High-Performance Image Processing

Frédo Durand
most slides by Jonathan Ragan-Kelley
MIT CSAIL

Lytro

http://www.bloomberg.com/news/2013-11-20/lytro-raises-40-million-after-leadership-change-at-camera-maker.html

Lytro Raises $40 Million After Leadership Change at Camera Maker

By Adam Satariano - Nov 20, 2013 9:00 AM ET

0 COMMENTS + QUEUE

Lytro Inc., the maker of a camera that lets pictures be refocused after they are taken, pulled in an additional $40 million in funding to release a redesigned device and get its technology into smartphones.

Halide in the real world

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/

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

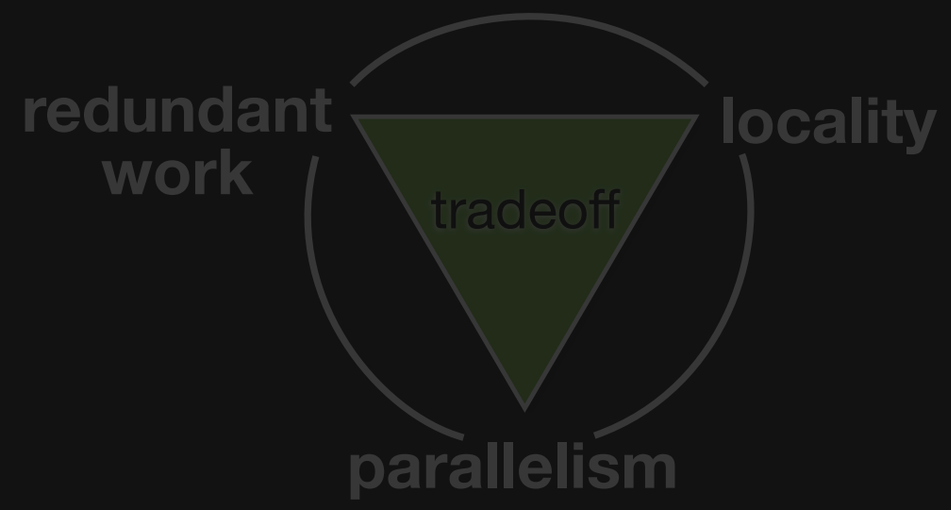
Vectorized

Multithreaded

Redundant computation

Near roof-line optimum

Where does performance come from?

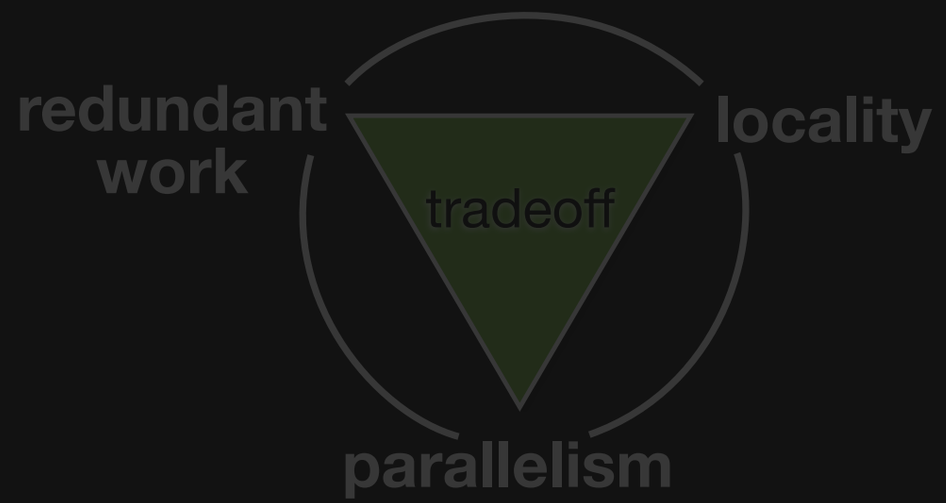


Message #2: organization of computation
is a first-class issue

Program:

Program

Hardware



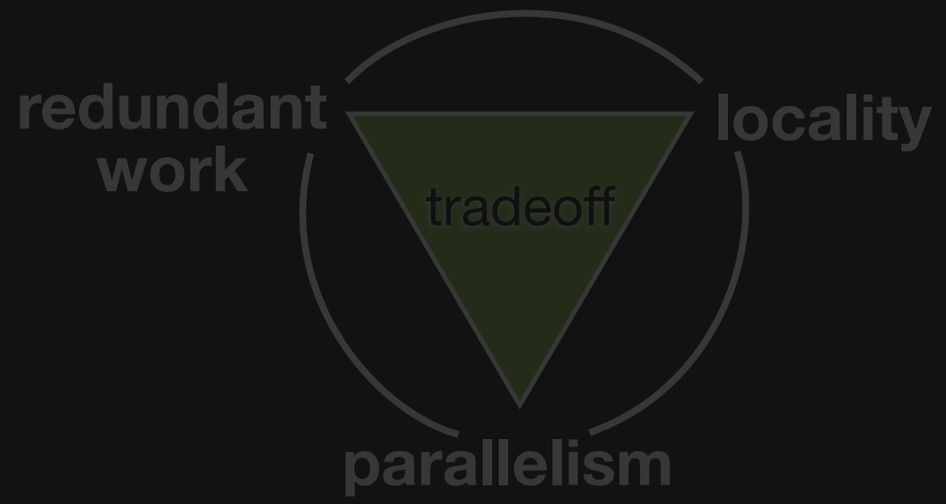
Message #2: organization of computation
is a first-class issue

