High-Performance Image Processing

Frédo Durand
most slides by Jonathan Ragan-Kelley
MIT CSAIL

Always remember

No for loop in Halide

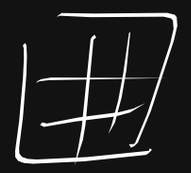
no if either (but there is a select)

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

tricky

don't worry about it in psets.

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://people.csail.mit.edu/jrk/halide-pldi13.pdf

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)

```
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

```
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)

```
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

```
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
```

produce

consumer

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.

```
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
```

Inline schedule equivalent Python

```
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
```

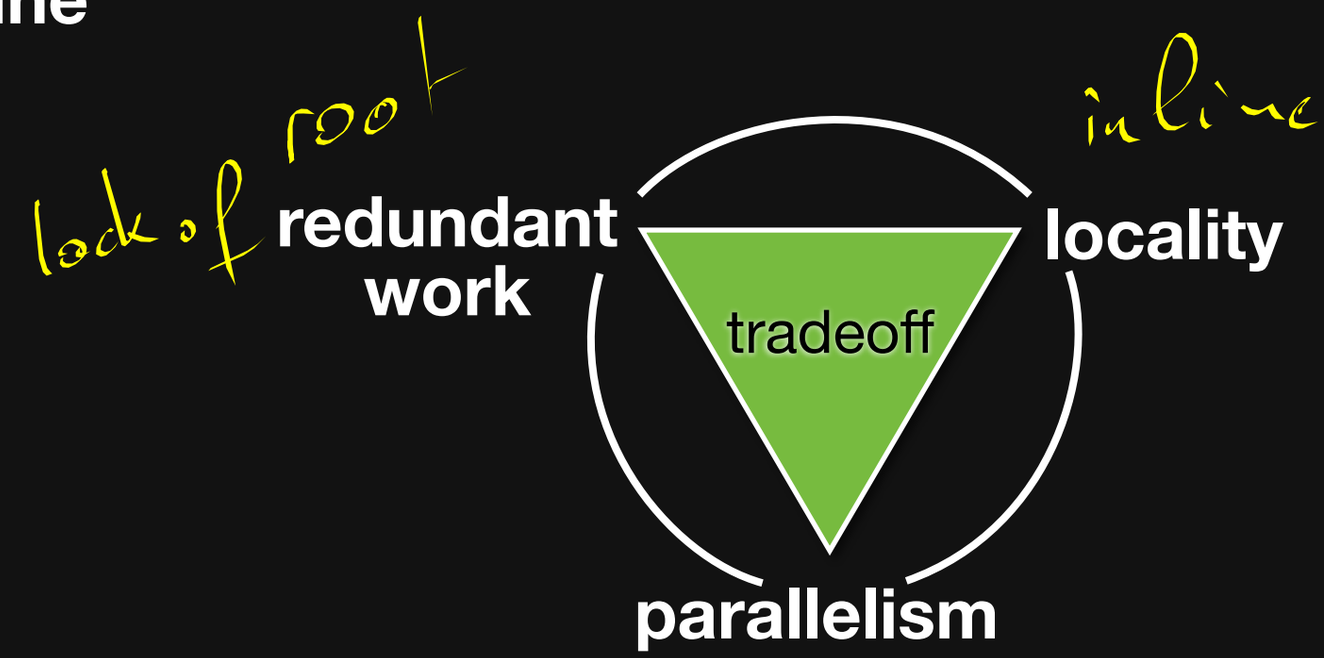
producer is computed right when needed

Root vs. Inline

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

good locality like inline

limit redundancy like root



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)

```
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)
blur_x.compute_at(blur_y, xo)
```

tile consumer
compute producer at
tile granularity

Tiling and Fusion: Python equivalent

```
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

indexing code affected by schedule in green

```
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)
```

tile

produce

consume

Halide does this automatically

Schedule visualization

blur$_y$

root

x_o

x_i

compute and
store at x_o of blur$_2$

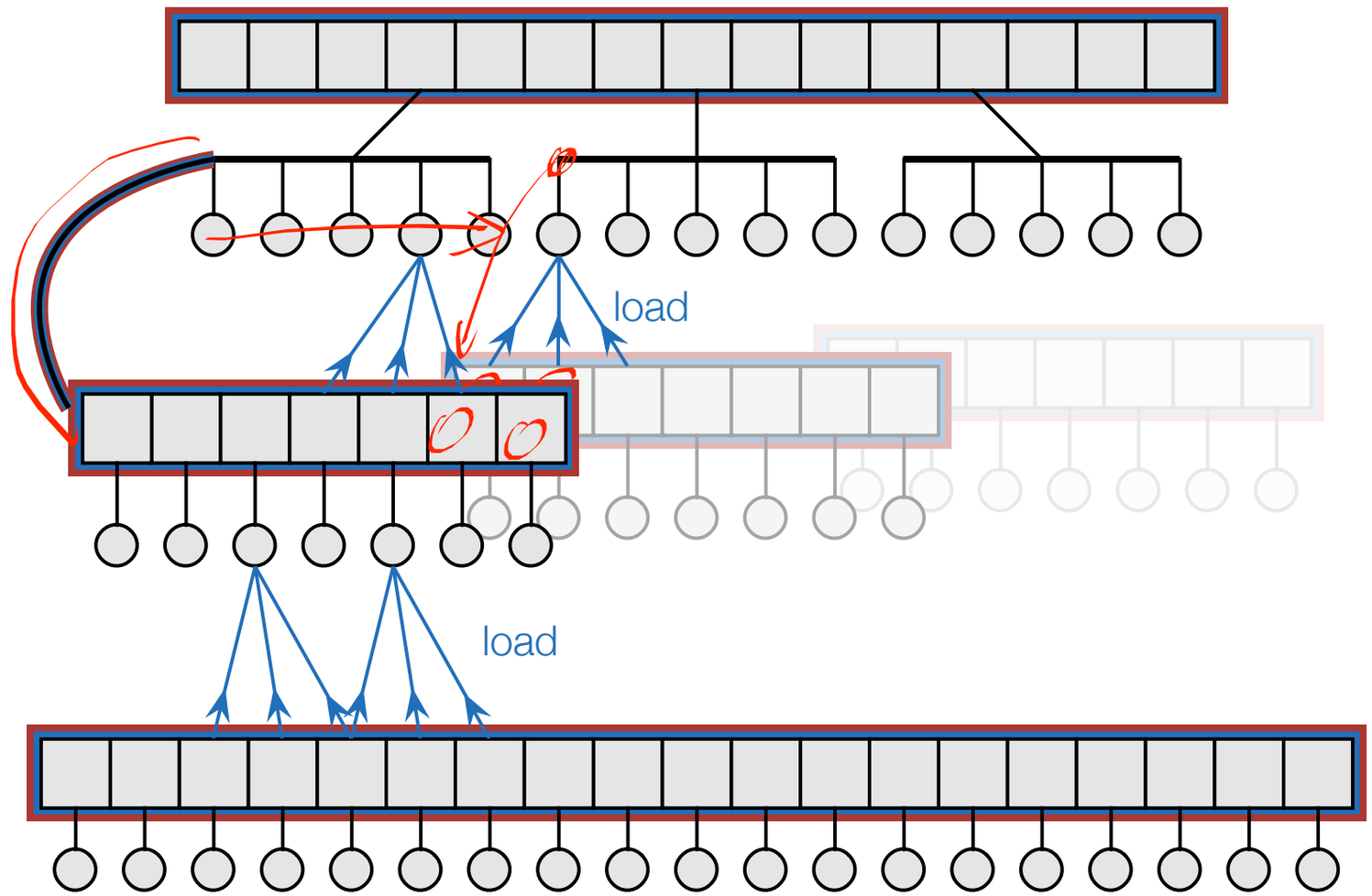
x

blur$_x$

compute and
store at root

x

input

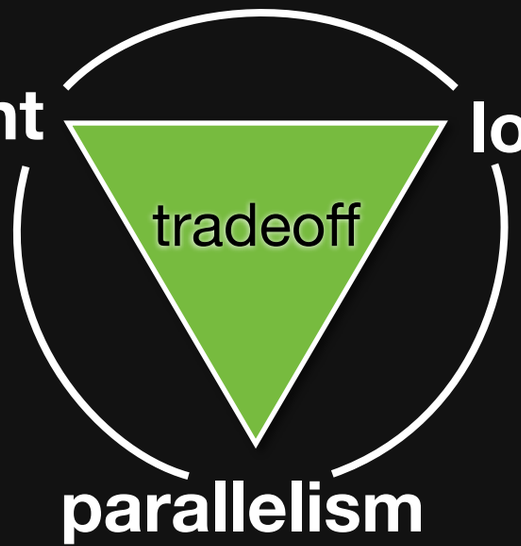


Tiling and ~~fusion~~ pros and cons

interleaving

big tiles ← root is best

redundant work



locality

inline is best

small tiles

tile size affect

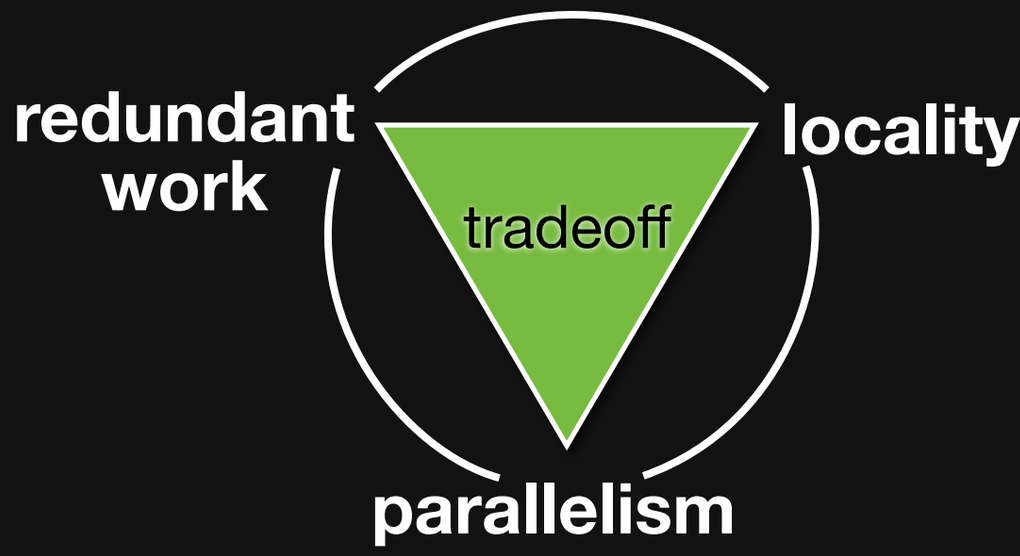
- redundancy length vs area
- locality if tile is bigger than cache: trouble

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

tile = split + reorder

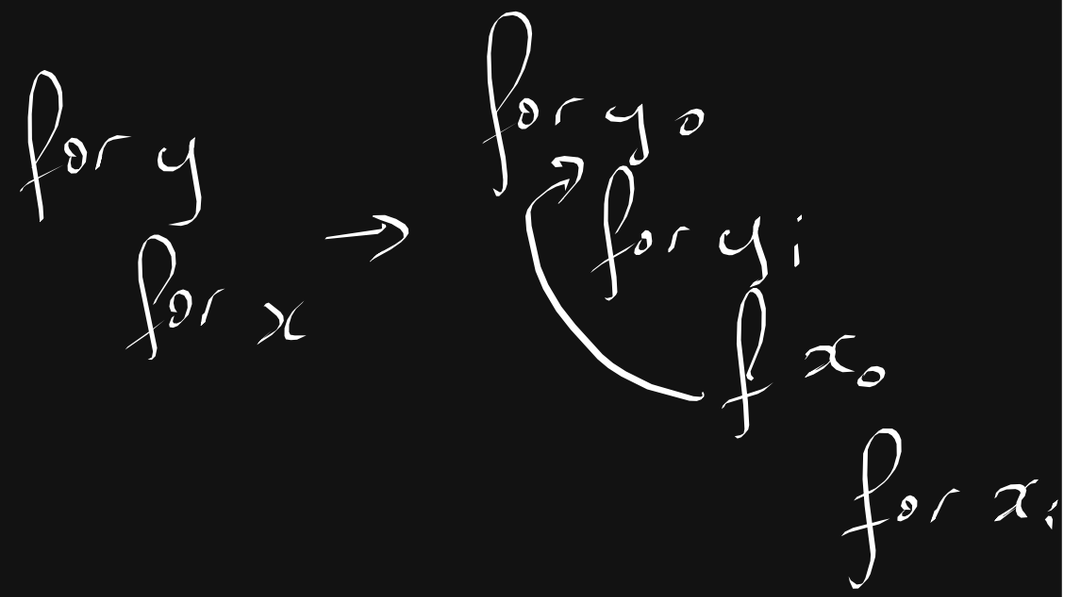
Reorder

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-specifc

Final 3x3 blur: add parallelism and vectorization

```
x, y = Var('x'), Var('y')
blur_x, blur_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

xi, yi = Var('xi'), Var('yi')
blur_y.tile(x, y, xi, yi, 8, 4) \
      .parallel(y) \
      .vectorize(xi, 8)
blur_x.compute_at(blur_y, x) \
      .vectorize(x, 8)

output=blur_y.realize(input.width()-2, input.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

blur$_y$

root

x_o

x_i

compute and
store at x_o of blur$_2$

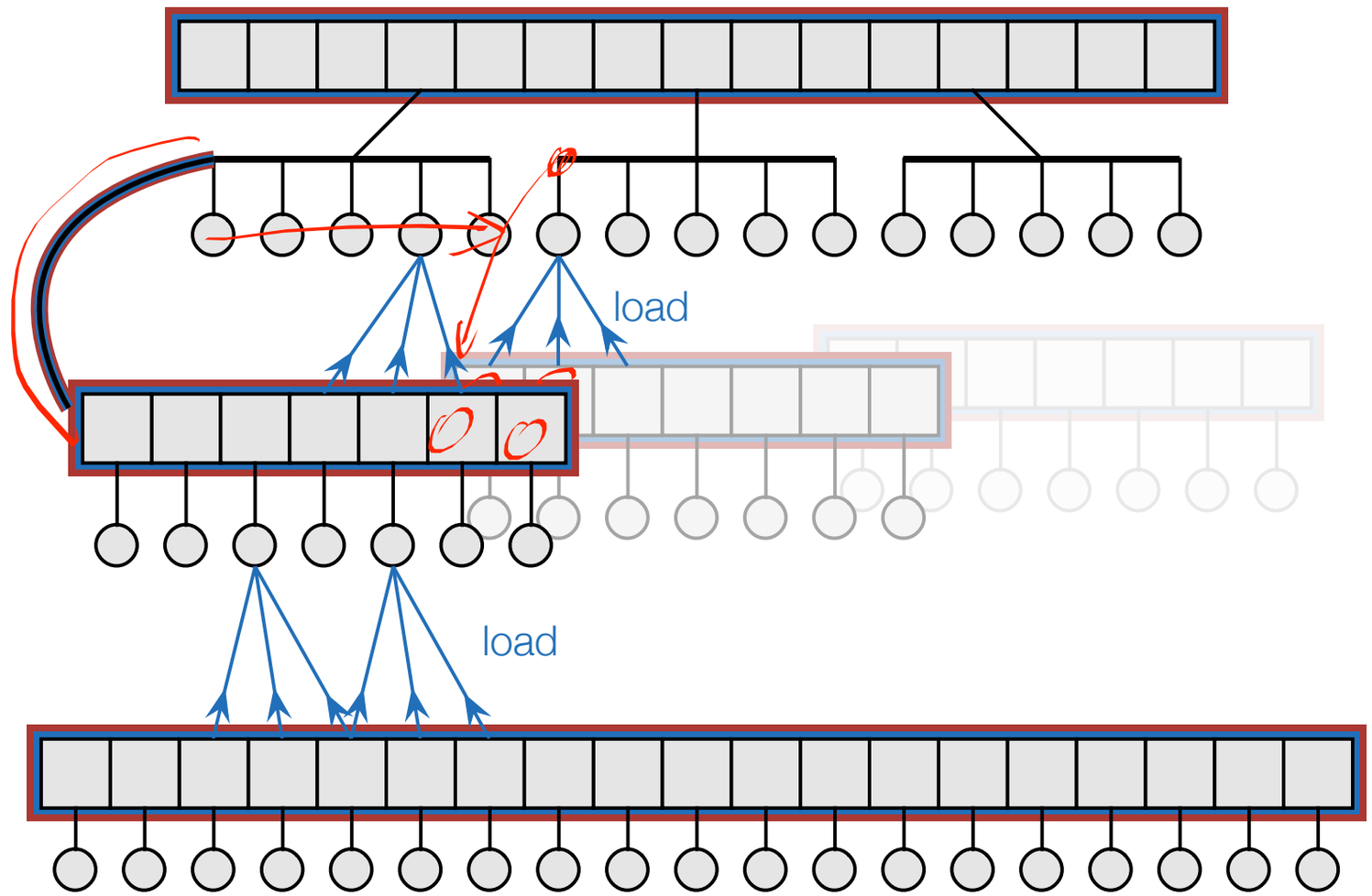
x

blur$_x$

compute and
store at root

x

input



Tiling and Fusion: Python equivalent

indexing code affected by schedule in green

```
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$_y$

root

x_o

x_i

compute and
store at x_o of blur$_2$

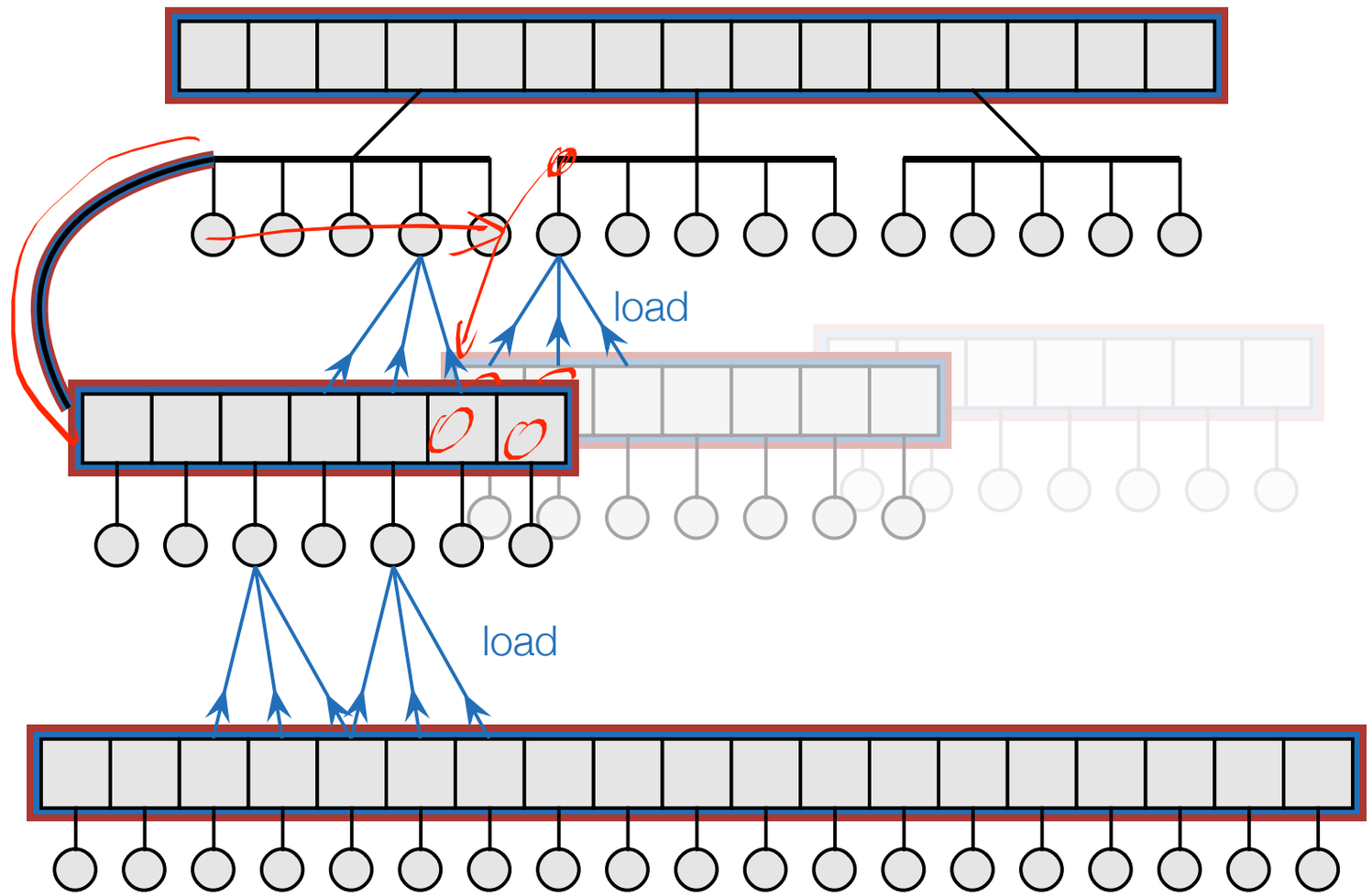
x

blur$_x$

compute and
store at root

x

input

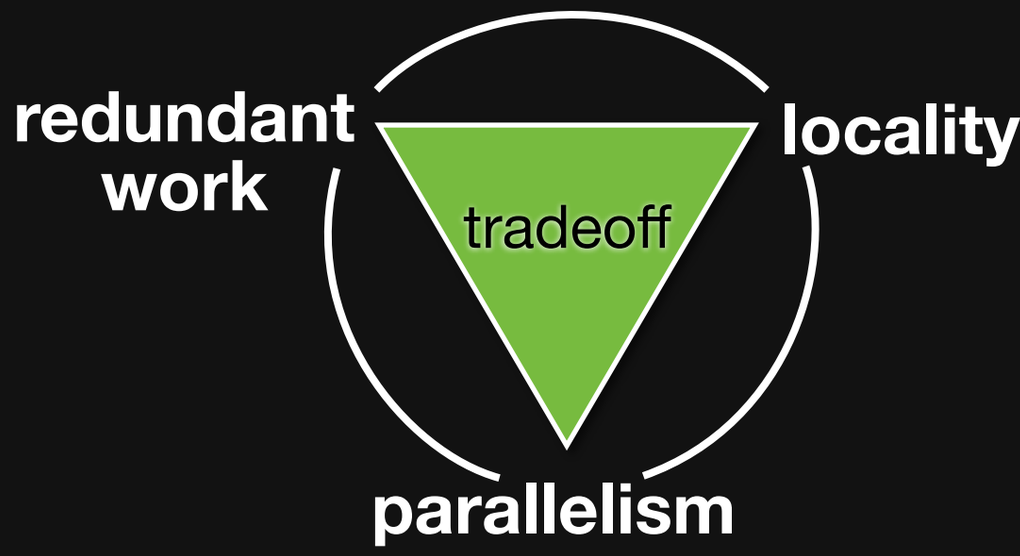


Tiling and fusion pros and cons

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

tile = split + reorder

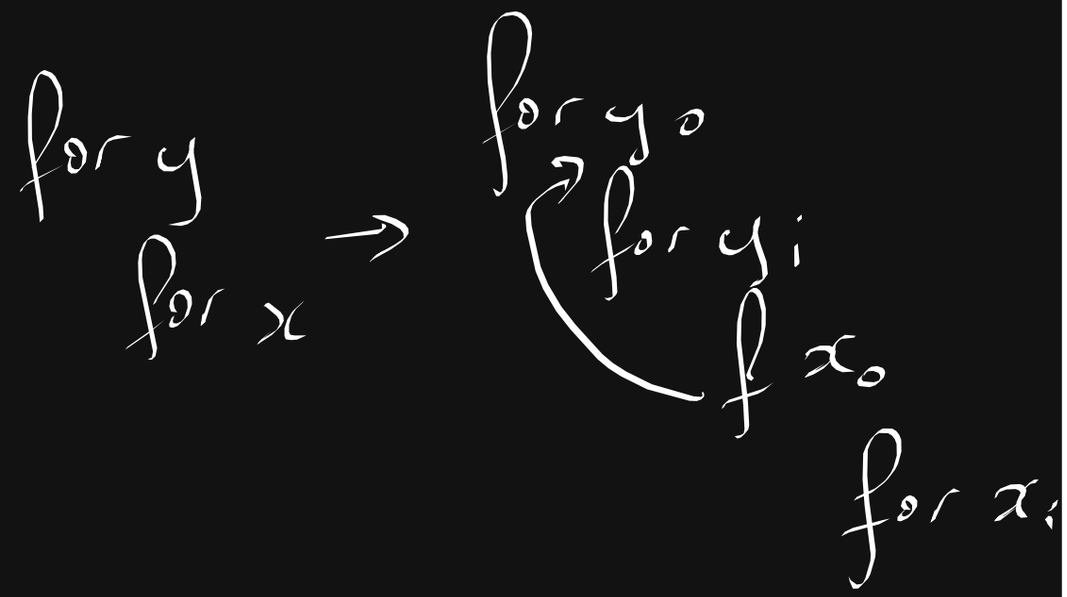
Reorder

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-specifc

Final 3x3 blur: add parallelism and vectorization

```
x, y = Var('x'), Var('y')
blur_x, blur_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

xi, yi = Var('xi'), Var('yi')
blur_y.tile(x, y, xi, yi, 8, 4) \
      .parallel(y) \
      .vectorize(xi, 8)
blur_x.compute_at(blur_y, x) \
      .vectorize(x, 8)

output=blur_y.realize(input.width()-2, input.height()-2)
```

More general box blur, 5x5, 35 MPixels on 12 cores

default schedule
took 5.80736255646 seconds

root first stage (blur x)
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

each core does 1 tile box right away tile blury

all tiles of blur x first, then tiles of blury

2x → memory bound

arithmetic intensity : low here

Pixels on 12 cores

each core does 1 tile by right away tile blurx

ut interleaving

all tiles of blurx first, then tiles of blury

here

dot product

x_1
x_2
x_3
x_4
$\vdots$

y_1
y_2
y_3
y_4
$\vdots$

1 mult, 1 add per pair of float

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

```
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);
    }}}}}
```

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

```
input = Image(Float(32), im)

sel=Func()

x, y, c = Var(), Var(), Var()

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

output = sel.realize(input.width(), input.height(),
                     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

```
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

reduction

update

Sum in Halide

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

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

usual

equivalent to the extent of reduction loops

```
mySum[c]=0.0


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


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

Sum in 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]
```

Sum in Halide

```
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());
```

usual

equivalent to the extent of reduction loops

init

update

reduction variables

free variable

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,...)

Initialilize 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

```
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());
```

Equivalent Sum in 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]
```

free variable

outer loop always reduction loops.

Equivalent Sum in 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

```
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

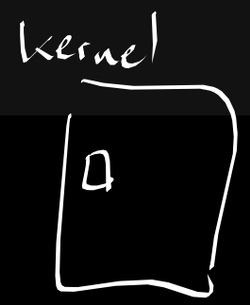
Convolution as reduction

```
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]
```



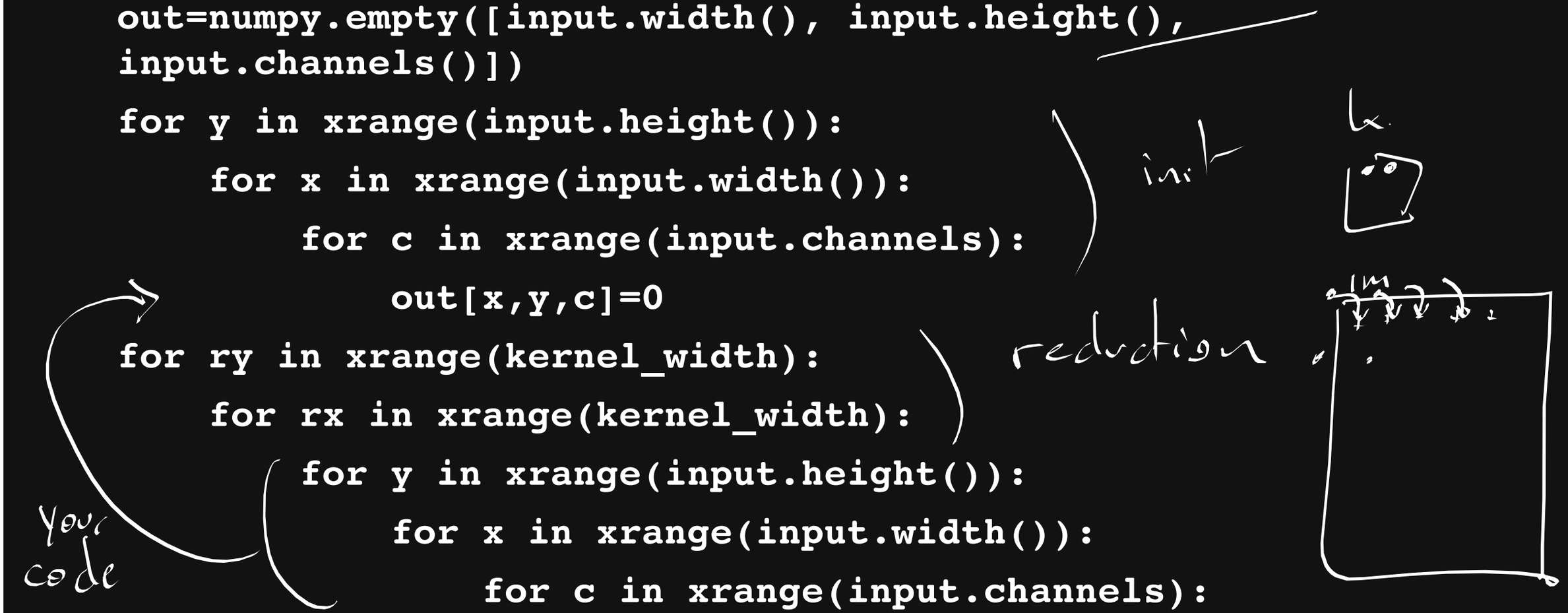
Note

Again, no loop!

Here, reduction over 2 of the 3 dimensions

Equivalent Python

```
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

```
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]
```

identity

blur will be inlined

Python equivalent

```
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 ther fixed value

Python equivalent

```
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]
```

I know x, y, c

loop size 1

size 1

size 1

1 value

reduction loop

free variable

loop size 1

Python equivalent without 1-iteration loops

```
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):
```

free variable

```
            tmp=0
            for ry in xrange(kernel_width):
                for rx in xrange(kernel_width):


                    tmp+=clampedInput[x,y,c]
            superBlur[x,y,c]=tmp
```

reduction loops

init

update

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

reduction domain

Expr

Under the hood, creates a helper/inline

More complex reductions

Histograms

Reduction over

Update equation:

RDom

Histogram[intensity]

for y

for x

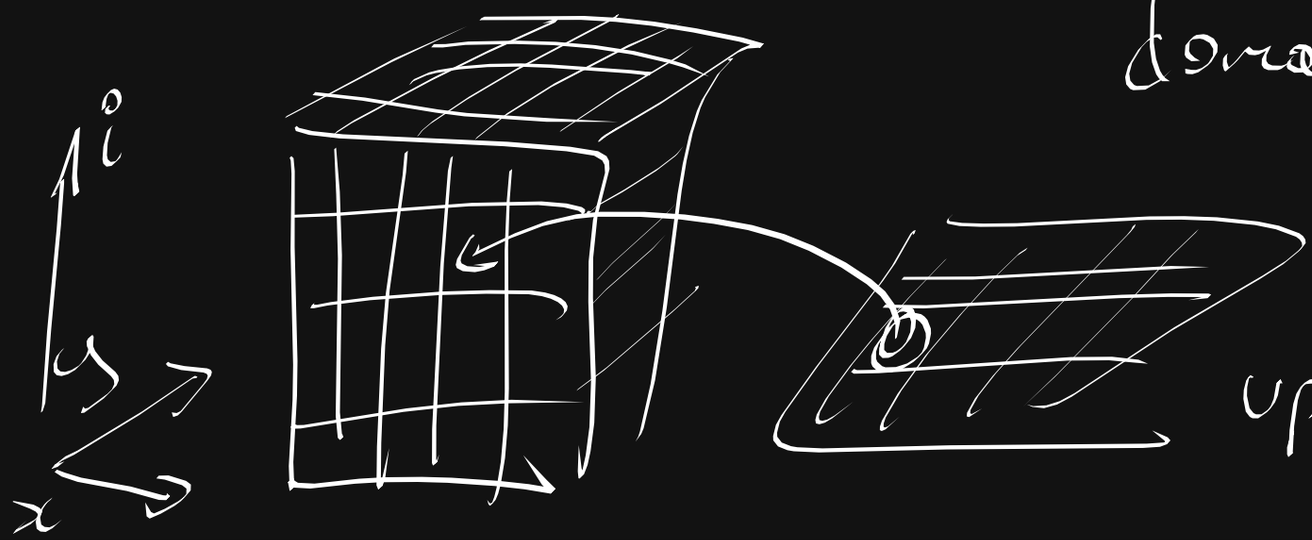
hist[im[x,y]] +=

bin[intensity[rx, ry]] += 1

Bilateral grid

Reduction over

Update equation:



domain: x y

update = grid[rx,ry,i[rx,ry]]
+= i[rx,ry]