High-Performance Image Processing

With Halide

Frédo Durand
MIT CSAIL

Participation grade?

Apologies

Last lecture had residual python syntax

if it has [ ] and missing ; it’s a bad sign

Also Apple’s spellcheck wants to replace Funcs by Fun

Scheduling recap

Within stage

reorder, split, tile

parallel, vectorize

Across stages: producer wrt consumer

compute_root: everything needed for whole image

no redundant computation, terrible locality

compute_inline: jus-in-time for one pixel at a time

lots of redundant computation, excellent locality

compute_at: somewhere in between (tile, scanline, etc)

compute_at

one of the tricky things also one of the most powerful ones

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

for box

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

root & inline as compute_at

compute_at specify at which loop level we compute

compute_root means above the outermost loop

inline kind of means below the innermost one

but in practice slightly different from compute_at(consumer, xi)
because it doesn’t necessarily compute a whole rectangle, just the needed values.

Scheduling

Two issues: within stage & across stages

Think about scheduling from output first, from consumer to producers

but the order in the source file doesn't matter.

But the code itself will run producers first.

Common mistake

```
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;
Var xo, yo, xi, yi;
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);
Var xo2, yo2, xi2, yi2;
blur_x.tile(x, y, xo2, yo2, xi2, yi2, 256, 32);
```

tile consumer

tile producer

need to add : blur_x.compute_root();
or better : blur_x.compute_at(blur_y, xo);
blur_y : output
(no need to further)

across → always root
within → tile

blur_x : producer
across → default : inline
→ no rectangle, no notion of within

Common mistake

```
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;
Var xo, yo, xi, yi;
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);
Var xo2, yo2, xi2, yi2;
blur_x.tile(x, y, xo2, yo2, xi2, yi2, 256, 32);
```

blur_x is scheduled inline in terms of “across” (default)
you can’t tile an inline computation
you need to at least add `blur_x.compute_root()`
and you should probably just use `compute_at(blur_y, xo)`
and not tile blur_x itself.

Not covered

Different storage and compute granularity

e.g. `producer.store_root().compute_at(x)`

e.g. store producer for full image but fill values one at a time

avoids recomputation

Common mistake

```
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;
Var xo, yo, xi, yi;
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);
Var xo2, yo2, xi2, yi2;
blur_x.tile(x, y, xo2, yo2, xi2, yi2, 256, 32);
```

blur_x is scheduled inline in terms of “across” (default)
you can’t tile an inline computation
you need to at least add `blur_x.compute_root()`
and you should probably just use `compute_at(blur_y, xo)`
and not tile blur_x itself.

Common mistake

```
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;
Var xo, yo, xi, yi;
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);
Var xo2, yo2, xi2, yi2;
blur_x.tile(x, y, xo2, yo2, xi2, yi2, 256, 32);
```

tile consumer

tile producer

need to add : blur_x.compute_root();
or better : blur_x.compute_at(blur_y, xo);
Eno need to further)

blur_y : output

across ⇒ always root
within ⇒ tile

xouter (= ~~coord~~ index of tile of blur_y)

blur_x : producer
across ⇒ default : inline
⇒ no rectangle, no notion of within

Common mistake

```
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;
Var xo, yo, xi, yi;
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);
Var xo2, yo2, xi2, yi2;
blur_x.tile(x, y, xo2, yo2, xi2, yi2, 256, 32);
```

blur_x is scheduled inline in terms of “across” (default)
you can’t tile an inline computation
you need to at least add `blur_x.compute_root()`
and you should probably just use `compute_at(blur_y, xo)`
and not tile blur_x itself.