Program:

Algorithm

Organization of computation

Hardware



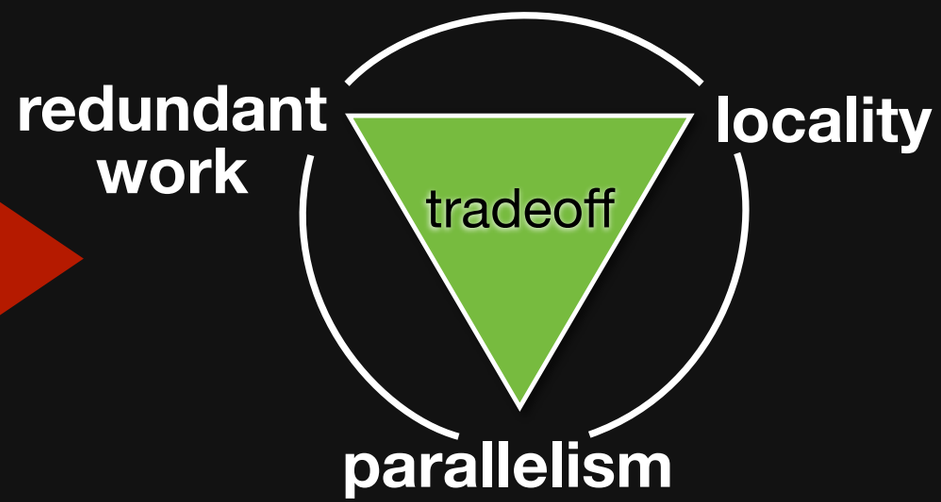
Message #2: organization of computation
is a first-class issue

Program:

Algorithm

Organization of computation

Hardware



Halide *algorithm*:

```
blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
blury[x, y] = (blurx[x, y-1] + blurx[x, y] + blurx[x, y+1])/3;
```

Halide *schedule*:

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

Halide good news and bad news

Halide syntax is easy

Halide is cross platform (Linux, Mac, Windows)

Although some better than others

Supports many targets (x86, GPU, ARM)

Halide generates really fast executables

Halide is easy to embed in Python and C++

Halide is not Turing complete and cannot represent everything

e.g. the iteration loop of conjugate gradient would have to be in the host language, not Halide

Halide doesn’t always give useful error messages

And IDLE hides them from you. Don’t use IDLE

Today

Halide syntax for algorithms

Halide schedule

Performance and reorganization of computation

Halide is a purely Functional language

No side effect, no environments!

Define your algorithm as a pipeline of pure functions

Loops are NOT specified in the Halide algorithm

That’s the job of the schedule

Halide is embedded in C++ or Python

Called Metaprogramming

We'll construct and use Halide programs inside Python code

Under the hood:

Our Python code creates Python objects representing the Halide program (Abstract Syntax Tree, etc.)

We call the Halide compiler on these objects to compile and execute

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)
```

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)
```

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] + 3.14 * blur_x[2*x+1, y] + cos(2*x)

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

from halide import *

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)
```

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

Under the hood

Func, Var, Expr, Image are Python classes representing Halide programs

and under the first hood, there is another hood hiding C++ classes ;-)

We declare Halide Funcs, Vars, etc. by calling the Python constructor of these classes

We manipulate Halide programs (and in particular their schedules) by calling methods of the Func class

Halide Box Blur (Python embedding)

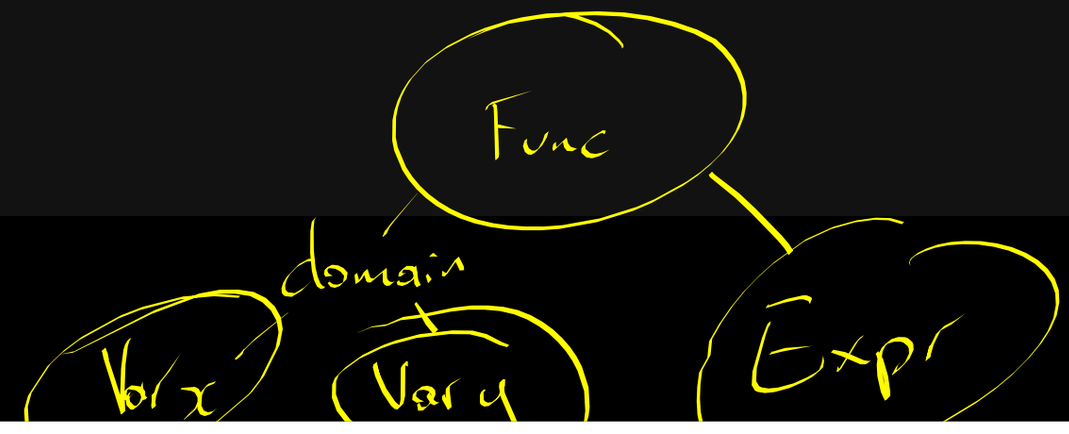
```
inputP=imageIO.imread('rgb.png')[:,:,1]
input=Image(Float(32), inputP)

