Halide Scheduling

Frédo Durand
most slides by Jonathan Ragan-Kelley
MIT CSAIL

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/

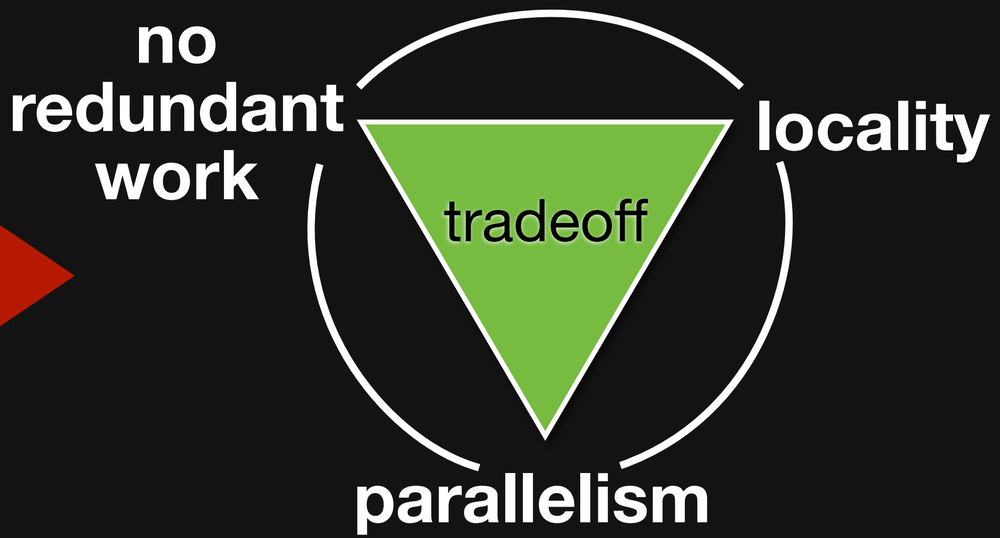
Tradeoffs matter; organization of computation is a first-class issue

Program:

Algorithm

Organization of computation

Hardware



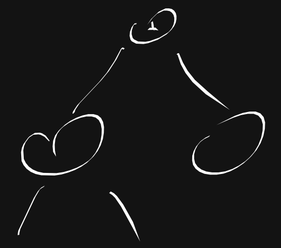
Halide Box Blur

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

Var x, y;
Func blur_x;
Func blur_y;


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


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



Halide *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);
```

Main Halide elements

NO loops → implicit

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

(mathematical)

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

only for input/output
NOT for intermediate stages (use Func instead)

Under the hood

Func, Var, Expr, Image are C++ classes representing Halide programs

Declare Halide Funcs, Vars, etc. by calling the C++ constructor of these classes

Define Funcs and Expr with operator overloading: (), =, +,*, etc.

Manipulate Halide schedules with methods of the Func class

Realize

Func.realize calls the Halide compiler on the C++ 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

Swept under the rug so far

Reductions

sum, histogram, general convolution

tricky

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.

easy

and of course, scheduling not explained yet

Halide good news and bad news

Halide syntax is easy

Halide is cross platform (Linux, Mac, Windows) kind of

Although some better than others (I'm looking at you Windows)

Supports many targets (x86, GPU, ARM)

Halide generates really fast executables

Halide is easy to embed in 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

Recap & Questions?

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

Embedded in C++:
Under the hood: C++ classes represent the Halide program

Simple syntax: Func, Var, Expr

Call realize to compile and execute

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

Schedule

Given an algorithm defined as a pipeline of Func

Specifies the order of computation

- within stages
- across stages (more tricky)

specifying loop nest

Schedule

Given an algorithm defined as a pipeline of Func

Specifies the order of computation

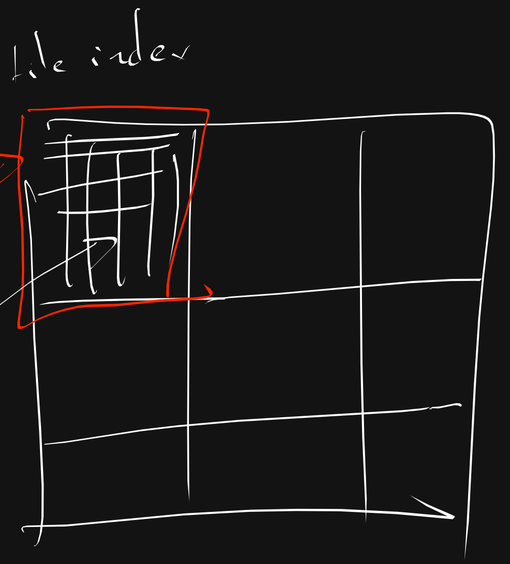
within each Func

across Func (the more interesting part)

Boils down to specifying nested loops

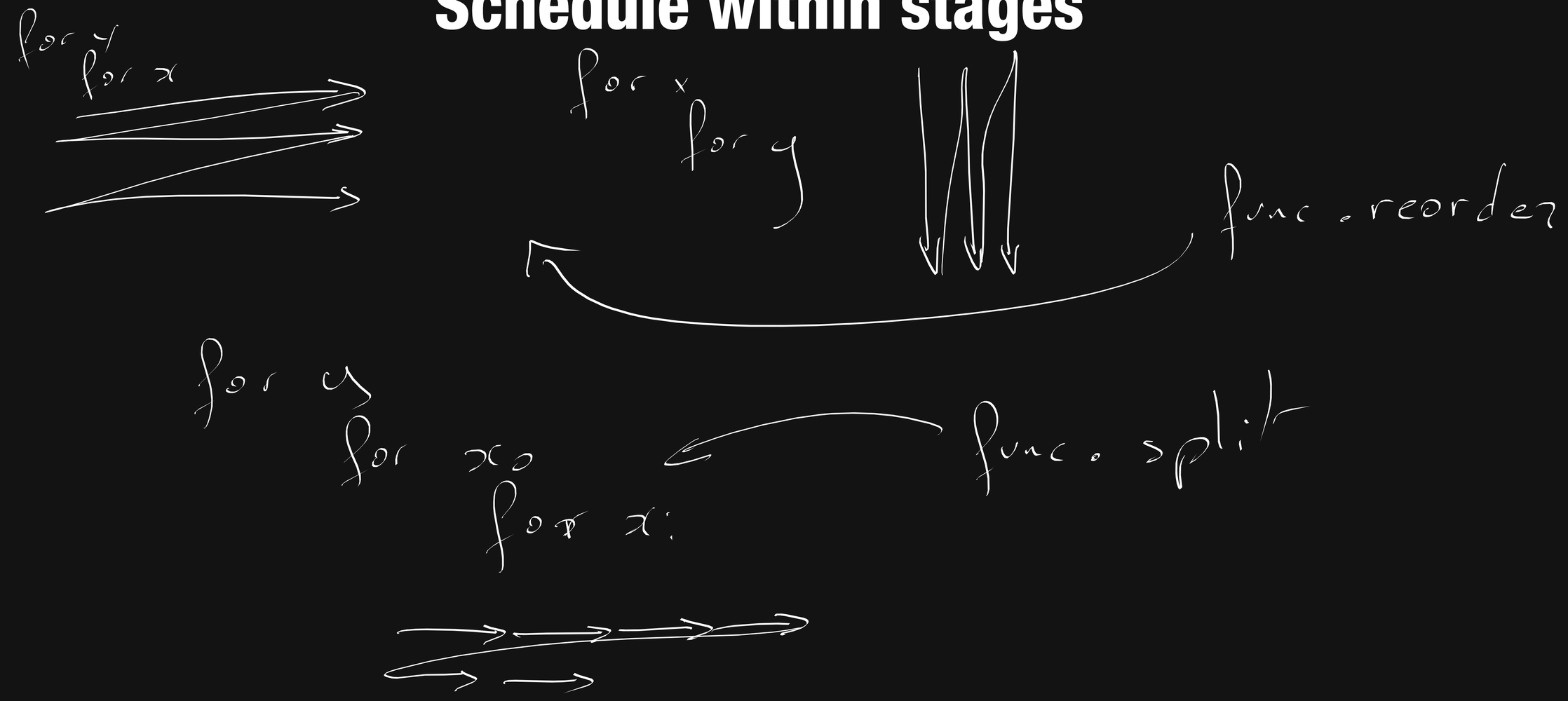
e.g. tiled computation for one stage:

```
for y_tile_index:
    for x_tile_index:
        for y_within_tile:
            for x_within_tile:
                out(y_tile_index*tile_height+y_within_tile,
                    x_tile_index*tile_width+x_within_tile)=…
```



hidden under hood

Schedule within stages



Schedule across stages

control locality

e.g. separable blur: blur_y of blur_x

blur_y (later stage) is the consumer

blur_x (earlier) is the producer

↳ controls whether produced values are still in cache when consumed

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

Why start with consumer

In general, schedule from the output to the input

at least intellectually

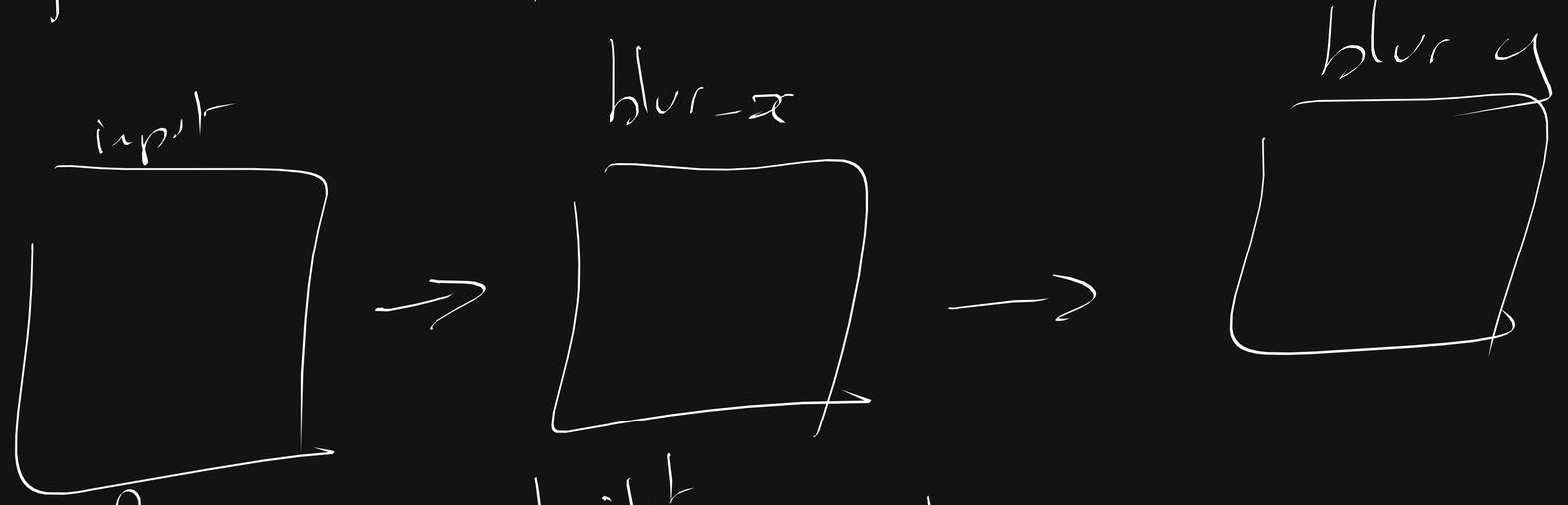
Because consumer knows dependencies, producer doesn't

while data flow from producer to consumer,
dependencies flow from consumer to producer

essentially pull perspective rather than push

Root schedule (across)

compute producer for entire image
before starting consumer



for y 0... height
for x 0... width
compute producer

for y for x
compute consumer

Root schedule

Most similar to what you'd do in C++

Compute the producer on the entire image before starting the consumer

Specified with Func.compute_root()

Default and only option 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();
```

optional

Root schedule equivalent

Root schedule equivalent

```
allocate tmp(width, height+2)
for y in(0..height+2):
    for x in(0..width):
        tmp[x,y]=(input[x,y]+input[x+1,y]+input[x+2,y])/3

for y in(0..height):
    for x in (0..width):
        out[x,y] = (tmp[x,y]+tmp[x,y+1]+tmp[x,y+2])/3
```

whole image size

// blur x

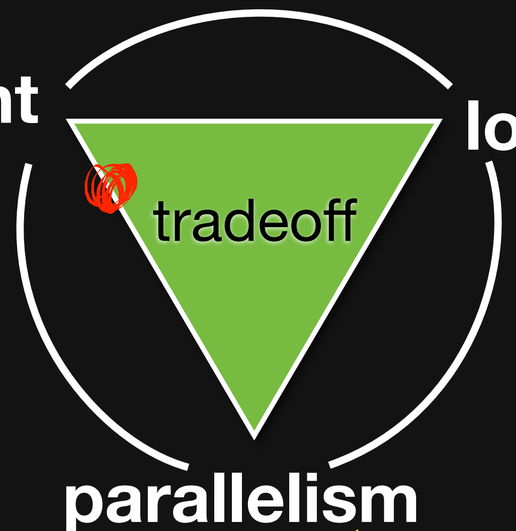
// blur y

Root pross and cons

for y
  for x
    tmp = - - -

each pixel x,y is only computed once

A+

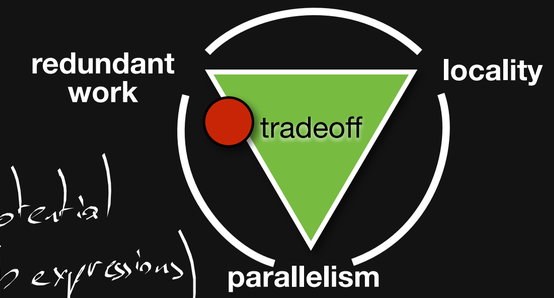


no parallelism yet

F or G

compute entire image worth of produces before consuming it

Recap: root



Most similar to what you'd do in C++

(ignoring potential common sub expressions)

breadth first

Compute the producer on the entire image before starting the consumer

Specified with Func.compute_root()

Default and only option 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();
```

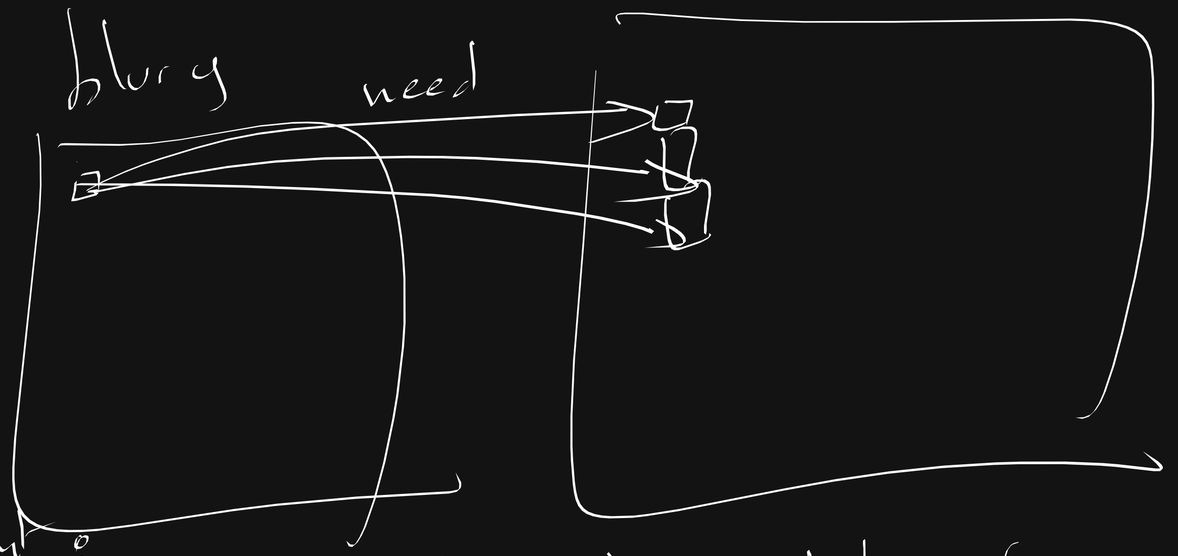
Inline schedule (across)

depth first

last minute computation

for y
for x
blur y

blur_y(x, y)



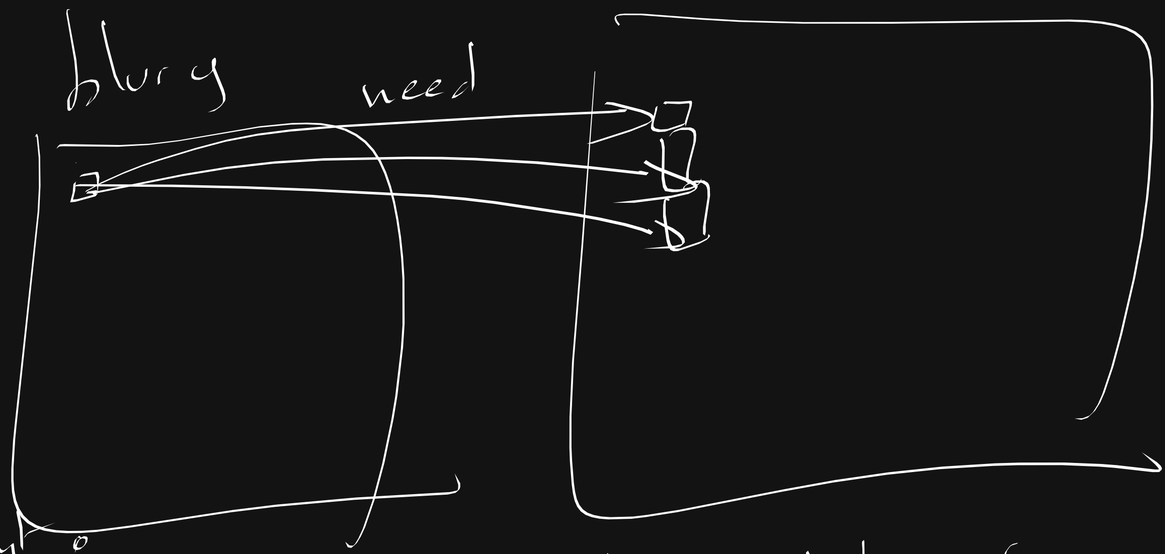
what I want

= blurx(x, y-1) + blurx(x, y) + blurx(x, y+1)

→ input(x-1, y-1) + input(x, y-1) + input(x+1, y-1)

last minute computation