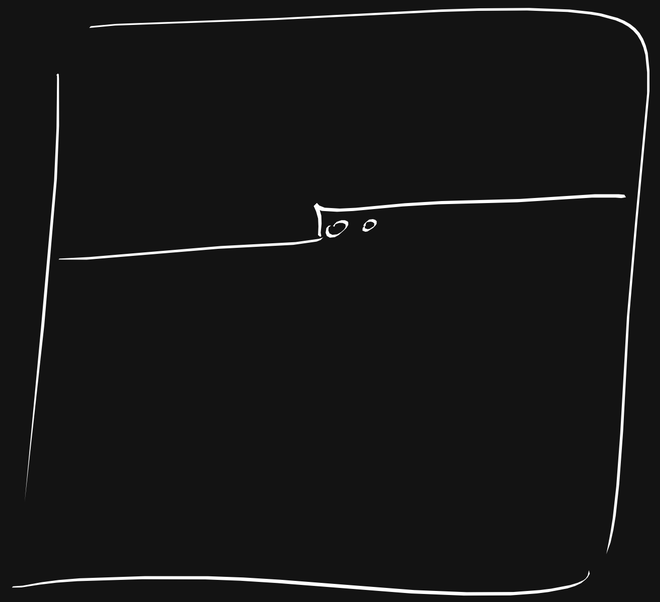
Not covered

Different storage and compute granularity

e.g. `producer.store_root().compute_at(x)`

e.g. store producer for full image but fill values one at a time

avoids recomputation



Naming Vars and Funcs

```
Var x(“x”);

Func f(“f”);
```

helps with debugging messages

Compile to HTML

Compile Halide pipeline to HTML pseudocode

Visualize loop nest

Especially if you have used compute_at, store_at

Visualize where buffers are getting allocated

Visualize which loops are marked parallel

Very helpful for debugging schedules

Compile to HTML: Box blur

```
Func blur_x("blur_x"), blur_y("blur_y");
Var x("x"), y("y"), xi("xi"), yi("yi");


// The 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;


// How to schedule it
blur_y.split(y, y, yi, 8).parallel(y);
blur_x.store_at(blur_y, y).compute_at(blur_y, yi);
```

Visualize this schedule in HTML

```
// create a dummy output buffer just so that compile to HTML knows sizes
Buffer out(type_of<float>(), 512, 512, 0, 0, NULL, "out_buffer");
blur_y.compile_to_simplified_lowered_stmt("halide_blur.html", out, HTML);
```