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)
```

Var constructor
it binds the python variable x
with a Halide notion of variable
x = Var('x')

overloaded

overloaded

Expr



```
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)
```

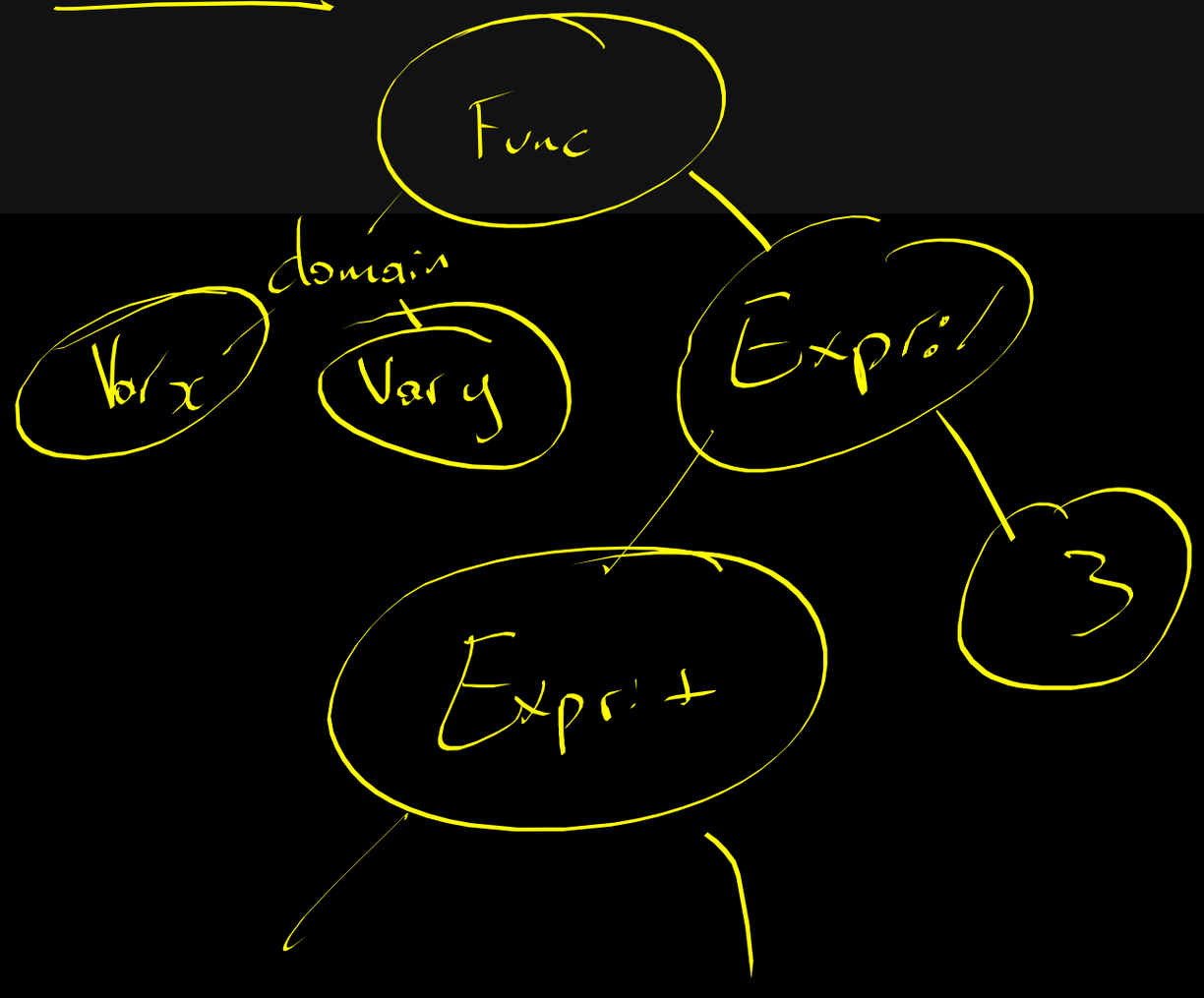
with a 'Halide notion of variable'

x = Var('x')

overloaded

overloaded

Expr



Realize

Func.realize calls the Halide compiler on the Python representation

Lots of magic happens behind the scene

Will depend on the schedule

More later

then executes your code and returns a Halide Image

Always call on last stage

unless you’re debugging earlier stages

Quirks

Declare variables and functions before defining or using them.

e.g.:

```
blur_x = Func() #declare the horizontal blur function
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3 #define it
```

Use brackets for functions

The Halide C++ syntax is slightly different from the Python one

Because different constraints of the host languages affect the ease of metaprogramming

The last part of the slides use the C++ syntax. Sorry

Reductions are a little more complicated

e.g. sums, histograms, general convolutions

More later

More quirk

Halide functions in Python tend to override standard functions (e.g. max, min...)

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)
```

Why Embedded / Metaprogramming?

Makes it easy to embed in an existing language and codebase

Avoids the need to parse

Enables cool tricks

e.g. use a Python for loop to programmatically generate a complex Halide program

that’s the ‘meta’ part

It’ll blow your mind

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)
```

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)
```

Swept under the rug so far

Reductions
sum, histogram, general convolution

Types and precision
floats, 16 bits, 8 bits, etc.
It’s annoying to work with 8-bit integers because of overflow

Clamping to image boundary
like black/edge padding. easy.