```
for y
  for x
    blur_y(x, y)
```



= what I want:

$$= \text{blur}_x(x, y-1) + \text{blur}_x(x, y) + \text{blur}(x, y+1)$$

$$\rightarrow \Big[\big(\text{input}(x-1, y-1) + \text{input}(x, y-1) + \text{input}(x+1, y-1) \big)/3 + \big(\text{input}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y) \big)/3 + \big(\text{in}(x-1, y+1) + \text{in}(x, y+1) + \text{in}(x+1, y+1) \big)/3 \Big]$$

Inline schedule

Opposite of root: compute producer right when needed, for each value of the consumer

No loop for the producer, all inside consumer

```
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

```
//reminder, Halide algorithm:
// 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

allocate out (width, height)
```

Inline schedule equivalent

```
//reminder, Halide algorithm:
// 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

allocate out (width, height)
for y in (0..height):
    for x in(0..width):
        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
```

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)

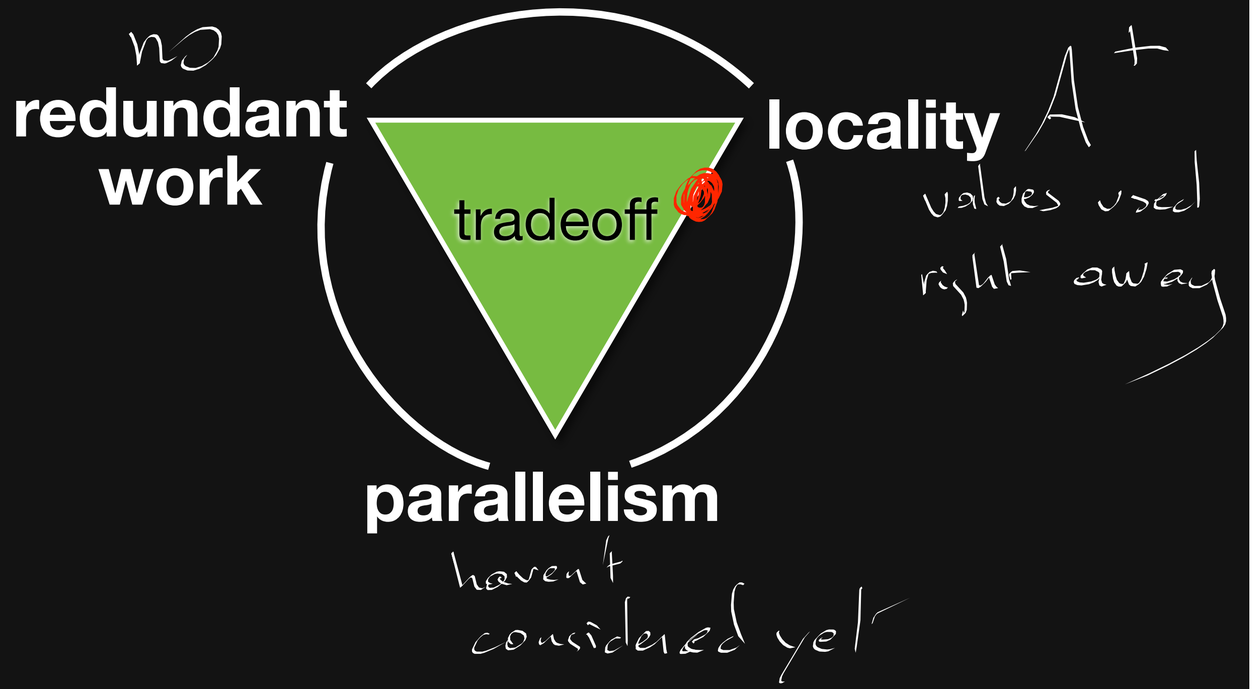
This makes it easier to use many intermediate Funcs for complex algebraic calculations.

```
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 pros and cons

each producer value computed 3 times

D

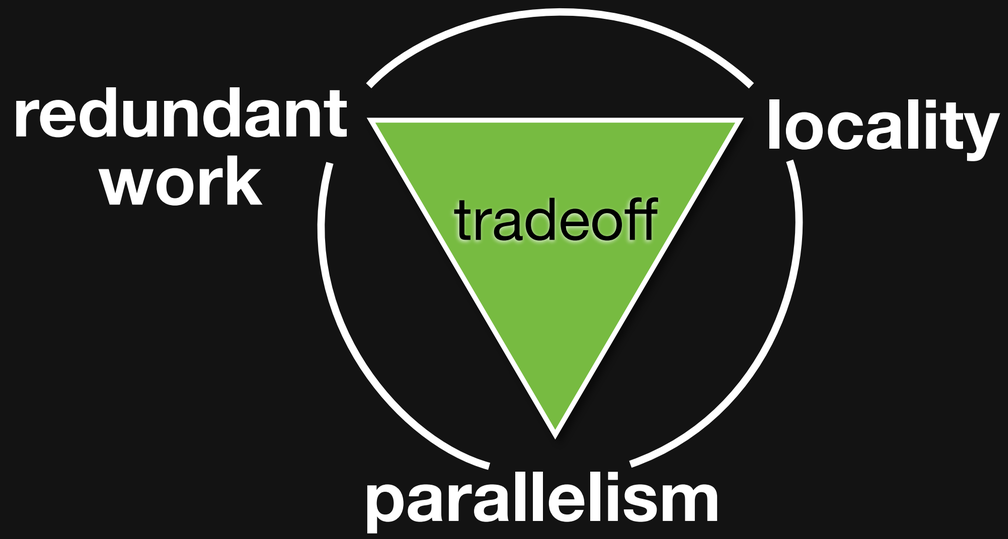


Root vs. Inline

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

good locality like inline

limit redundancy like root



Producer-consumer

First schedule consumer (start with output)

so far trivial, root (more interesting soon)

For producer, at what granularity of the consumer is it computed?

root: prepare all producer values needed for entire image

inline: produce on the fly, for each individual consumer pixel
(lots of redundancy but super local)

more options soon

Once you specify consumer granularity
Halide figures out how much is needed, allocates buffers, etc.

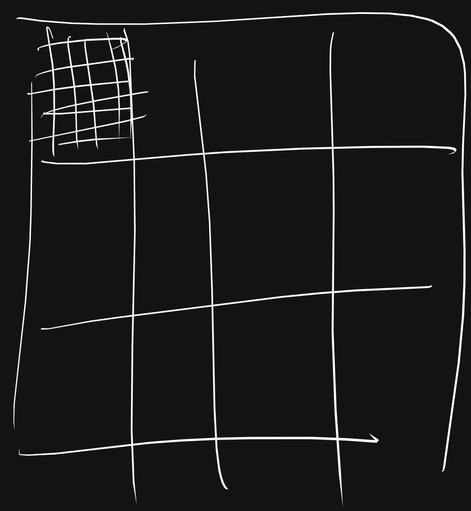
Then maybe schedule within producer

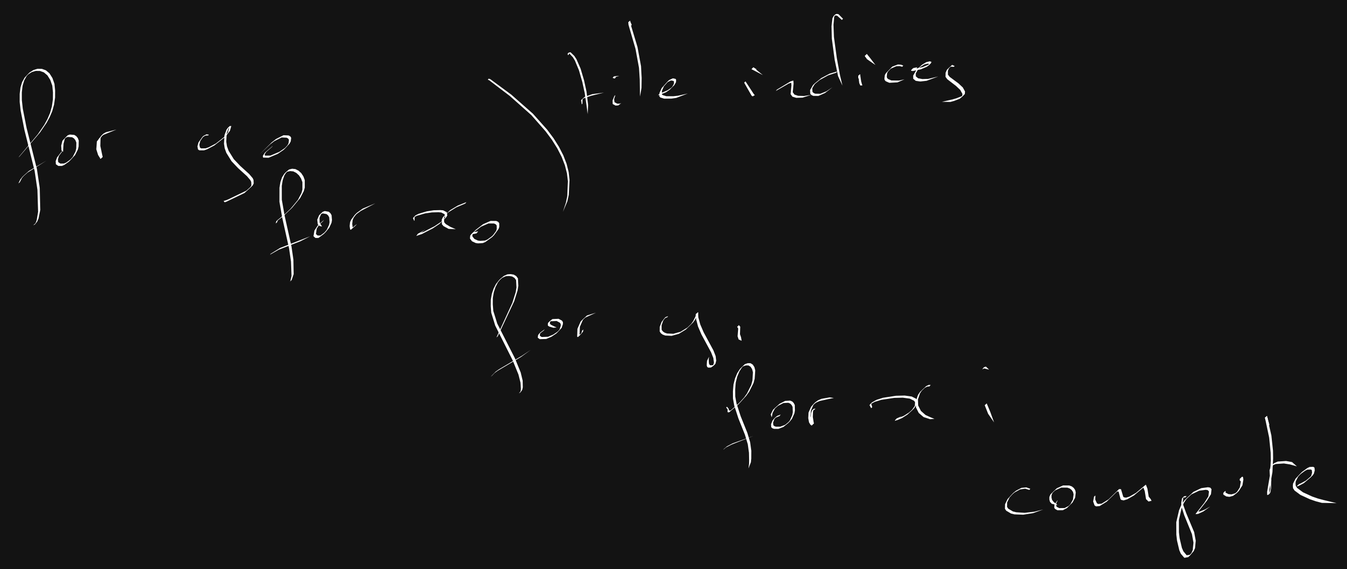
Tiling (within)

Split the domain into inner and outer loops

Specified with Func.tile()

requires new Vars for inner and outer indices





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

