y < 32+y; y++) {
        const uint16_t *inPtr = (&(in[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)(inPtr+(2*256)));
          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;
        }
      }
      bluryxPtr = bluryx;
      for (int y = 0; y < 32+y; y++) {
        __m128i *outPtr = (&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(bluryxPtr+(2*256/8));
          b = _mm_load_si128(bluryxPtr+256/8);
          c = _mm_load_si128(bluryxPtr++);
          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?
Hand-optimized C++

9.9 → 0.9 ms/megapixel

```cpp
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 = &((in.height()+32) / 8); // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32; y++) {
                const uint16_t *inPtr = &((yTile+y)[xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(inPtr-1));
                    b = _mm_loadu_si128((__m128i*)(inPtr+1));
                    c = _mm_load_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurxPtr, avg);
                    inPtr += 8;
                }
            }
            blurxPtr = blurx;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (_m128i*)(blury[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadl_epi64((blurxPtr+2*256)/8);
                    b = _mm_loadl_epi64((blurxPtr+256)/8);
                    c = _mm_loadl_epi64((blurxPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_storel_epi64(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*?
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:

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

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

Halide schedule:

\[
\text{blury.} .tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
\]

\[
\text{blurx.} .compute_at(blury, x).store_at(blury, x).vectorize(x, 8);
\]