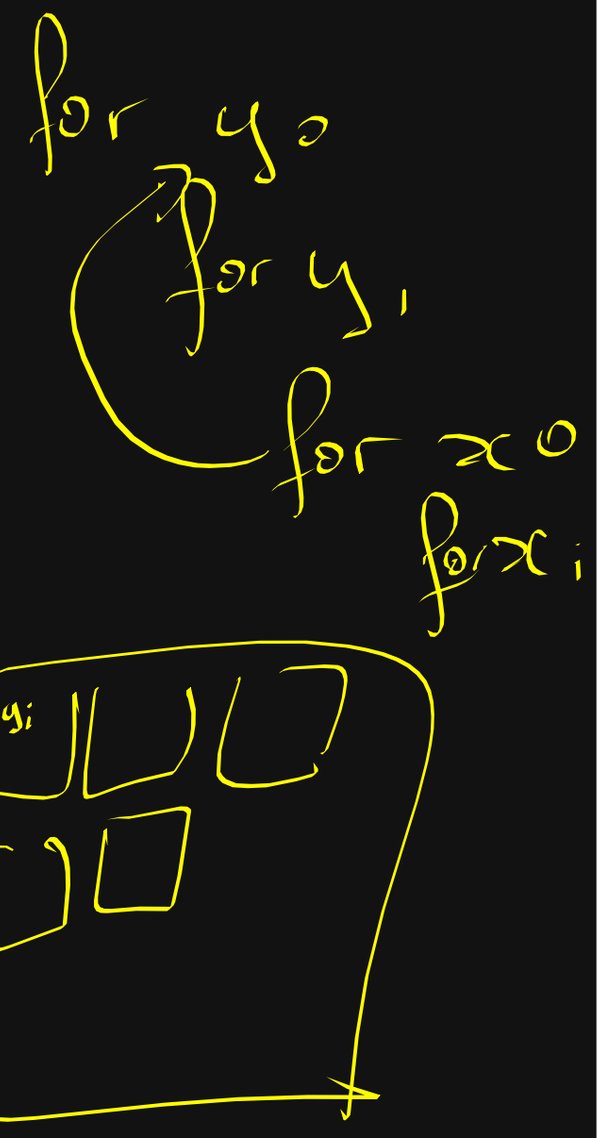
Schedule

Given an algorithm given as a pipeline of Func

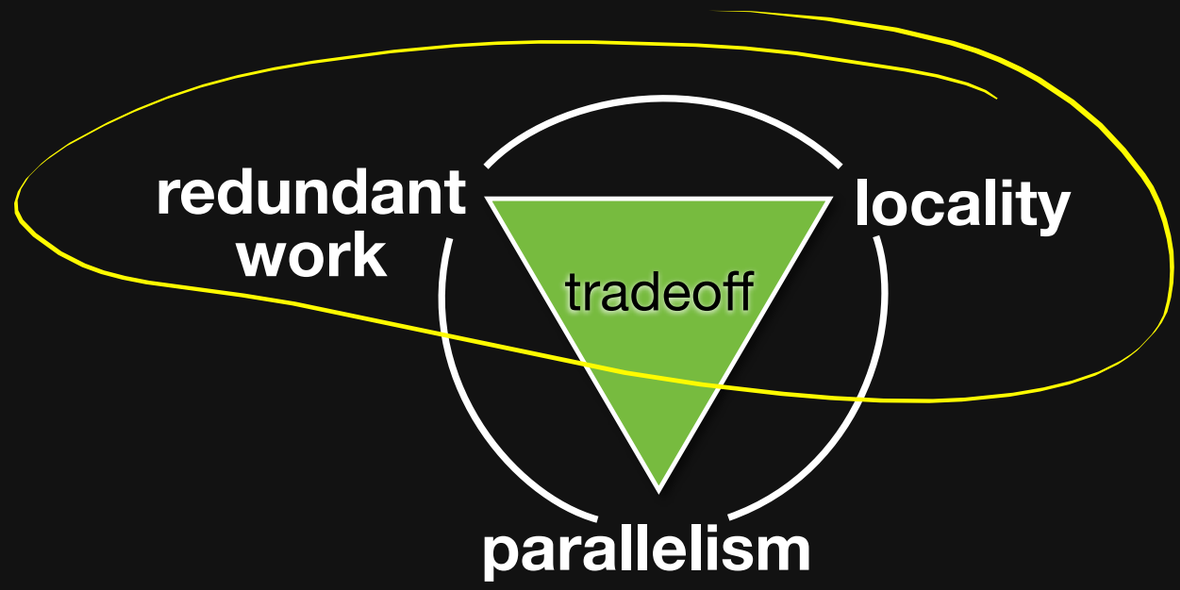
Specifies the order of computation
within each Func
across Func (the more interesting part)

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()
```



Scheduling seeks to optimize tradeoffs



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()
```