```
Var x, y; Func blur_y;
blur_y(x,y) = (input(x,y)+input(x,y+1)+input(x,y+2))/3;

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

Tiling: equivalent

```
allocate out (width, height)
```

Tiling: equivalent

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

Tiling: equivalent for non-integer multiple

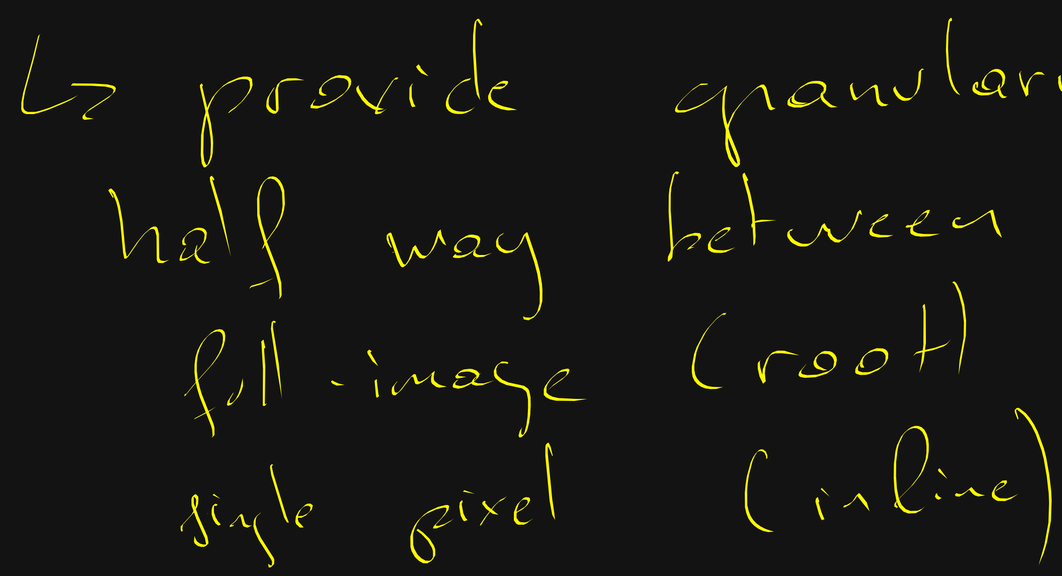
```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in (0..(height+31)/32): #+31 to get the ceiling int
   for xo in(0..(width+255)/256):#+255 to get the ceiling int

      for yi in(0..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(0..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
```

Why tile?

