High-Performance Image Processing

Frédo Durand
most slides by Jonathan Ragan-Kelley
MIT CSAIL
Always remember

No for loop in Halide

no if either (but there is a select)

Halide is not Turing complete
Schedule

Default schedule can be a disaster
inlines everything, lots of redundancy

First non-dumb schedule:
When a function’s consumer has a footprint, schedule as root
Otherwise inline

Focus on locality and redundancy first (tile)
Although you won’t gain as much from locality without parallelism

worry about parallelism next

Do vectorization last
Doesn’t always pay off

Performance is a non-linear business
Jonathan’s scheduling strategy

Schedule root for stencil producers

Basic parallelization over scanlines or tiles

Then worry about fusion/interleaving
Worry only about producer that are consumed by stencils
Don’t worry about pointwise
Tips

print a lot, after every stage

name your Funcs and Var

as usual, debug on small images
save images as numpy array. Much faster to load than png

but scheduling depends on image size (locality...)
References

http://halide-lang.org/

http://people.csail.mit.edu/jrk/halide12


http://www.connellybarnes.com/documents/halide/
Main Halide elements

**Func**: pure functions defined over integer domain
These are the central objects in Halide
  - e.g. blur, brighten, harris, etc.

**Var**: abstract variable representing the domain
  - e.g. x, y, c

**Expr**: Algebraic expression of Halide Funcs and Vars
  - e.g. x**2+y+blur[x,y,c]
  - Most operators and functions you expect are available (+, -, *, /, **, sqrt, cos..)

**Image**: represents inputs and output image
Can be created from our numpy arrays.
  - Careful: convention is that order is x, y
Halide Box Blur (Python embedding)

```python
inputP=imageIO.imread('rgb.png')[::,::,1]
input=Image(Float(32), inputP)

x, y = Var(), Var()
blur_x = Func()
blur_y = Func()

blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3

output=blur_y.realize(input.width()-2, input.height()-2)
```
Recap & Questions?

Functional (no side effect, no loop in algorithms)
Pipeline of functions

Embedded in Python:
Under the hood: Python classes represent the Halide program

Simple syntax: Func, Var, Expr

Call realize to compile and execute

```python
input=Image(Float(32), inputP)
x, y = Var(), Var()
blur_x, blur_y = Func(), Func()
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3
output=blur_y.realize(input.width()-2, input.height()-2)
```
Schedule

Given an algorithm as a pipeline of Func

**Specifies the order of computation**
within each Func
across Func (the more interesting part)

**Boils down to specifying nested loops**
e.g. tile computation:
for y_tile_index:
  for x_tile_index:
    for y_within_tile:
      for x_within_tile:
        compute_stuff()
Schedule across stages

e.g. separable blur: blur_y of blur_x

blur_y (later stage) is the **consumer**

blur_x (earlier) is the **producer**

Locality: have values consumed soon after they are produced

Scheduling is driven by the consumer

Specify when the producer is computed with respect to the consumer
Schedule

Given an algorithm as a pipeline of Func

Specifies the order of computation within each Func
across Func (the more interesting part)

Boils down to specifying nested loops
e.g. tile computation:
for y_tile_index:
  for x_tile_index:
    for y_within_tile:
      for x_within_tile:
        compute_stuff()
Schedule across stages

e.g. separable blur: blur_y of blur_x

blur_y (later stage) is the consumer

blur_x (earlier) is the producer

Locality: have values consumed soon after they are produced

Scheduling is driven by the consumer

Specify when the producer is computed with respect to the consumer
Root schedule

Most similar to what you’d do in Python or C

Compute each stage entirely before computing the next one

Specified with Func.compute_root()

Default for the output (last stage)

```python
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3
blur_y.compute_root()
blur_x.compute_root()
```
Root schedule equivalent Python

```python
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
# compute blur_x at root
# Note the +2 in both the array allocation and the for loop,
# because blur_y needs has a 3-pixel high footprint
tmp=numpy.empty((width, height+2))
for y in xrange(height+2):
    for x in xrange(width):
        tmp[x,y]=(inputP[x,y]+inputP[x+1,y]+inputP[x+2,y])/3
# compute blur_y
for y in xrange(height):
    for x in xrange(width):
        out[x,y] = (tmp[x,y]+tmp[x,y+1]+tmp[x,y+2])/3
```
Inline schedule