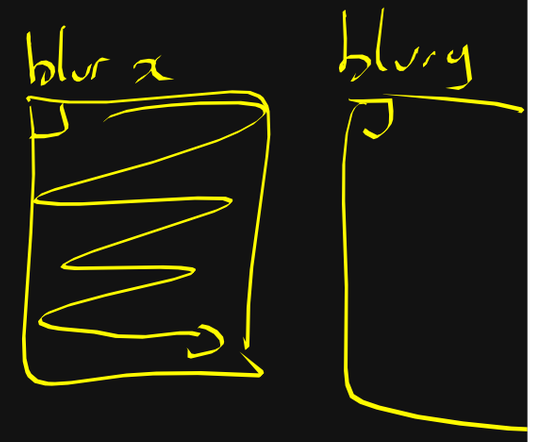
Root schedule equivalent Python

```
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
```

blur_x

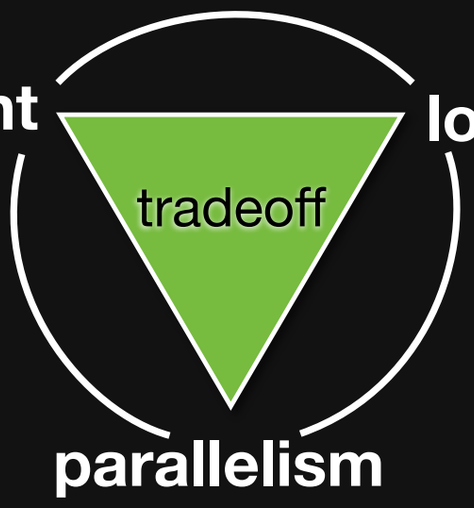
blur_y

Root pross and cons



A+

redundant work



locality

E-

parallelism

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()

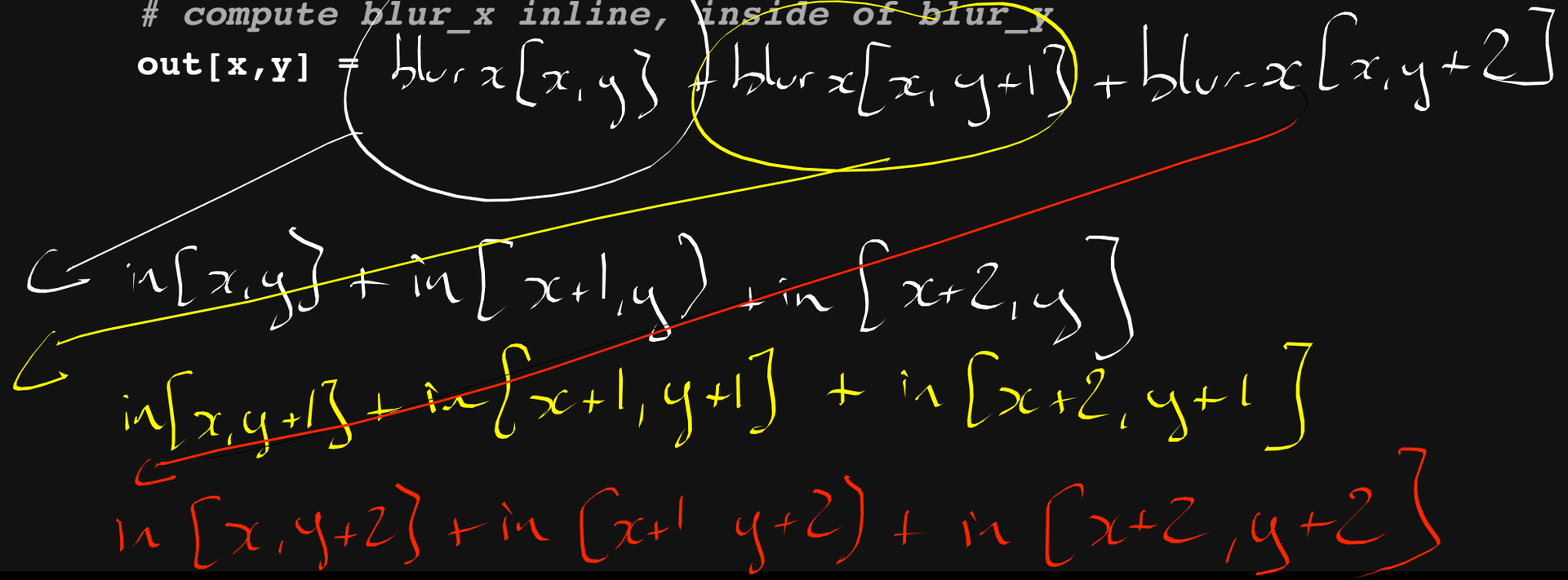
Default schedule (except output)

