High-Performance Image Processing

last lecture on Halide

Frédo Durand
MIT CSAIL

High-Performance Image Processing

last lecture on Halide

Frédo Durand
MIT CSAIL

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

Sum in Halide

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

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

mySum(c)=0.0;

mySum(c) +=input(r.x, r.y, c);

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

r.x

r.y

reduction domain

→ first dim of an RDom is always called x

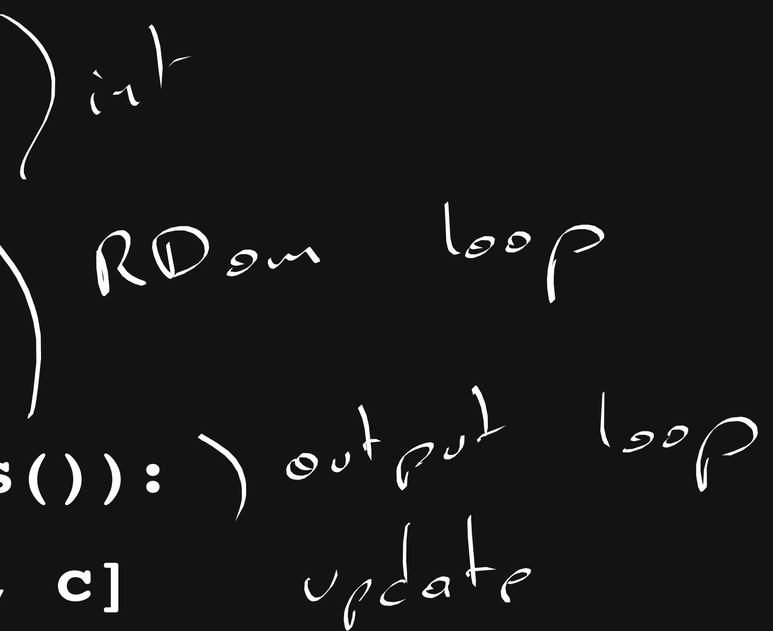
init

update

→ implicit loop

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]
```



Note that the reduction loops are the outer loop

In general, updates

Funcs must always have a first pure-function initialization

similar to what we've done so far

e.g. f(x,y)=input(x,y)*2; or f(x,y)=0.0;

Then they can have multiple updates

either at specific locations: f(32,46) = 1.0;

or sets of locations: f(x, 17)=x+1; (applies to all x, implicit loop)

or with RDoms (implicit loops, more flexibility for arithmetics)

In general:

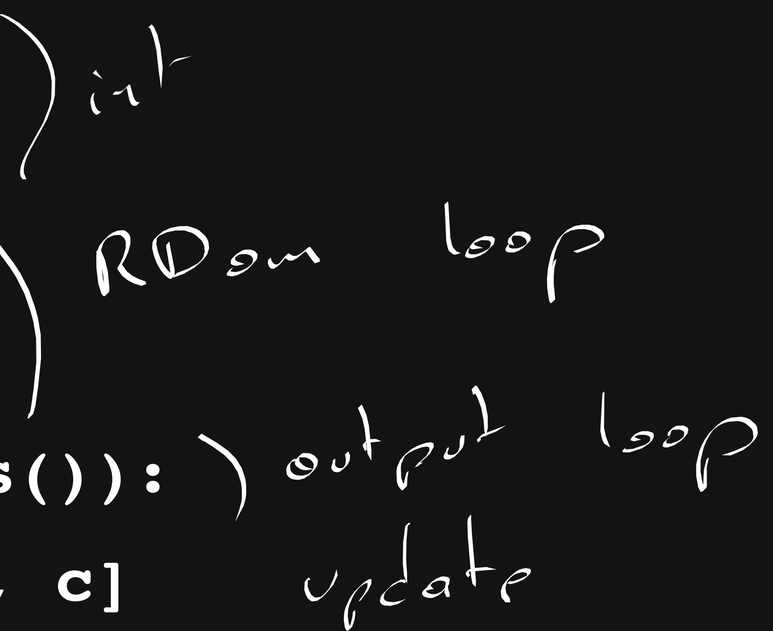
myFunc(Expr,Expr) = f (myFunc(Expr,Expr), Rdom);

will be called for each RDom location

Unlike Func definitions, updates can have Expr on LHS

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]
```