Box blur HTML pseudo code

Show actual HTML file

Important excerpts on next slides

Box blur HTML pseudo code

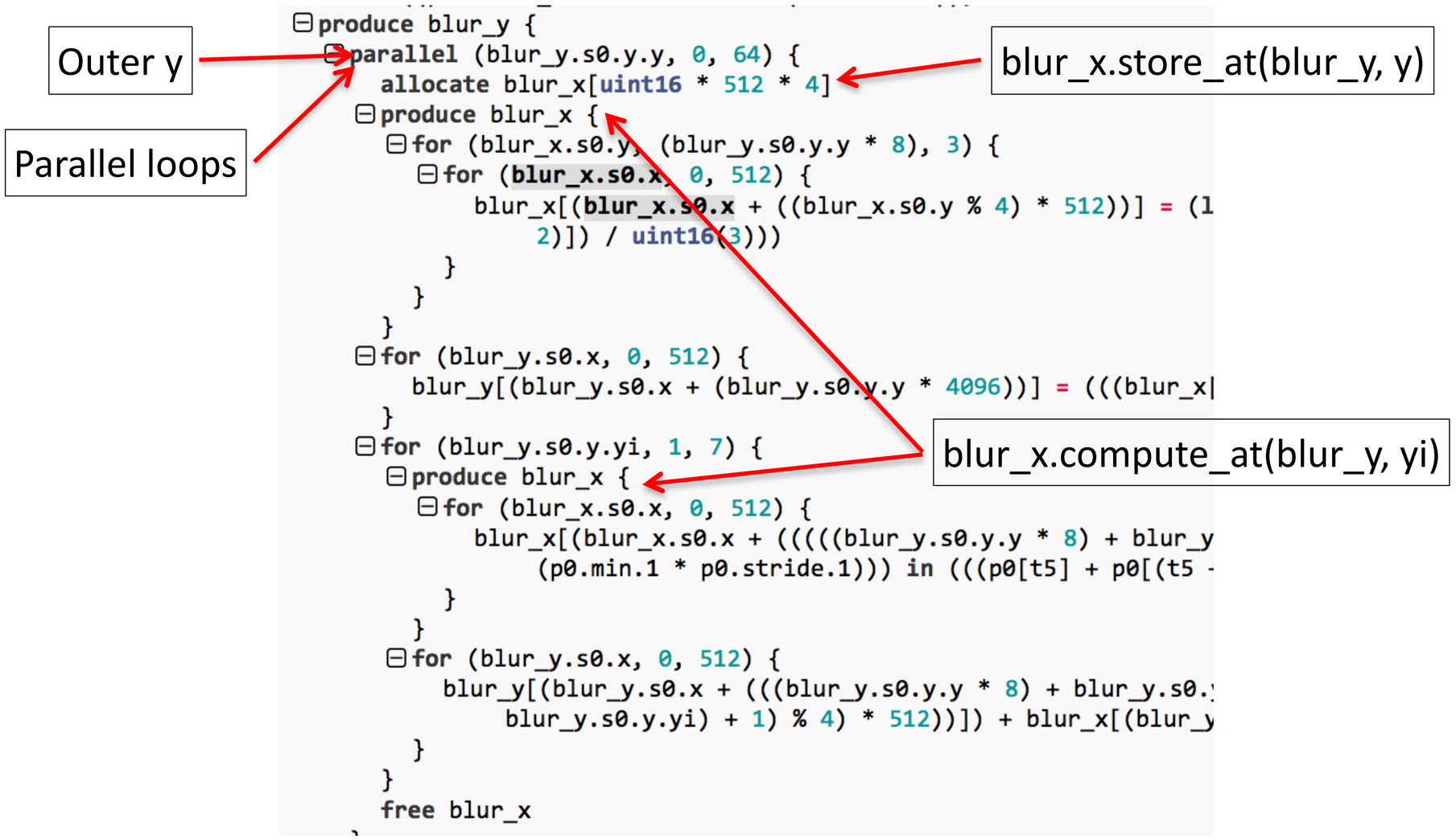
```
produce blur_y {
  parallel (blur_y.s0.y.y, 0, 64) {
    allocate blur_x[uint16 * 512 * 4]
    produce blur_x {
      for (blur_x.s0.y, (blur_y.s0.y.y * 8), 3) {
        for (blur_x.s0.x, 0, 512) {
          blur_x[(blur_x.s0.x + ((blur_x.s0.y % 4) * 512))] = (1
            2)]) / uint16(3)))
        }
      }
    }
    for (blur_y.s0.x, 0, 512) {
      blur_y[(blur_y.s0.x + (blur_y.s0.y.y * 4096))] = (((blur_x|
    }
    for (blur_y.s0.y.yi, 1, 7) {
      produce blur_x {
        for (blur_x.s0.x, 0, 512) {
          blur_x[(blur_x.s0.x + (((((blur_y.s0.y.y * 8) + blur_y
            (p0.min.1 * p0.stride.1))) in (((p0[t5] + p0[(t5 -
        }
      }
      for (blur_y.s0.x, 0, 512) {
        blur_y[(blur_y.s0.x + (((blur_y.s0.y.y * 8) + blur_y.s0.
          blur_y.s0.y.yi) + 1) % 4) * 512))]) + blur_x[(blur_y
      }
    }
    free blur_x
```

Loop min, size

Inner yi=0

Inner yi=
1 to 7
Optimize boundary
conditions for yi=0

Box blur HTML pseudo code



Compile to HTML

Halide infers size of buffers, extent of loops etc. from out_buffer

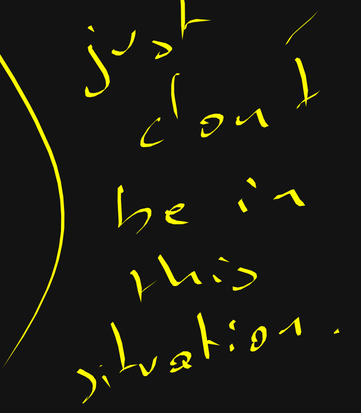
```
Func::compile_to_simplified_lowered_stmt("stmt.html", out_buffer, HTML);
```

Most complete debugging output

- If you don't know size of output buffer

```
Func::compile_to_lowered_stmt("stmt.html", HTML);
```

Uses symbolic variables for extents, buffer sizes but not quite as good.



across

at which level of loop nest
do we insert produce computation

within

what loops do we use for
the corresponding rectangle/box of produce

→ implies two hierarchies

consume > produce

across > within

When to use what?