Parallelism

Soon: parallelism+locality+limited redundant work



Tiling under the hood

Tiling under the hood

Tiling is a sugar for split+reorder

split y into yo, yi

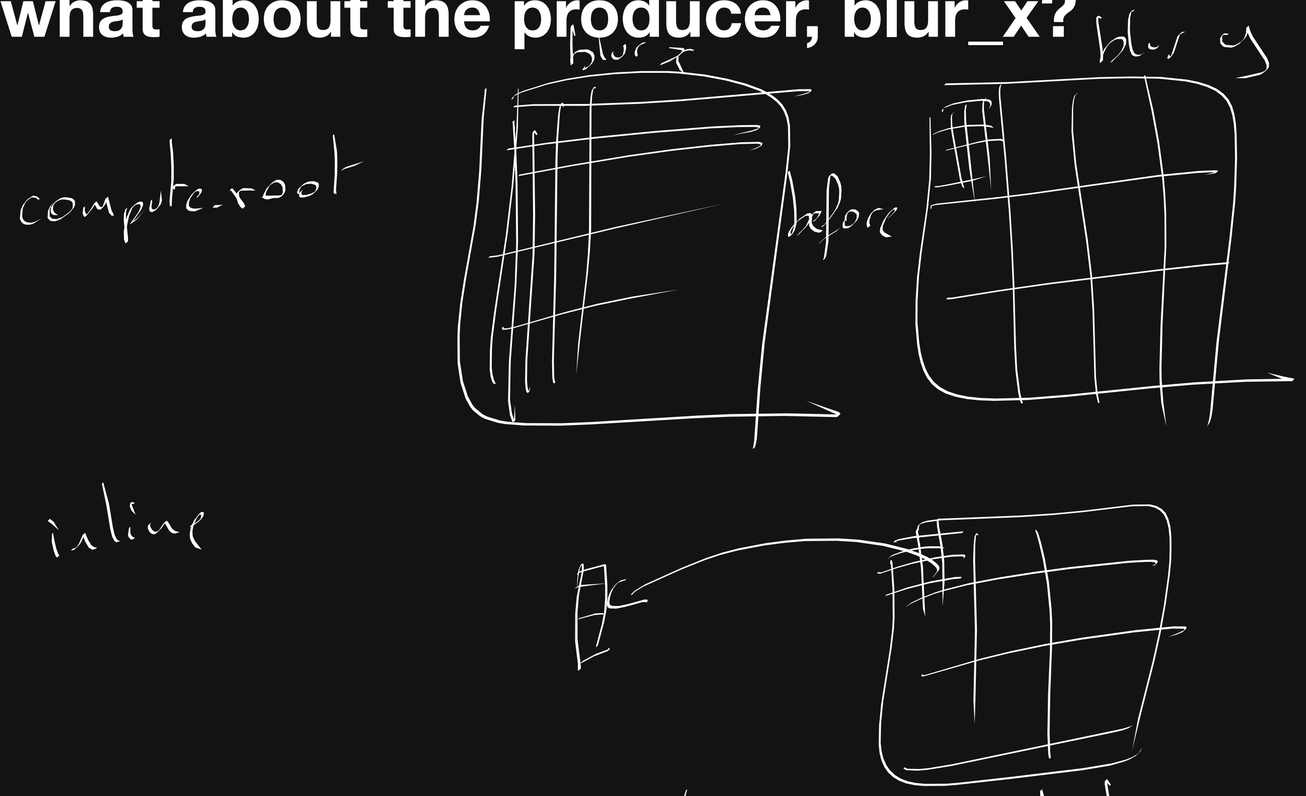
split x into xo, xi

reorder from yo, yi, xo, xi
to yo, xo, yi, xi

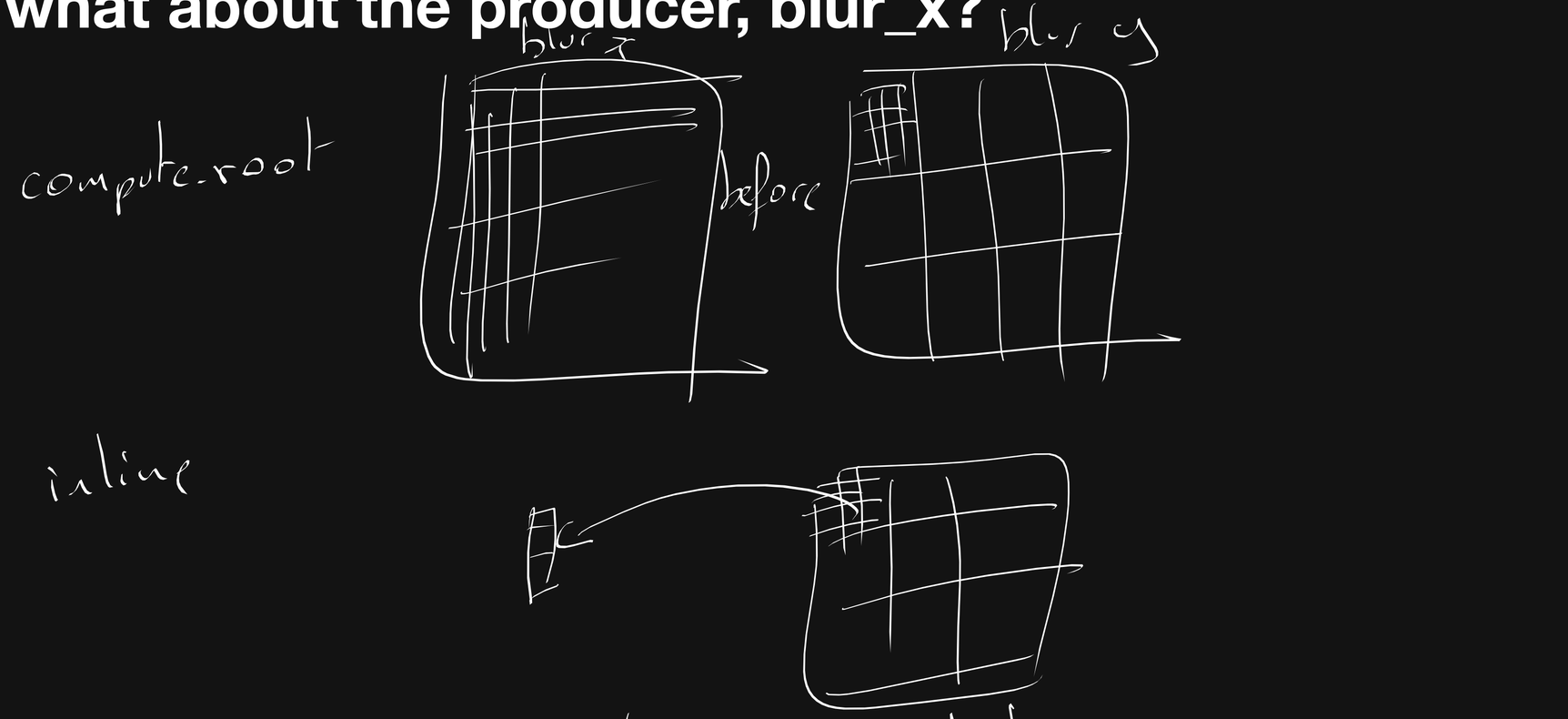
Tiling II (within+across)

Back to 2-stage pipeline, tile the consumer (blur_y)

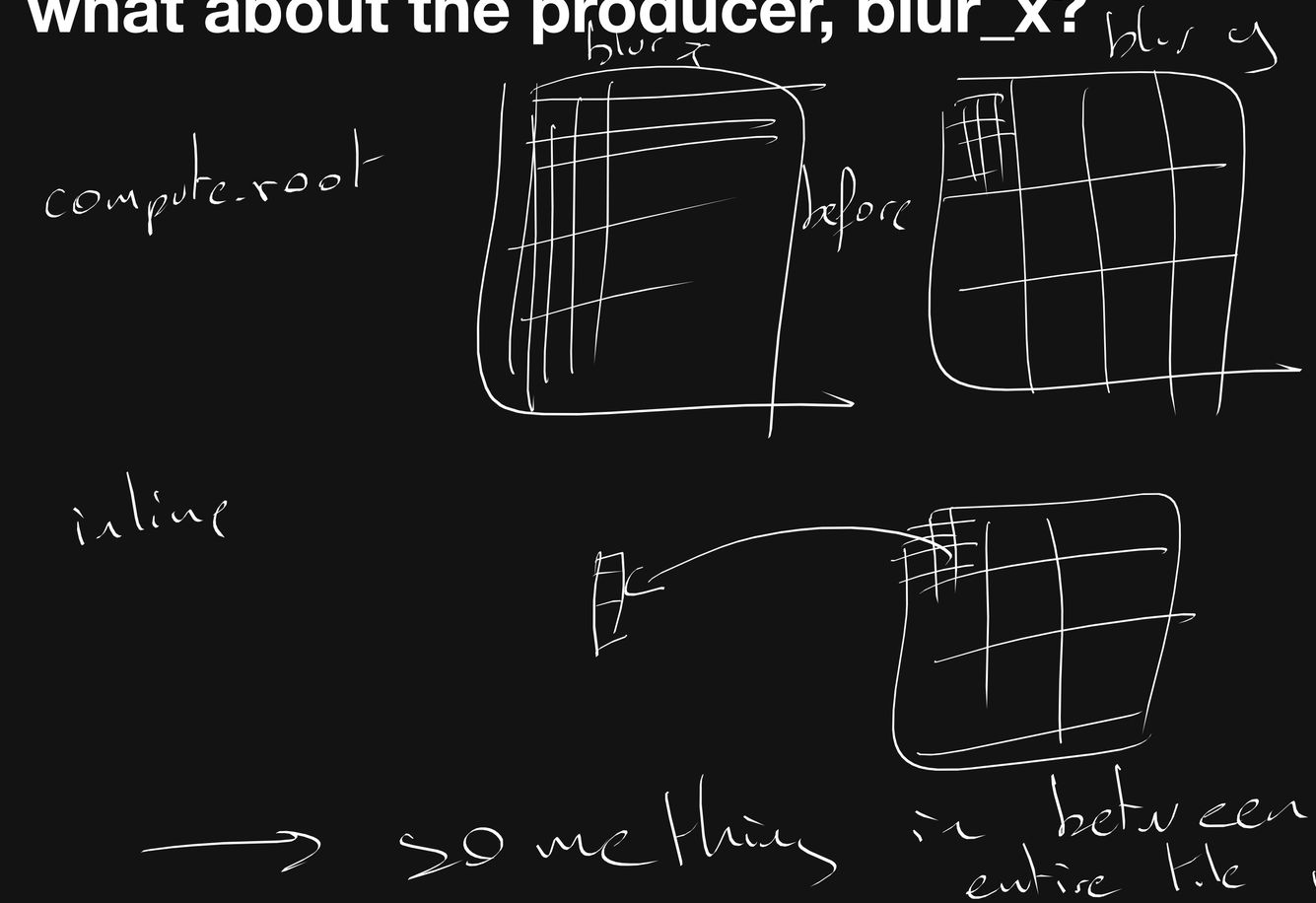
what about the producer, blur_x?