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] = blurx[x,y] + blurx[x,y+1] + blurx[x,y+2]
```

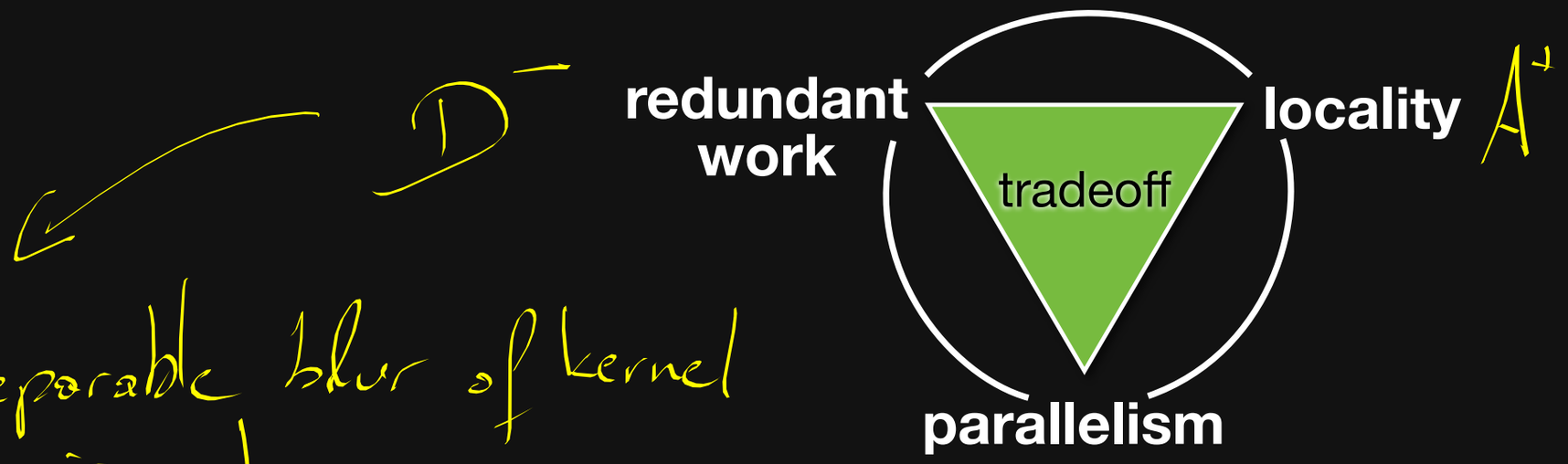


Inline schedule equivalent Python

```
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
```

turns out to be brute-force 2D blur

Inline pros and cons

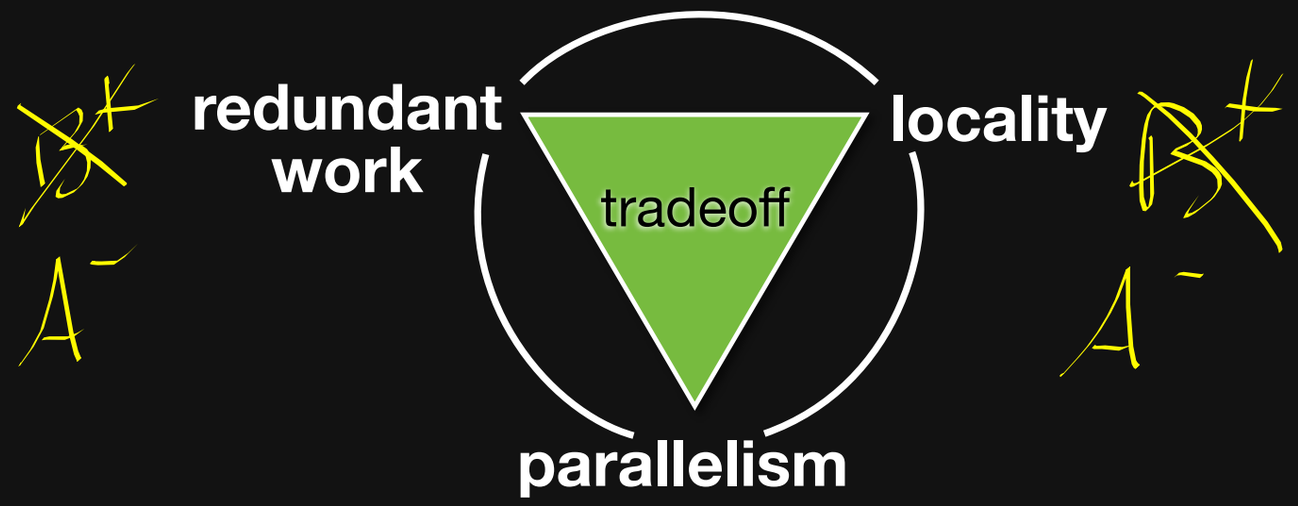


Root vs. Inline

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

good locality like inline

limit redundancy like root



Tiling (within stage)

Split the domain into inner and outer loops

Specified with Func.tile()
requires new Vars for inner and outer indices

Let's look at a single-stage pipeline function that only blurs vertically

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

xo, yo, xi, yi = Var(), Var(), Var(), Var()
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32)
```

optional

Tiling: Python equivalent

```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange(height/32): #assuming height%32=0
    for xo in xrange(width/256): #assuming width%256=0
        for yi in xrange(32):
            y=yo*32+yi
            for xi in xrange(256):
                x=xo*256+xi
                out[x,y] = (input[x,y]+input[x,y+1]+input[x,y+2])/3
```

tile index

original indices

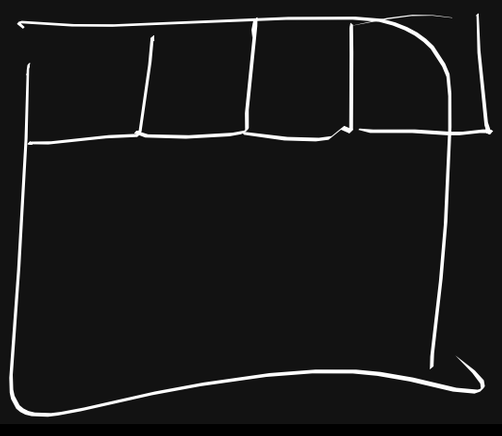
within tile

Tiling: Python equivalent: non-integer multiple

```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange((height+31)/32): #+31 to get the ceiling int
   for xo in xrange((width+255)/256):#+255 to get the ceiling int

      for yi in xrange(32):
         y=yo*32+yi
         # make sure y doesn't go beyond height
         # worst case we'll recompute the last rows multiple times
         if y>=height: y=height-1
         for xi in xrange(256):
            x=xo*256+xi
            # make sure x doesn't go beyond width
            # worst case we'll recompute the last columns
            if x>=width: x=width-1
            out[x,y] = (input[x,y]+input[x,y+1]+input[x,y+2])/3
```

Tiling: Python equivalent

```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange(height/32): #assuming height%32=0
    for xo in xrange(width/256): #assuming width%256=0
        for yi in xrange(32):
            y=yo*32+yi
            for xi in xrange(256):
                x=xo*256+xi
                out[x,y] = (input[x,y]+input[x,y+1]+input[x,y+2])/3
```



Tiling: Python equivalent: non-integer multiple

```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange((height+31)/32): #+31 to get the ceiling int
   for xo in xrange((width+255)/256):#+255 to get the ceiling int

      for yi in xrange(32):
         y=yo*32+yi
         # make sure y doesn't go beyond height
         # worst case we'll recompute the last rows multiple times
         if y>=height: y=height-1
         for xi in xrange(256):
            x=xo*256+xi
            # make sure x doesn't go beyond width
            # worst case we'll recompute the last columns
            if x>=width: x=width-1
            out[x,y] = (input[x,y]+input[x,y+1]+input[x,y+2])/3
```

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

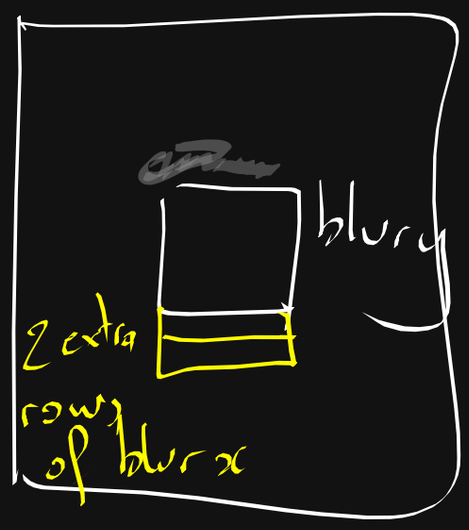
Halfway between root and inline:
root within tile: produce full tile before consuming it
inline across tile: compute a tile when it is needed

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)
```

consumer: tile
producer: at tile granu

Tiling and Fusion: Python equivalent

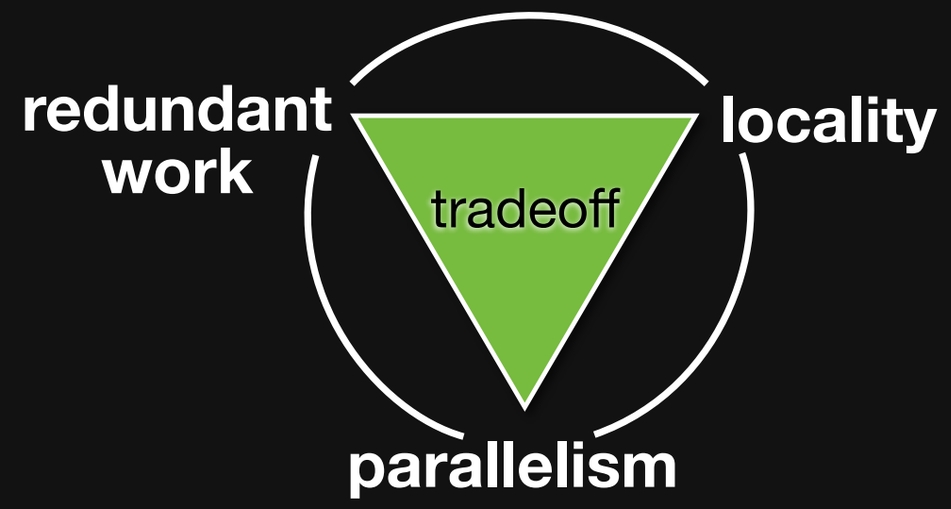
```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange((height+31)/32):
    for xo in xrange((width+255)/256):
        tmp=numpy.empty((256, 32+2))
        for yi in xrange(32+2):
            y=yo*32+yi
            if y>=height: y=height-1
            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)
```

Tiling and fusion pros and cons



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)
```

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)
```

consumer: tile
producer: at tile granu

Tiling and Fusion: Python equivalent

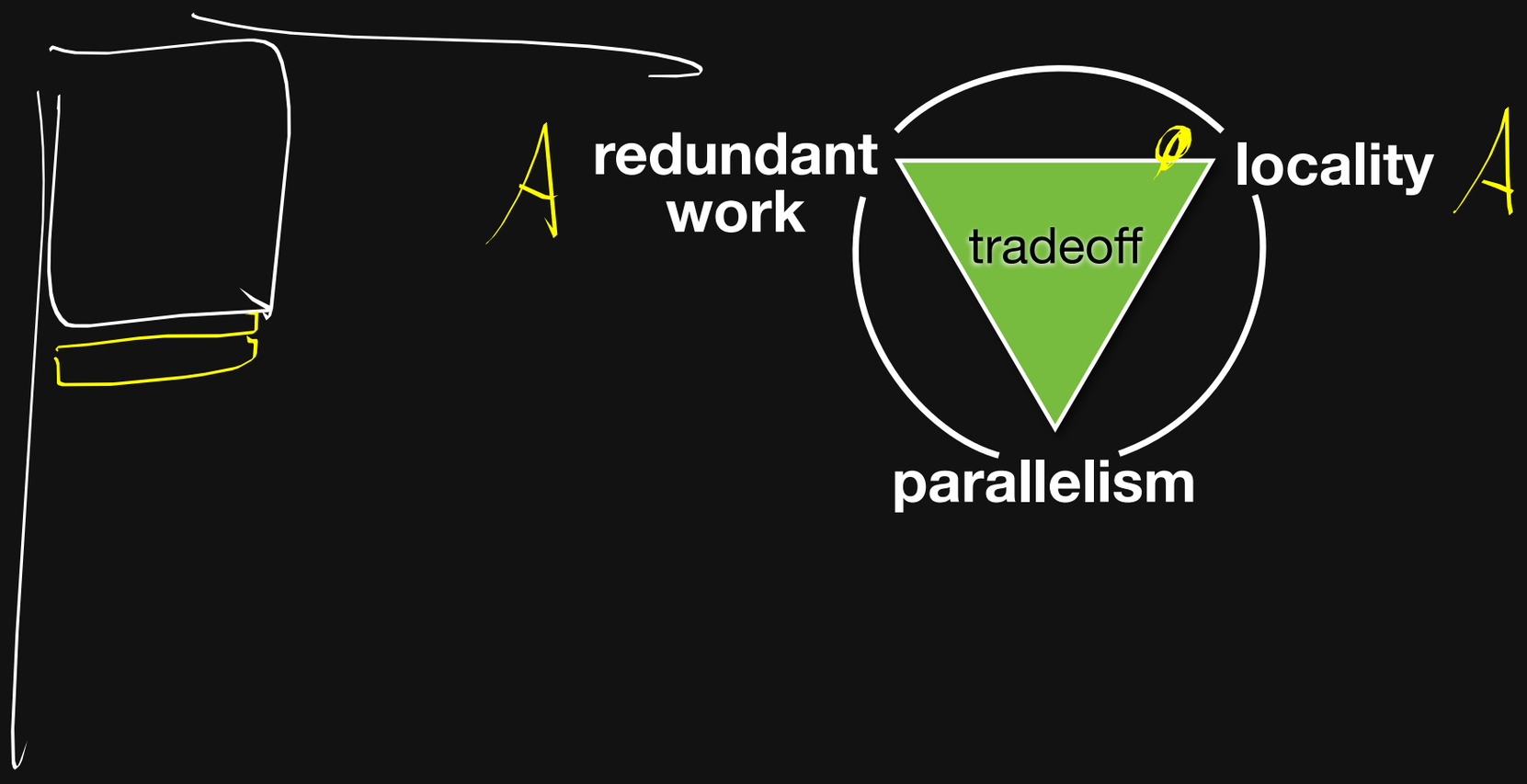
```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in xrange((height+31)/32):
    for xo in xrange((width+255)/256):
        tmp=numpy.empty((256, 32+2))
        for yi in xrange(32+2):
            y=yo*32+yi
            if y>=height: y=height-1
            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)
```

Tiling and fusion pros and cons

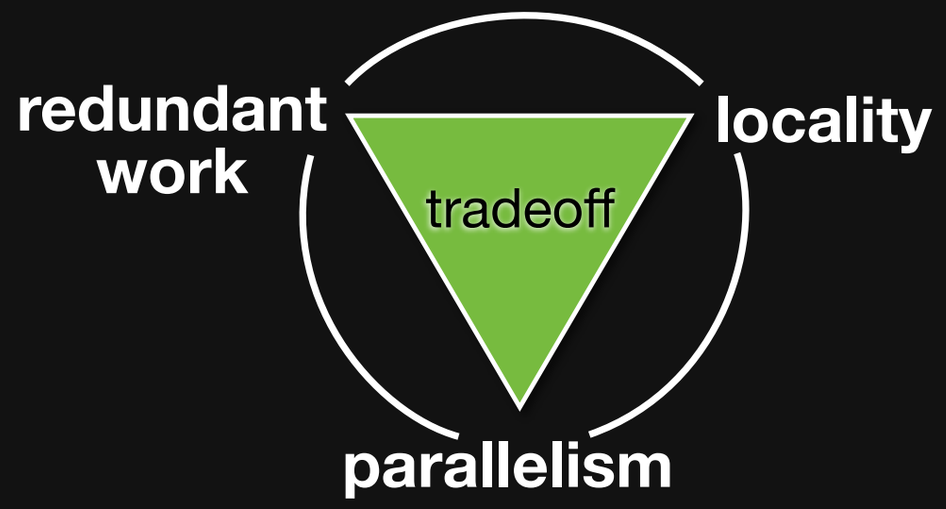


Tiling and fusion pros and cons

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-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)
```

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)