Opposite of root: compute producer right when needed, for each value of the consumer
No loop for the producer, all inside consumer

Specified with Func.compute_inline()

Default schedule (except output)
This makes it easier to use many intermediate Funcs for complex algebraic calculations.

```python
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3
blur_y.compute_root()
blur_x.compute_inline()
# both could be skipped since they are the default
```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
#compute blur_y
for y in xrange(height):
    for x in xrange(width):
        # compute blur_x inline, inside of blur_y
        out[x,y] = ((inputP[x,y]+inputP[x+1,y]+inputP[x+2,y])/3
                        + (inputP[x,y+1]+inputP[x+1,y+1]+inputP[x+2,y+1])/3
                        + (inputP[x,y+2]+inputP[x+1,y+2]+inputP[x+2,y+2])/3 )/3

producer is computed right when needed
Root vs. Inline

Scheduling is often about finding a compromise halfway between root and inline:
- Good locality like inline
- Limit redundancy like root

![Diagram](image-url)
Tiling and Fusion (within and across tiles)

Split consumer into tiles and compute producer for the whole tile just before computing the consumer tile.

Specified using `.tile()` on consumer and `.compute_at()` on producer

compute_at inserts the producer loop at a given level of the nested consumer loops

Halfway between root and inline:
root within tile: produce full tile before consuming it
inline across tile: compute a tile when it is needed
Tiling and Fusion (within and across tiles)

Split consumer into tiles and compute producer for the whole tile just before computing the consumer tile.

Often requires to enlarge the producer tile
Because the consumer footprint might need extra pixels
  e.g. blur_y needs extra pixels vertically
Inferred automatically by Halide
Tiling and Fusion (within and across tiles)

Split consumer into tiles and compute producer for the whole tile just before computing the consumer tile.
Specified using `.tile()` on consumer and `.compute_at()` on producer

`compute_at` insert the producer loop at a given level of the nested consumer loops (xo here, x coordinate of tile)

```python
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3
xo, yo, xi, yi = Var(), Var(), Var(), Var()
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32)  # Tile consumer
blur_x.compute_at(blur_y, xo)  # Compute producer or tile granularity
```
Tiling and Fusion: Python equivalent

```python
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange((height+31)/32):
    for xo in xrange((width+255)/256):
        tmp=numpy.empty((256, 32+2))
        for yi in xrange(32+2):
            y=yo*32+yi
            if y>=height: y=height-1
            for xi in xrange(256):
                x=xo*256+xi
                if x>=width: x=width-1
                tmp[xi,yi]=(inputP[x,y]+inputP[x+1,y]+inputP[x+2,y])/3
        for yi in xrange(32):
            y=yo*32+yi
            if y>=height: y=height-1
            for xi in xrange(256):
                x=xo*256+xi
                if x>=width: x=width-1
                out[x,y] = (tmp[xi,yi]+tmp[xi,yi+1]+tmp[xi,yi+2])/3
```
Tiling and Fusion: Python equivalent

```python
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange((height+31)/32):  # loops over tile
    for xo in xrange((width+255)/256):
        tmp=numpy.empty((256, 32+2))  # tile to store blur_x. Note +2 for enlargement
        for yi in xrange(32+2):  # loops for blur_x nested inside xo yo of blur_y
            y=yo*32+yi
            if y>=height: y=height-1  # for boundary tiles
            for xi in xrange(256):
                x=xo*256+xi
                if x>=width: x=width-1  # for boundary tiles
                tmp[xi, yi]=(inputP[x,y]+inputP[x+1,y]+inputP[x+2,y])/3
                # computation on x,y but store at xi, yi
        for yi in xrange(32):  # loops for blur_y
            y=yo*32+yi
            if y>=height: y=height-1  # for boundary tiles
            for xi in xrange(256):
                x=xo*256+xi
                if x>=width: x=width-1  # for boundary tiles
                out[x, y] = (tmp[xi, yi]+tmp[xi, yi+1]+tmp[xi, yi+2])/3
                # computation on xi,yi but store at x, y (opposite of blur_x)
```
Schedule visualization

blur

root

\( x_0 \)