Schedule producer as a function of the stencil of consumer

how many producer pixels are needed for one consumer pixel?

example : blur_y needs 3 blur_x

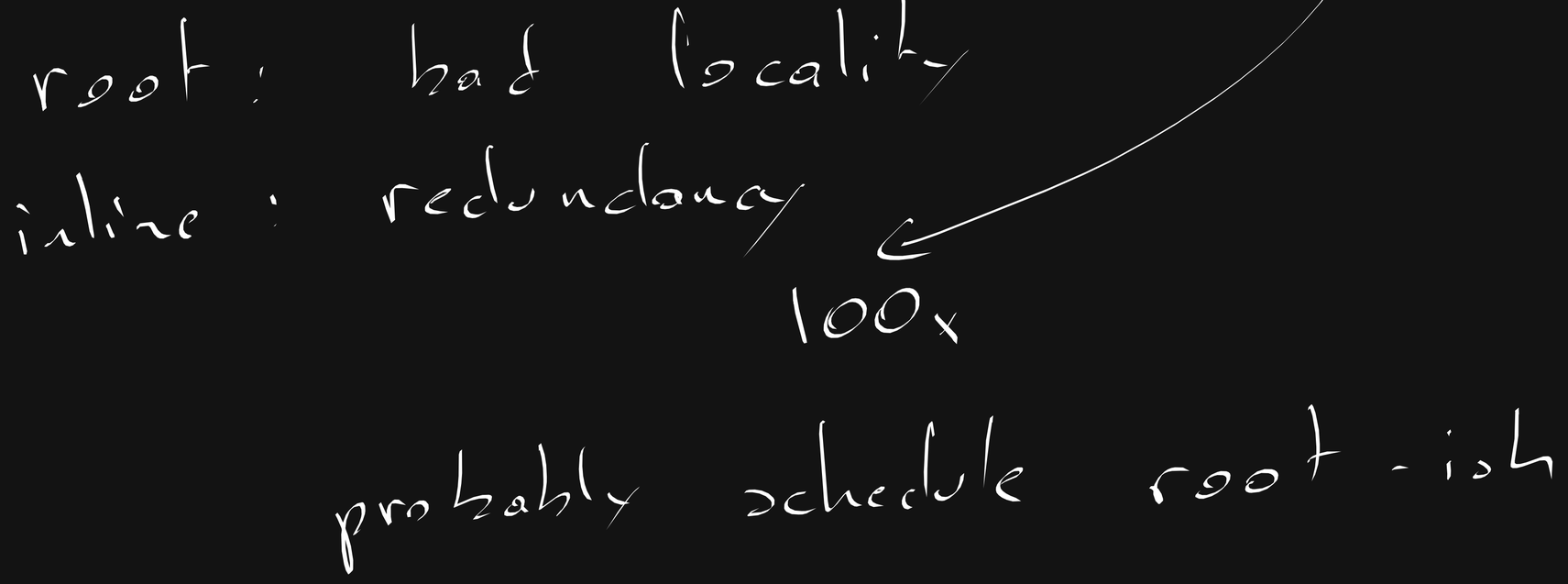
value->value 1 (pointwise)

Let's first look at root vs. inline

Big stencil

consumer: vertical Gaussian blur 100 pixels

producer: horizontal Gaussian blur 100 pixels



Small stencil (pointwise)

consumer: gamma remapping

value → value &
stencil size 1

producer: horizontal Gaussian blur 100 pixels

inline : redundancy ✓ 1x
good

→ inline is best!