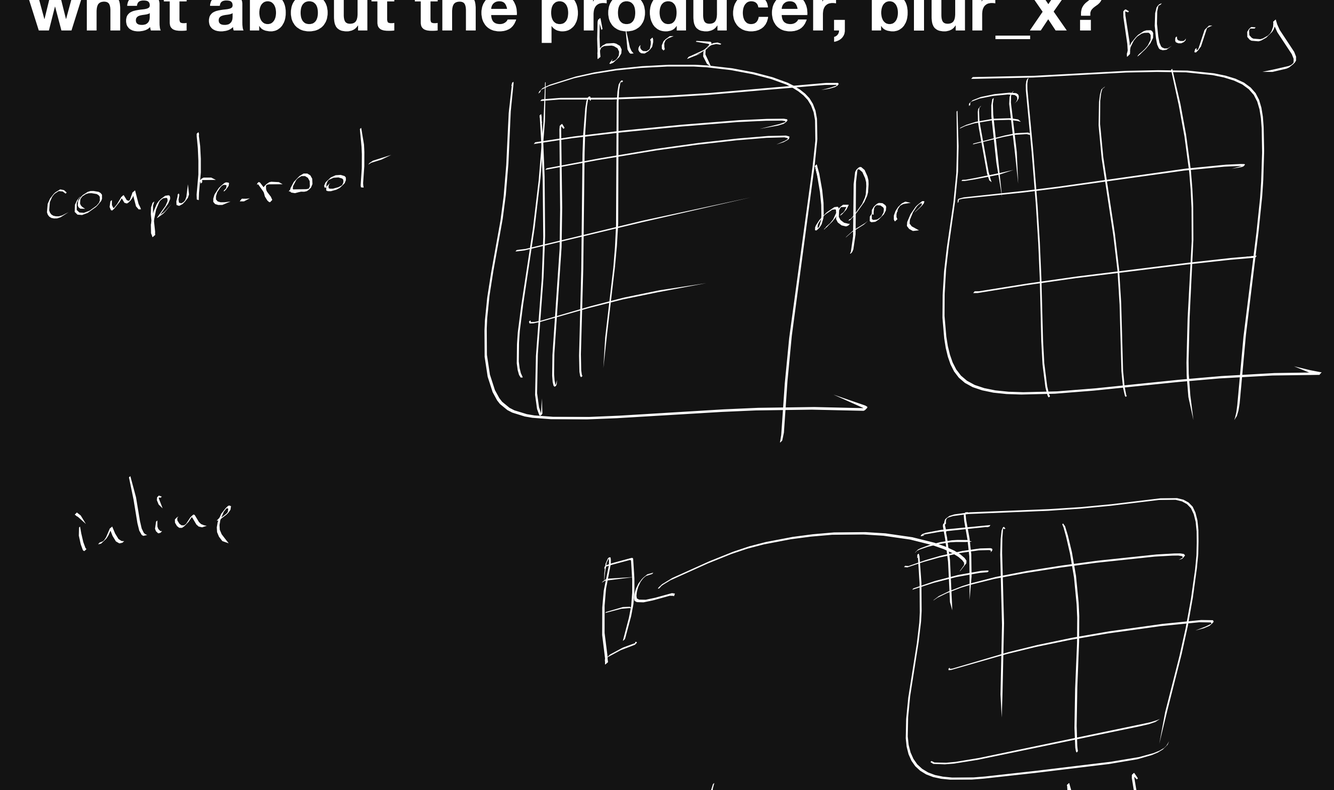
what about the producer, blur_x?



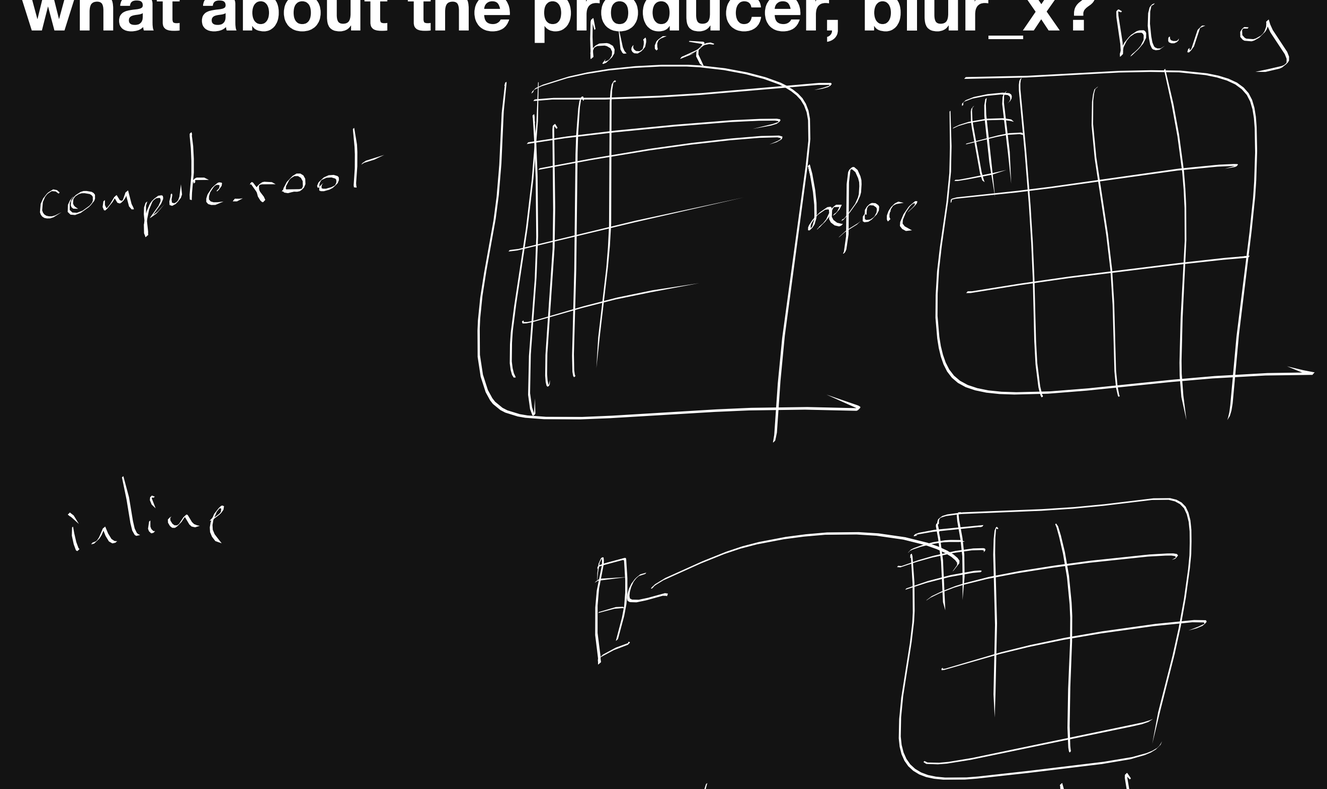
what about the producer, blur_x?

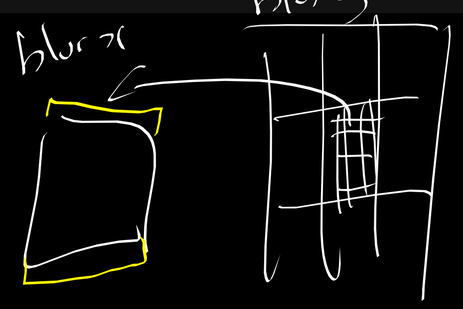


what about the producer, blur_x?



what about the producer, blur_x?