\( x_i \)

compute and store at \( x_0 \) of blur_2

\( x \)

load

blur

input

compute and store at root

\( x \)
Tiling and fusion: pros and cons

- **Fusion**: interleaving
- **Redundant work**: big tiles -> root is best
- **Parallelism**: tile size affects
  - redundancy length
  - locality if tile is bigger than cache trouble
- **Locality**: small tiles

Tradeoff between these aspects.
Tiling and fusion pros and cons

Scheduling is often about finding a compromise halfway between root and inline

good locality like inline
limit redundancy like root

Tile size controls the tradeoff
Extra scheduling options

Reorder
e.g. for x for y => for y for x

Split
e.g. for x => for xo for xi
Tiling combines a set of splits and a reorder

Unroll

Vectorize

Parallelize

some CUDA-specific
Final 3x3 blur: add parallelism and vectorization

\[
x, y = \text{Var}('x'), \text{Var}('y') \\
\text{blur}_x, \text{blur}_y = \text{Func}('\text{blur}_x'), \text{Func}('\text{blur}_y') \\
\text{blur}_x[x,y] = (\text{input}[x,y]+\text{input}[x+1,y]+\text{input}[x+2,y])/3.0 \\
\text{blur}_y[x,y] = (\text{blur}_x[x,y]+\text{blur}_x[x,y+1]+\text{blur}_x[x,y+2])/3.0 \\
\]

\[
\text{x}_i, \text{y}_i = \text{Var}('\text{x}_i'), \text{Var}('\text{y}_i') \\
\text{blur}_y\text{.tile}(x, y, x_i, y_i, 8, 4) \ \backslash \\
\hspace{1cm} .\text{parallel}(y) \ \backslash \\
\hspace{1cm} .\text{vectorize}(x_i, 8) \\
\text{blur}_x\text{.compute_at}(\text{blur}_y, x) \ \backslash \\
\hspace{1cm} .\text{vectorize}(x, 8) \\
\]

\[
\text{output} = \text{blur}_y\text{.realize}(\text{input}.\text{width}()-2, \text{input}.\text{height}()-2) \\
\]
More general box blur, 5x5, 35 MPixels on 12 cores

- default schedule
  took 5.80736255646 seconds

- root first stage
  took 1.99803357124 seconds

- tile 256 x 256 + interleave
  took 1.7552740097 seconds

- tile 256 x 256 + parallel+vector
  took 0.550438785553 seconds

- tile 256 x 256 + parallel+vector without interleaving
  took 1.10970659256 seconds
Schedule visualization

Blurred images can be computed and stored at $x_0$ of blur2.

Input images are computed and stored at the root.

Blurred images are loaded into memory and then used for further processing.

Load arrows indicate the flow of information from memory to computations.
Tiling and Fusion: Python equivalent

```python
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange((height+31)/32):  # loops over tile
    for xo in xrange((width+255)/256):
        tmp=numpy.empty((256, 32+2))  # tile to store blur_x. Note +2 for enlargement
        for yi in xrange(32+2):  # loops for blur_x nested inside xo yo of blur_y
            y=yo*32+yi
            if y>=height: y=height-1  # for boundary tiles
            for xi in xrange(256):
                x=xo*256+xi
                if x>=width: x=width-1  # for boundary tiles
                tmp[xi, yi]=(inputP[x,y]+inputP[x+1,y]+inputP[x+2,y])/3  # computation on x,y but store at xi, yi
        for yi in xrange(32):  # loops for blur_y
            y=yo*32+yi
            if y>=height: y=height-1  # for boundary tiles
            for xi in xrange(256):
                x=xo*256+xi
                if x>=width: x=width-1  # for boundary tiles
                out[x, y] = (tmp[xi, yi]+tmp[xi, yi+1]+tmp[xi, yi+2])/3  # computation on xi,yi but store at x, y (opposite of blur_x)
```

indexing code affected by schedule in green
Schedule visualization

- **blur**
  - Compute and store at $x_0$ of blur2
  - $x$

- **root**
  - $x_0$
  - $x_i$

- **input**
  - Compute and store at root
  - $x$

Load arrows indicate the flow of data.
Tiling and fusion pros and cons

Scheduling is often about finding a compromise halfway between root and inline
- good locality like inline
- limit redundancy like root