leave it as default
no need for within
not even meaningful

It’s the consumer that matters most

What matter is how many times the producers would be recomputed

This depends on consumer, not producer

Exception:
When producer is very cheap, recomputation may be OK

Multistage(>2) pipeline

Recursive decisions, starting from output

start

often good solution:

tile output (or similar granularity, e.g. full rows)

leave producers of pointwise consumers as inline (default)

use compute_at to fuse producers

except if tile expansion becomes too large (cumulative stencil too big)

then use compute_root

start by making list of producers for consumers of stencils > 1

start by scheduling them as root
then start thinking

Multistage(>2) pipeline

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

Final 3x3 blur: add parallelism and vectorization

```
Var x(“x”), y(“y”);
Func blur_x(“blur_x”), blur_y(“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;

Var xi(“xi”), yi(“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);
```

shorter version of tile

SIMD

SIMD

Parallel

func.parallel(var)

turns the for of this var into a parallel for

vectorize

Gives you SIMD instructions easily

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

It can be finicky

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

vs. Halide version, same performance

```
Var x(“x”), y(“y”);
Func blur_x(“blur_x”), blur_y(“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;

Var xi(“xi”), yi(“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

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

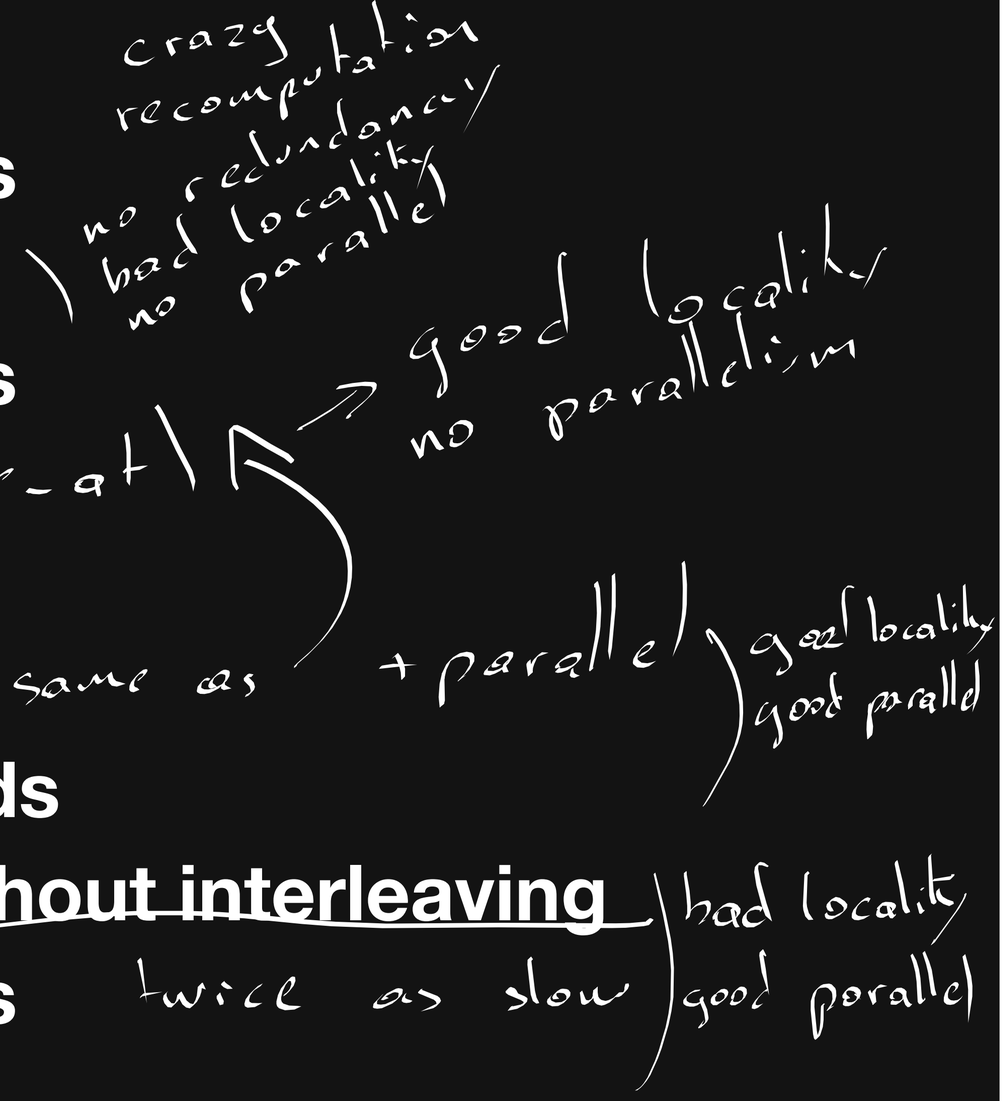
default schedule
took 5.80736255646 seconds

root first stage
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



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)

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 (using compute_at)

+ others (vectorize, parallelize)

Within vs. across

Within

tile, reorder, split

Across

root, inline, compute_at

Note: the “within” schedule operates on the granularity given by the “across” schedule

e.g. if you schedule the producer “compute_at” the level of a tile of the consumer, and then try to tile the producer, the tiling will apply within a tile the size of what the c

First decision is across (wrt consumer), then within

Always start with consumer!

In general, schedule from the output to the input

Because consumer knows dependencies, producer doesn’t

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

How can we determine *good* schedules?

How can we determine *good* schedules?

Explicit programmer control
The compiler does *exactly what you say*.
Schedules cannot influence correctness.
Exploration is fast and easy.

How can we determine *good* schedules?

Explicit programmer control
The compiler does *exactly what you say*.
Schedules cannot influence correctness.
Exploration is fast and easy.

Stochastic search *(autotuning)*
Pick your favorite high-dimensional search.
(We used Petabricks' GA tuner *[Ansel et al. 2009]*)

Open Tuner

Heuristic scheduling ongoing work

Schedule tips

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

Worry only about producer that are consumed by stencils > 1

where one consumer pixel needs multiple producer pixels

Don’t worry about point operations

one consumer pixel only needs one producer pixel

leave inline

Jonathan's scheduling strategy

Schedule root for stencil producers

Basic parallelization over scanlines or tiles

Then worry about fusion/interleaving

Andrew’s wisdom

But with all of these things it's very hard to say without trying it. There are so many factors at play. I'm good at justifying results after-the-fact, but pretty poor at predicting performance, which makes me little better than an astrologer. If only we had a language that made it easy to experiment quickly in this space :)

One more keyword: select

no if in Halide