Note that the reduction loops are the outer loop

Sum in Halide

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

RDom r(0, input.width(), 0, input.height());
mySum(c)=0.0;

mySum(c) +=input(r.x, r.y, c);

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

r.x

r.y

reduction domain

→ first dim of an RDom is always called x

init

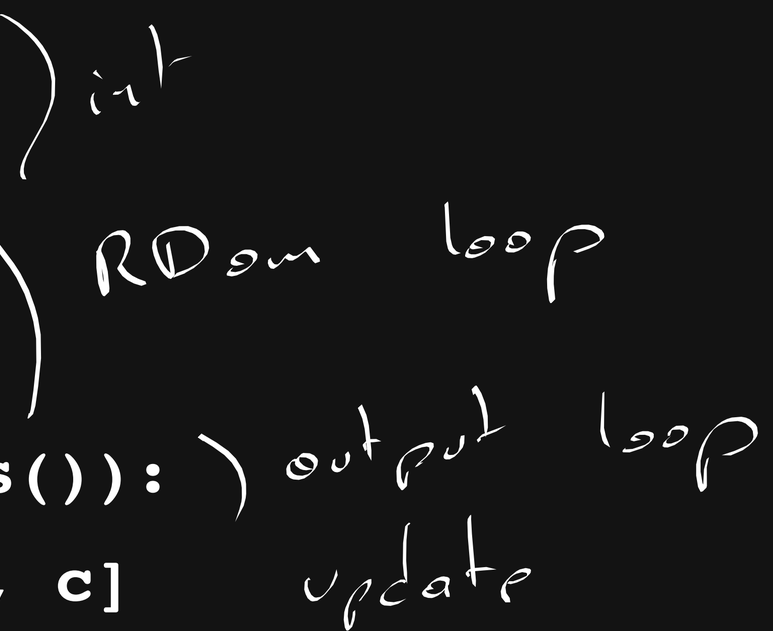
update

→ implicit loop

3

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]
```



Note that the reduction loops are the outer loop

In general, updates

Funcs must always have a first pure-function initialization

similar to what we've done so far

e.g. f(x,y)=input(x,y)*2; or f(x,y)=0.0;

Then they can have multiple updates

either at specific locations: f(32,46) = 1.0;

or sets of locations: f(x, 17)=x+1; (applies to all x, implicit loop)

or with RDoms (implicit loops, more flexibility for arithmetics)

In general:

myFunc(Expr,Expr) = f (myFunc(Expr,Expr), Rdom);

will be called for each RDom location

Unlike Func definitions, updates can have Expr on LHS

Pure Var vs. RDom

Pure Var update: f(x, 17)=x+6.

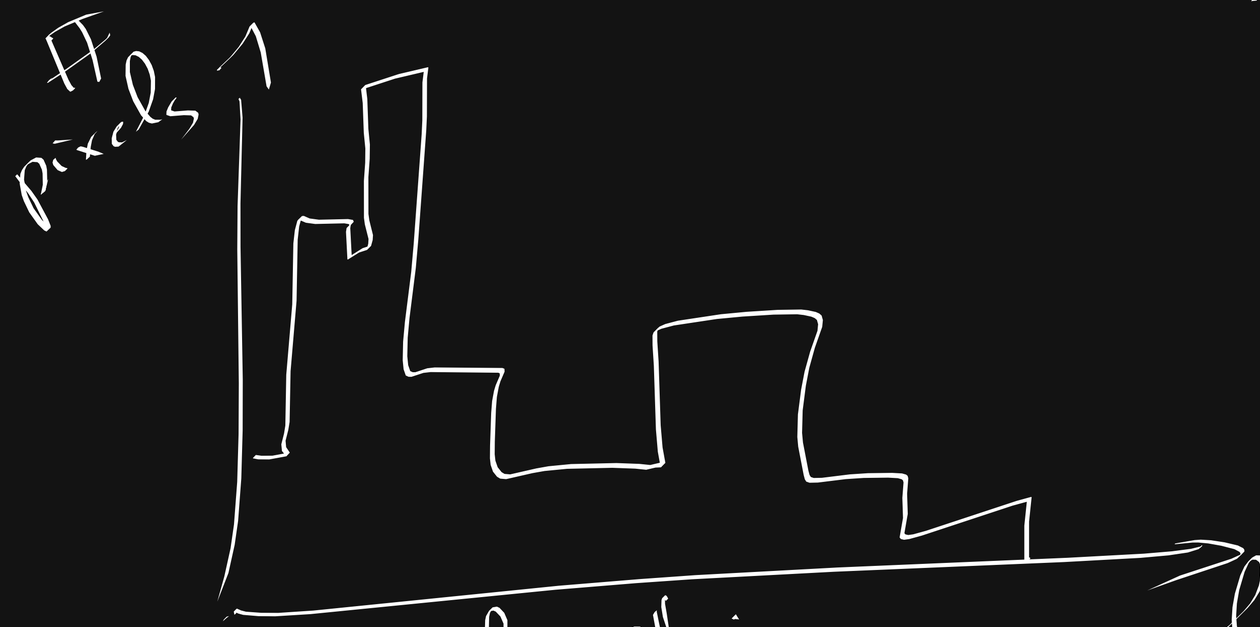
only for output pixel coordinates

pure Var (x) can only appear unmodified on LHS,
and at the same location as in pure definition.
e.g. f(x+1)=x*7; or f(y, x)=16; are invalid (assuming it's f(x, y))

RDoms can be used more arbitrarily

typically iterate over input pixels

Histograms (for 8 bit images)



loop over input pixels) reduction domain

updates (increment counter for bin

hist [intensity (x,y)] ++

Histogram in C (for 8 bit images)