Tile size controls the tradeoff
Extra scheduling options

Reorder
e.g. for x for y => for y for x

Split
e.g. for x => for xo for xi
Tiling combines a set of splits and a reorder

Unroll

Vectorize

Parallelize

some CUDA-specific
Final 3x3 blur: add parallelism and vectorization

\[
x, y = \text{Var('x')}, \text{Var('y')}
\]
\[
\text{blur}_x, \text{blur}_y = \text{Func('blur}_x'), \text{Func('blur}_y')
\]
\[
\text{blur}_x[x,y] = (\text{input}[x,y]+\text{input}[x+1,y]+\text{input}[x+2,y])/3.0
\]
\[
\text{blur}_y[x,y] = (\text{blur}_x[x,y]+\text{blur}_x[x,y+1]+\text{blur}_x[x,y+2])/3.0
\]
\[
\text{xi, yi} = \text{Var('xi')}, \text{Var('yi')}
\]
\[
\text{blur}_y\text{.tile}(x, y, \text{xi}, \text{yi}, 8, 4) \\ 
\quad .\text{parallel}(y) \ 
\quad .\text{vectorize}(\text{xi}, 8)
\]
\[
\text{blur}_x\text{.compute_at} (\text{blur}_y, x) \\ 
\quad .\text{vectorize}(x, 8)
\]
\[
\text{output} = \text{blur}_y\text{.realize}(\text{input}.\text{width}()-2, \text{input}.\text{height}()-2)
\]
More general box blur, 5x5, 35 MPixels on 12 cores

default schedule
  took  5.80736255646 seconds
root first stage
  took  1.99803357124 seconds
tile  256 x 256  + interleave
  took  1.7552740097 seconds
tile  256 x 256  + parallel+vector
  took  0.550438785553 seconds
tile  256 x 256  + parallel+vector without interleaving
  took  1.10970659256 seconds

arithmetic intensity: low here

each core does 1 tile box right away
first tile blurry
all tiles of blur x
memory bound
Each core does 1 tile box right away tile blurry output interleaving all tiles of blur x first, then tiles of blurry here

\[
\text{dot product} \quad x_1, x_2, x_3, x_4, \ldots, y_1, y_2, y_3, y_4.
\]

1 mult, 1 add per pair of looks.
More general box blur, 5x5, 35 MPixels on 12 cores

- default schedule
  took 5.80736255646 seconds

- root first stage
  took 1.99803357124 seconds

- tile 256 x 256 + interleave
  took 1.7552740097 seconds

- tile 256 x 256 + parallel+vector
  took 0.550438785553 seconds

- tile 256 x 256 + parallel+vector without interleaving
  took 1.10970659256 seconds

Note the exact doubling (memory bound)
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[(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);
      }
    }
  }
}
```

11x faster (quad core x86)

Tiled, fused
Vectorized
Multithreaded
Redundant computation
Near roof-line optimum
Some of the benefits of Halide

Keeps algorithm clean and orthogonal to schedule

Systematic organization of scheduling/performance

Automatically does low-level stuff for you
indexing logic, including when the image is not divisible by the tile size
tile expansion inference
vectorization
translation to CUDA

Enables quick exploration of possible schedules
Recap

Scheduling is about generating nested loops

1/ Within stages

2/ Across stages: when is the producer computed with respect to the consumer

Compromise between root (no redundancy but bad locality) and inline (lots of redundancy, perfect locality)

Root, Inline, Tile + fusion

+ others (vectorize, parallelize)
Thresholding using select

\[
\text{input} = \text{Image(Float(32), im)}
\]

\[
\text{sel=Func()}
\]

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

\[
\text{sel}[x, y, c] = \text{select}(\text{input}[x,y,1]<0.5, 0.0, 1.0)
\]

\[
\text{output} = \text{sel}.\text{realize(input.width()), input.height()}, \text{input.channels()};
\]
Reductions

Very overloaded term
http://en.wikipedia.org/wiki/Reduction
Here, aka fold, accumulate, aggregate, ...
Same as reduce in map-reduce

Aggregates multiple values
cases where you would want to use a for loop

For image processing:
average of an image, max
histogram
convolution
max over windows
Sum in Python