```
select(predicate, Expr1, Expr2)
```

if predicate is true, returns Expr1, and Expr2 otherwise

There is no if in Halide

Thresholding using select

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

Func sel;

Var x, y, c;

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

output = sel.realize(input.width(), input.height(),
                     input.channels());
```

green channel

Clamping (padding)

Halide has a builtin “clamp” function that clamps indices

edge padding

Clamping (padding)

Halide has a builtin “clamp” function that clamps indices

edge padding

Add an extra stage that does the clamping

and leave it inline

new stage just after input → padded input

```
Func clamped("clamped");

clamped(x, y, c) = input(clamp(x, 0, im.width()-1),
                         clamp(y, 0, im.height()-1),
                         c);

blur_x(x,y,c) = (clamped(x,y,c)+clamped(x+1,y,c)+clamped(x+2,y,c))/3.0;

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

Reductions

2nd hard thing in Halide

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 over input pixels

For image processing:

average of an image, max

histogram

convolution

max over windows

Sum pseudocode

```
for c in (0..input.channels()):
    out[c]=0.0
for ry in (0..input.height()):
    for rx in (0..input.width()):
        for c in (0..input.channels()):
            out[c] += input[rx, ry, c]
```

init

loops over input pixels

loop over output channels

accumulation

Sum in Halide

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

Reduction Domain: specifies implicit loops over input or anything else

2D

x

y

init, implicit loop over c

implicit loop over output

implicit loop over RDom

update equation for each value of r in RDom

Halide reductions

Loops are implicit !!

Reduction domain: all the location that will be aggregated

Multidimensional

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

Initialilize your Func

myFunc(Var1, Var2)=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