```
int c_result[256];
for (int x = 0; x < 256; x++) {
    c_result[x] = 0;
}
for (int r_y = 0; r_y < input.height(); r_y++) {
    for (int r_x = 0; r_x < input.width(); r_x++) {
        c_result[input(r_x, r_y)] += 1;
    }
}
```

output array

init

reduction loops

update

Histogram in Halide (assuming 8 bit images)

assume input, call it in

```
Func histogram("histogram");
```

(output)

```
Var i("i");
histogram(i) = 0;
RDom r(0, in.width(), 0, in.height(), "r");
histogram(in(r.x, r.y)) += 1
```

update

— realize - - (256)

Histogram in Halide

```
Func histogram("histogram");
Var x("x");
histogram(x) = 0;

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

histogram(input(r.x, r.y)) += 1;

Image<int> halide_result = histogram.realize(256);
```

init

implicit loops

update

C equivalent

```
int c_result[256];
for (int x = 0; x < 256; x++) {
    c_result[x] = 0;
}
for (int r_y = 0; r_y < input.height(); r_y++) {
    for (int r_x = 0; r_x < input.width(); r_x++) {
        c_result[input(r_x, r_y)] += 1;
    }
}
```

reduction domain

Note: no loop over output indices because no Var was used in update equation (unlike sum)

Convolution as reduction

0

(for each output pixel at x, y

pure Var

init

init output(x, y) = 0

loop over kernel rx ry

RDom

add pixel (x - rx, y - ry)

+ maybe divide by normalization

Convolution as reduction

box blur

```
Func blur ("blur"), clamped("clamped");
Var x("x"), y("y"), c("c");
clamped(x, y, c) = input(clamp(x, 0, input.width()-1),
                         clamp(y, 0, input.height()-1), c);
```

```
blur(x, y, c) = 0.0;
RDom r(0, kernel_size, 0, kernel_size);
blur(x, y, c) += clamped(x + r.x - kernel_size/2,
                         y + r.y - kernel_size/2,
                         c) / pow(kernel_size, 2);
blur.realize(____
```

* kernel(r.x, r.y) for general case

Convolution as reduction

```
Func blur (“blur”), clamped(“clamped”);
Var x(“x”), y(“y”), c(“c”);
clamped(x, y, c) = input(clamp(x, 0, input.width()-1),
                         clamp(y, 0, input.height()-1), c);

RDom r(0, kernel_width, 0, kernel_width, “r”);

blur(x,y,c) = 0.0;
blur(x,y,c) += clamped(x+r.x-kernel_width/2,
                       y+r.y-kernel_width/2,
                       c) / pow(kernel_width,2);
```

Note

Again, no loop!

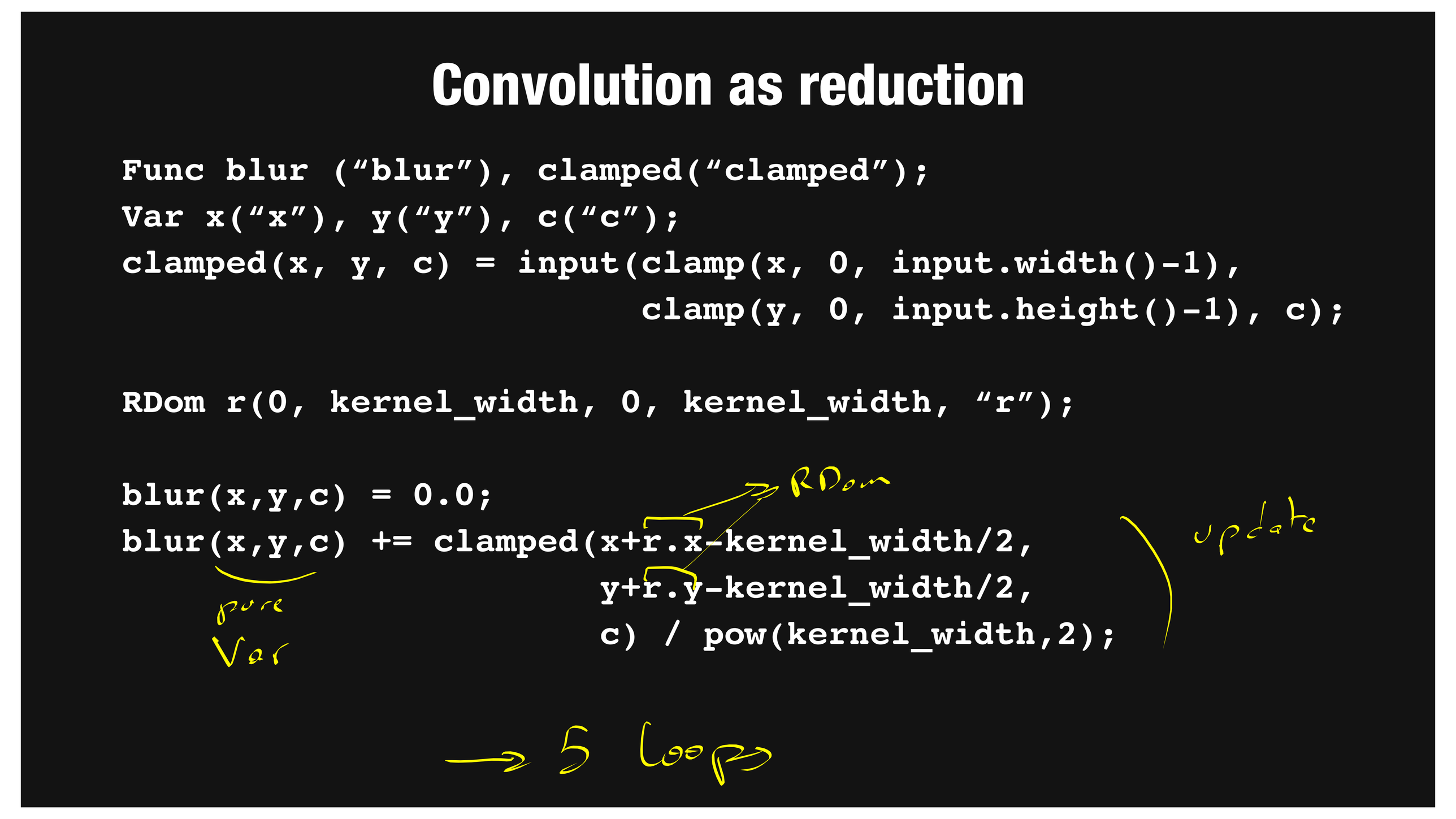
Here, reduction over 2 of the 3 dimensions

Convolution as reduction

```
Func blur (“blur”), clamped(“clamped”);
Var x(“x”), y(“y”), c(“c”);
clamped(x, y, c) = input(clamp(x, 0, input.width()-1),
                         clamp(y, 0, input.height()-1), c);

RDom r(0, kernel_width, 0, kernel_width, “r”);

blur(x,y,c) = 0.0;
blur(x,y,c) += clamped(x+r.x-kernel_width/2,
                       y+r.y-kernel_width/2,
                       c) / pow(kernel_width,2);
```



Note

Again, no loop!

Here, reduction over 2 of the 3 dimensions

Equivalent

```
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+rx kernel_width/2,
                                             y+ry-kernel_width/2,
                                             c] / kernel_width**2
```

init

RDom

Pure Var

update

Convolution as reduction - alternative

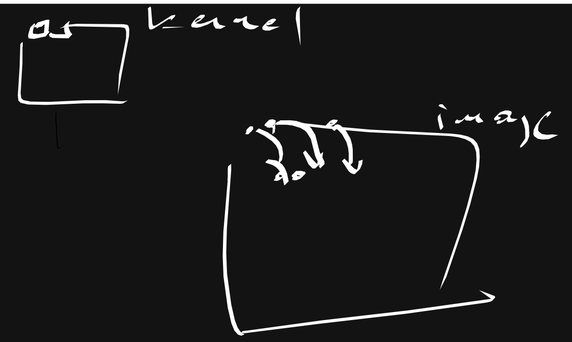
```
Func blur (“blur”), clamped(“clamped”);
Var x(“x”), y(“y”), c(“c”);
clamped(x, y, c) = input(clamp(x, 0, input.width()-1),
                         clamp(y, 0, input.height()-1), c);

RDom r(0, kernel_width, 0, kernel_width, “r”);

blur(x,y,c) = 0.0;

blur(x,y,c) += clamped(x+r.x-kernel_width/2,
                       y+r.y-kernel_width/2,
                       c);
blur(x,y,c) /= pow(kernel_width,2);
```

Loop 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+rx kernel_width/2,
                                             y+ry-kernel_width/2,
                                             c] / kernel_width**2
```

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+rx kernel_width/2,
                                             y+ry-kernel_width/2,
                                             c] / kernel_width**2
```

Reduction loops are always outside

Because otherwise the update semantics might be changed

Can result in very bad locality for convolution

Cannot always be reordered directly (semantics should not change)

But there is a trick

Inline/helper trick

Trick to make Halide do the right thing with reductions

I’ll show you how it works

Then I’ll show you a sugar that hides it from you

The helper/inline trick

same code as before

```
Func blur (“blur”), clamped(“clamped”);
Var x(“x”), y(“y”), c(“c”);
clamped(x, y, c) = input(clamp(x, 0, input.width()-1),
                         clamp(y, 0, input.height()-1), c);

RDom r(0, kernel_width, 0, kernel_width, “r”);

blur(x,y,c) = 0.0;
blur(x,y,c) += clamped(x+r.x-kernel_width/2,
                       y+r.y-kernel_width/2,
                       c) / pow(kernel_width,2);
```

dummy

```
Func superBlur(“superBlur”);
superBlur(x,y,c)=blur(x,y,c)
```

Trick (default) schedule equivalent

produce:

```
RDom r(0, kernel_width, 0, kernel_width, 'r');
blur(x,y,c) = 0.0;
blur(x,y,c) += clamped(x+r.x-kernel_width/2, y+r.y-kernel_width/2,  c) /
pow(kernel_width,2);
```

simplified version of code

→ inline

task [`superBlur(x,y,c)=blur(x,y,c)` output, final consumer → root

```
for y
    for x
        for c
            // then produce all need values of produce
            // for 1 consumer at x, y, c
            // i.e one value of blur at x, y, c
```

produce

```
RDom r(0, kernel_width, 0, kernel_width, 'r');
blur(x,y,c) = 0.0;
blur(x,y,c) += clamped(x+r.x-kernel_width/2, y+r.y-kernel_width/2,  c) /
pow(kernel_width,2);
```

simple version of code

→ inline

finish `superBlur(x,y,c)=blur(x,y,c)` output, final consumer → root

```
for y
    for x
        for c
            // here produce all need values of produce
            // for 1 consumer at x,y,c
            // i.e one value of blur at x,y,c
```

```
for r.y in (0...kernel_w)
    for r.x (0...kernel_w)
```

produce

```
RDom r(0, kernel_width, 0, kernel_width, 'r');
blur(x,y,c) = 0.0;
blur(x,y,c) += clamped(x+r.x-kernel_width/2, y+r.y-kernel_width/2,  c) /
pow(kernel_width,2);
```

vectorize of code

→ inline

```
func [ superBlur(x,y,c)=blur(x,y,c)
```

output, final consumer → root

```
for y
    for x
        for c
            // now produce all need values of produce
            // for 1 consumer at x, y, c
            // i.e one value of blur at x, y, c
```

```
for r.y in (0 ... kernel_w)
    for r.x (0 ... kernel w)
        for y' in (y, y)
            for x' in (x, x)
```

```
    pow(kernel_width,2);
id [ superBlur(x,y,c)=blur(x,y,c)
```

output final consumer → root