→ something in between
entire tile worth of blur_x
just before corresponding
blur_y tile

Tiling II

Back to 2-stage pipeline, tile the consumer (blur_y)

what about the producer, blur_x?

default: inline

could do : compute_root + tile

but Halide will compute all tiles of blur_x before starting blur_y
bad locality
note that tile without compute_root doesn't make sense:
you can't tile something that's inline

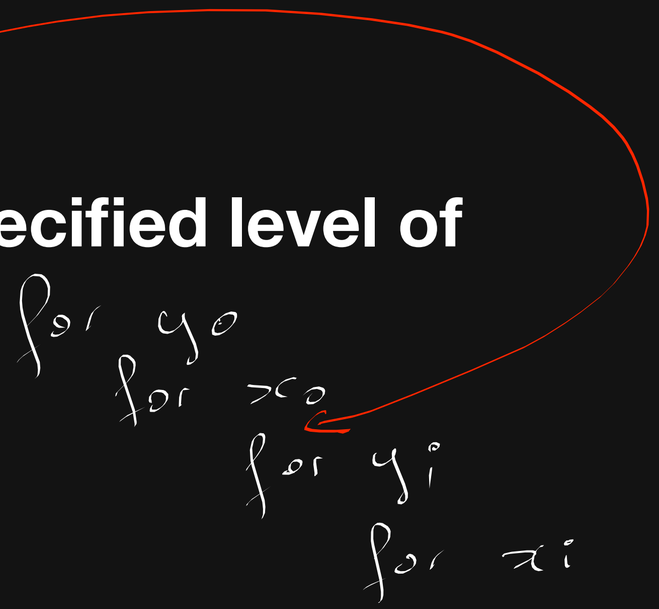
Instead, we want to compute each tile of blur_x before the corresponding tile of blur_Y

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 specified 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)

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)

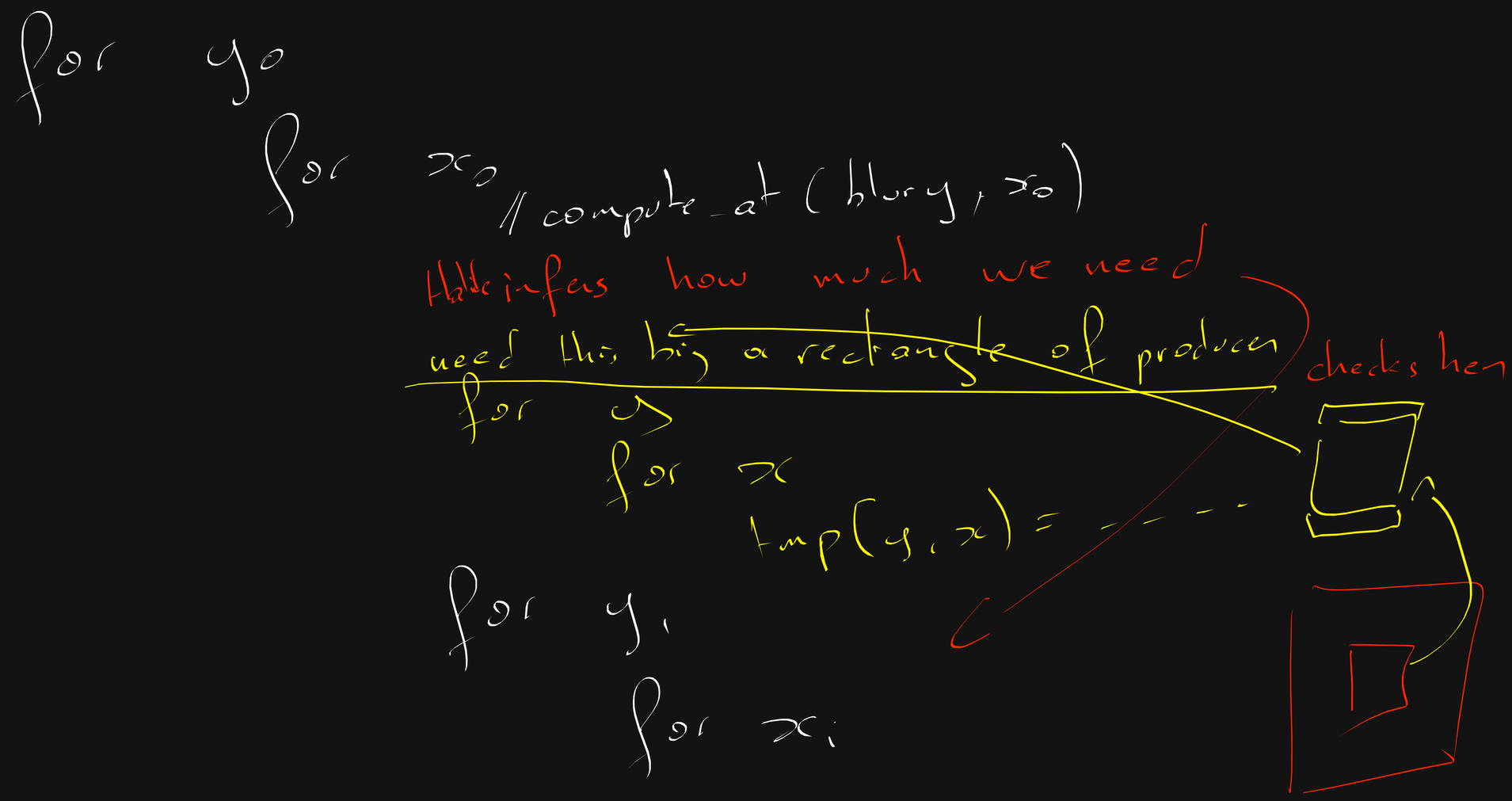
Split consumer into tiles and compute producer for the whole tile just before computing the consumer tile.

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

```
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)
```

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

Tiling and Fusion: equivalent



Tiling and Fusion: equivalent

```
width, height = input.width()-2, input.height()-2
allocate out((width, height))
for yo in(0..(height+31)/32):
    for xo in (0..(width+255)/256):
        allocate tmp((256, 32+2))
        for yi in (0..32+2):
            y=yo*32+yi
            if y>=height: y=height-1
            for xi in (0..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 (0..32):
            y=yo*32+yi
            if y>=height: y=height-1
            for xi in (0..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
```