```python
out = numpy.empty((3));
for c in xrange(input.channels()):
    out[c]=0.0
for ry in xrange(0, input.height()):
    for rx in xrange(0, input.width()):
        for c in xrange(input.channels()):
            out[c] += input[rx, ry, c]
```

- **Init**: `out` array is initialized with zeros.
- **Reduction**: Iterating over rows and columns, the value at each position is accumulated.
- **Update**: The accumulated value is stored in the `out` array.

This code snippet demonstrates how to sum the elements along the channels of a 3D array in Python using `numpy`.
Sum in Halide

```python
input = Image(Float(32), im)
x, y, c = Var(), Var(), Var()
mySum = Func()

r = RDom(0, input.width(), 0, input.height())
    equivalent to the extent of reduction loop

mySum[c]=0.0

mySum[c] += input[r.x, r.y, c]

output = mySum.realize(input.channels());
```
Sum in Python

```python
out = numpy.empty((3));
for c in xrange(input.channels()):
    out[c]=0.0
for ry in xrange(0, input.height()):
    for rx in xrange(0, input.width()):
        for c in nxrange(input.channels()):
            out[c] += input[rx, ry, c]
```

- **Init**: `out[c]=0.0`
- **Reduction Domain**: Input range
- **Reduction**: Accumulate values
- **Update**: `out[c] += input[rx, ry, c]`
Sum in Halide

```python
input = Image(Float(32), im)
x, y, c = Var(), Var(), Var()
mySum = Func()

r = RDom(0, input.width(), 0, input.height())

mySum[c] = 0.0

mySum[c] += input[r.x, r.y, c]

output = mySum.realize(input.channels());
```
Halide reductions

Loops are implicit !!

Reduction domain: all the location that will be aggregated similar to Python’s range/xrange for the loop you feel like writing

Multidimensional

RDom(baseX, extentX, baseY, extentY,...)

Initialize your Func

myFunc[Var, Var]=initialValue

Update equation

myFunc[Expr,Expr] = f ( myFunc[Expr,Expr], Rdom )

will be called for each RDom location

arbitrary Expr of the Func, its Var, and the RDom. The RDom can be on the left and right
Sum in Halide

```python
input = Image(Float(32), im)
x, y, c = Var(), Var(), Var()
mySum = Func()

r = RDom(0, input.width(), 0, input.height())

mySum[c]=0.0

mySum[c] += input[r.x, r.y, c]

output = mySum.realize(input.channels());
```
out = numpy.empty((3));
for c in xrange(input.channels()):
    out[c] = 0.0
for ry in xrange(0, input.height()):
    for rx in xrange(0, input.width()):
        for c in nxrange(input.channels()):
            out[c] += input[rx, ry, c]
Equivalent Sum in Python

```python
out = numpy.empty((3));
for c in xrange(input.channels()):
    out[c]=0.0
for ry in xrange(0, input.height()):
    for rx in xrange(0, input.width()):
        for c in nxrange(input.channels()):
            out[c] += input[rx, ry, c]

Note that the reduction loops are ALWAYS the outer loop
More about this soon
```
Equivalent Sum in Python

```python
out = numpy.empty((3));
for c in xrange(input.channels()):
    out[c] = 0.0
for ry in xrange(0, input.height()):
    for rx in xrange(0, input.width()):
        for c in nxrange(input.channels()):
            out[c] += input[rx, ry, c]
```

Note that the reduction loops are ALWAYS the outer loop
More about this soon
Convolution as reduction

```python
input = Image(Float(32), im)
blur=Func('blur')
x, y, c = Var('x'), Var('y'), Var('c')
clamped = Func('clamped')
clamped[x, y, c] = input[clamp(x, 0, input.width()-1),
                        clamp(y, 0, input.height()-1), c]

r = RDom(0, kernel_width, 0, kernel_width, 'r')

blur[x,y,c] = 0.0

blur[x,y,c] += clamped[x+rx.x-kernel_width/2, y, c]
```

Note

Again, no loop!

Here, reduction over 2 of the 3 dimensions
Equivalent Python