```
for y
    for x
        for c
            // then produce all need values of produce
            // for 1 consumer at x, y, c
            // i.e one value of blur at x, y, c
```

```
for r.y in (0 ... kernel_w)
    for r.x in (0 ... kernel w)
        for y' in (y, y)
            for x' in (x, x)
                for c' in (c, 0)
```

```
superBlur(x,y,c)=blur(x,y,c)
```

output: final consumer → root

```
for y
    for x
        for c
            // then produce all need values of producer
            // for 1 consumer at x,y,c
            // i.e one value of blur at x,y,c
```

```
for ry in (0 ... kernel_w)
    for rx in (0 ... kernel_w)
        for y' in (y,y)
            for x' in (x,x)
                for c' in (c,c)
```

```
for y
    for x
        for c
            // Now produce all need values of producer
            // for 1 consumer at x, y, c
            // i.e one value of blur at x, y, c
```

```
for ry in (0 ... kernel_w)
    for rx in (0 ... kernel_w)
        for y' in (y, y)
            for x' in (x, x)
                for c' in (c, c)
```

```
for y
    for x
        for c
            // Now produce all need values of producer
            // for 1 consumer at x, y, c
            // i.e one value of blur at x, y, c
```

```
for ry in (0 ... kernel_w)
    for rx in (0 ... kernel_w)
        for y' in (y, y)
            for x' in (x, x)
                for c' in (c, c)
```

```
for y
    for x
        for c
            // then produce all need values of producer
            // for 1 consumer at x, y, c
            // i.e one value of blur at x, y, c
            for y' in y, y
                for x' in x, x
                    for c' in c, c
                        blur(x', y', c') = 0

for ry in (0 ... kernel_w)
    for rx in (0 ... kernel_w)
        for y' in (y, y)
            for x' in (x, x)
                for c' in (c, c)
                    accumulate
```

Equivalent

```
for y in xrange(input.height()):
    for x in xrange(input.width()):
        for c in xrange(input.channels):
```

Python equivalent

```
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+rx kernel_width/2,
                                                            y+ry-kernel_width/2,
                                                            c] / kernel_width**2

            superBlur[x,y,c]=tmp[0,0,0]
```

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):



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



                                tmp          +=clampedInput[x+rx kernel_width/2,
                                                            y+ry-kernel_width/2,
                                                            c] / kernel_width**2
            superBlur[x,y,c]=tmp
```

Rule of thumb

For stencil reduction

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

Use the helper/inline trick

Encapsulate into an extra Func scheduled as default (inline)

In general, true for reduction where the free variables are independent of the reduction variables.

Sugar: sum (includes helper/inline trick)

```
Func blur("blur");
Var x("x"), y("y"), c("c");
Func clamped("clamped");
clamped(x, y, c) = input(clamp(x, 0, input.width()-1),
                         clamp(y, 0, input.height()-1), c);

RDom r(0, kernel_width, 0, kernel_width, "r");

blur(x,y,c) = sum(clamped(x+r.x-kernel_width/2, y, c));
```

y + r.y - kernel_width/2

Under the hood: initializes, creates a helper/inline

For hardcore Halide

(beyond the scope of this class & a little dirty)

↳ conflates schedule & algorithm

`unroll_sum`

does what it says

Computing the kernel

for this class, I advise you do it in C++ and pass it to Halide as an image (possibly of size 1xkernel_size)

Could be done in Halide

how should it be scheduled?

compute_root()

More complex reductions

Bilateral grid

Output over : grid

Reduction over : input pixels

Update equation: grid(r.x, r.y, intensity(r.x, r.y)) += intensity(r.x, r.y)

% grid construction

+ probably downsampling

More complex reductions

Bilateral grid

Output over grid cells x, y, I

Reduction over input image x,y

Update equation: grid(rx,ry, intensity(rx,ry))+=intensity(rx,ry);
(plus another one for sum of weights)

Note how the left-hand side is an Expr of the input image

Only case where LHS can have something else than Var to index Funcs

Compilation

We'll just JIT things (just in time)

- your code calls the Halide library when it runs.
- the Halide library includes a full compiler (LLVM) and generates the binary on the fly

Alternative: traditional static compilation

- compile_to_file
- useful to ship code, slightly more work to do the bindings .

Other goodies

store_at

GPU specific (CUDA, OpenCL)

tends to be faster

gpu_threads, gpu_blocks, gpu_tile

tuples

extern

parameters