tiles

blur x

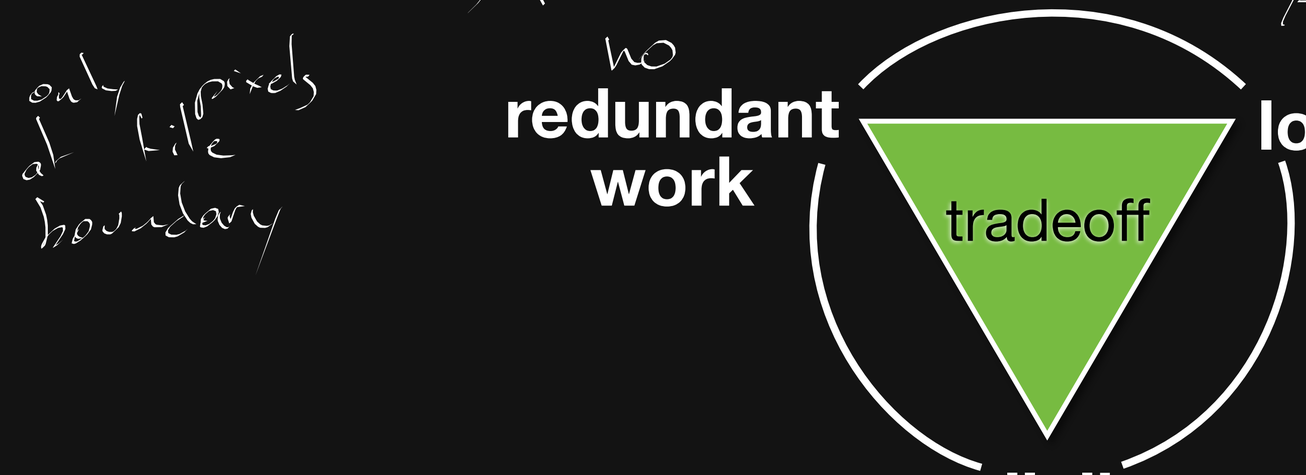
blur y

Tiling and Fusion: equivalent

indexing code affected by schedule in green

```
width, height = input.width()-2, input.height()-2
allocate out((width, height))
for yo in (0..(height+31)/32):      #loops over tile
    for xo in (0..(width+255)/256):
        allocate tmp((256, 32+2)) #tile to store blur_x. Note +2 for enlargement
        for yi in (0..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 (0..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 (0..32):      #loops for blur_y
            y=yo*32+yi
            if y>=height: y=height-1 # for boundary tiles
            for xi in (0..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



A⁻ (same as A)

no redundant work

only pixels at tile boundary

A⁻ (→ A)

locality

size of tile

not quite as good as inline

but much better than root

parallelism

lots of potential

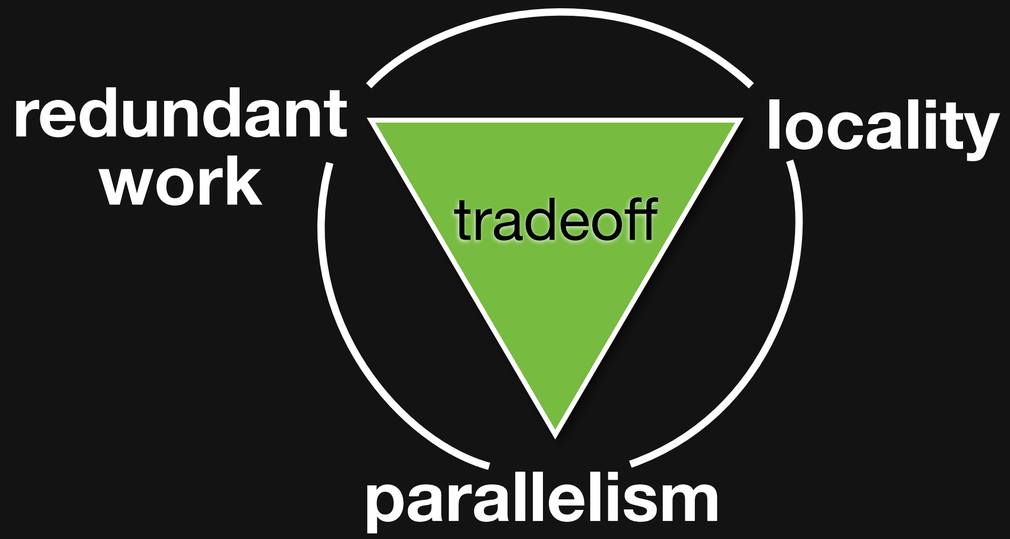
will probably get an A eventually

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

Warning

There are two hard things about Halide:

compute_at

reductions (histogram, convolutio, etc.)

compute_at : across

producer.compute_at(some_consumer, var)

compute_at

producer.compute_at(some_consumer, var)

given the nested loops of the consumer,
performs computation of producer at loop for var

Halide infers everything needed by consumer for loops below var

i.e. rectangle xmin, xmax, ymin, ymax (and other coordinates if applicable)

The resulting rectangle can then be scheduled within

producer.compute_at(consumer, var_k)

```
for var1
    …
    for var_k
      //Insert computation of producer here
      // whatever is needed for the consumer loops below
      allocate temp buffer of appropriate size
      for producer_var1
          for producer_var2
                …
                temp(var1, var2, …) = …
      for var_k+1
        …
        for var_n
            consumer(index1, index2,…) =…
```

compute needed rectangle

compute_at example

```
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, yo )
```

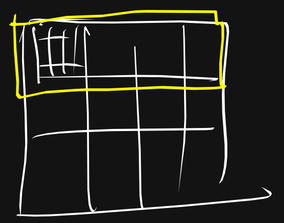
for yo

for xo

for yi

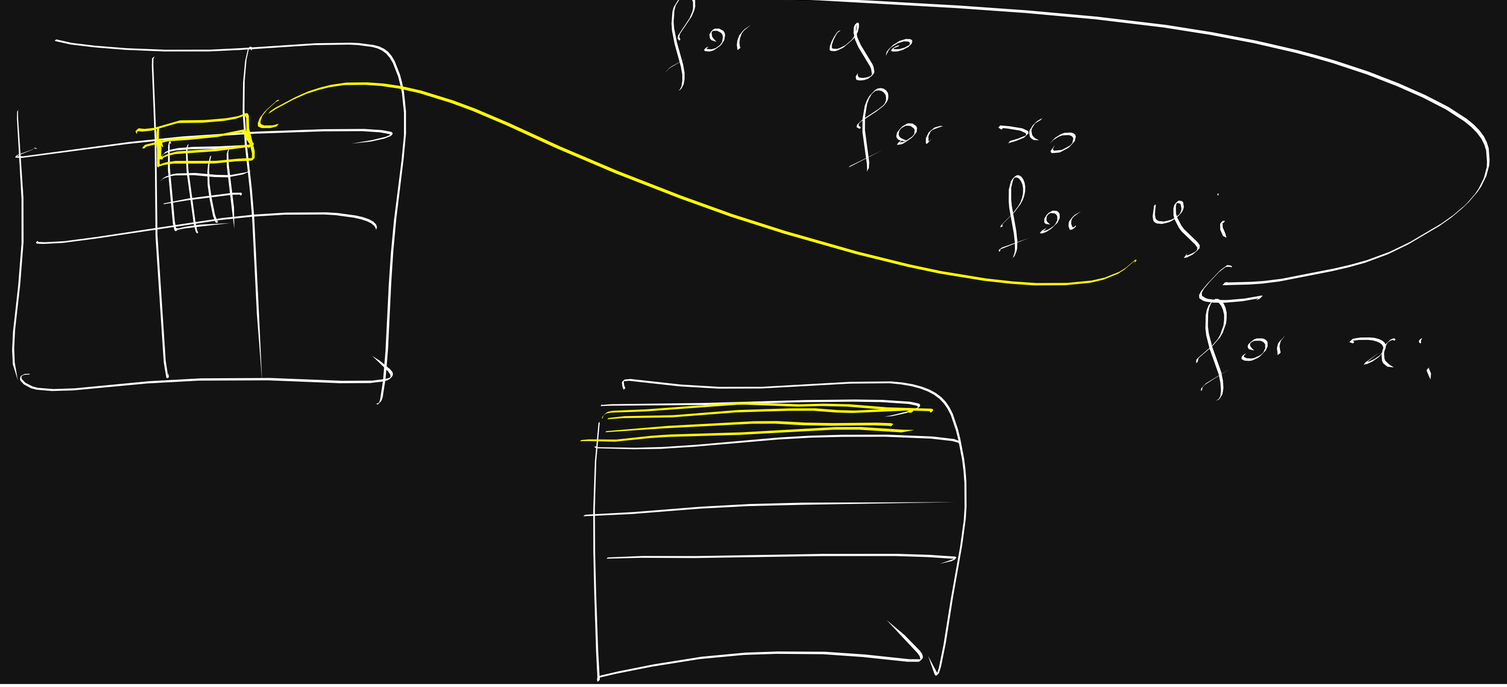
for xi

→ rectangle for entire row of tiles



compute_at example

```
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, yi)
```



compute_at

producer.compute_at(some_consumer, var)

given the nested loops of the consumer,
performs computation of producer at loop for var

Halide infers everything needed by consumer for loops below var

i.e. rectangle xmin, xmax, ymin, ymax (and other coordinates if applicable)

The resulting rectangle can then be scheduled within

Could be any consumer above the producer

typically in multi-stage pipeline,
there are key consumers that everybody compute_at

multistage(>2) pipeline

root & inline as compute_at

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

Parallel

some CUDA-specifc

Parallel

func.parallel(var)

turns the for of this var into a parallel for

vectorize

Don’t worry too much about vectorize for 6.815/6.865

It can be finicky

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

default schedule

took 5.80736255646 seconds

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

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

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

default schedule

took 5.80736255646 seconds

root first stage (producer)

took 1.99803357124 seconds

tile 256 x 256 + interleave

took 1.75527400097 seconds

tile 256 x 256 + parallel+vector

took 0.550438785553 seconds

tile 256 x 256 + parallel+vector without interleaving

took 1.10970659256 seconds

order of magnitude

no locality gain

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 1.75527400097 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)