```python
out = numpy.empty([input.width(), input.height(), input.channels()])

for y in xrange(input.height()):
    for x in xrange(input.width()):
        for c in xrange(input.channels):
            out[x, y, c] = 0

for ry in xrange(kernel_width):
    for rx in xrange(kernel_width):
        for y in xrange(input.height()):
            for x in xrange(input.width()):
                for c in xrange(input.channels):
                    out[x, y, c] += clampedInput[x, y, c]
```
Problem, bad order

```
for ry in xrange(kernel_width):
    for rx in xrange(kernel_width):
        for y in xrange(input.height()):
            for x in xrange(input.width()):
                for c in xrange(input.channels):
                    out[x,y,c]+=clampedInput[x,y,c]
```

The reduction loops are always outside
Because otherwise the update semantics might be changed

Can result in very bad locality for convolution

Cannot be reordered directly (semantics should not change)
But there is a trick
The helper/inline trick

```python
input = Image(Float(32), im)
blur=Func('blur')
x, y, c = Var('x'), Var('y'), Var('c')
clamped = Func('clamped')
clamped[x, y, c] = input[clamp(x, 0, input.width()-1),
                        clamp(y, 0, input.height()-1), c]
r = RDom(0, kernel_width, 0, kernel_width, 'r')
blur[x,y,c] = 0.0
blur[x,y,c] += clamped[x+rx.x-kernel_width/2, y, c]

superBlur=Func('superBlur')
superBlur[x,y,c]=blur[x,y,c]
```

superBlur will be inlined
Python equivalent

```python
superBlur = numpy.empty([input.width(), input.height(), input.channels()])
for y in xrange(input.height()):
    for x in xrange(input.width()):
        for c in xrange(input.channels):
            # inline blur here
            values y, x, c are fixed
            105, 35, 2
            copy-paste the code for blur replacing y, x, c by their fixed value
```
superBlur=numpy.empty([input.width(), input.height(), input.channels()])

for y in xrange(input.height()):
    for x in xrange(input.width()):
        for c in xrange(input.channels):
            tmp=numpy.empty([1,1,1])
            for yi in xrange(1):
                for xi in xrange(1):
                    for ci in xrange(1):
                        tmp[xi,yi,ci]=0
            for ry in xrange(kernel_width):
                for rx in xrange(kernel_width):
                    for yi in xrange(1):
                        for xi in xrange(1):
                            for ci in xrange(1):
                                tmp[xi,yi,ci]+=clampedInput[x,y,c]

superBlur[x,y,c]=tmp[0,0,0]
Python equivalent without 1-iteration loops

```python
superBlur = numpy.empty([input.width(), input.height(), input.channels()])
for y in xrange(input.height()):
    for x in xrange(input.width()):
        for c in xrange(input.channels):
            tmp = 0
            for ry in xrange(kernel_width):
                for rx in xrange(kernel_width):
                    tmp += clampedInput[x, y, c]
            superBlur[x, y, c] = tmp
```
Rule of thumb

For stencil reduction
i.e. each output pixel is a reduction over multiple input pixels

Use the helper/inline trick

Encapsulate into an extra Func scheduled as default (inline)

In general, true for reduction where the free variables are independent of the reduction variables.
Sugar: sum (includes helper/inline trick)

input = Image(Float(32), im)
blur=Func('blur')
x, y, c = Var('x'), Var('y'), Var('c')
clamped = Func('clamped')
clamped[x, y, c] = input[clamp(x, 0, input.width()-1),
                           clamp(y, 0, input.height()-1), c]
r = RDom(0, kernel_width, 0, kernel_width, 'r')
   \[\text{reduction domain}\]
blur[x,y,c] = sum(clamped[x+rx.x-kernel_width/2, y, c])
   \[\text{Expr}\]

Under the hood, creates a helper/inline
More complex reductions

**Histograms**
Reduction over
Update equation:

**Bilateral grid**
Reduction over
Update equation: