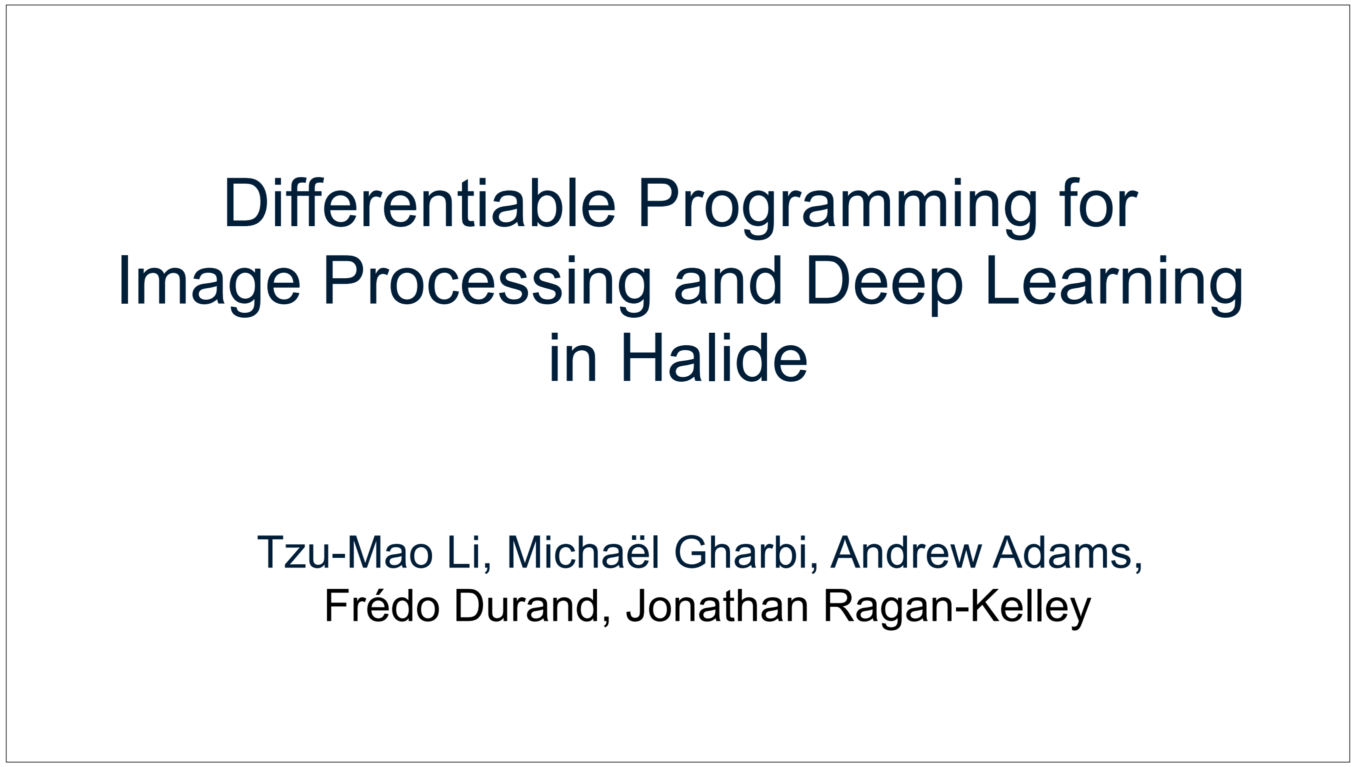
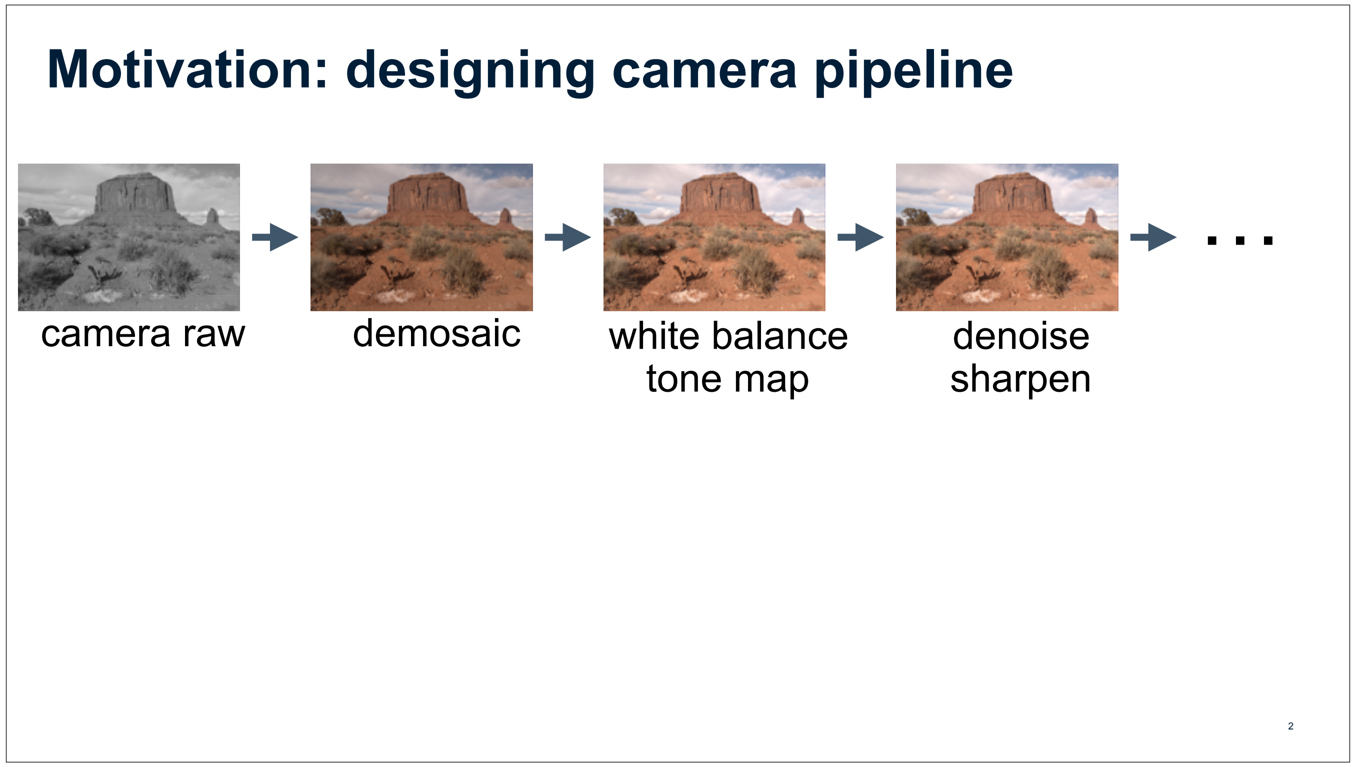
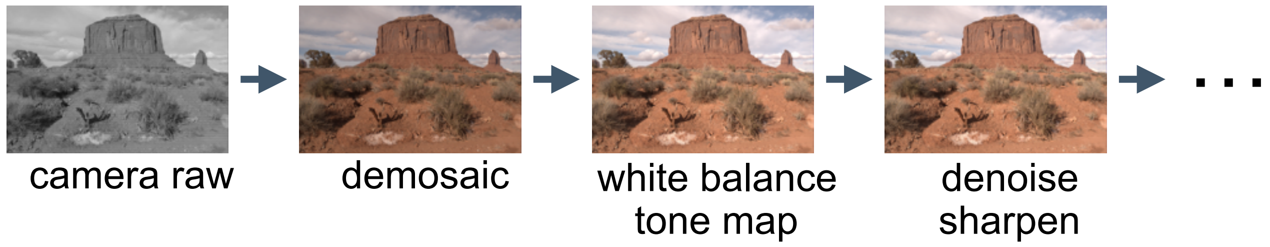


Differentiable Programming for Image Processing and Deep Learning in Halide

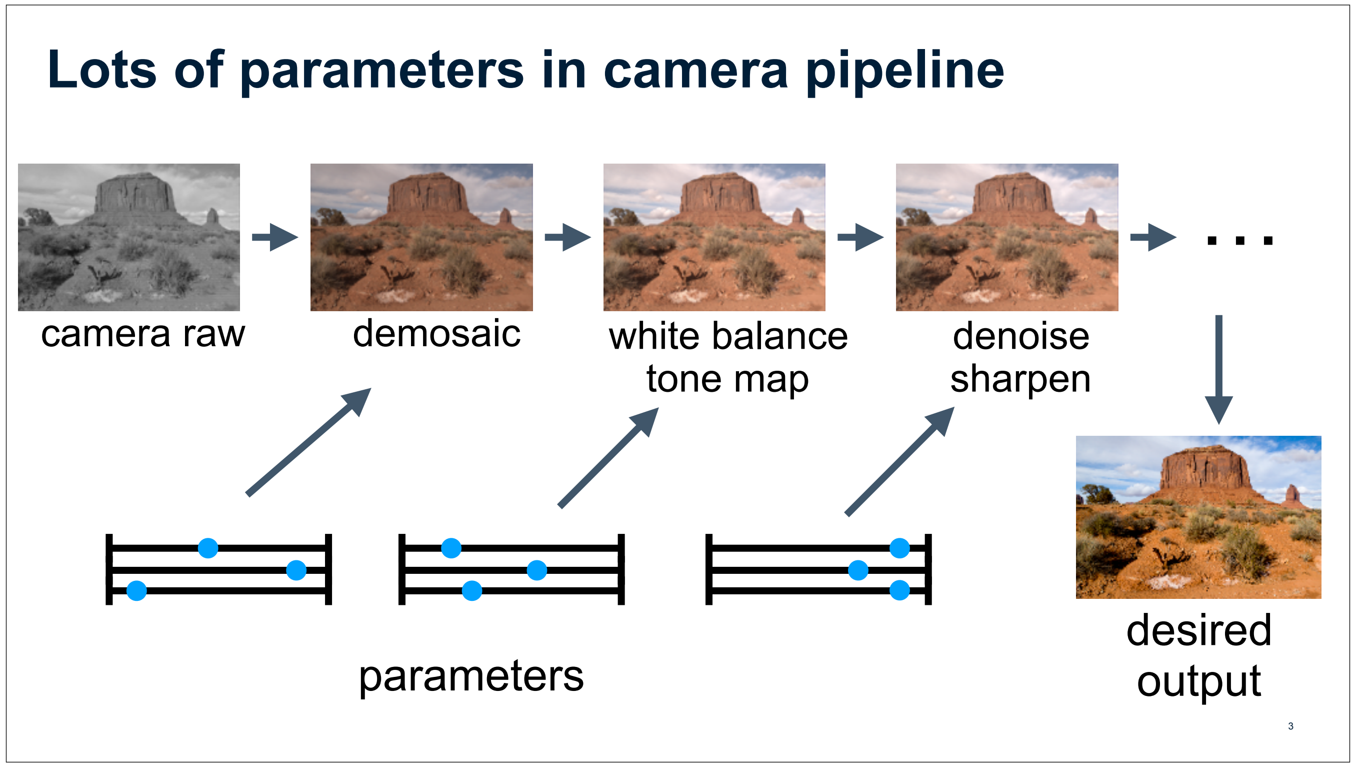
Tzu-Mao Li, Michaël Gharbi, Andrew Adams,
Frédo Durand, Jonathan Ragan-Kelley



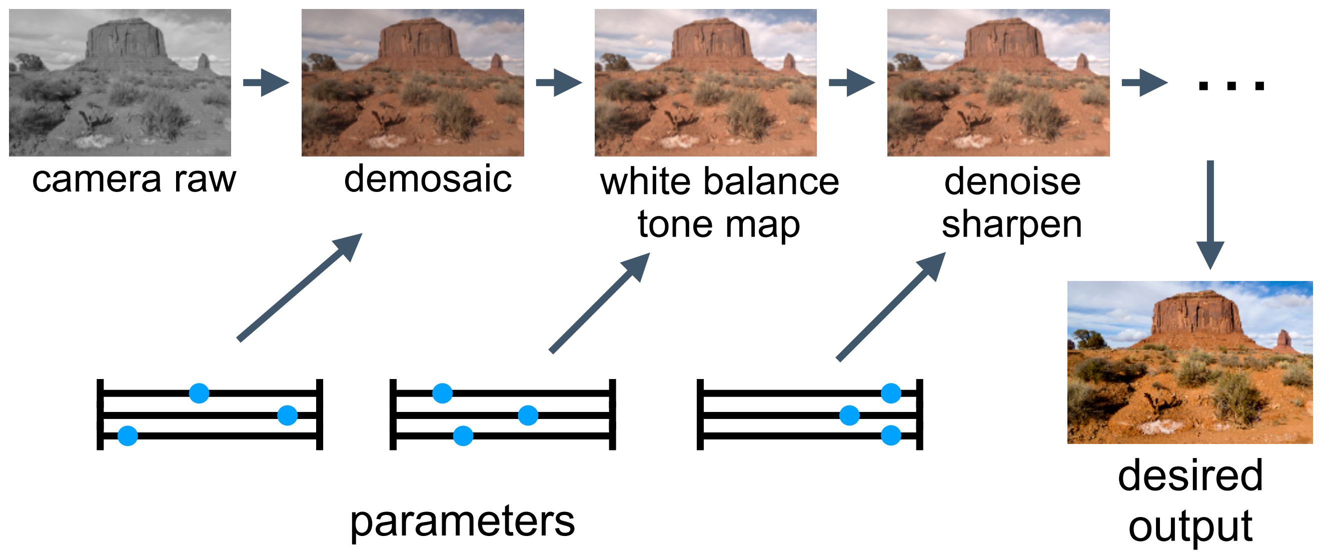
Motivation: designing camera pipeline



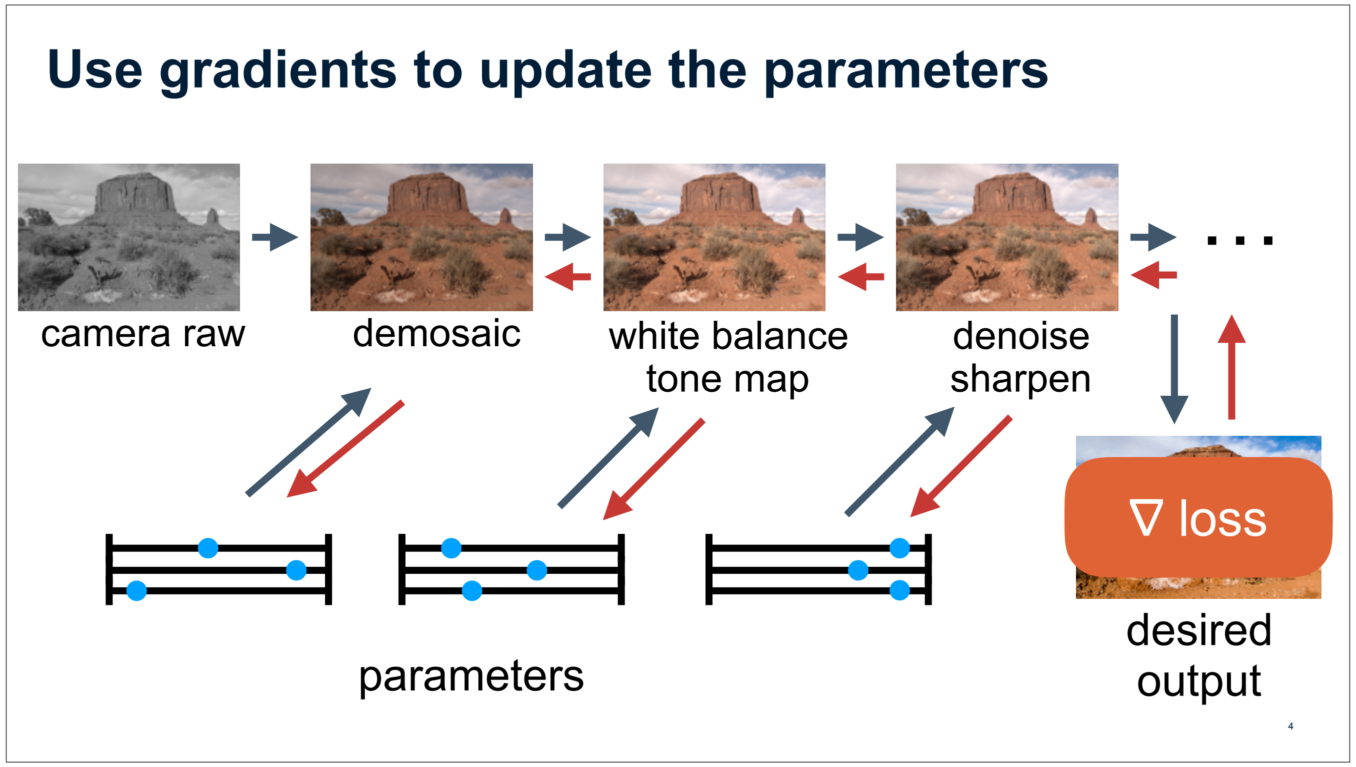
Modern camera pipelines contain many post-processing steps after the photons are recorded in the camera sensor.



Lots of parameters in camera pipeline



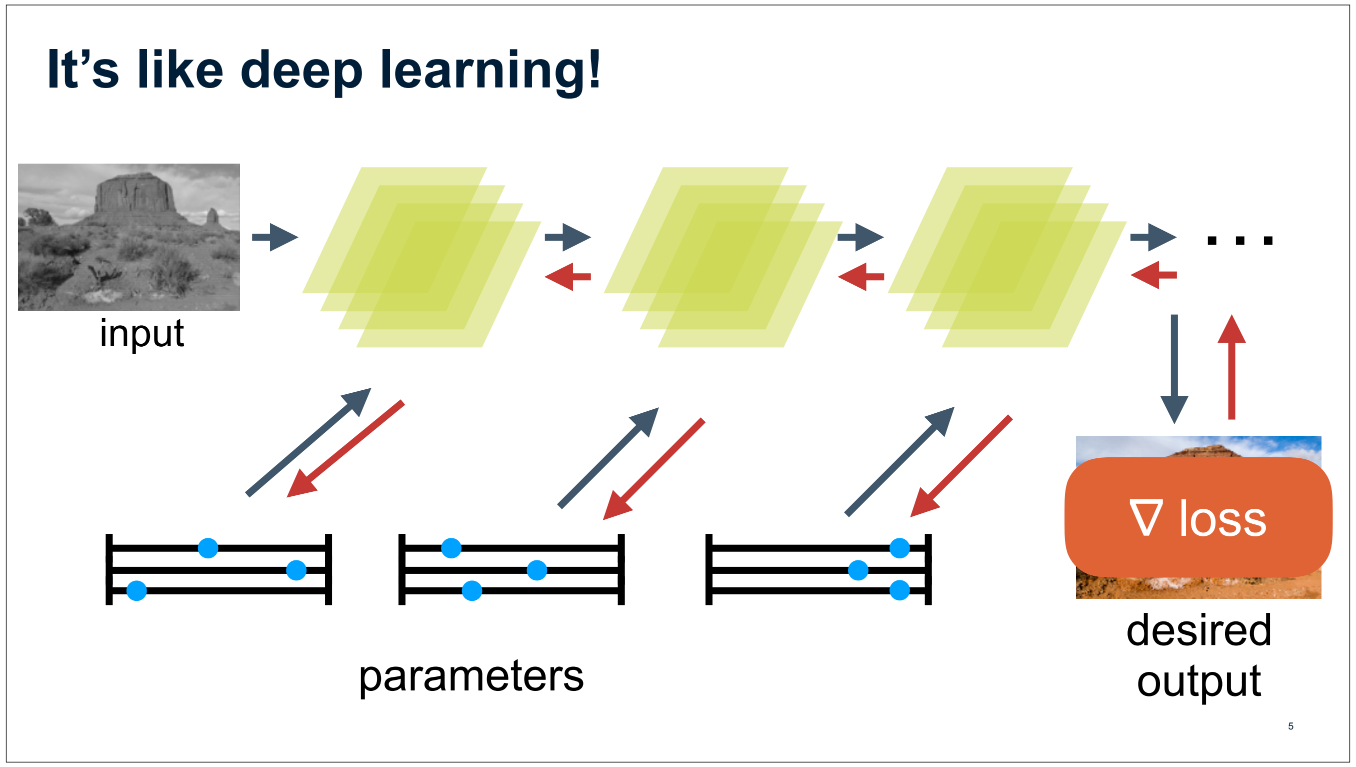
Each step of the post processing involves quite a few parameters to tune.
Like, which filters you use for demosaicing, what is the tone mapping curve, etc
What people often do is to prepare a set of test images,
and manually adjust the parameters until the images look good to them.



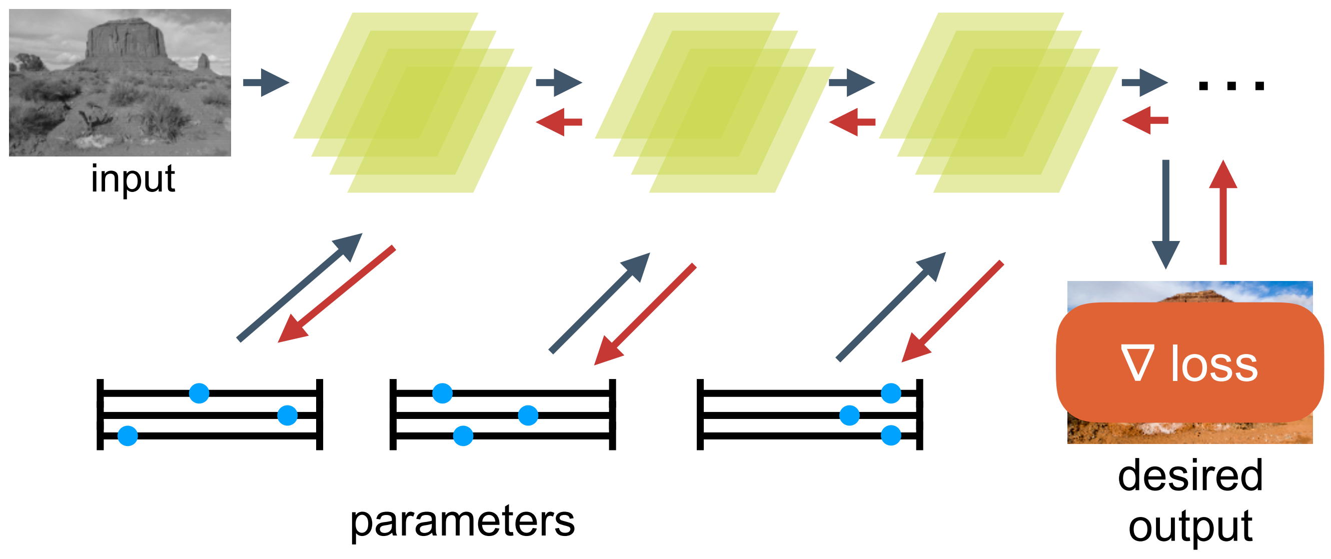
Use gradients to update the parameters

A more automatic way to do this is to define a loss function, and use gradient descent to update all parameters.

If you think about it, you are optimizing some highly parametrized functions using gradient descent…

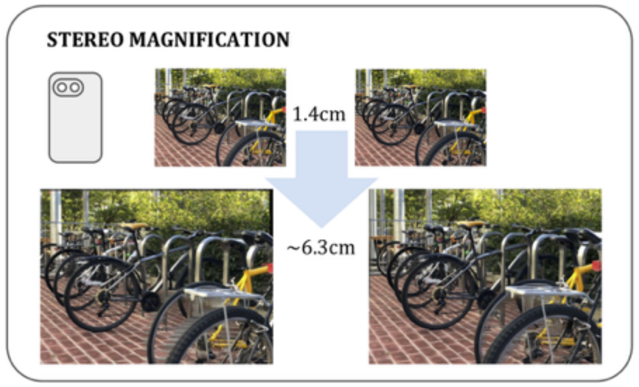


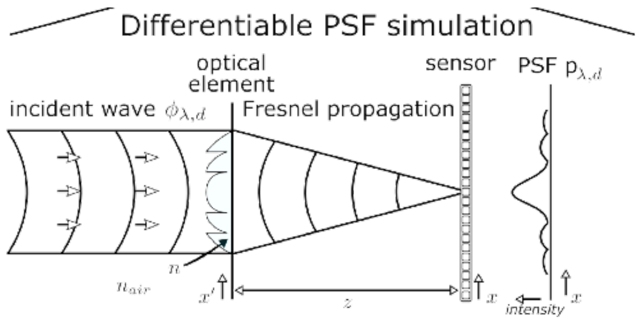
It’s like deep learning!



so it’s pretty much like deep learning, and all the stages are your “layers”.

Differentiable image processing @ SIGGRAPH 2018



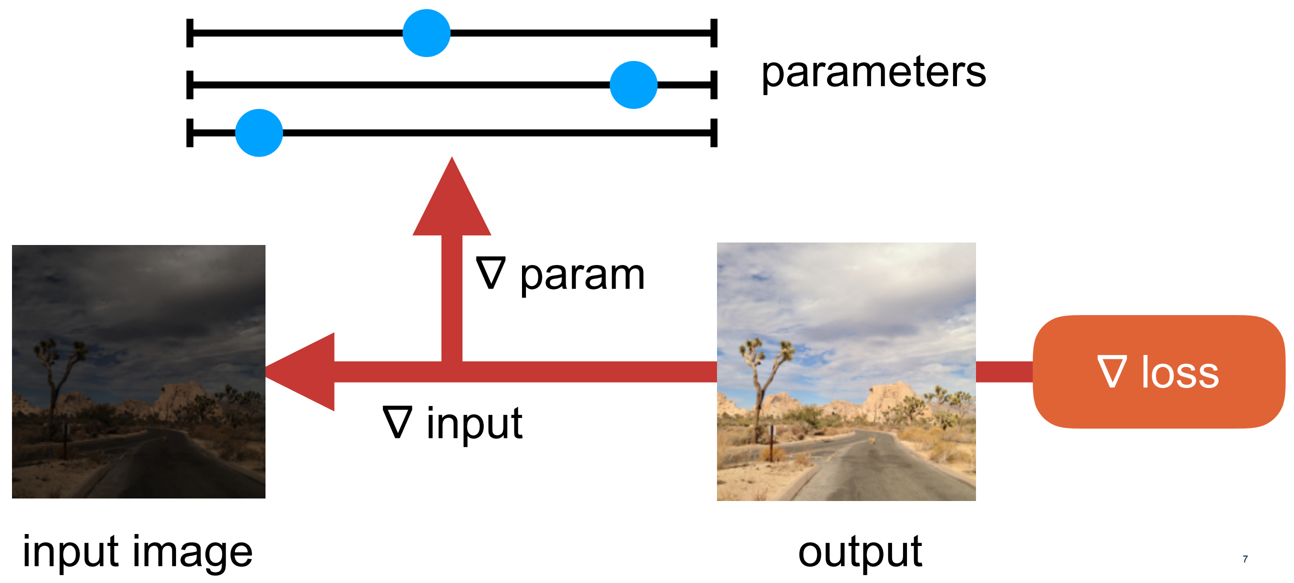


Zhou et al.
stereo magnification
differentiable rendering layer

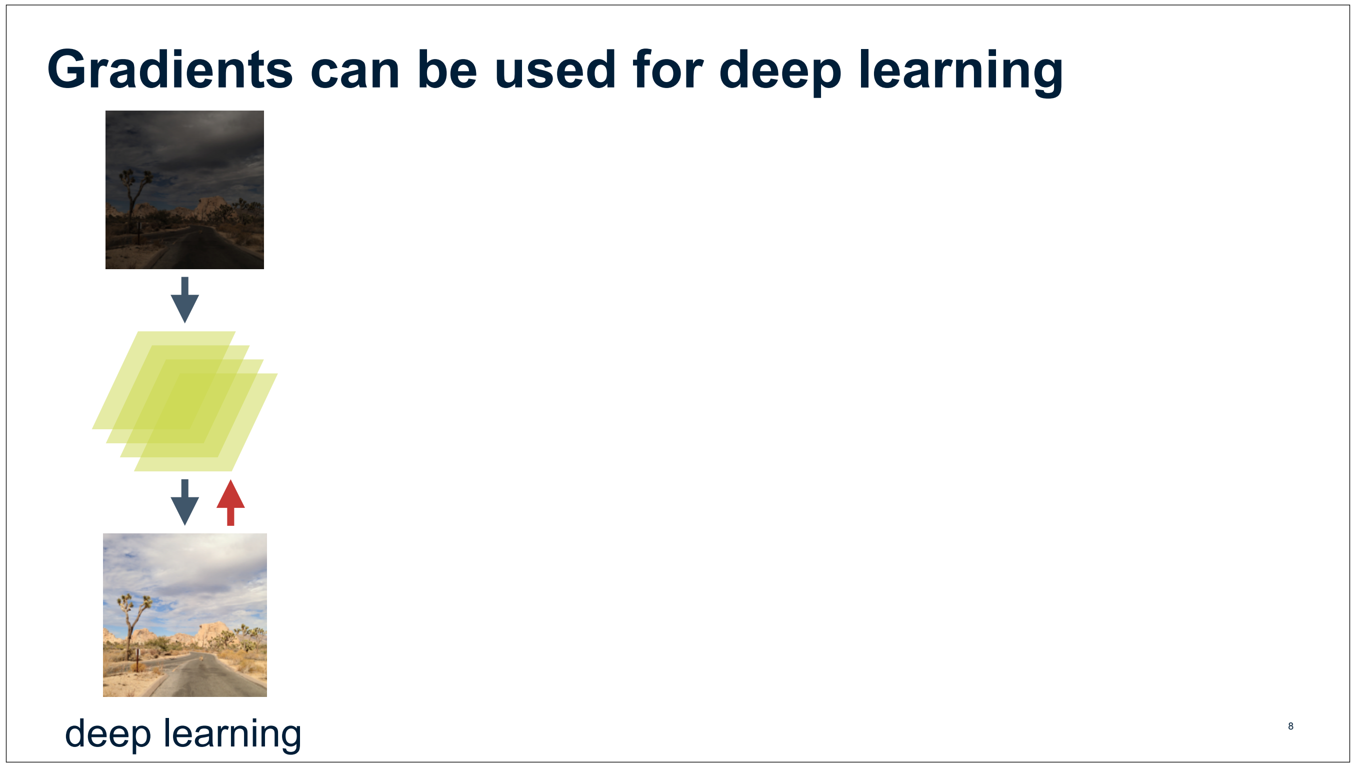
Sitzmann et al.
end-to-end optics design
differentiable optics

We are seeing an emerging new paradigm for differentiable image processing in this year SIGGRAPH.
People are starting to design their own end-to-end differentiable
pipelines,
like Zhou et al.’s stereo magnification work, or Sitzmann et al.’s work on end-to-end optics design.

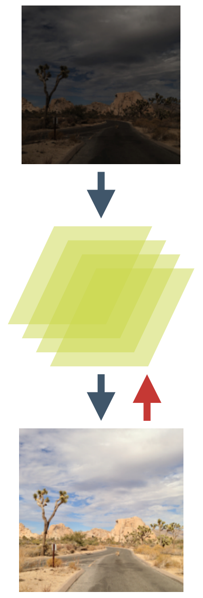
Differentiable image processing



Usually you would define a loss function for outputs from a pipeline.
The example we just showed use gradients to update the parameters to minimize the loss, but you can also update the input image to minimize the loss, solving an inverse problem.



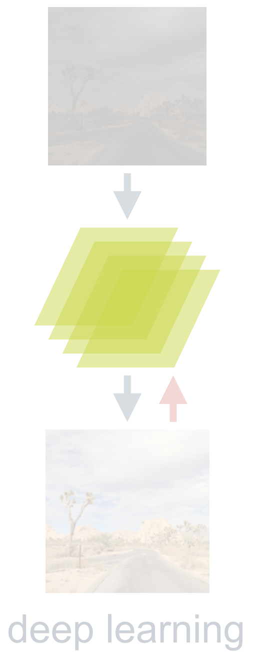
Gradients can be used for deep learning

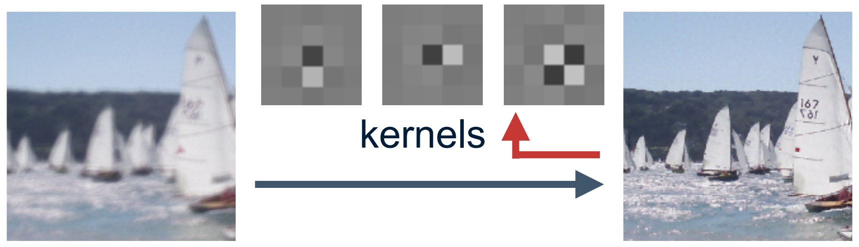


deep learning

So deep learning heavily uses gradients, obviously.

Gradients can be used for tuning existing pipelines

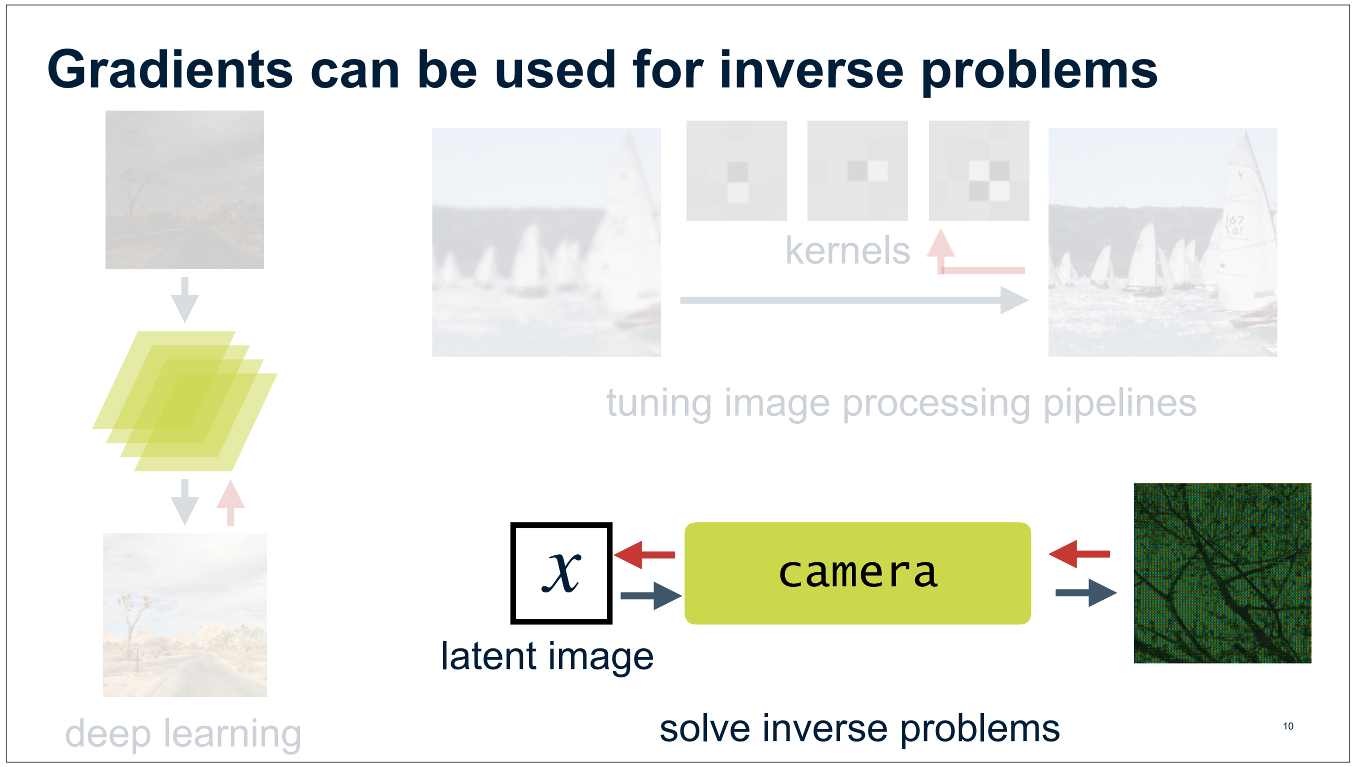




tuning image processing pipelines

But even more, you can also learn any differentiable function that is more than just a convolutional neural network.
For example you can optimize for the kernels or other parameters used in an image processing pipeline.

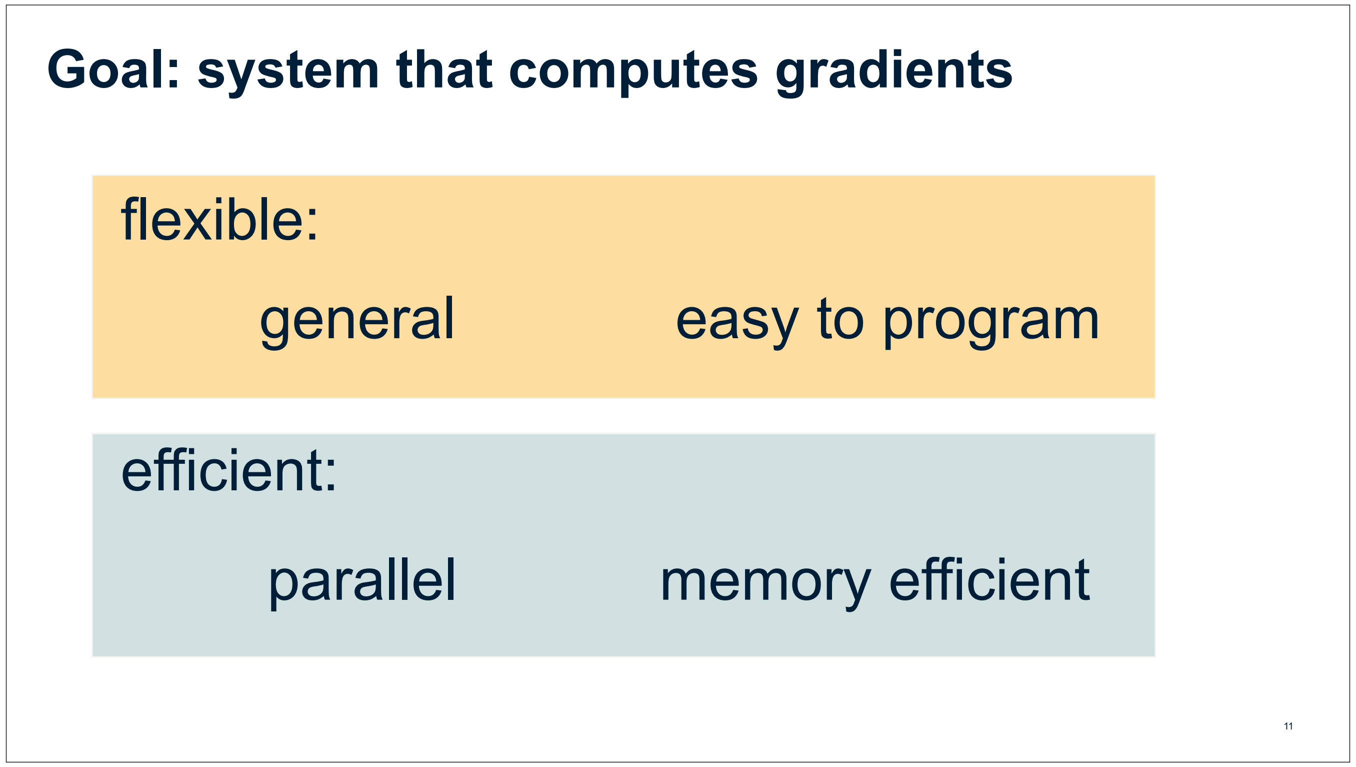
An important message we want to deliver today is that you shouldn’t draw a thick line between deep learning and traditional image processing.



Gradients can be used for inverse problems

Another application of gradient is you can use it for solving inverse problems by figuring out the latent input of a forward model.
You would model, for example, how a camera record and degrade an image, and try to solve for the original.
No training data is involved in this case.

I’ll get back to these applications later.



Goal: system that computes gradients

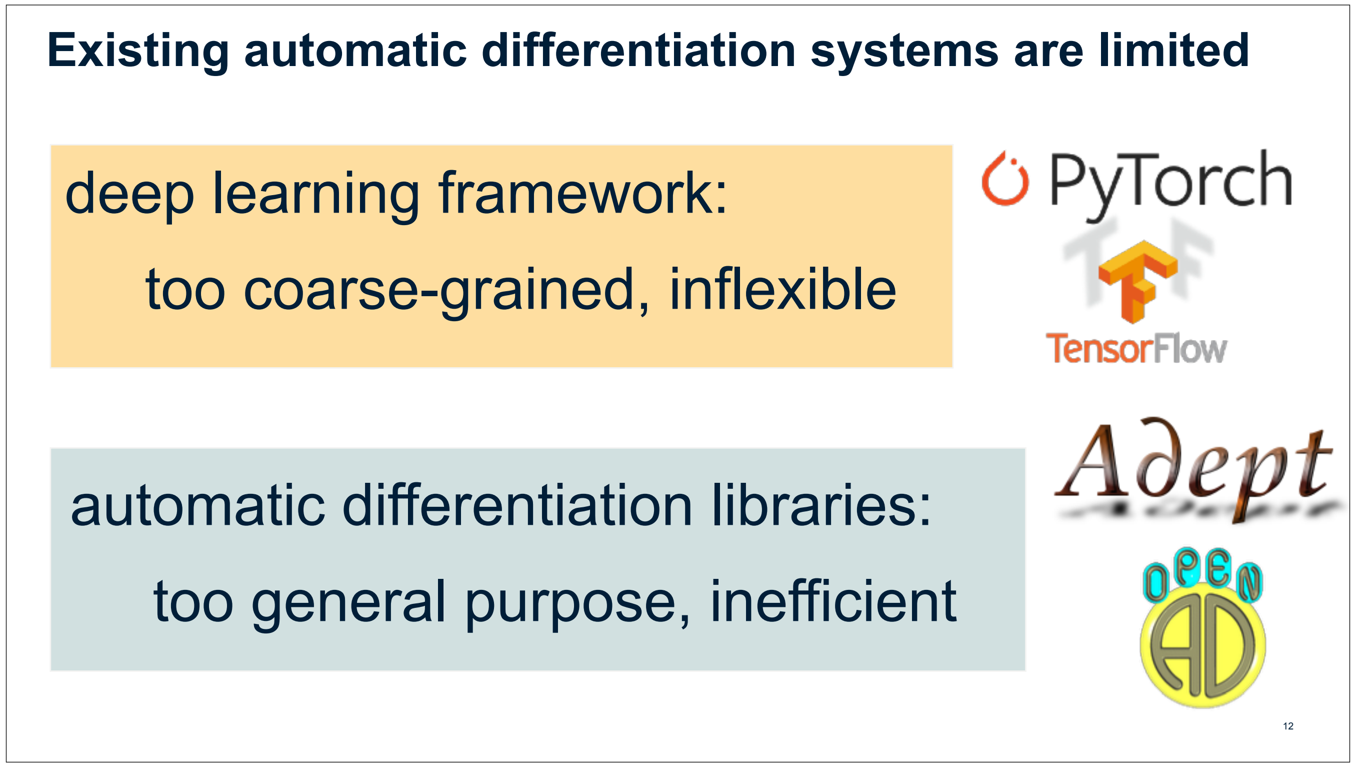
flexible:

general | easy to program

efficient:

parallel | memory efficient

All these applications require the gradients of a pipeline.
Our goal is to develop a system that computes gradients, such that the system is flexible and efficient.
We want the system to be general so that it supports arbitrary image processing pipelines you can come up with, but we also want it to be easy to program.
We want the system to be efficient, because we want short training time for fast iteration of ideas.
It means we want to maximize parallelism and be memory efficient.



Existing automatic differentiation systems are limited

deep learning framework:

too coarse-grained, inflexible

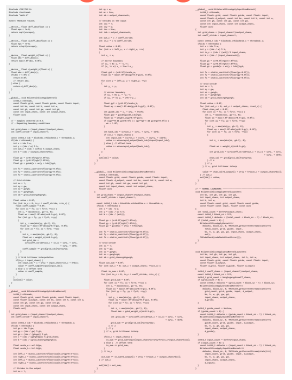
automatic differentiation libraries:

too general purpose, inefficient

There are existing systems that compute gradients, but they do not meet the criteria we listed.
Deep learning frameworks like PyTorch or Tensorflow are composed of coarse-grained operators, where operators are written by experts, like the 14 specialized implementation of convolution in Tensorflow.
It is often awkward to use these operators to assemble something new for research.
There are other more traditional automatic differentiation libraries that transform your C++ or Python code.
The problem with these libraries is that they are too general purpose, which means they don’t have the domain specific knowledge to speed up the code, so it’s difficult to make them efficient.

Developing custom operators is tedious

red region: gradient code



CUDA 308 lines

derive the gradient (manually)

implement

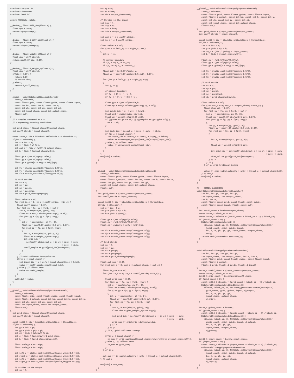
debug

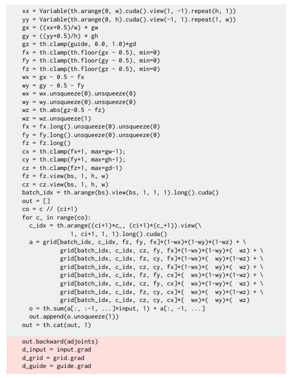
repeat

code from Gharbi et al. 2017

Here is a case study.
One of our coauthor developed his custom deep learning operator for his work last year at SIGGRAPH.
At that time Tensorflow lacked the necessary operators to implement his idea.
So he had to manually implement his custom operator in CUDA. This is bad because you have to derive the gradients either manually or in Mathematica, then write the code for the gradients and debug.
Once you change anything in the forward code you have to rederive and reimplement the gradients.

Deep learning frameworks are limited





```
xx = Variable(th.arange(0, w).cuda().view(1, -1).repeat(h, 1))
yy = Variable(th.arange(0, h).cuda().view(-1, 1).repeat(1, w))
gx = ((xx+0.5)/w) * gw
gy = ((yy+0.5)/h) * gh
gz = th.clamp(guide, 0.0, 1.0)*gd
fx = th.clamp(th.floor(gx - 0.5), min=0)
fy = th.clamp(th.floor(gy - 0.5), min=0)
fz = th.clamp(th.floor(gz - 0.5), min=0)
wx = gx - 0.5 - fx
wy = gy - 0.5 - fy
wx = wx.unsqueeze(0).unsqueeze(0)
wy = wy.unsqueeze(0).unsqueeze(0)
wz = th.abs(gz-0.5 - fz)
wz = wz.unsqueeze(1)
fx = fx.long().unsqueeze(0).unsqueeze(0)
fy = fy.long().unsqueeze(0).unsqueeze(0)
fz = fz.long()
cx = th.clamp(fx+1, max=gw-1);
cy = th.clamp(fy+1, max=gh-1);
cz = th.clamp(fz+1, max=gd-1)
fz = fz.view(bs, 1, h, w)
cz = cz.view(bs, 1, h, w)
batch_idx = th.arange(bs).view(bs, 1, 1, 1).long().cuda()
out = []
co = c // (ci+1)
for c_ in range(co):
  c_idx = th.arange((ci+1)*c_, (ci+1)*(c_+1)).view(\
            1, ci+1, 1, 1).long().cuda()
  a = grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, fy, fx]*(1-wx)*(1-wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, cy, fx]*(1-wx)*(  wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, fy, cx]*(  wx)*(1-wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, fy, cx]*(  wx)*(1-wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, cy, cx]*(  wx)*(  wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, cy, cx]*(  wx)*(  wy)*(  wz)
  o = th.sum(a[:, :-1, ...]*input, 1) + a[:, -1, ...]
  out.append(o.unsqueeze(1))
out = th.cat(out, 1)

out.backward(adjoints)
d_input = input.grad
d_grid = grid.grad
d_guide = guide.grad
```

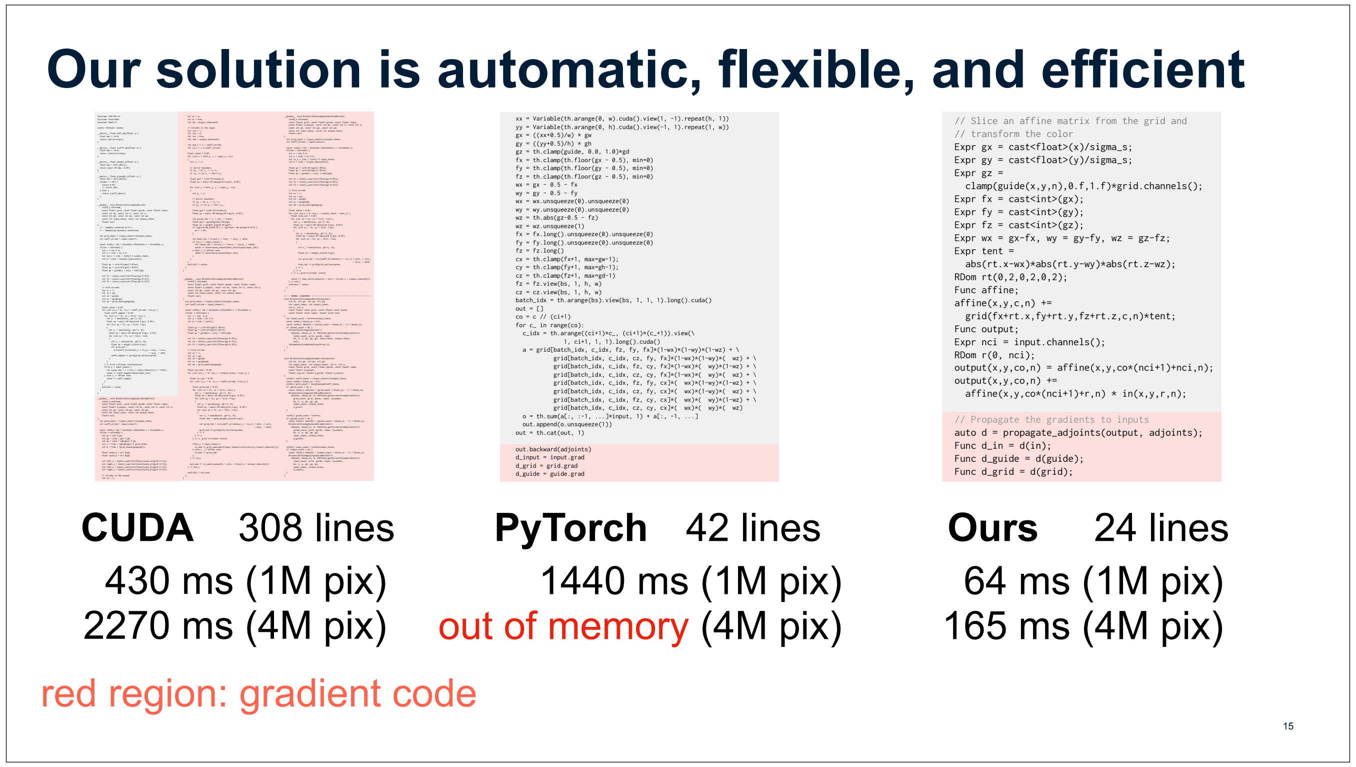
CUDA 308 lines
430 ms (1M pix)
2270 ms (4M pix)

PyTorch 42 lines
1440 ms (1M pix)
out of memory (4M pix)

- coarse-grained operators
- no control over performance

red region: gradient code

One year after the development of his operator, it is possible to implement the operators in both Tensorflow and PyTorch.
It is still a bit awkward to code it in PyTorch because everything is so coarse-grained.
Even worse, it is slower than the manually implemented version and ran out of memory when we feed a larger image to it.
And we have pretty much no control to this.



Our solution is automatic, flexible, and efficient

```
[illegible]
```

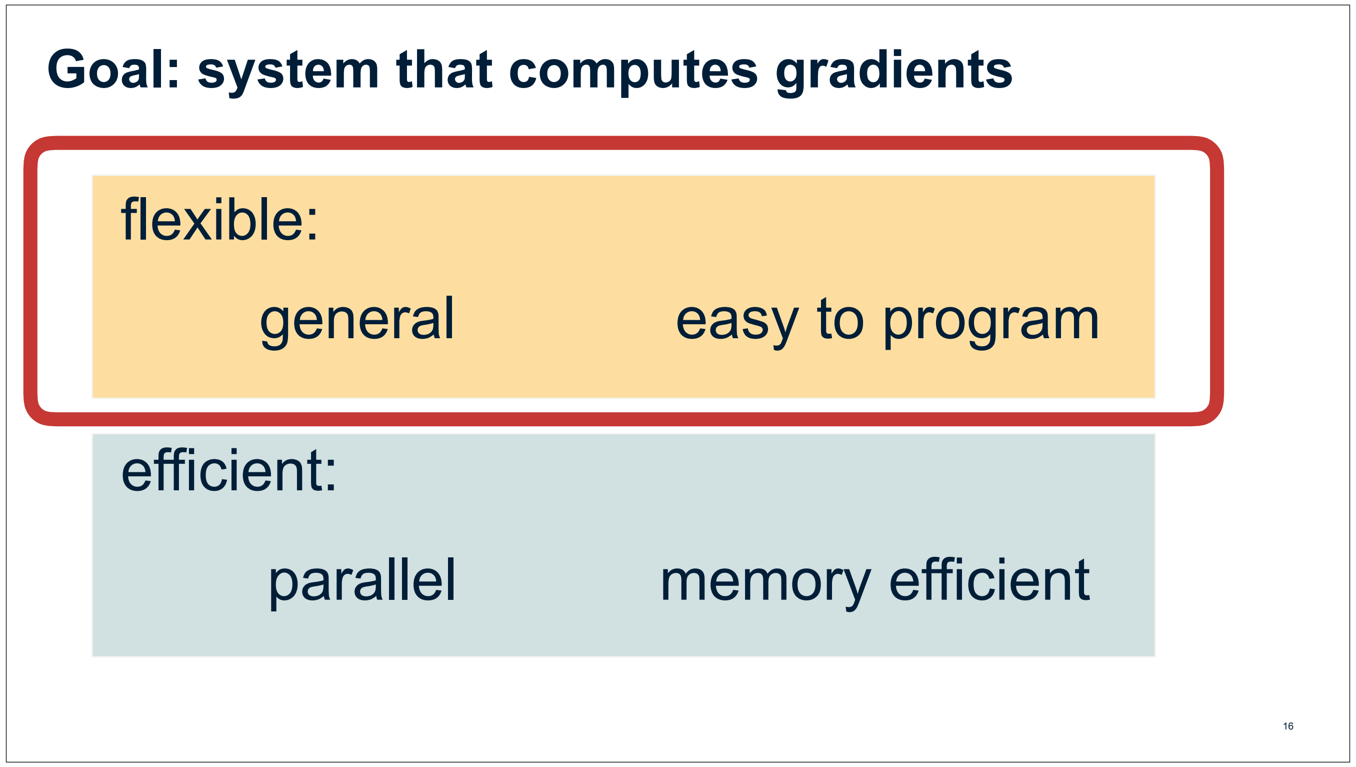
```
xx = Variable(th.arange(0, w).cuda().view(1, -1).repeat(h, 1))
yy = Variable(th.arange(0, h).cuda().view(-1, 1).repeat(1, w))
gx = ((xx+0.5)/w) * gw
gy = ((yy+0.5)/h) * gh
gz = th.clamp(guide, 0.0, 1.0)*gd
fx = th.clamp(th.floor(gx - 0.5), min=0)
fy = th.clamp(th.floor(gy - 0.5), min=0)
fz = th.clamp(th.floor(gz - 0.5), min=0)
wx = gx - 0.5 - fx
wy = gy - 0.5 - fy
wx = wx.unsqueeze(0).unsqueeze(0)
wy = wy.unsqueeze(0).unsqueeze(0)
wz = th.abs(gz-0.5 - fz)
wz = wz.unsqueeze(1)
fx = fx.long().unsqueeze(0).unsqueeze(0)
fy = fy.long().unsqueeze(0).unsqueeze(0)
fz = fz.long()
cx = th.clamp(fx+1, max=gw-1);
cy = th.clamp(fy+1, max=gh-1);
cz = th.clamp(fz+1, max=gd-1)
fz = fz.view(bs, 1, h, w)
cz = cz.view(bs, 1, h, w)
batch_idx = th.arange(bs).view(bs, 1, 1, 1).long().cuda()
out = []
co = c // (ci+1)
for c_ in range(co):
  c_idx = th.arange((ci+1)*c_, (ci+1)*(c_+1)).view(\
              1, ci+1, 1, 1).long().cuda()
  a = grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, fy, fx]*(1-wx)*(1-wy)*(  wz) + \
           grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, cy, fx]*(1-wx)*(  wy)*(  wz) + \
           grid[batch_idx, c_idx, fz, fy, cx]*(  wx)*(1-wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, fy, cx]*(  wx)*(1-wy)*(  wz) + \
           grid[batch_idx, c_idx, fz, cy, cx]*(  wx)*(  wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, cy, cx]*(  wx)*(  wy)*(  wz)
  o = th.sum(a[:, :-1, ...]*input, 1) + a[:, -1, ...]
  out.append(o.unsqueeze(1))
out = th.cat(out, 1)

out.backward(adjoints)
d_input = input.grad
d_grid = grid.grad
d_guide = guide.grad
```

```
// Slice an affine matrix from the grid and
// transform the color
Expr gx = cast<float>(x)/sigma_s;
Expr gy = cast<float>(y)/sigma_s;
Expr gz =
  clamp(guide(x,y,n),0.f,1.f)*grid.channels();
Expr fx = cast<int>(gx);
Expr fy = cast<int>(gy);
Expr fz = cast<int>(gz);
Expr wx = gx-fx, wy = gy-fy, wz = gz-fz;
Expr tent =
  abs(rt.x-wx)*abs(rt.y-wy)*abs(rt.z-wz);
RDom rt(0,2,0,2,0,2);
Func affine;
affine(x,y,c,n) +=
  grid(fx+rt.x,fy+rt.y,fz+rt.z,c,n)*tent;
Func output;
Expr nci = input.channels();
RDom r(0, nci);
output(x,y,co,n) = affine(x,y,co*(nci+1)+nci,n);
output(x,y,co,n) +=
  affine(x,y,co*(nci+1)+r,n) * in(x,y,r,n);

// Propagate the gradients to inputs
auto d = propagate_adjoints(output, adjoints);
Func d_in = d(in);
Func d_guide = d(guide);
Func d_grid = d(grid);
```

CUDA 308 lines	**PyTorch** 42 lines	**Ours** 24 lines
430 ms (1M pix)	1440 ms (1M pix)	64 ms (1M pix)
2270 ms (4M pix)	out of memory (4M pix)	165 ms (4M pix)

red region: gradient code

We reimplement the same operator in our system. It only takes 24 lines and is much more efficient than both the manually written version and PyTorch. The manual version might not be the most optimized code but it already took the author considerable time to implement.



Goal: system that computes gradients

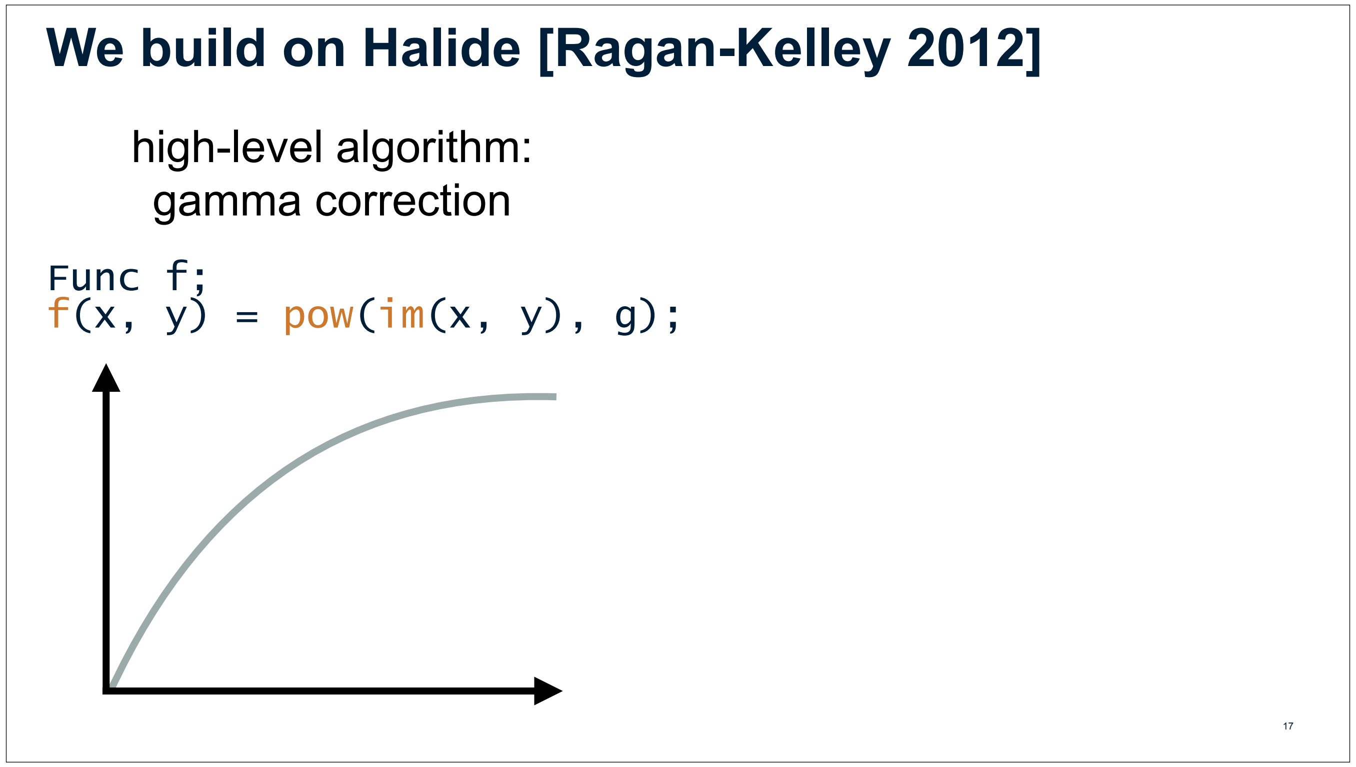
flexible:

general easy to program

efficient:

parallel memory efficient

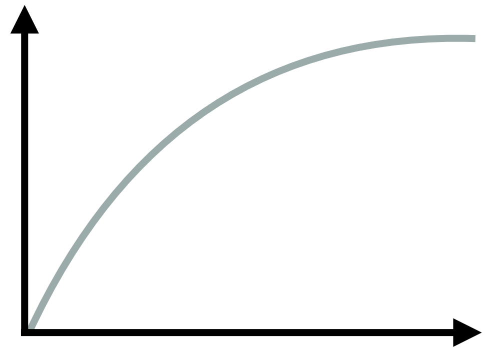
Let’s recap our goal again.
We want out system to be flexible and efficient.
For the flexibility we want the system to be general but easy to program, at least for all image processing tasks.



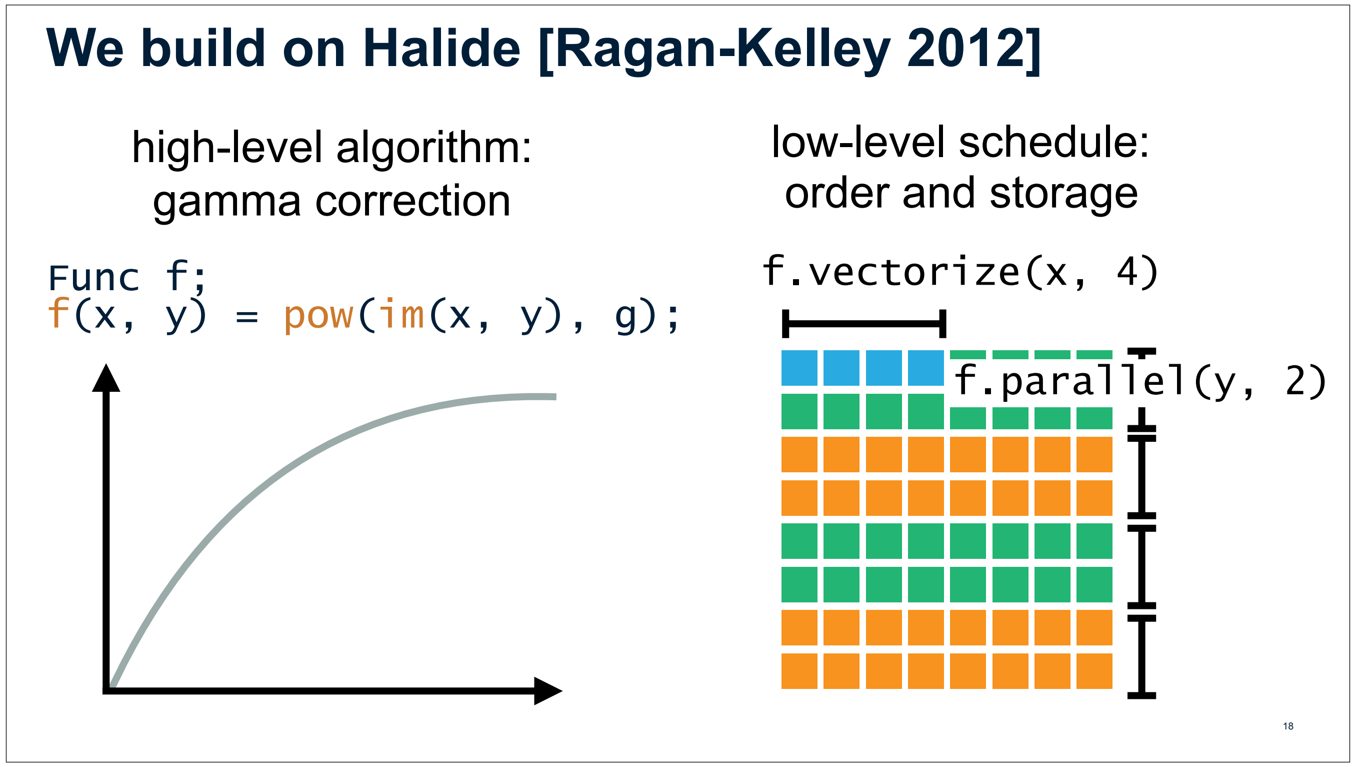
We build on Halide [Ragan-Kelley 2012]

high-level algorithm:
gamma correction

```
Func f;
f(x, y) = pow(im(x, y), g);
```



To achieve this we build on Halide, a programming language developed by a few of our coauthors, briefly mentioned in previous talk.
Halide achieves both generality and ease of programming by decomposing the code into two parts.
The first part is the high-level algorithm which is “what” you want to compute.



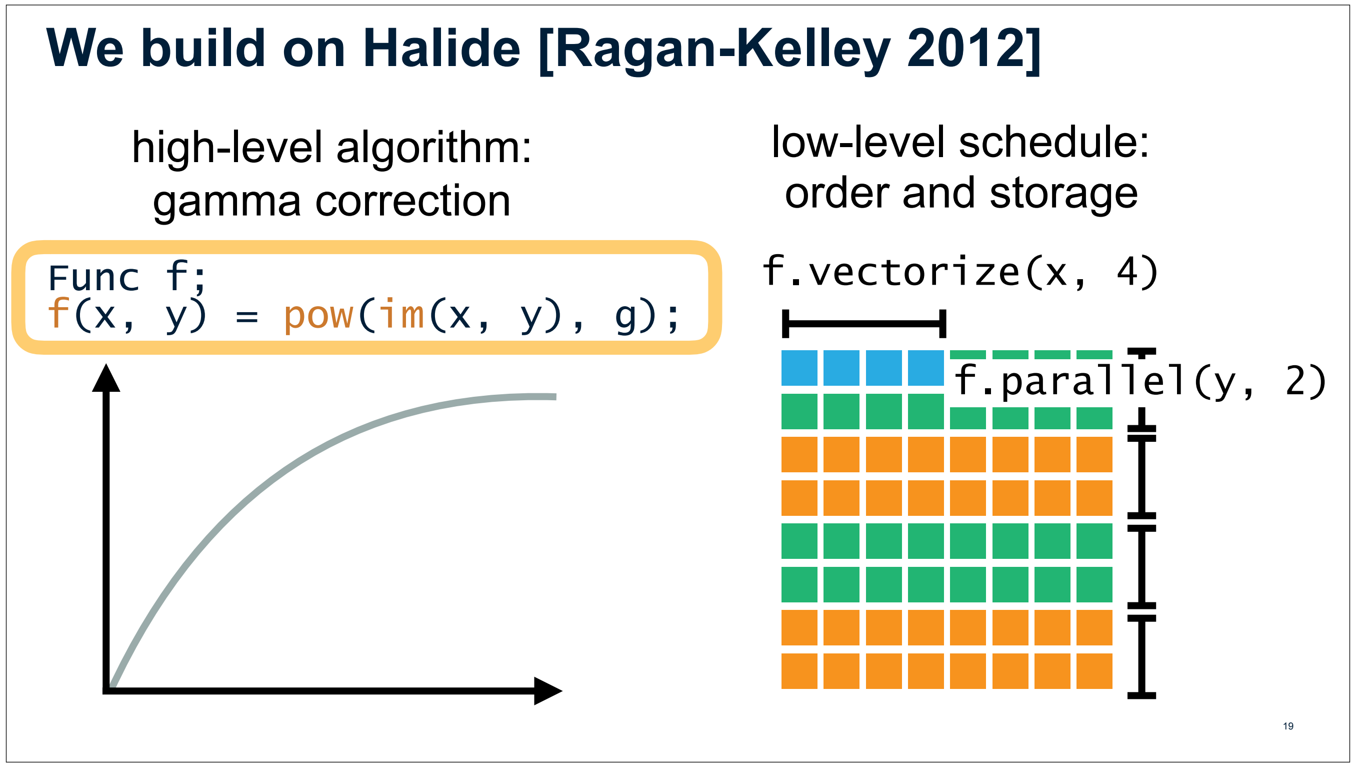
We build on Halide [Ragan-Kelley 2012]

high-level algorithm:
gamma correction

low-level schedule:
order and storage

```
Func f;
f(x, y) = pow(im(x, y), g);
```

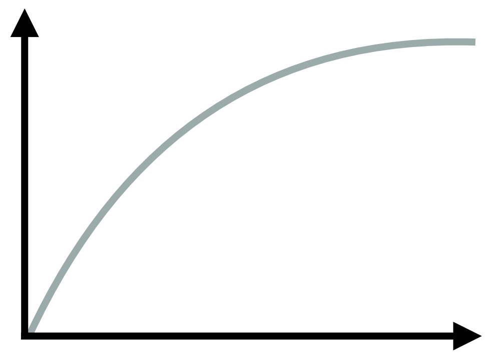
The second part is the lower-level schedule, which defines the order of computation and storage.



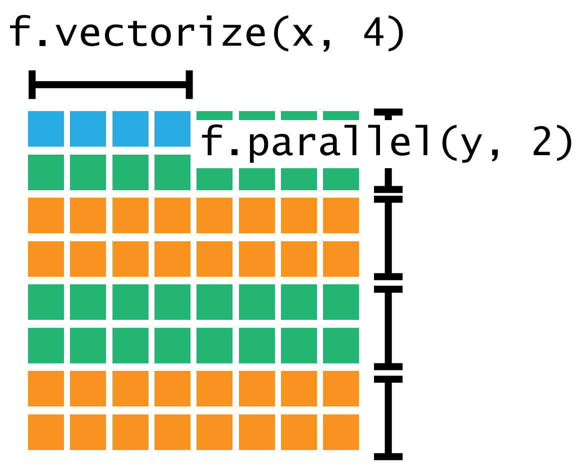
We build on Halide [Ragan-Kelley 2012]

high-level algorithm:
gamma correction

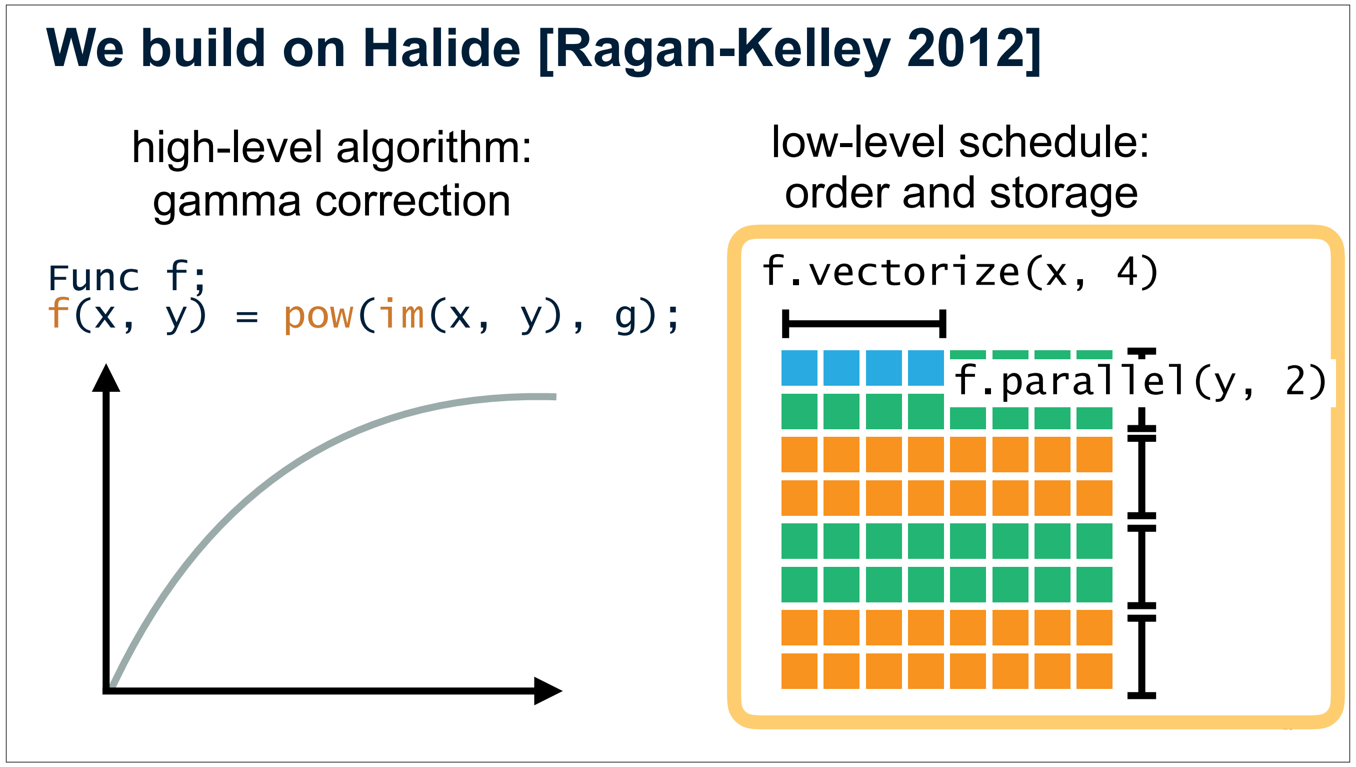
```
Func f;
f(x, y) = pow(im(x, y), g);
```



low-level schedule:
order and storage



For example for gamma correction, you would define a Halide function f and set each pixel to the power of g to the image.



We build on Halide [Ragan-Kelley 2012]

high-level algorithm:
gamma correction

```
Func f;
f(x, y) = pow(im(x, y), g);
```

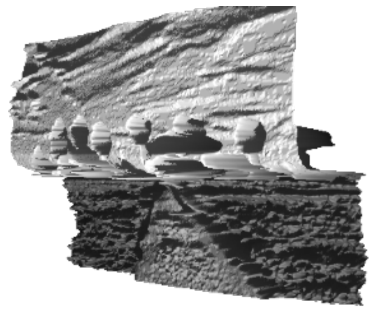
low-level schedule:
order and storage

```
f.vectorize(x, 4)
f.parallel(y, 2)
```

Then you would specify in the schedule how you want to compute f.
You can decide how you tile the domain for multi-threading or SIMD.
You can also decide whether to allocate storage for f, or just compute values of f on demand.

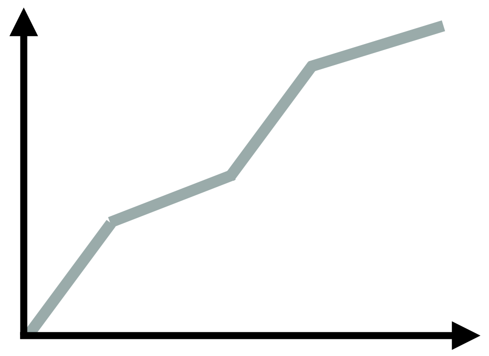
We can differentiate complex programs

- complex dependencies between pixels
  - e.g. edge aware filters
- deep learning frameworks are ill-suited
  - coarse-grained operators



A bunch of different image processing code is already implemented in Halide.
There are some edge aware filters like bilateral grid or local laplacian filters.
The code comparison I showed earlier from our co-author was one kind of edge-aware filter.
There’s also a full camera pipeline that transforms a camera raw image into a normal image.
Many of these applications have complex dependencies between pixels, which makes them either awkward or just impossible to implement in Tensorflow or PyTorch.
Usually in these frameworks you have to prepare a set of indices in numpy,
like which pixels are we gathering from or which pixels are we splatting to.
It is both difficult and inefficient doing things in this way.
With our framework you can get the gradients of all these applications for free.

We can differentiate through general programs

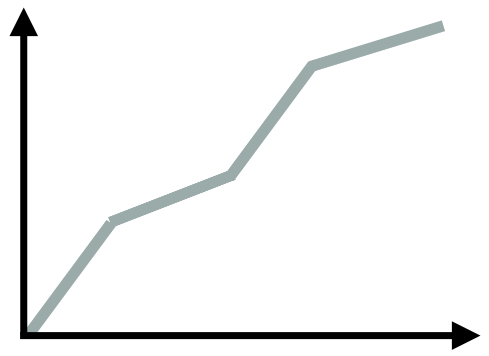


piecewise-linear tone mapping

I will give two simple examples which are difficult to express in deep learning frameworks.
The first example is we take the previous gamma correction example, but instead use a piecewise linear function.
This is very common in many tone mapping function.
Suddenly this becomes very difficult and inefficient in Tensorflow or Pytorch.
There are if/else operator in these frameworks but if you have many segments you'll have to stack up lots of if/else yourself.

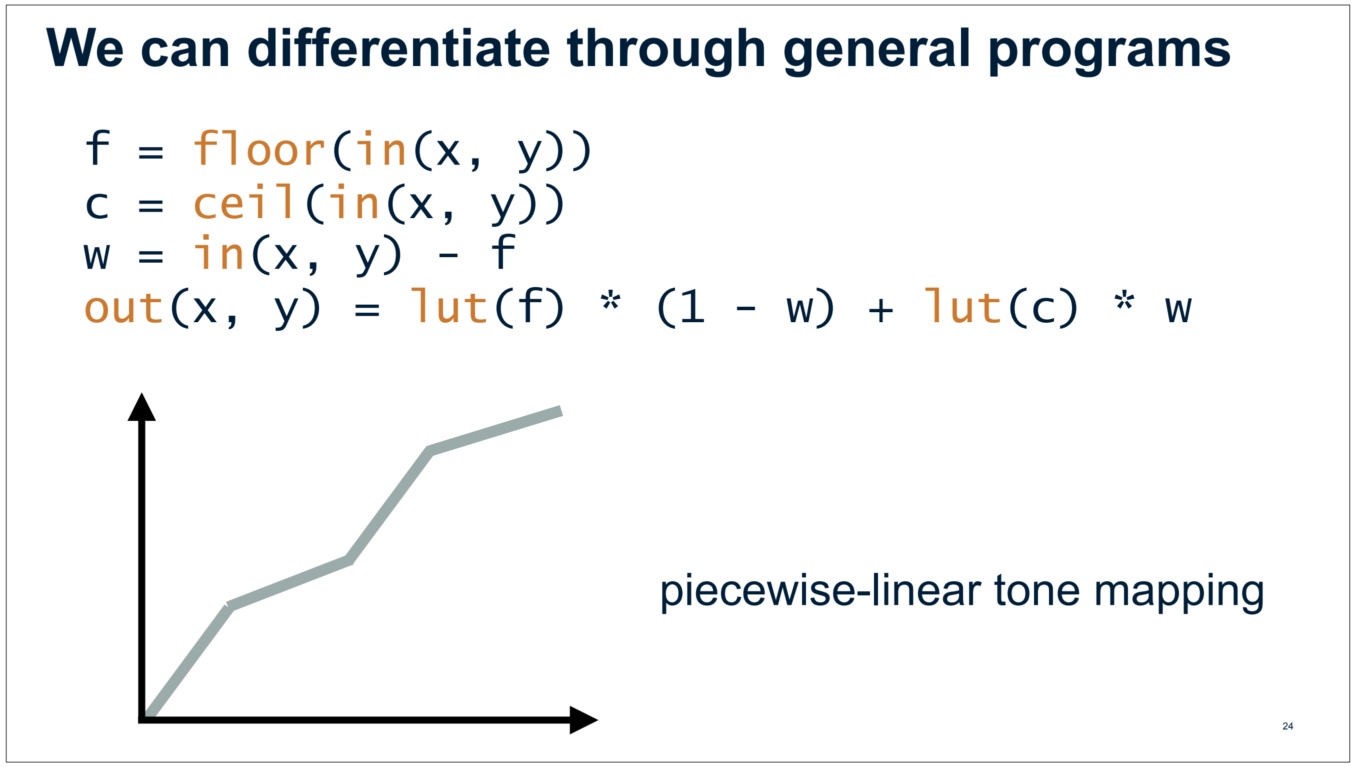
We can differentiate through general programs

```
f = floor(in(x, y))
c = ceil(in(x, y))
```



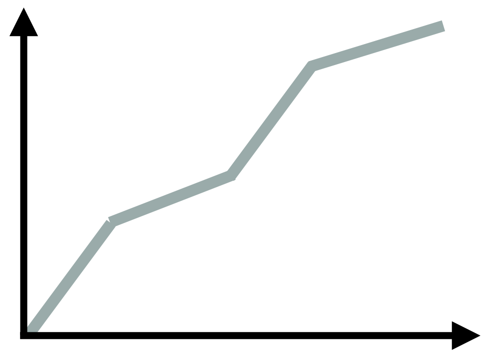
piecewise-linear tone mapping

I will show you how to do this in Halide.
First we take the floor and ceiling of our input pixels.



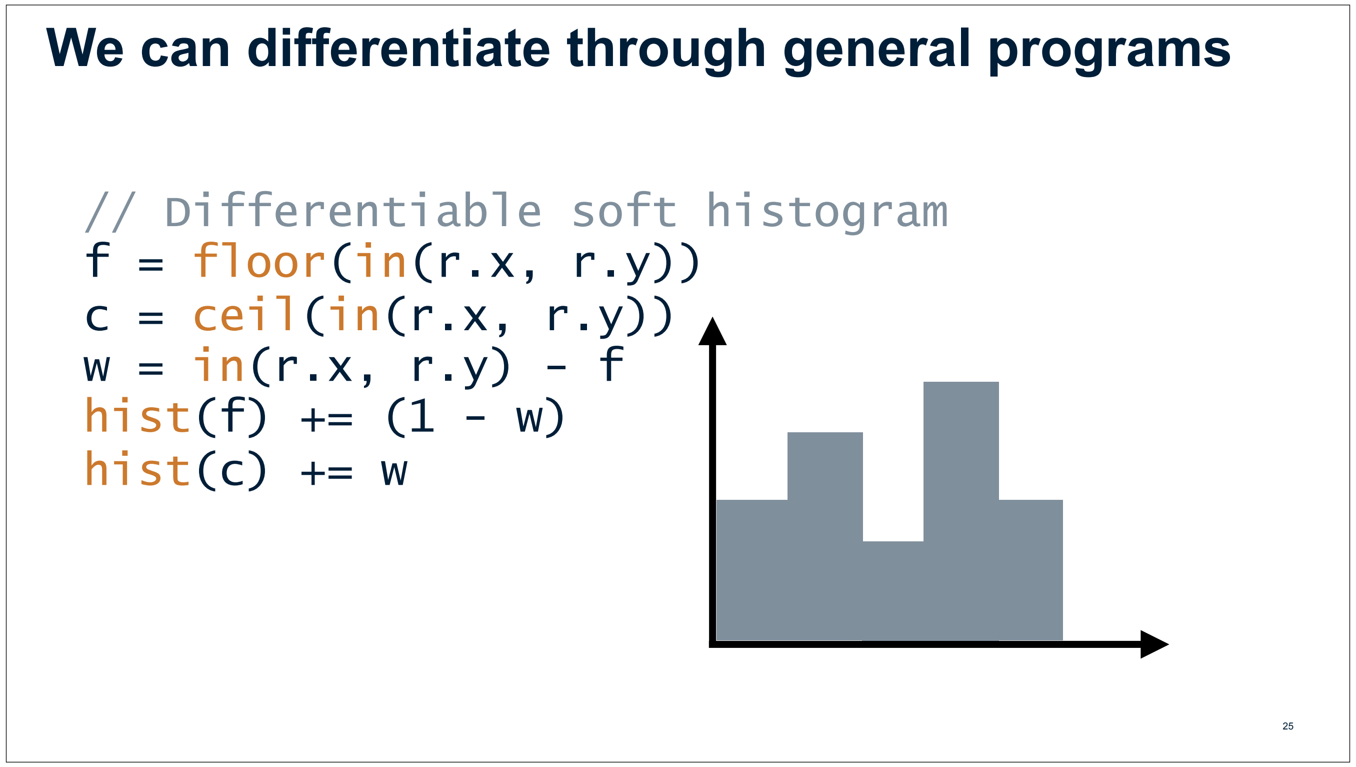
We can differentiate through general programs

```
f = floor(in(x, y))
c = ceil(in(x, y))
w = in(x, y) - f
out(x, y) = lut(f) * (1 - w) + lut(c) * w
```



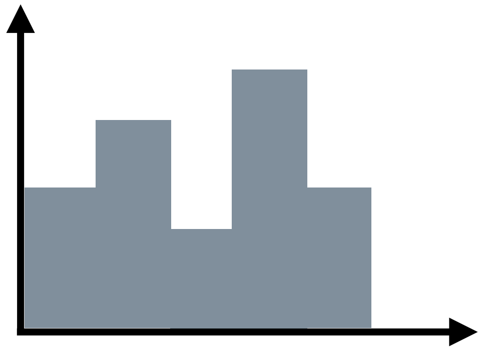
piecewise-linear tone mapping

Then we linearly interpolate the two element on the piecewise linear function, and we’re done.

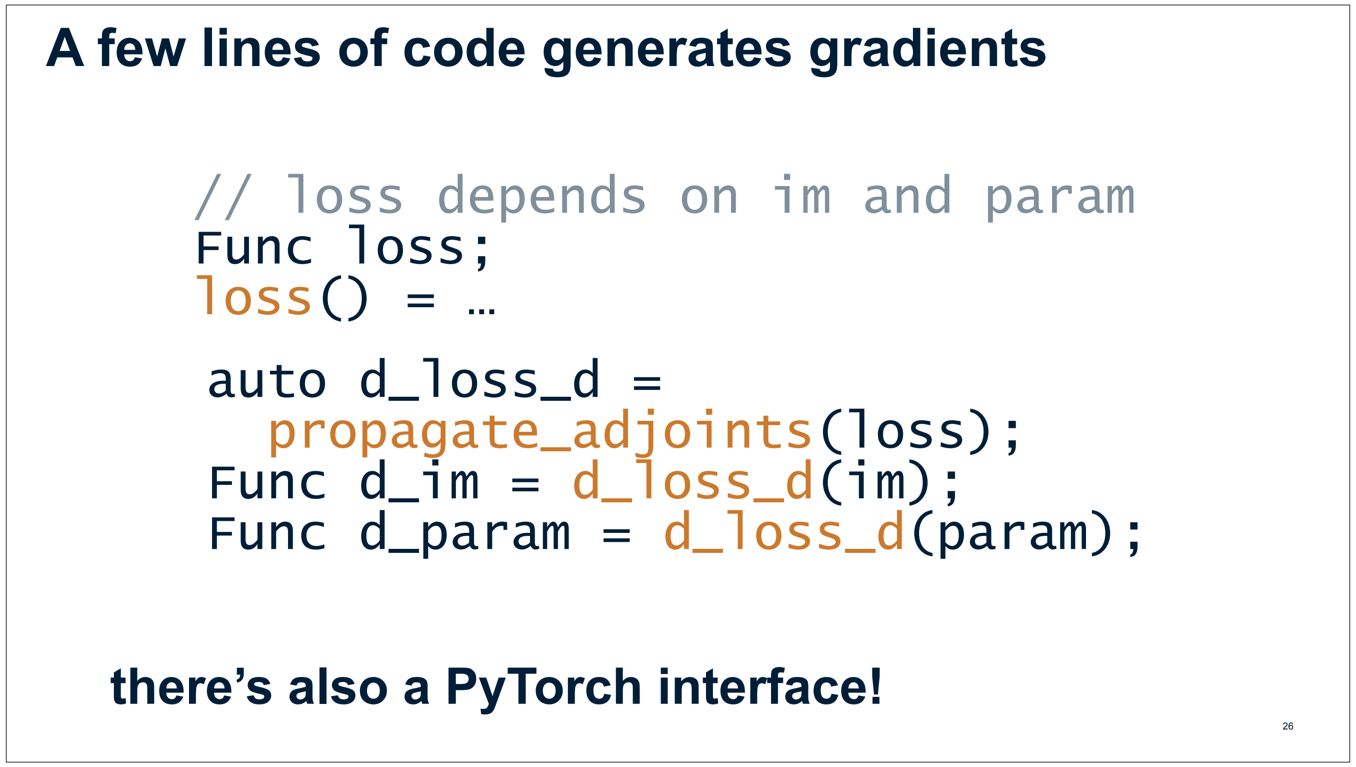


We can differentiate through general programs

```
// Differentiable soft histogram
f = floor(in(r.x, r.y))
c = ceil(in(r.x, r.y))
w = in(r.x, r.y) - f
hist(f) += (1 - w)
hist(c) += w
```



Here’s the second example.
Without much change of code, we can also have a differentiable soft histogram in Halide.
We still take the floor and ceiling of the pixel intensity, and we splat on the histogram using a tent filter.
This is an important operation in some edge aware filters.
And this one is even harder to do in deep learning frameworks,
we need to prepare a set of indices in numpy for computing position for the tent filters.



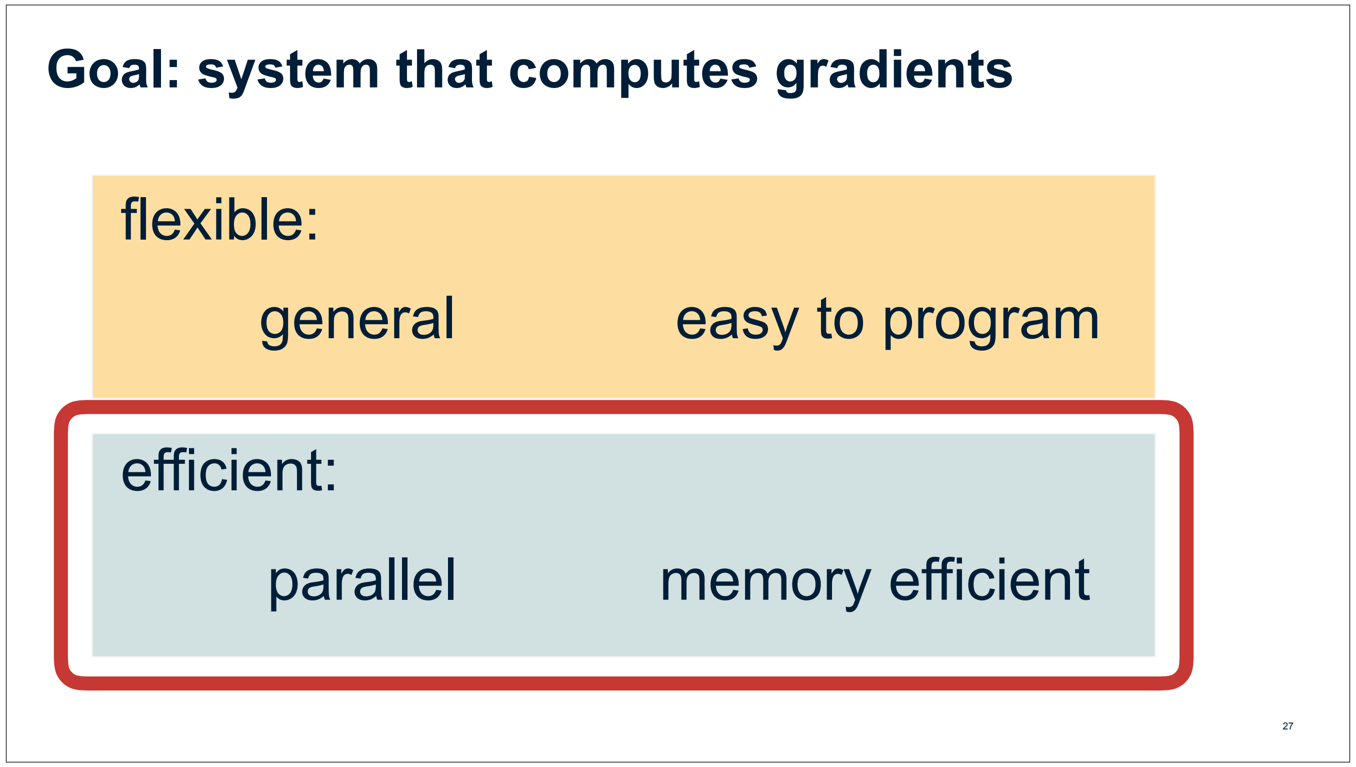
A few lines of code generates gradients

```
// loss depends on im and param
Func loss;
loss() = …

 auto d_loss_d =
   propagate_adjoints(loss);
 Func d_im = d_loss_d(im);
 Func d_param = d_loss_d(param);
```

there's also a PyTorch interface!

To get the gradients you just need a few extra lines of code.
You take a scalar loss function and you throw it to a function call propagate_adjoints.
It returns a mapping between the inputs and their gradients.
You can then extract the gradients by looking up the mapping.

And by the way we have an infrastructure to compile Halide directly to a new PyTorch operator.



Goal: system that computes gradients

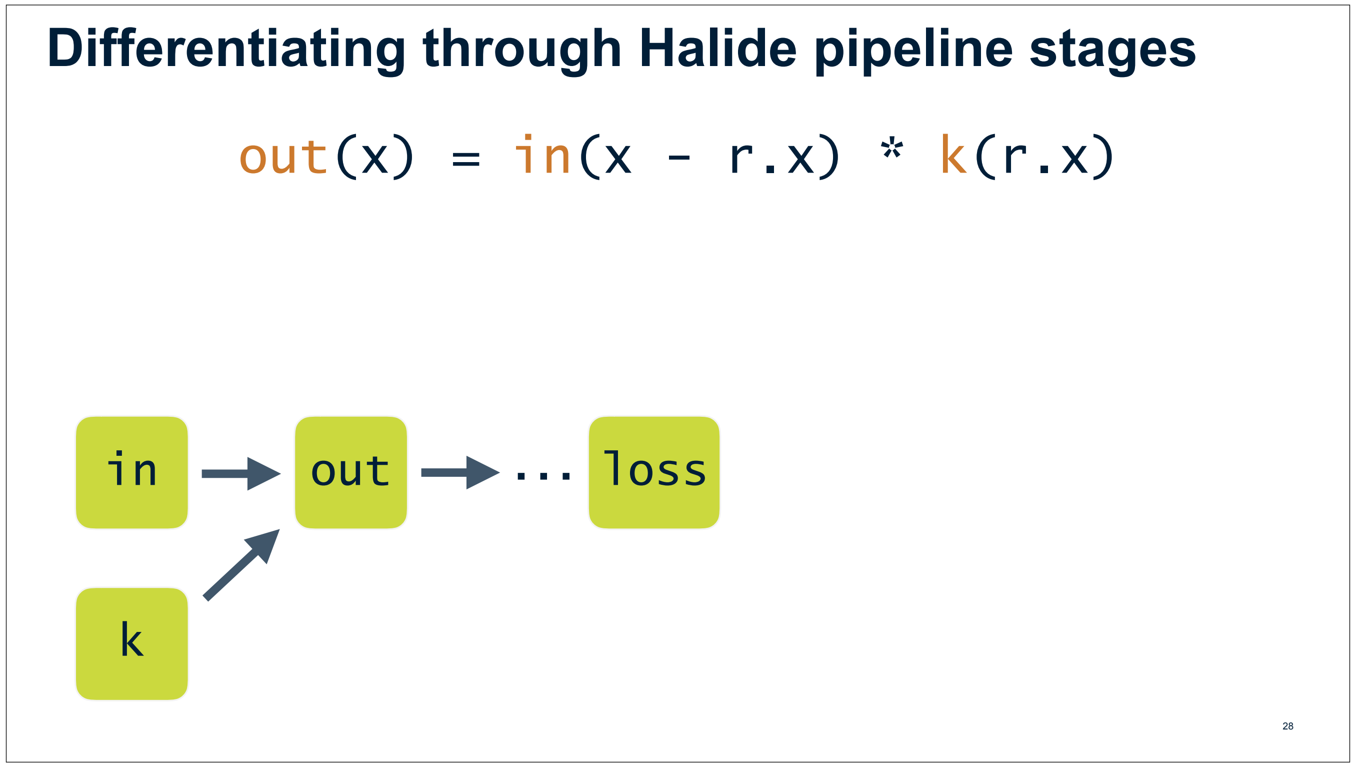
flexible:

general easy to program

efficient:

parallel memory efficient

By building on top of Halide and being able to take gradients with Halide programs,
we have a system computing gradients that is general and easy to program.
Now we turn the focus on the efficiency of the system, where you need for fast iteration.

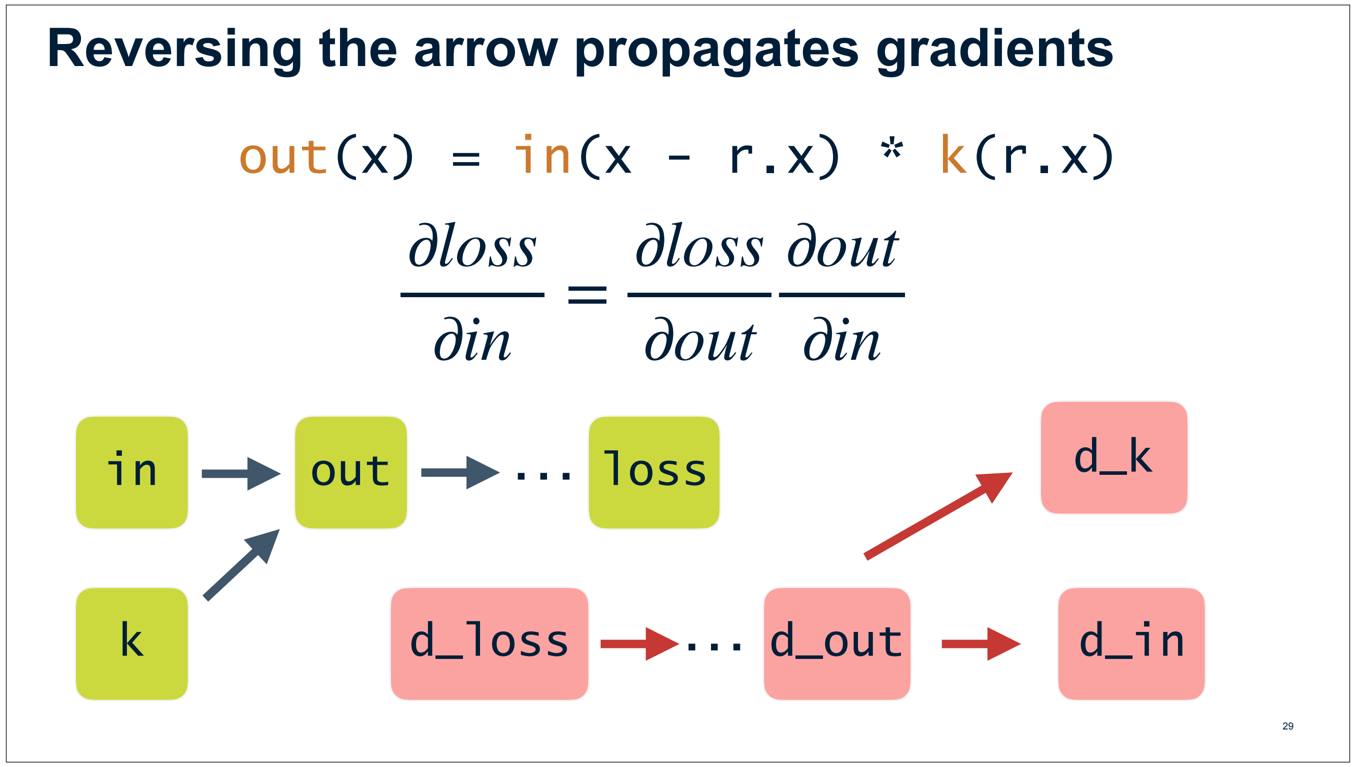


Differentiating through Halide pipeline stages

```
out(x) = in(x - r.x) * k(r.x)
```

$$\begin{tikzcd}
\boxed{\texttt{in}} \arrow[r] & \boxed{\texttt{out}} \arrow[r] & \cdots \boxed{\texttt{loss}} \\
\boxed{\texttt{k}} \arrow[ur] & &
\end{tikzcd}$$

To give you some contexts, I will explain how we propagate the gradients between the Halide pipeline stages first.
I will start with an example of convolution, where the output is a convolution between an input and a kernel.
The slide shows that the computational graph between the stages.

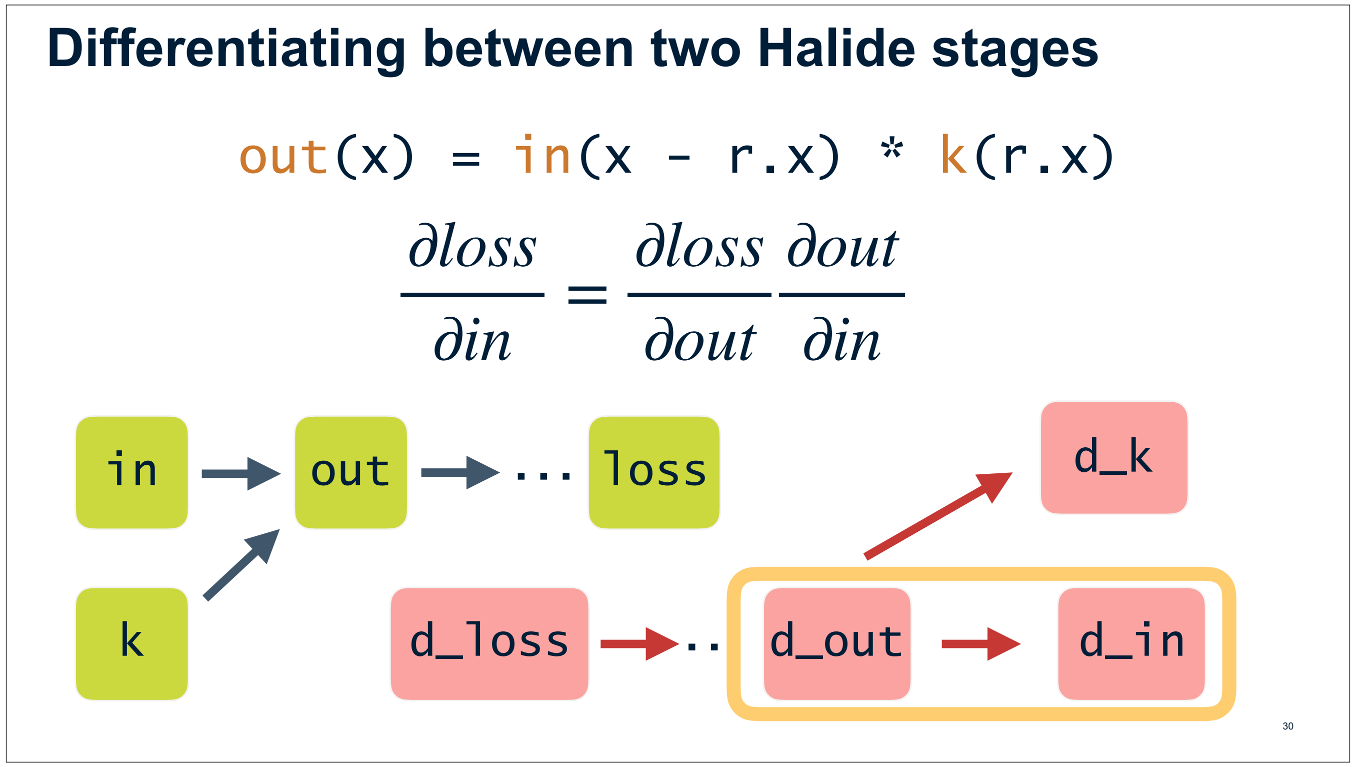


Reversing the arrow propagates gradients

```
out(x) = in(x - r.x) * k(r.x)
```

$$\frac{\partial loss}{\partial in} = \frac{\partial loss}{\partial out}\frac{\partial out}{\partial in}$$

Let’s say we are interested in the gradients of the input, and we already have the gradients of the output.
We can use chain rule to propagate the gradients from output to input.
In general you can just reverse the dependency arrows on the graph and get the gradients.
Sometimes the gradients require values computed from the forward pass. For example the gradients of the output might depends on the values of the output itself.
The gradient of input here depends on the value of the kernel.

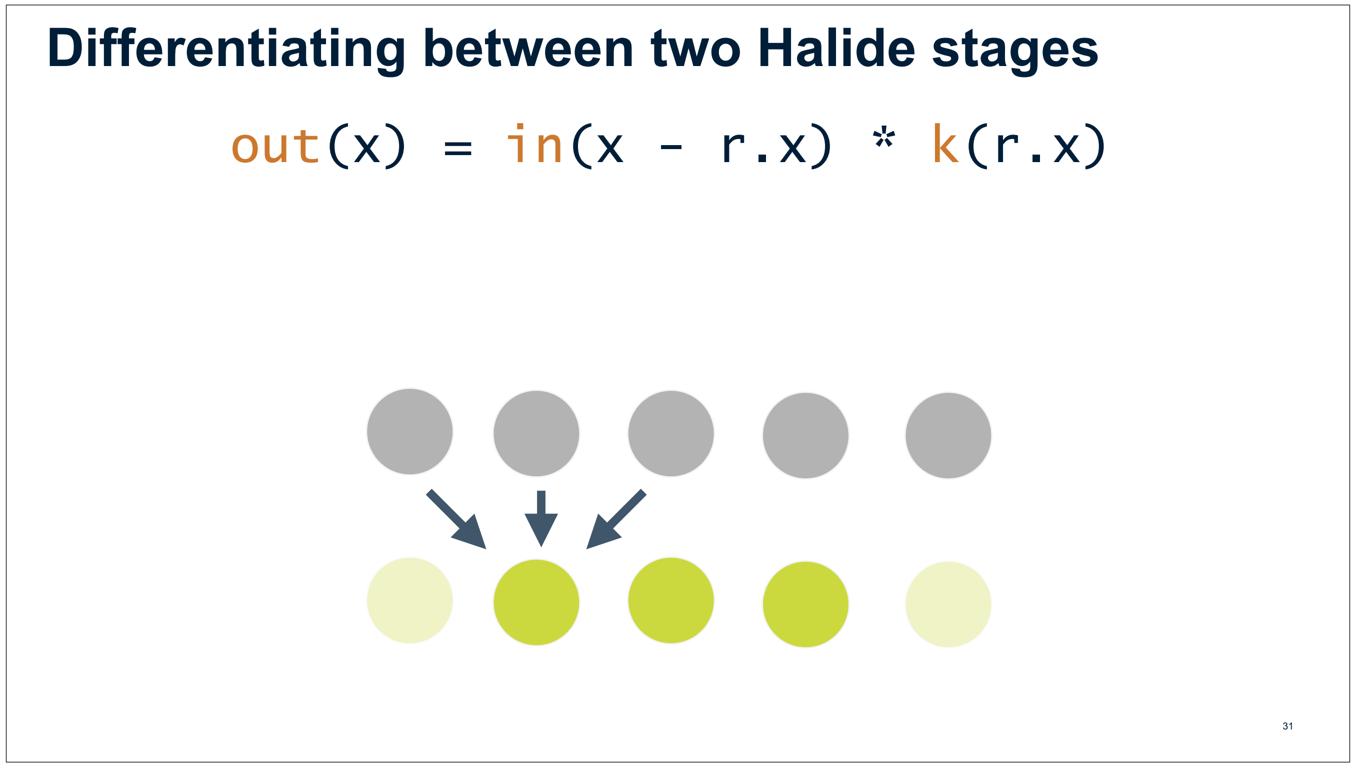


Differentiating between two Halide stages

```
out(x) = in(x - r.x) * k(r.x)
```

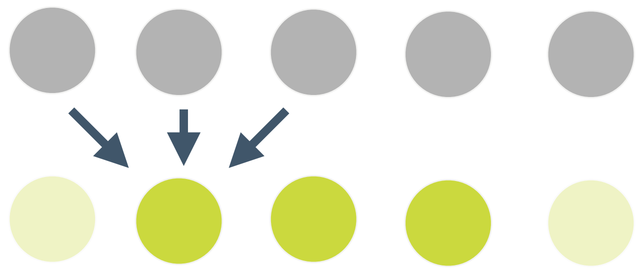
$$\frac{\partial loss}{\partial in} = \frac{\partial loss}{\partial out}\frac{\partial out}{\partial in}$$

Now the question is how you propagate the gradients between two stages, which might involve non-trivial interactions between pixels.



Differentiating between two Halide stages

```
out(x) = in(x - r.x) * k(r.x)
```

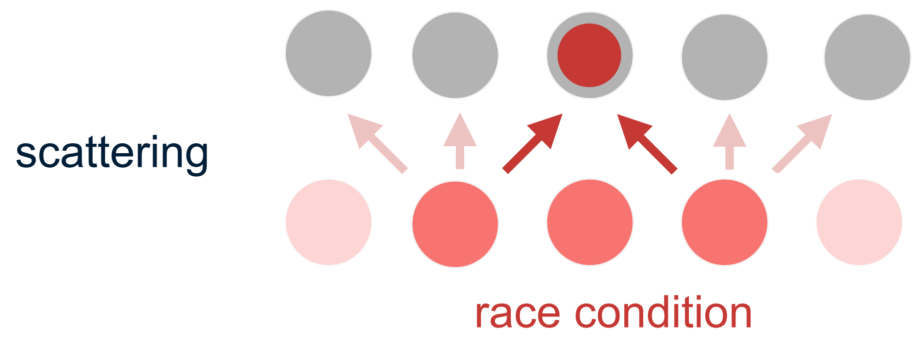


The dependencies of the elements between input and output look like this.

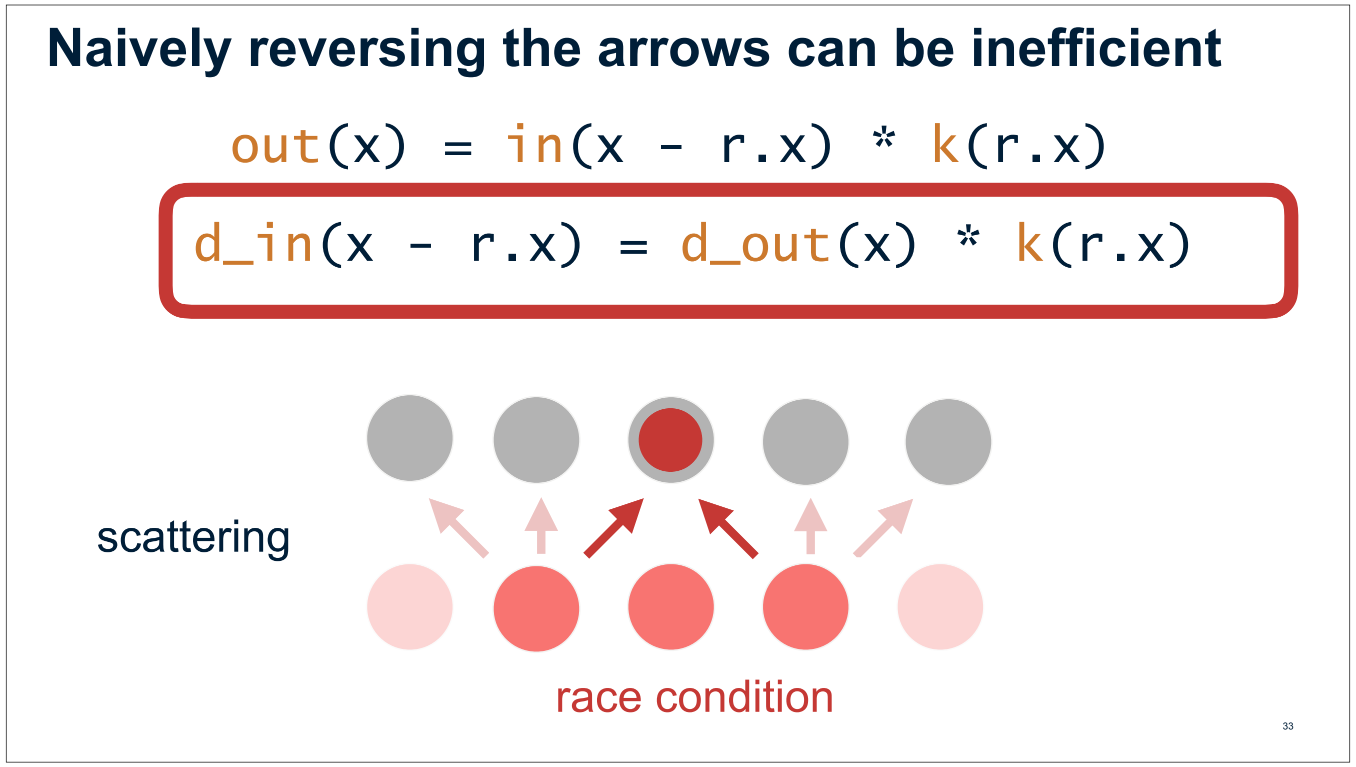
If you just reverse the arrow of the graph to compute gradients…

Naively reversing the arrows can be inefficient

```
out(x) = in(x - r.x) * k(r.x)
```

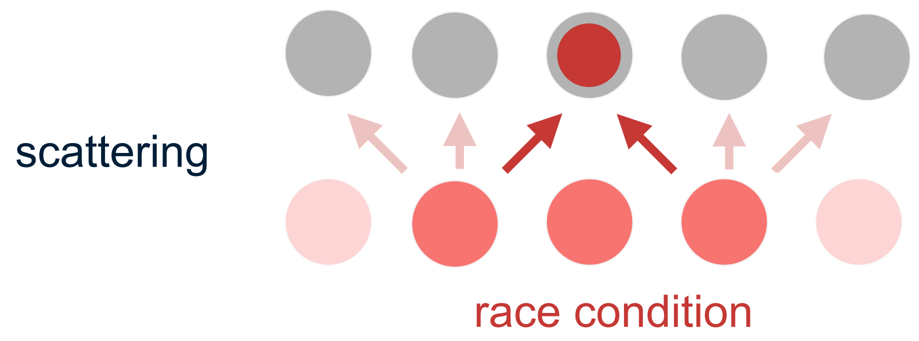


it becomes a scattering operation.
It can be inefficient because multiple outputs land on the same input, causing race condition.
Making it illegal to parallelize the program without atomics.



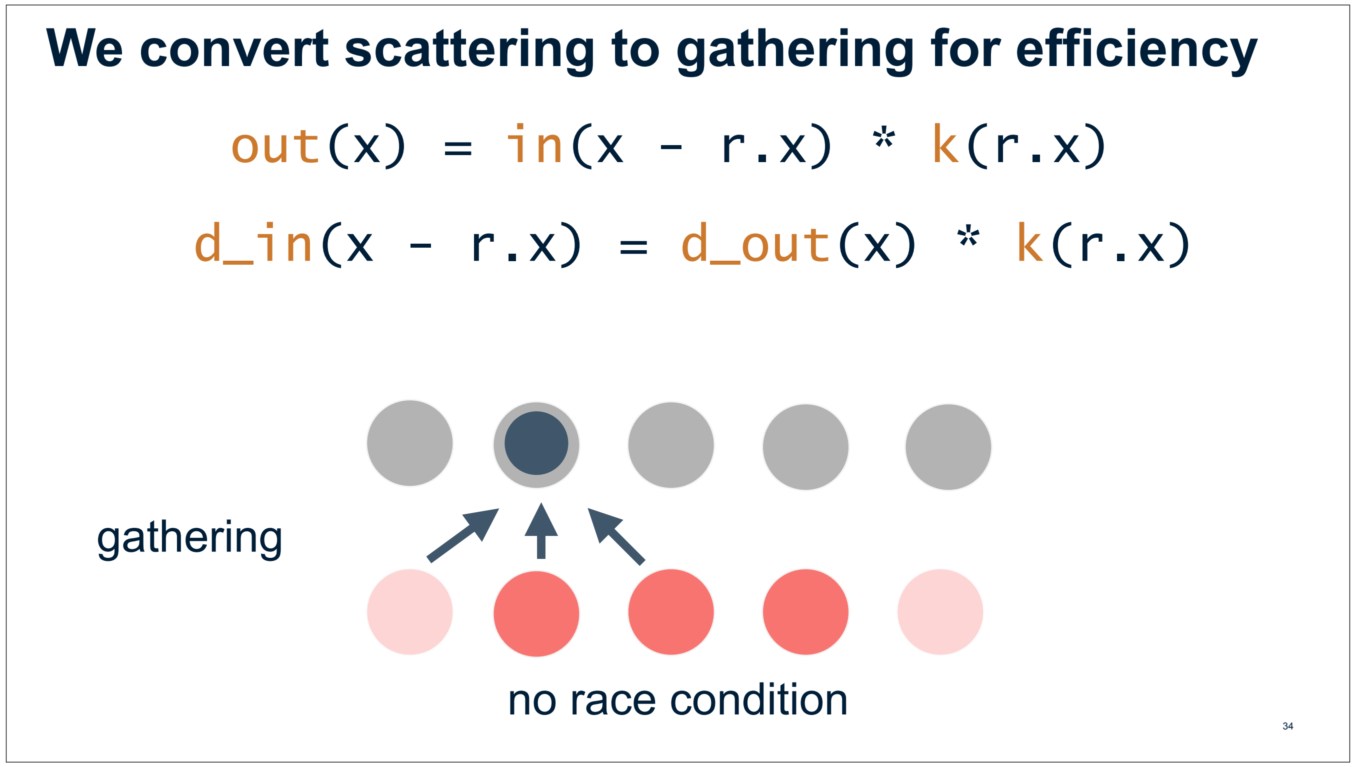
Naively reversing the arrows can be inefficient

```
out(x) = in(x - r.x) * k(r.x)
d_in(x - r.x) = d_out(x) * k(r.x)
```



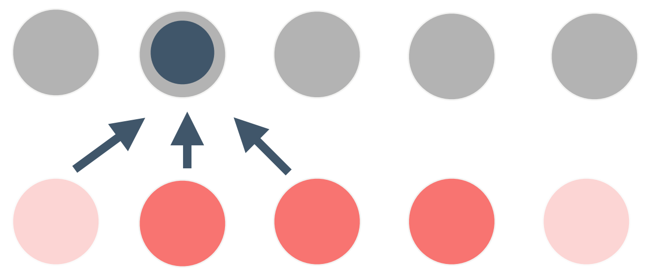
In terms of code it looks like this.
The left hand side arguments control which location you write to.
Each x in the gradients of the output scatters to three locations in the gradients of input.
Which means different elements on the gradients of output will scatter to the same location.
The goal is to make the argument on the left hand side a single variable, so we can guarantee there is no race condition.

So what we want to do is to rearrange the arrows a bit like this…

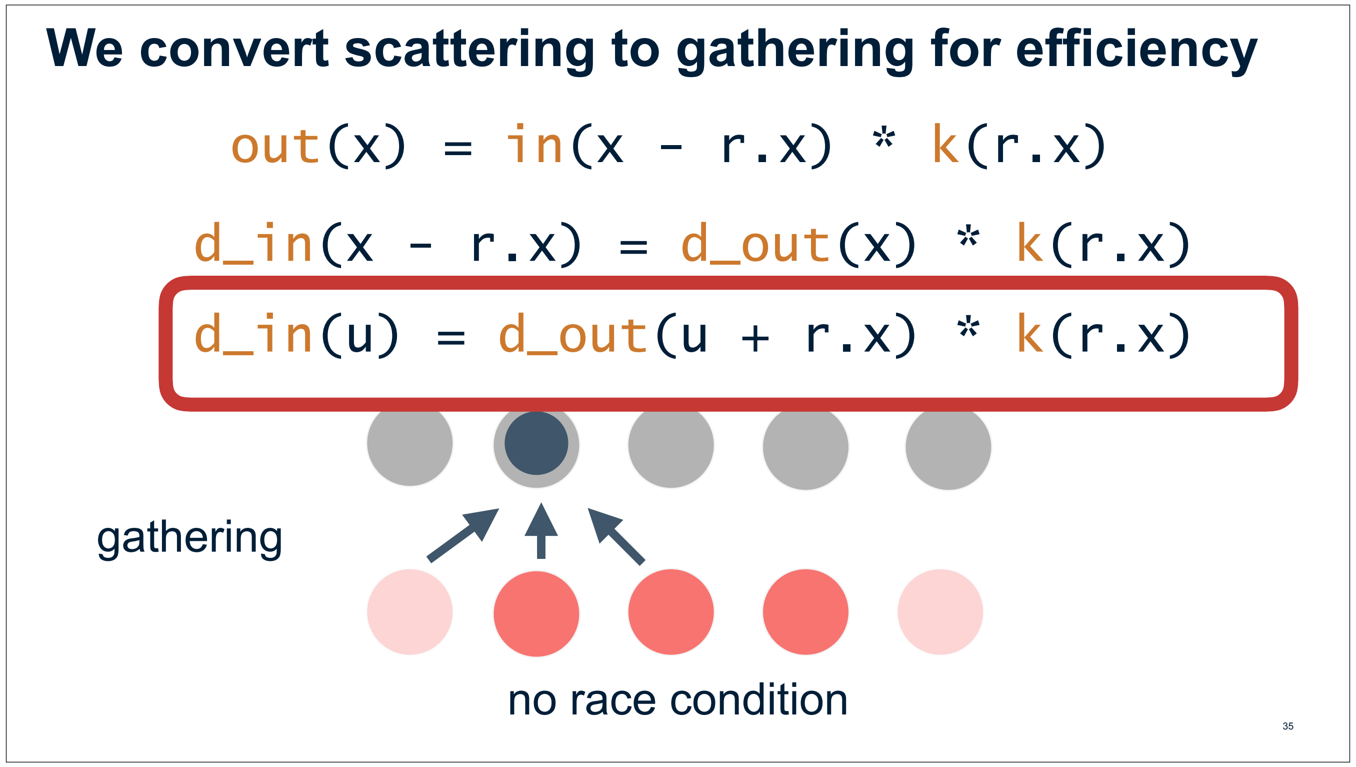


We convert scattering to gathering for efficiency

```
out(x) = in(x - r.x) * k(r.x)
d_in(x - r.x) = d_out(x) * k(r.x)
```

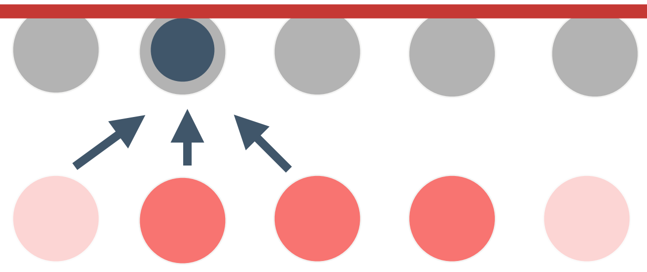


We introduce a transformation to convert the scattering operation back to a gathering operation, by finding all the output pixels contributing to each input pixels.
That is, we find the inverse of the indexing operation.
In this particular case the convolution becomes a correlation.
And the operation becomes parallelizable again.

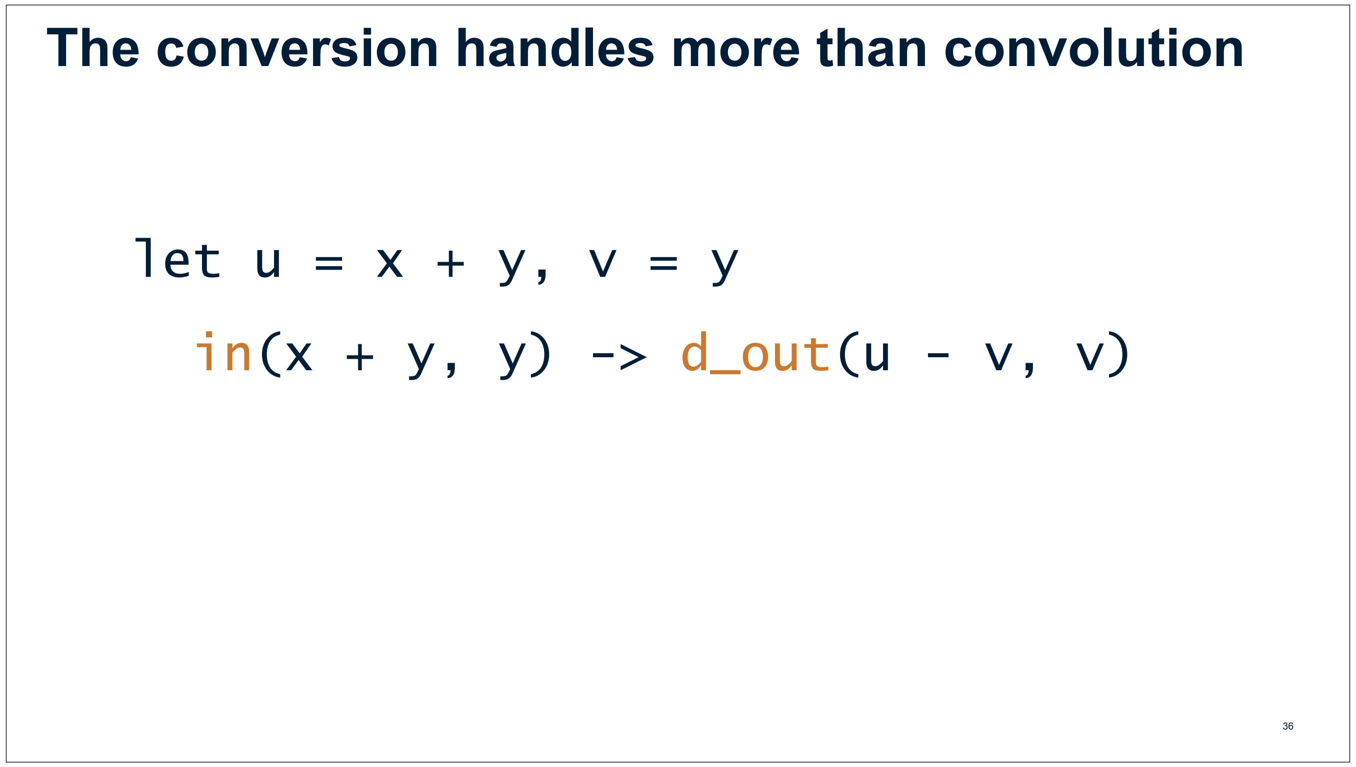


We convert scattering to gathering for efficiency

```
out(x) = in(x - r.x) * k(r.x)
d_in(x - r.x) = d_out(x) * k(r.x)
d_in(u) = d_out(u + r.x) * k(r.x)
```



In terms of code, we introduce a new variable u, and set u equals to x - r.x.
We then replace all x on the right hand side in terms of u.

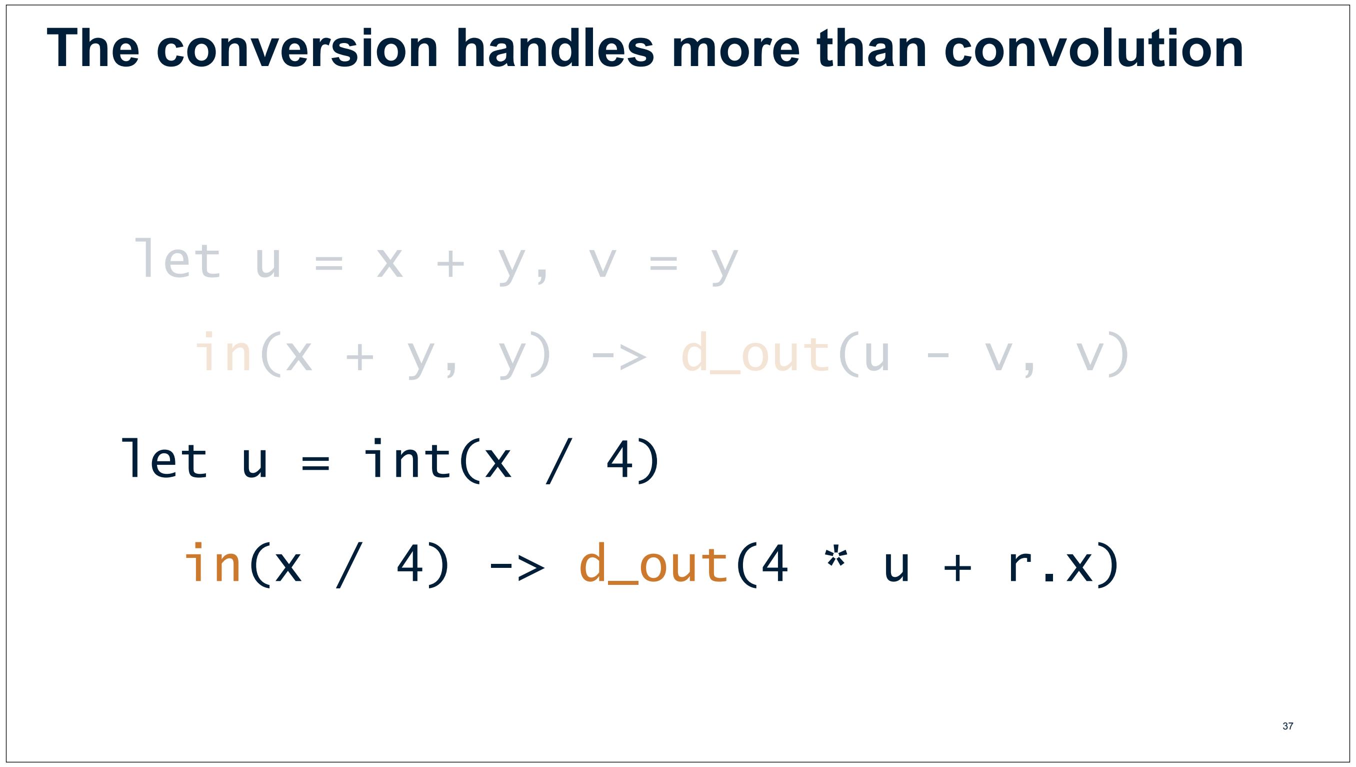


The conversion handles more than convolution

```
let u = x + y, v = y
  in(x + y, y) -> d_out(u - v, v)
```

36

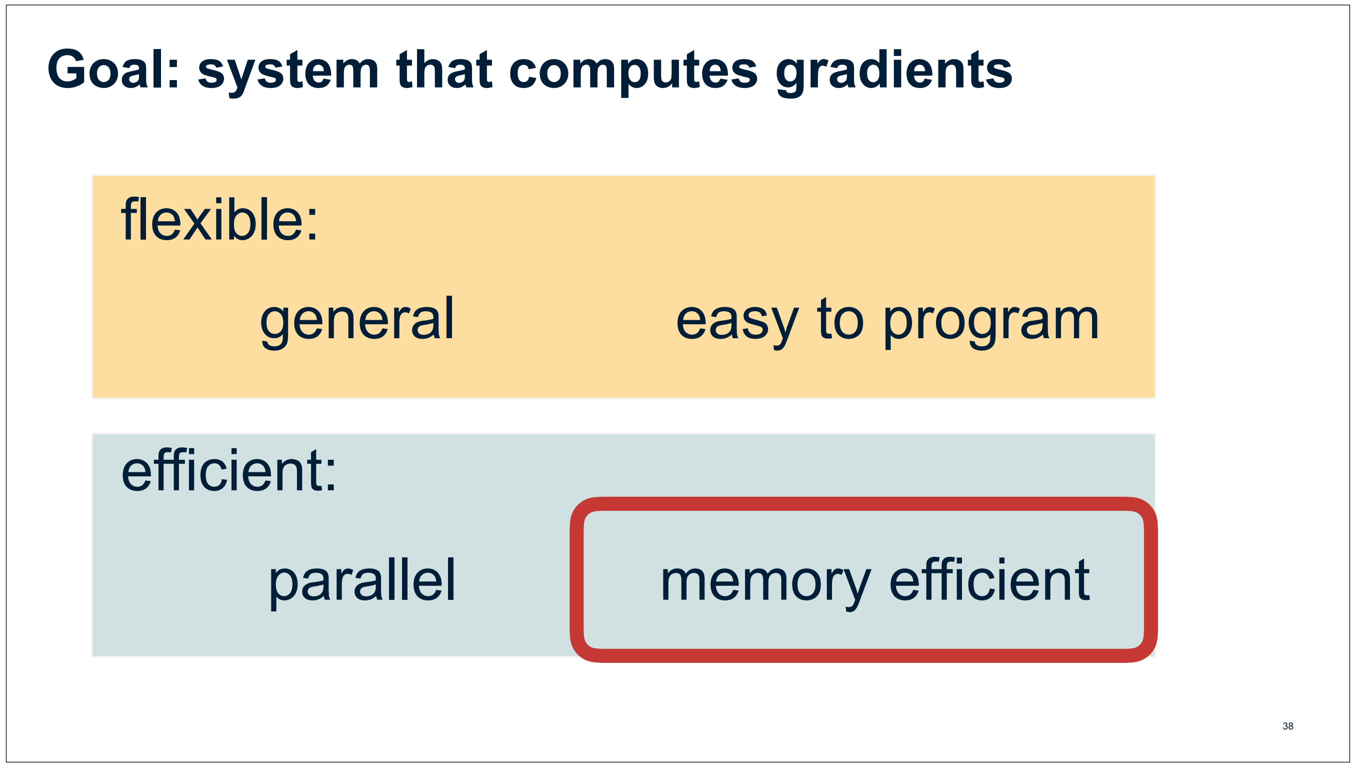
We can also handle the case where there are multiple variables and we can solve one of them.



The conversion handles more than convolution

```
let u = x + y, v = y
  in(x + y, y) -> d_out(u - v, v)
let u = int(x / 4)
  in(x / 4) -> d_out(4 * u + r.x)
```

And we can solve the case where the inverse has multiple correspondence. Like in this case each u corresponds to 4 different x.
This is a common operation in upsampling for example.



Goal: system that computes gradients

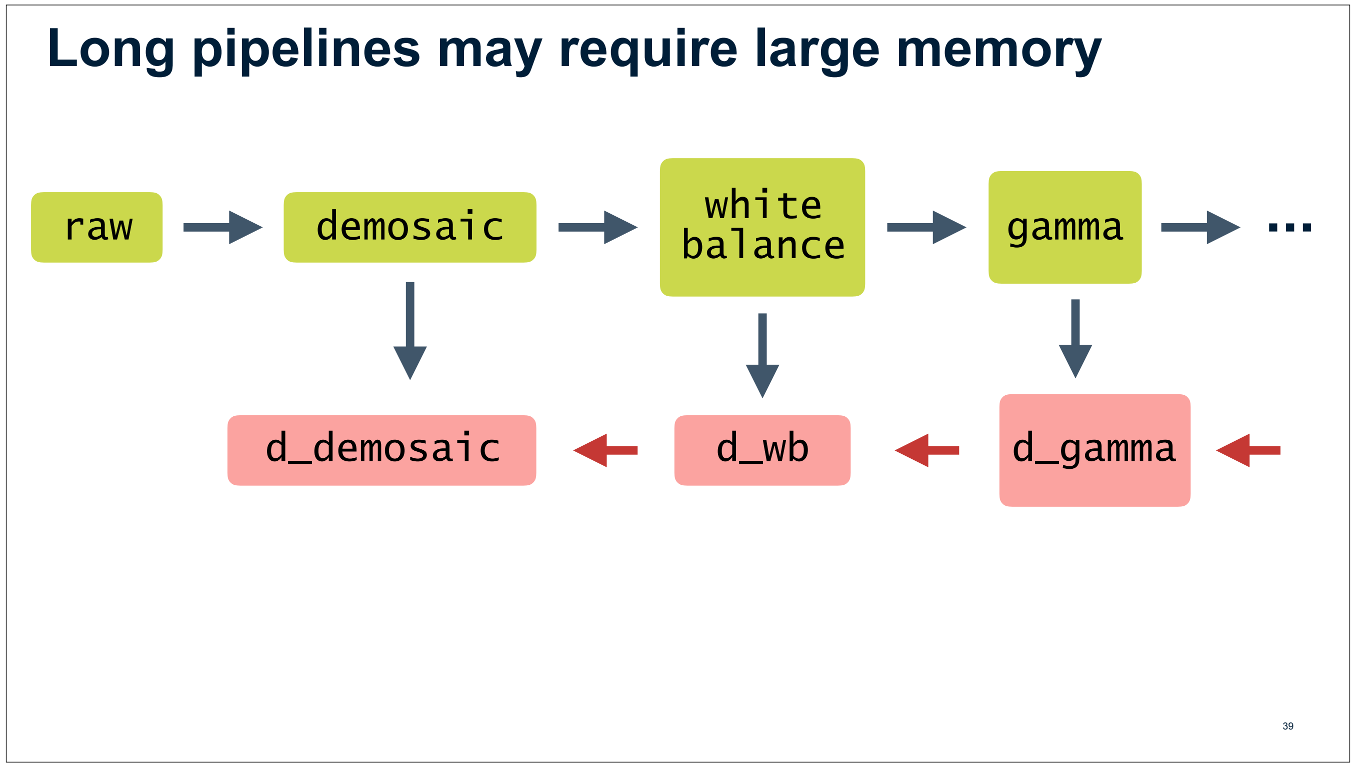
flexible:

general   easy to program

efficient:

parallel   memory efficient

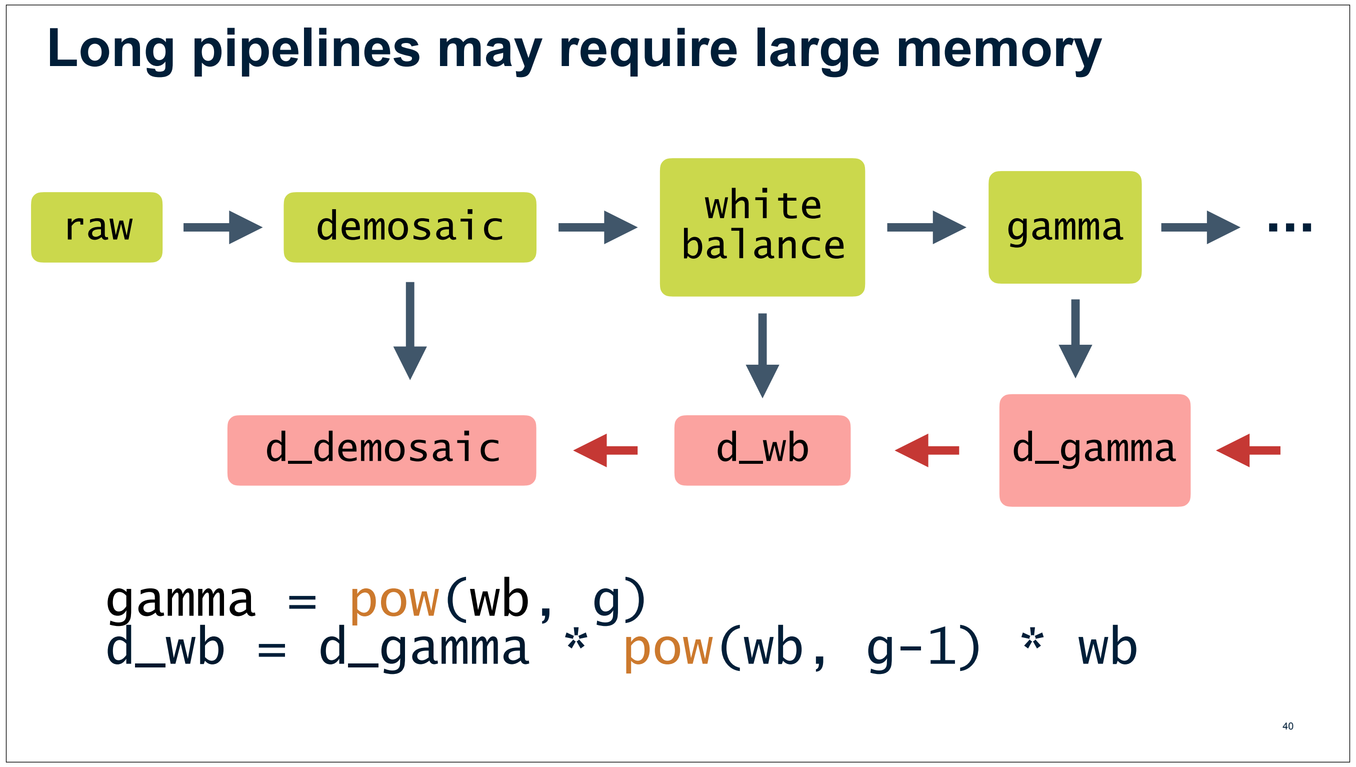
Memory efficiency is also a very important aspect of modern hardware.
We show that there is a strong connection between an existing
memory optimization for automatic differentiation and Halide’s scheduling constructs.



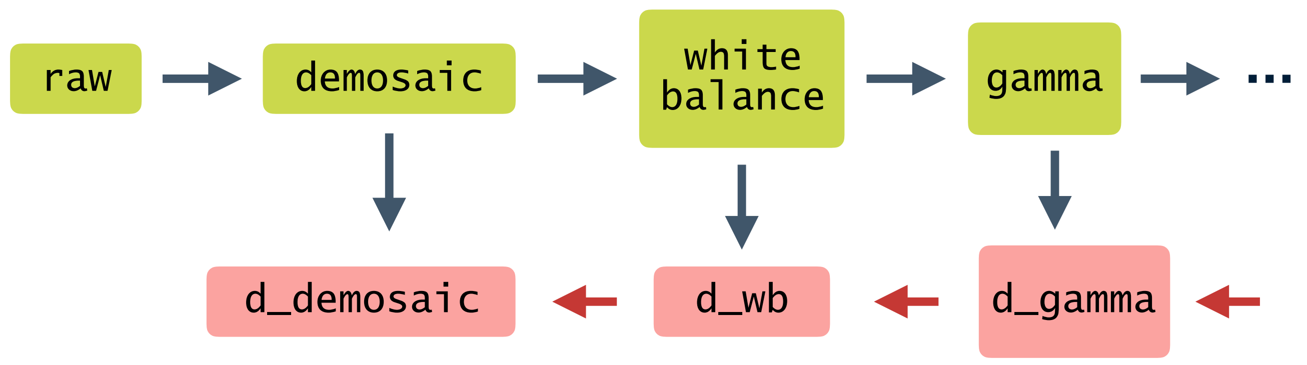
Long pipelines may require large memory

$$\begin{tikzcd}
\text{raw} \arrow[r] & \text{demosaic} \arrow[r] \arrow[d] & \text{white balance} \arrow[r] \arrow[d] & \text{gamma} \arrow[r] \arrow[d] & \cdots \\
& \text{d\_demosaic} & \text{d\_wb} \arrow[l] & \text{d\_gamma} \arrow[l] & \arrow[l]
\end{tikzcd}$$

Sometimes you will end up with a long pipeline, like the one we mentioned in the beginning. The gradients sometimes depend on the forward components.

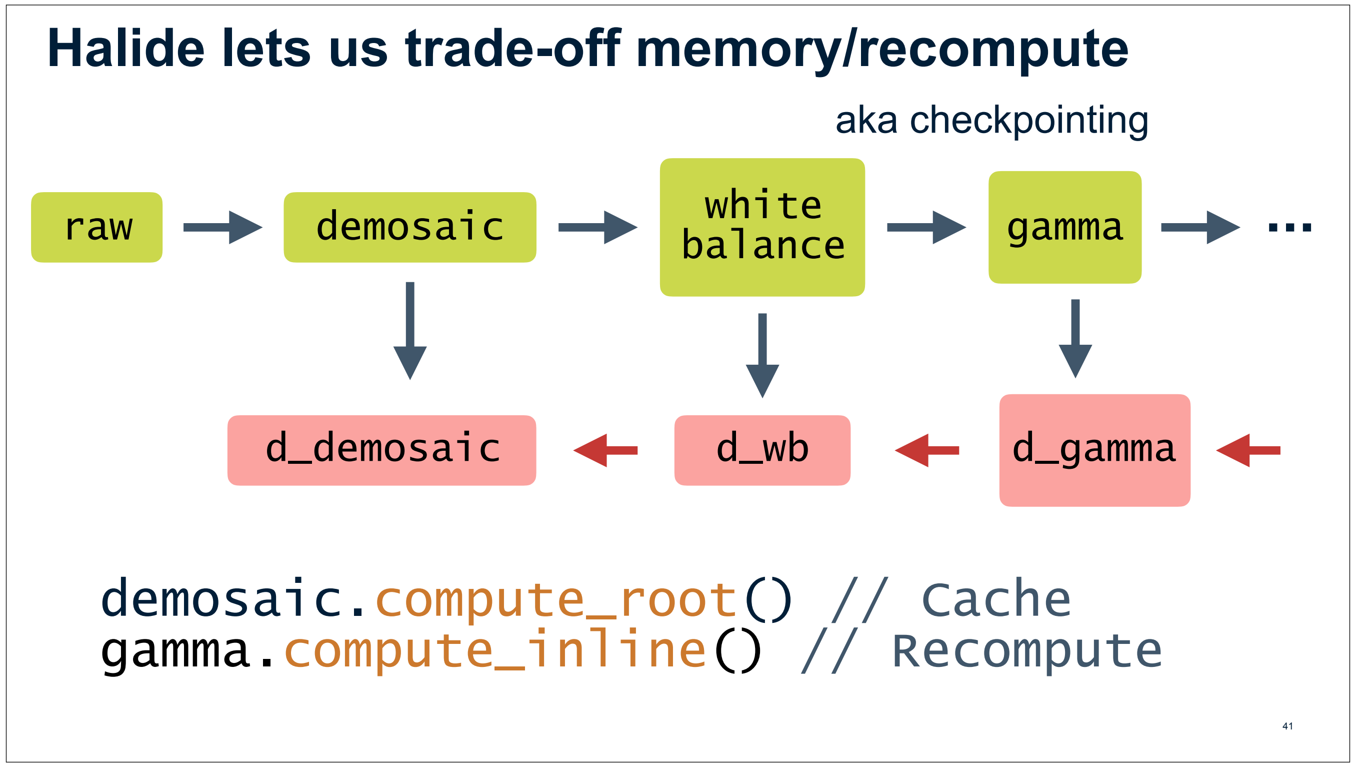


Long pipelines may require large memory



```
gamma = pow(wb, g)
d_wb = d_gamma * pow(wb, g-1) * wb
```

For example the gradients of the white balance stage may depend on its own result.
A problem when computing the gradients of a long pipeline is that you might end up using too much memory remembering the forward stages.
A remedy in autodiff, called checkpointing, is to only remember some of the forward components, and compute the rest on demand.
Most deep learning frameworks don’t have built-in mechanism for checkpointing. Sometimes you have to write a few hundred lines of code just for this.
For some frameworks it is just impossible.



Halide lets us trade-off memory/recompute

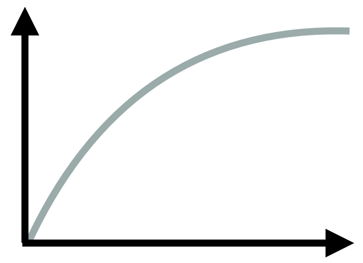
aka checkpointing

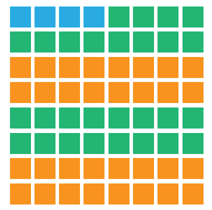
```
demosaic.compute_root() // Cache
gamma.compute_inline() // Recompute
```

Turns out Halide’s existing constructs are strongly connected to the idea of checkpointing.
You write demosaic.compute_root() during the scheduling and Halide compiler will allocate memory for the entire demosaic image buffer.
Or you can write gamma.compute_inline() and Halide compiler will compute gamma correction on demand.
Halide even provides an intermediate choice between compute root and compute inline.
More details are in the paper.

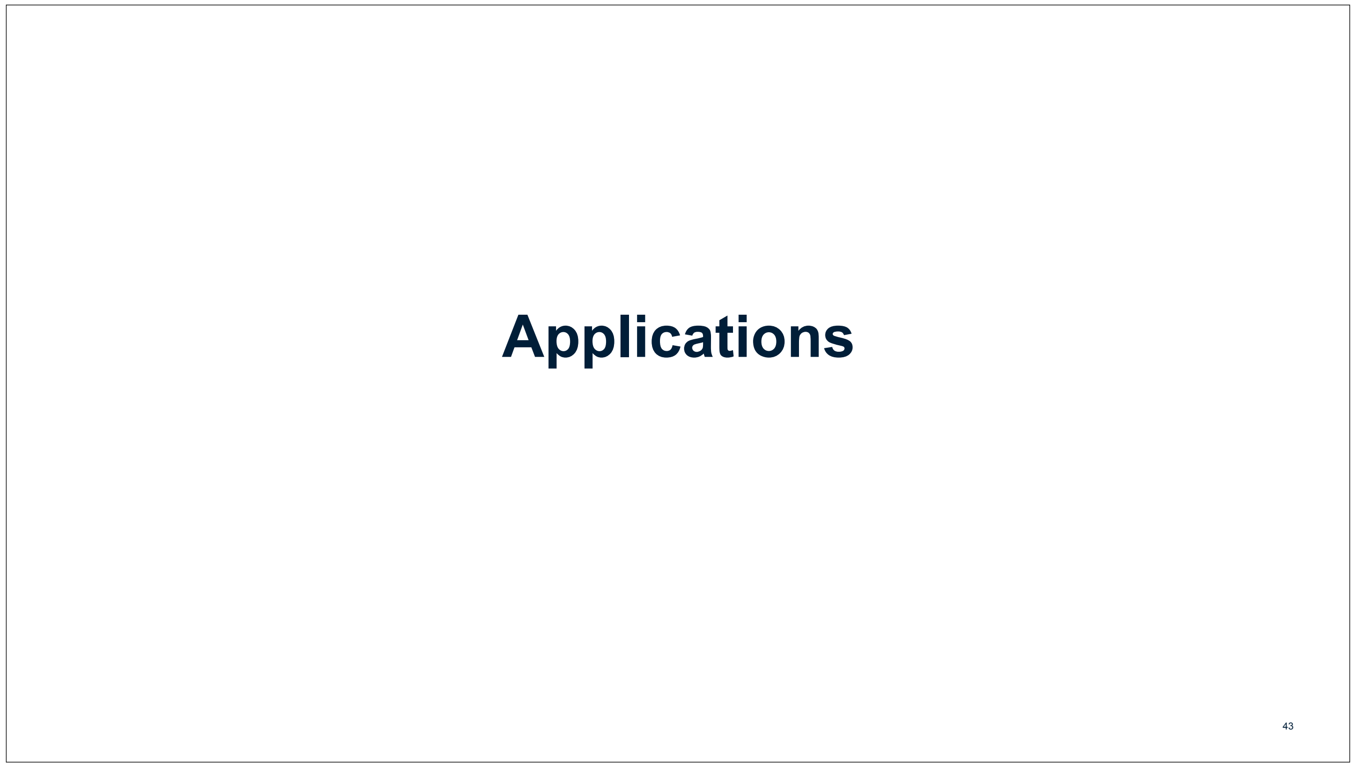
We implemented new automatic scheduling

- Halide’s auto scheduler [Mullapudi 2016] doesn’t handle GPU and parallel reduction (often appears in gradients).
- New auto scheduler that supports both these, using a new feature from Halide [Suriana 2017].



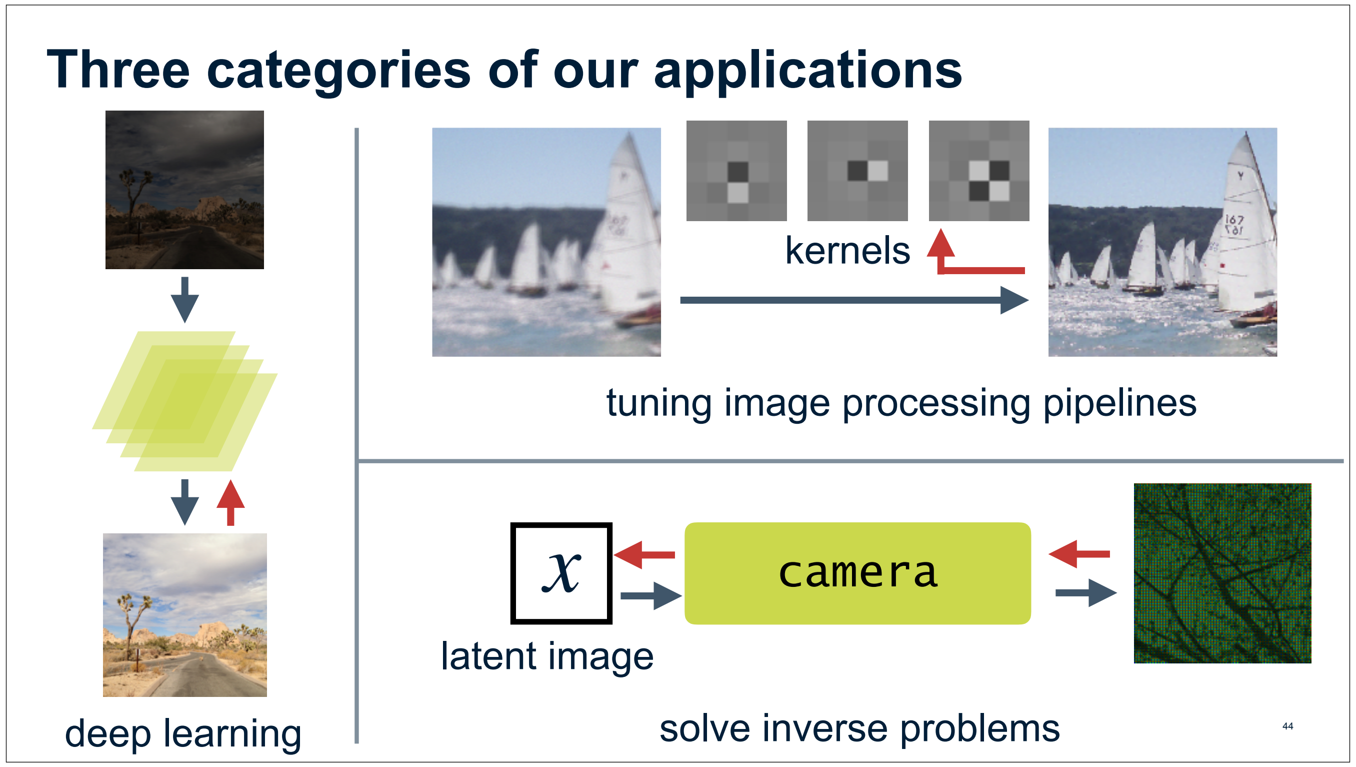


Remember that Halide has the high-level algorithm and lower-level scheduling decomposition.
There is a built-in tool in Halide developed by Mullapudi et al. that automatically generates a schedule for you.
However it doesn’t handle GPU schedules and the case where you have a big reduction, which often appears in gradients computation.
We implement a new auto scheduler that supports GPU schedules and parallel reduction.
All the results show in this talk later are automatically scheduled.



Applications

Now we show some applications to demonstrate the flexibility and efficiency of the system.

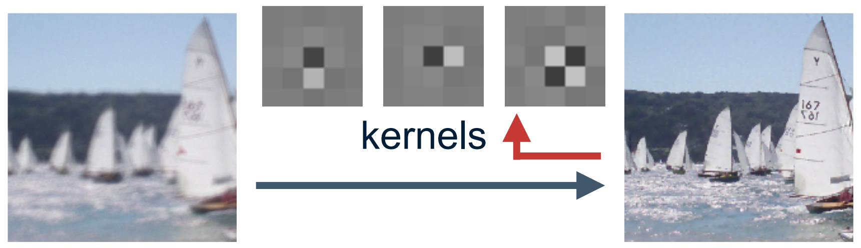


Three categories of our applications

Like I mentioned before, there are three different categories of the applications of gradients.

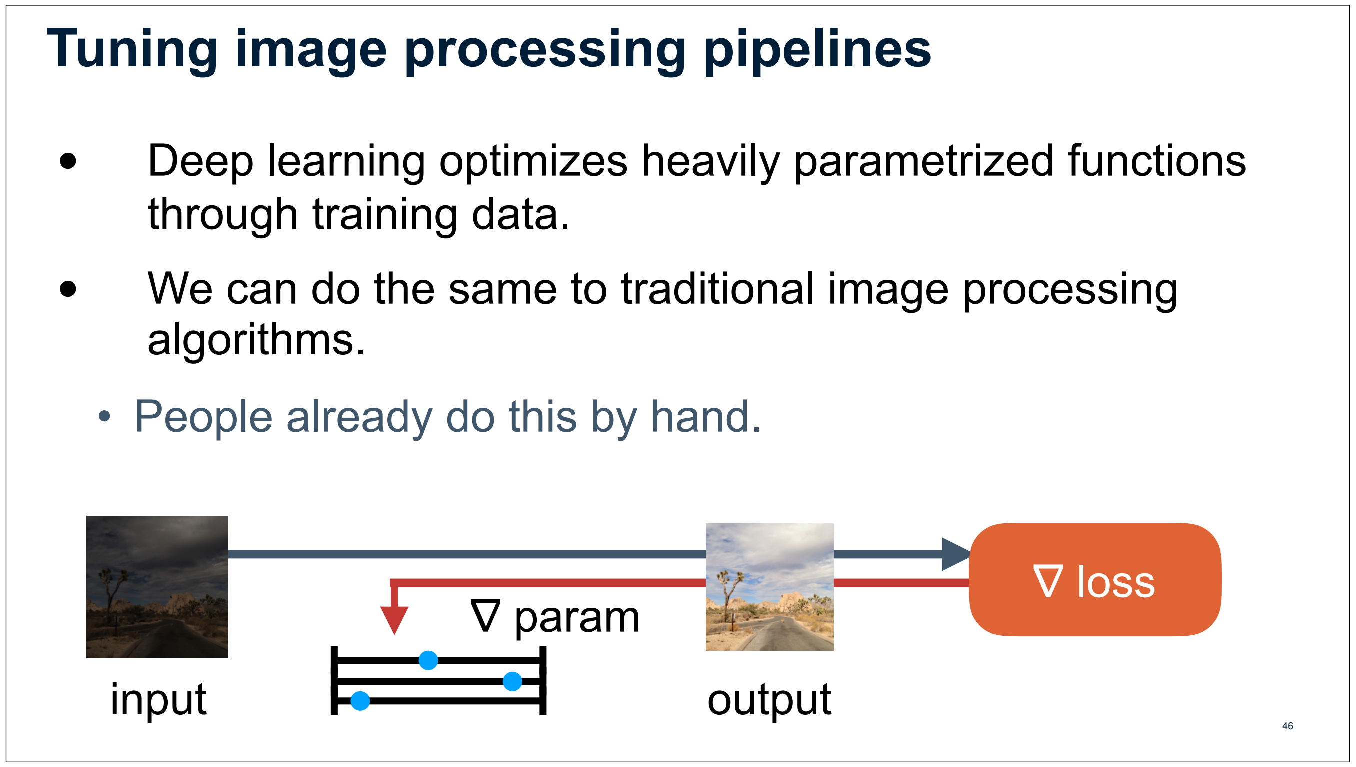
You can use it for deep learning, you can use it for tuning image processing pipelines, and you can use it to solve an inverse problem.

Gradients can be used for tuning existing pipelines



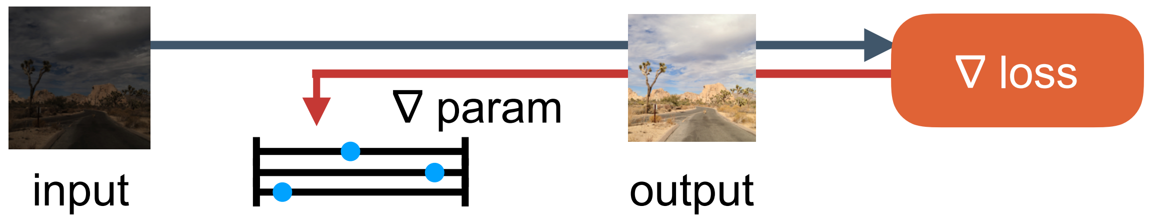
tuning image processing pipelines

I’ll talk about tuning image processing pipelines first.



Tuning image processing pipelines

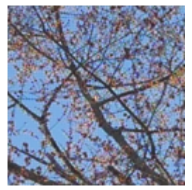
- Deep learning optimizes heavily parametrized functions through training data.
- We can do the same to traditional image processing algorithms.
  - People already do this by hand.

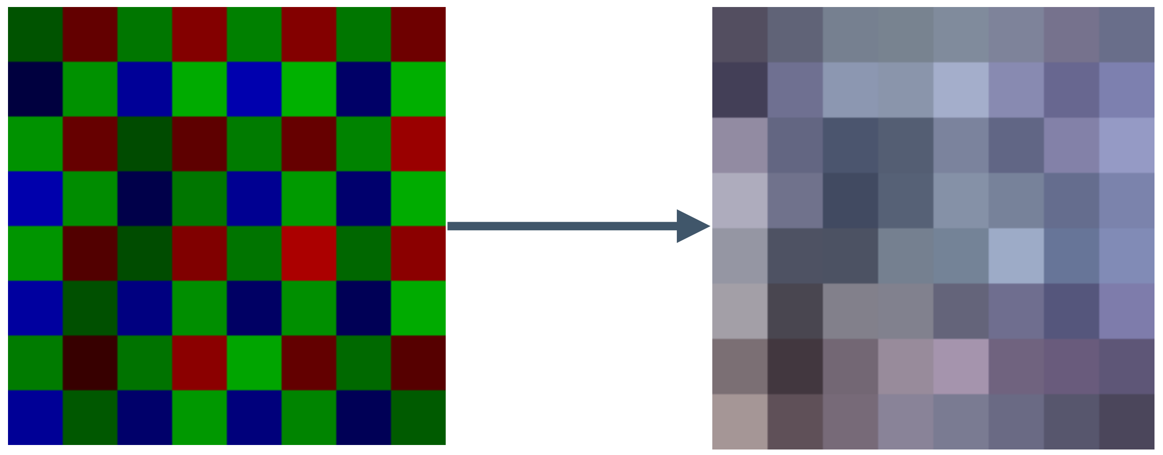


Like I said, deep learning is all about optimizing heavily parametrized functions through some training data.
Which means we can optimize the more traditional image processing algorithms in the same way.
People already do this by hand by tuning their algorithm on a small set of images.
Gradients allow us to do this automatically.
Even better we can add more parameters to your image processing algorithm and optimize them without worrying too much.

Tuning a popular demosaicking algorithm

AHD [Hirakawa and Parks 2005]

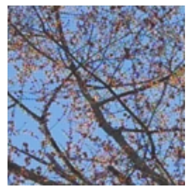


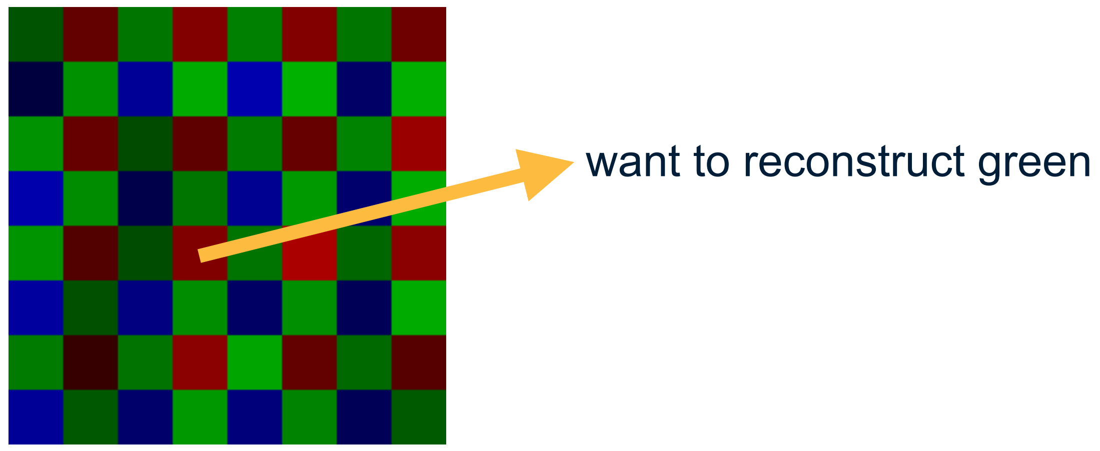


We will take an example of us modifying and tuning a popular demosaicking algorithm AHD, developed by Hirakawa and Parks.
In modern cameras your pixel usually can only record one channel of the color.
The task of demosaicking is given this incomplete information, we want to interpolate the recorded color to obtain the final image.

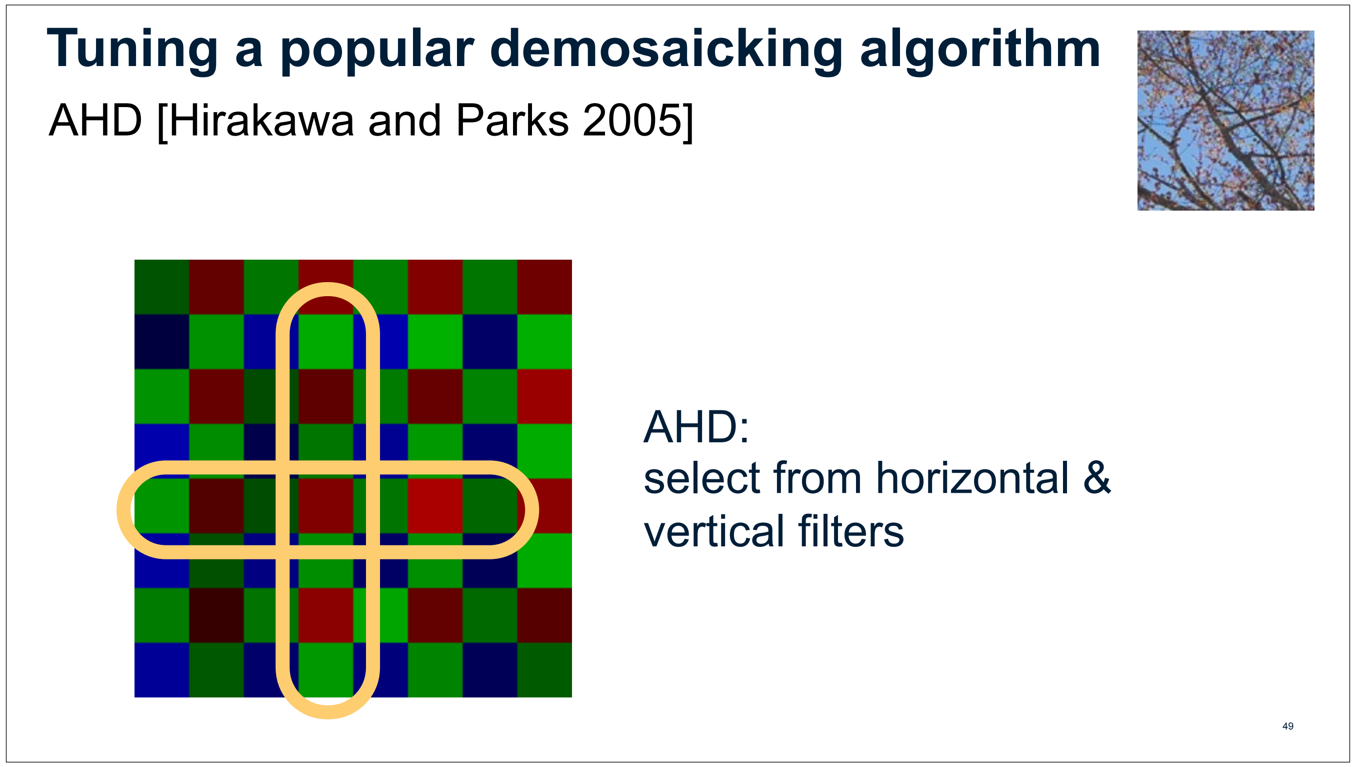
Tuning a popular demosaicking algorithm

AHD [Hirakawa and Parks 2005]



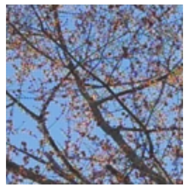


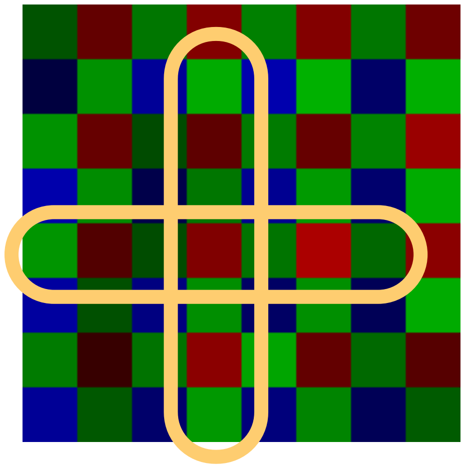
Let’s take this pixel in the center, which we only know the red color, and say we want to reconstruct the green color from the neighbors.



Tuning a popular demosaicking algorithm

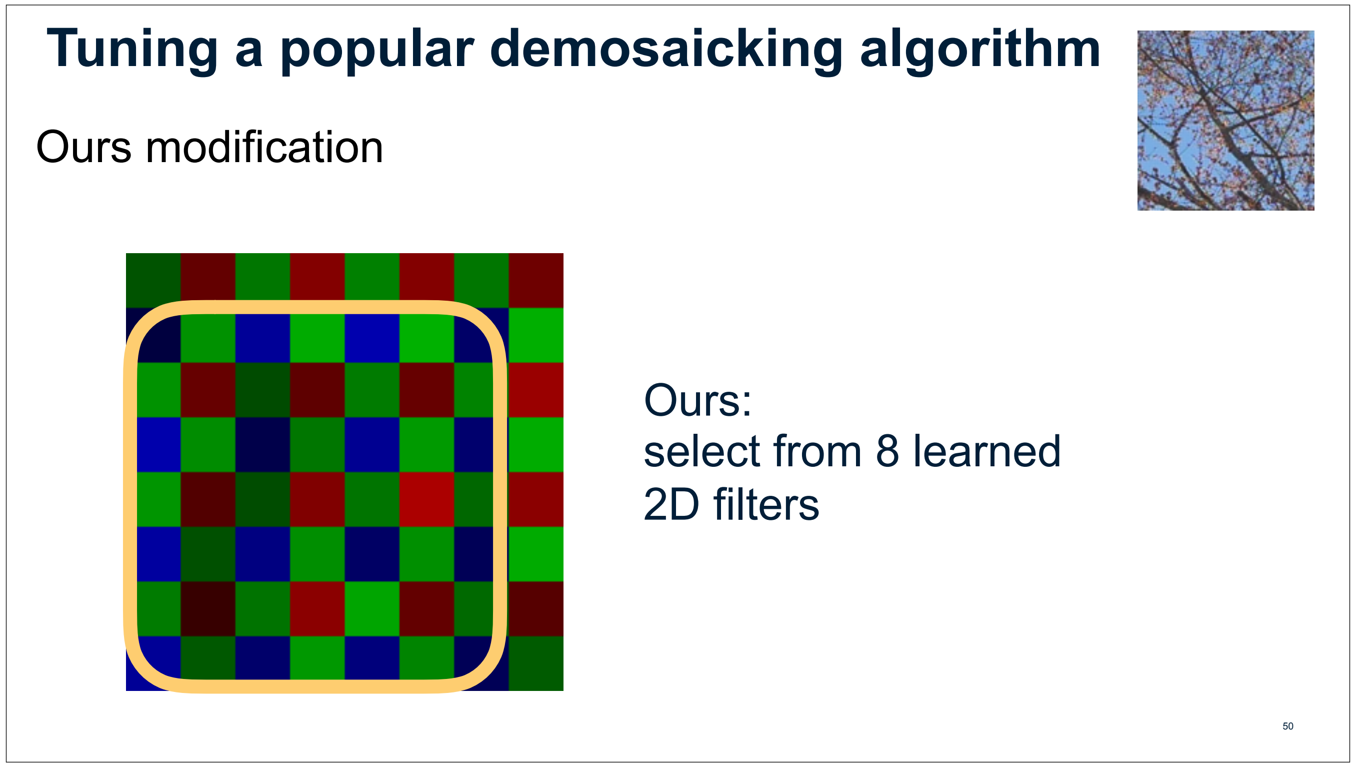
AHD [Hirakawa and Parks 2005]





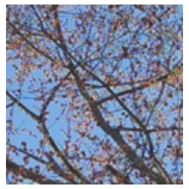
AHD:
select from horizontal &
vertical filters

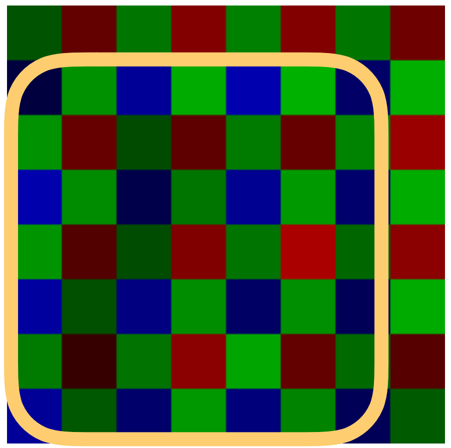
The AHD algorithm select from either a horizontal filter or a vertical filter, by choosing the direction which has less variation.



Tuning a popular demosaicking algorithm

Ours modification

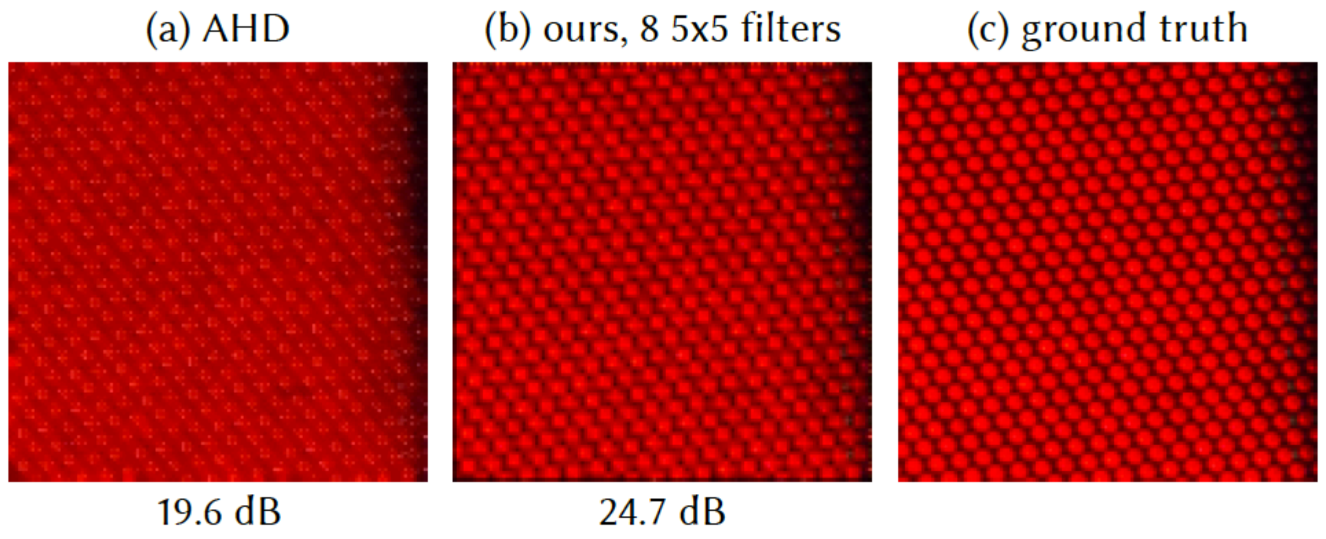




Ours:
select from 8 learned
2D filters

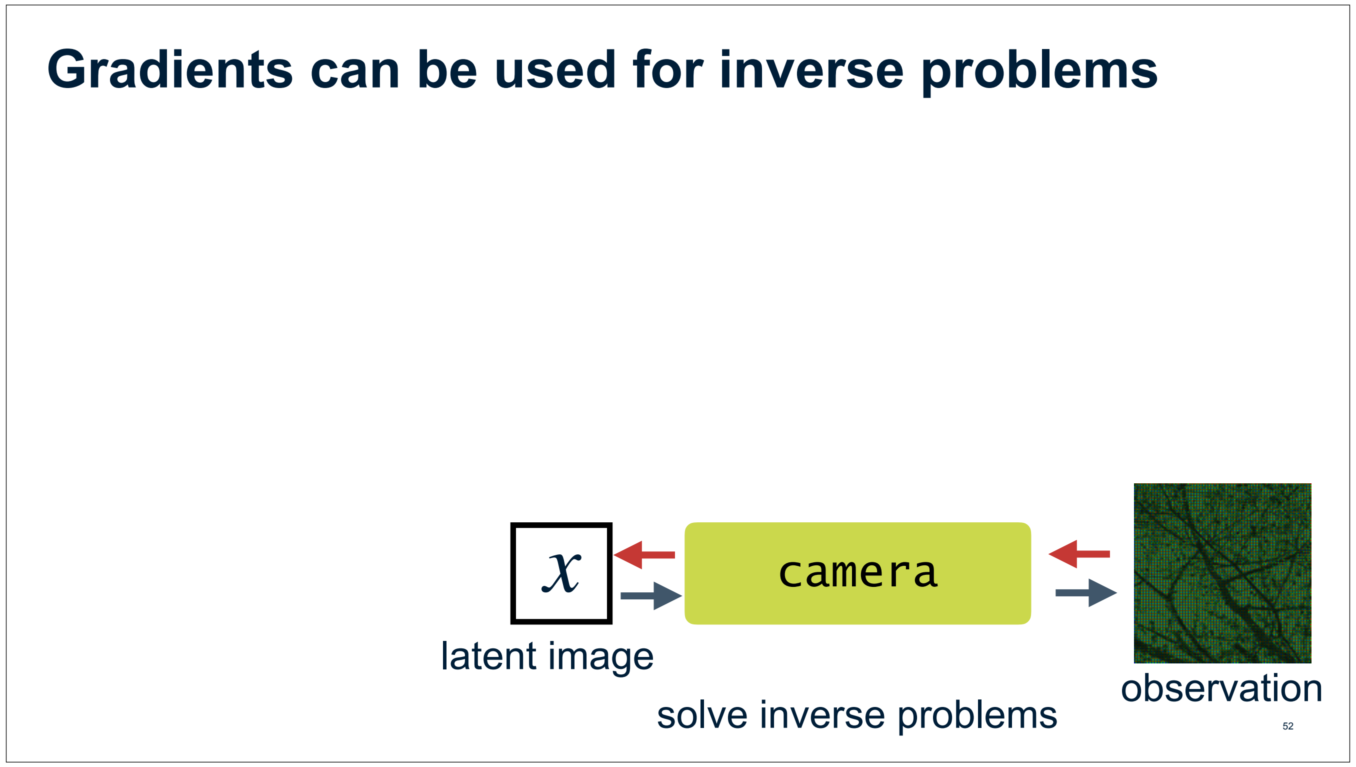
We modify the algorithm to select from 8 different 2D filters, where the parameters of all these filters are learned from data.
We use our system to generate the gradients of these parameters.

We improve AHD without much effort

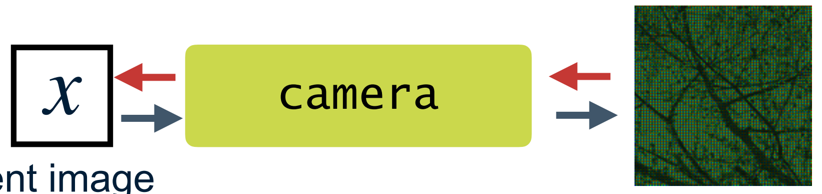


By training the algorithm on a dataset containing difficult demosaicking examples, we were able to get better result than the original AHD.
In this particular case it is very difficult to reconstruct the color just using horizontal or vertical filters due to the non-axis-aligned pattern.
All of this is kind of effortless: we first implement AHD in Halide, then we modify it to change the filters, then we get the gradients for free to optimize the filters.

I think this is really a new way to develop image processing algorithm where people haven’t explored enough yet.



Gradients can be used for inverse problems



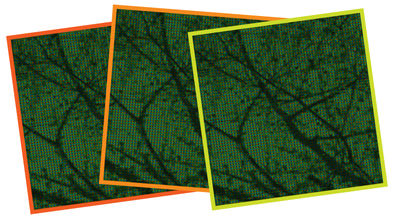
solve inverse problems

The next application I want to talk about is the case where you may or may not have training data, but you have some knowledge of how the data transform.
For example we want to recover the true image from the degradation of a camera.
We could model the camera as a differentiable function and solve for the unknown image using gradient descent.

Inverse problems: burst align & demosaick

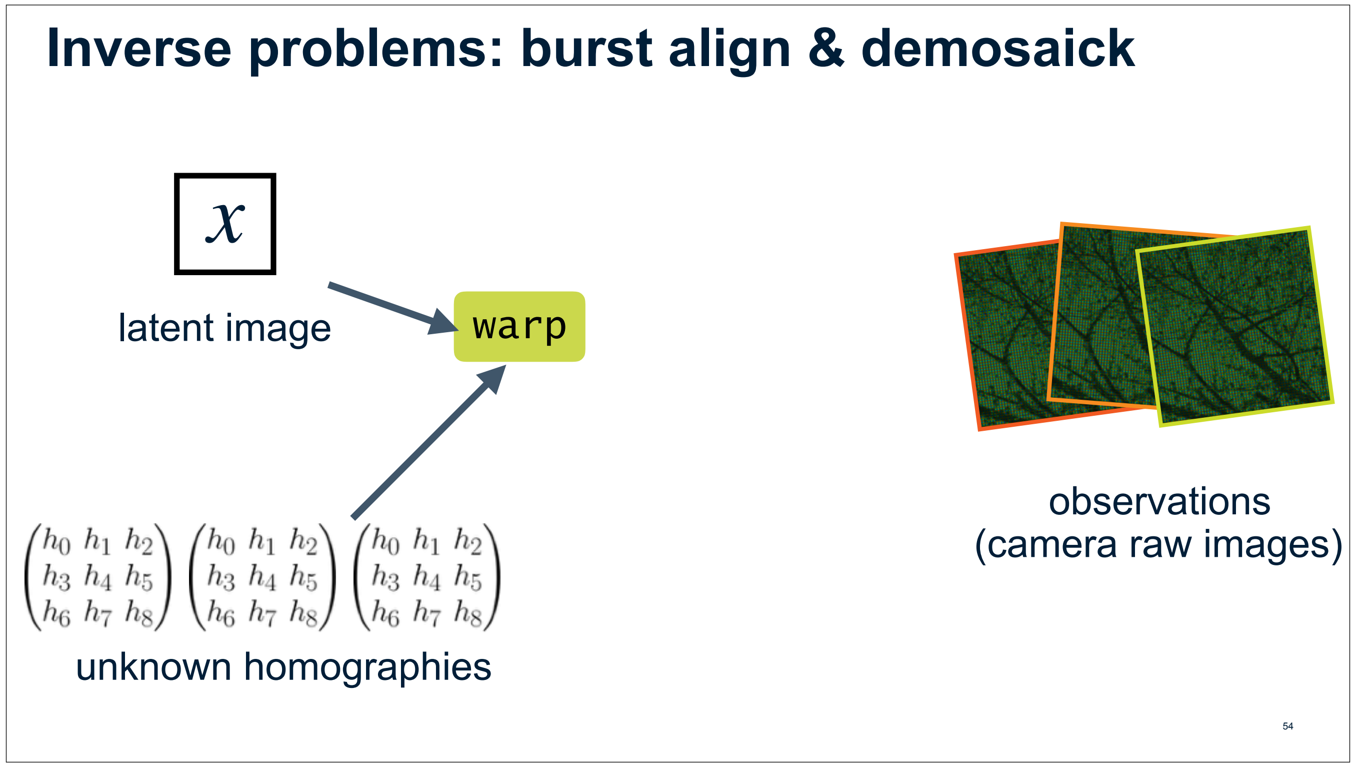
$\boxed{x}$

latent image



observations
(camera raw images)

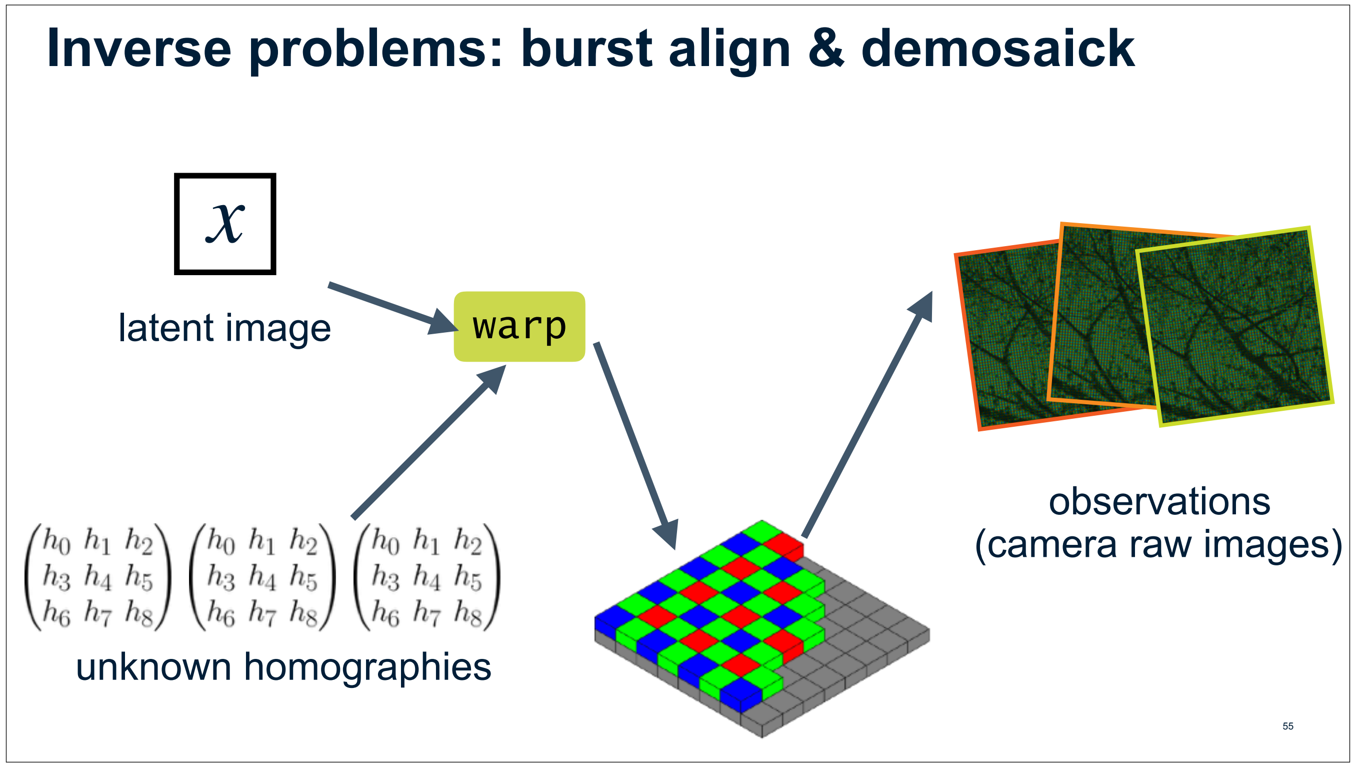
Our example consists of multiple camera raw images taken by some burst shots.
We want to find a latent image with full color that will give us these shots when taken from slightly different angles.



Inverse problems: burst align & demosaick

x
latent image
warp
$\begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix} \begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix} \begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix}$
unknown homographies
observations
(camera raw images)

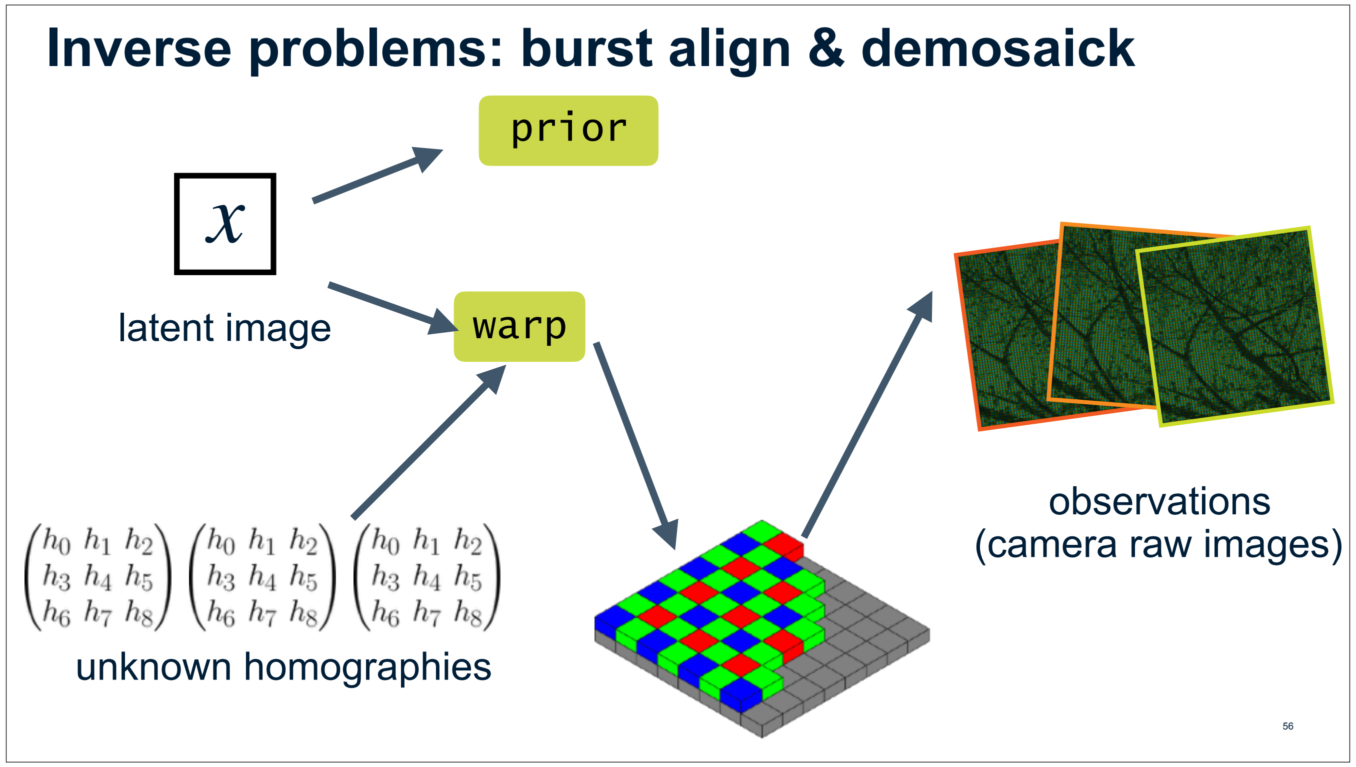
We assume that the latent image warps into multiple images by some unknown homographies.



Inverse problems: burst align & demosaick

x
latent image
warp
$\begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix}$ $\begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix}$ $\begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix}$
unknown homographies
observations
(camera raw images)

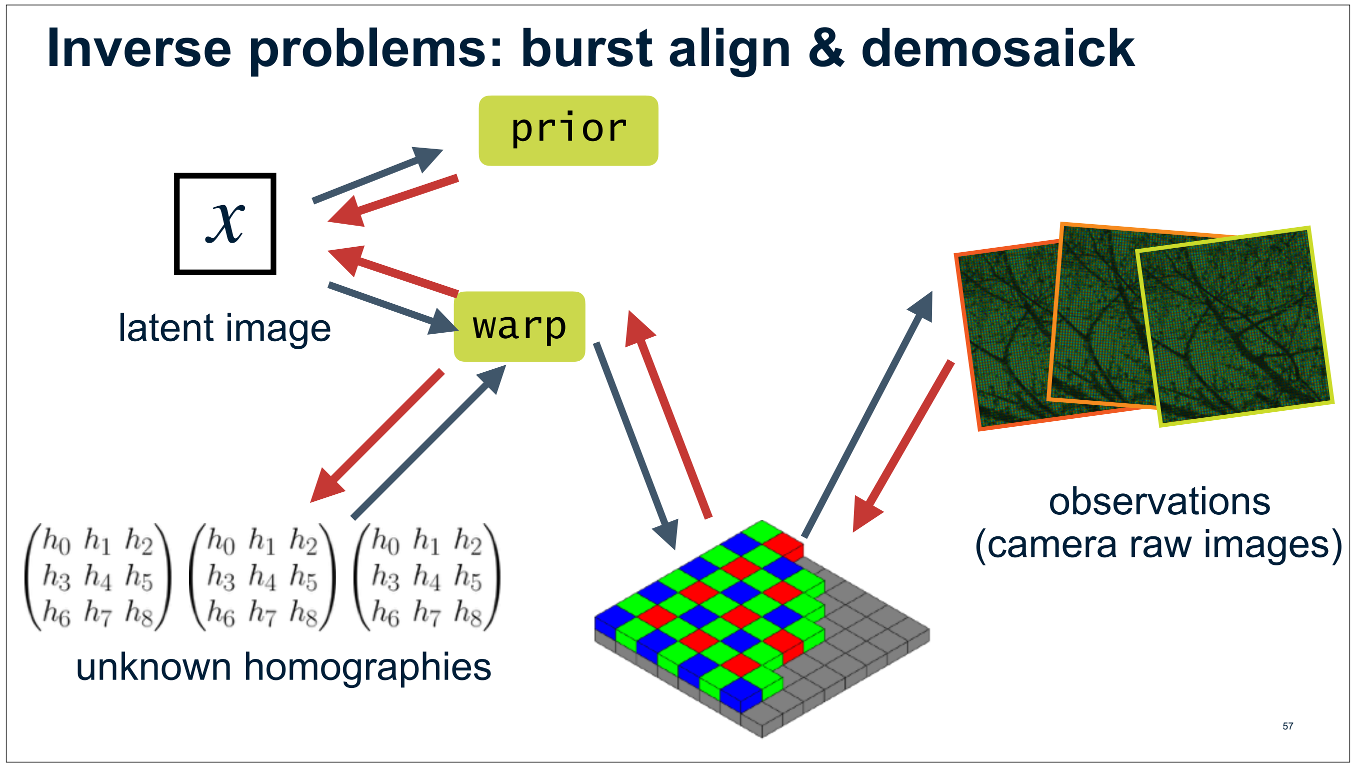
Remember the camera only record one channel per pixel.
So these warpped images are fed into a color filter array that leaves one channel of color per pixel.



Inverse problems: burst align & demosaick

prior
x
latent image
warp
$\begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix} \begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix} \begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix}$
unknown homographies
observations
(camera raw images)

The problem is ill-pose. There are many possible latent images that can give us the same result.
So we constraint the latent image by a simple smooth prior.

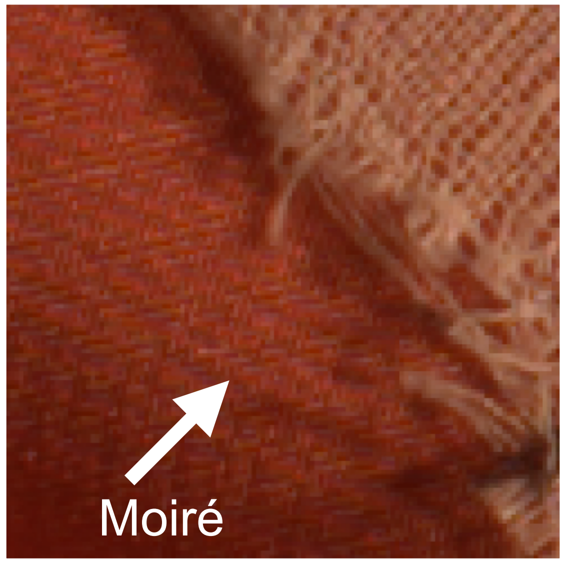


Inverse problems: burst align & demosaick

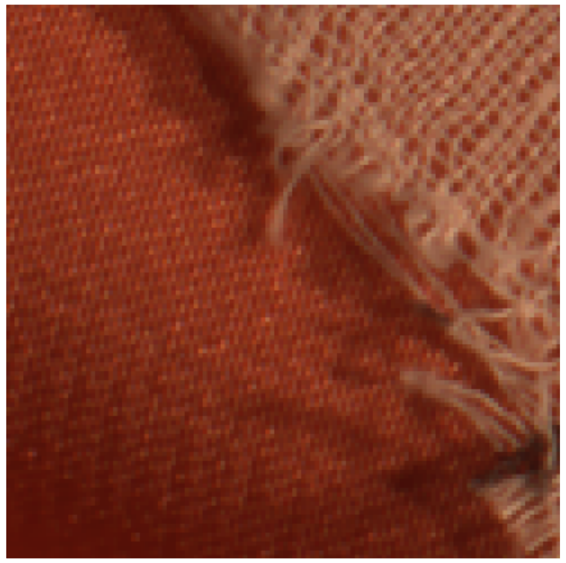
prior
x
latent image
warp
$\begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix} \begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix} \begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix}$
unknown homographies
observations
(camera raw images)

Then we compare the outputs with the actual observation and propagate the gradients all the way back to the unknown images and homographies to update them.

Inverse problems: burst align & demosaick

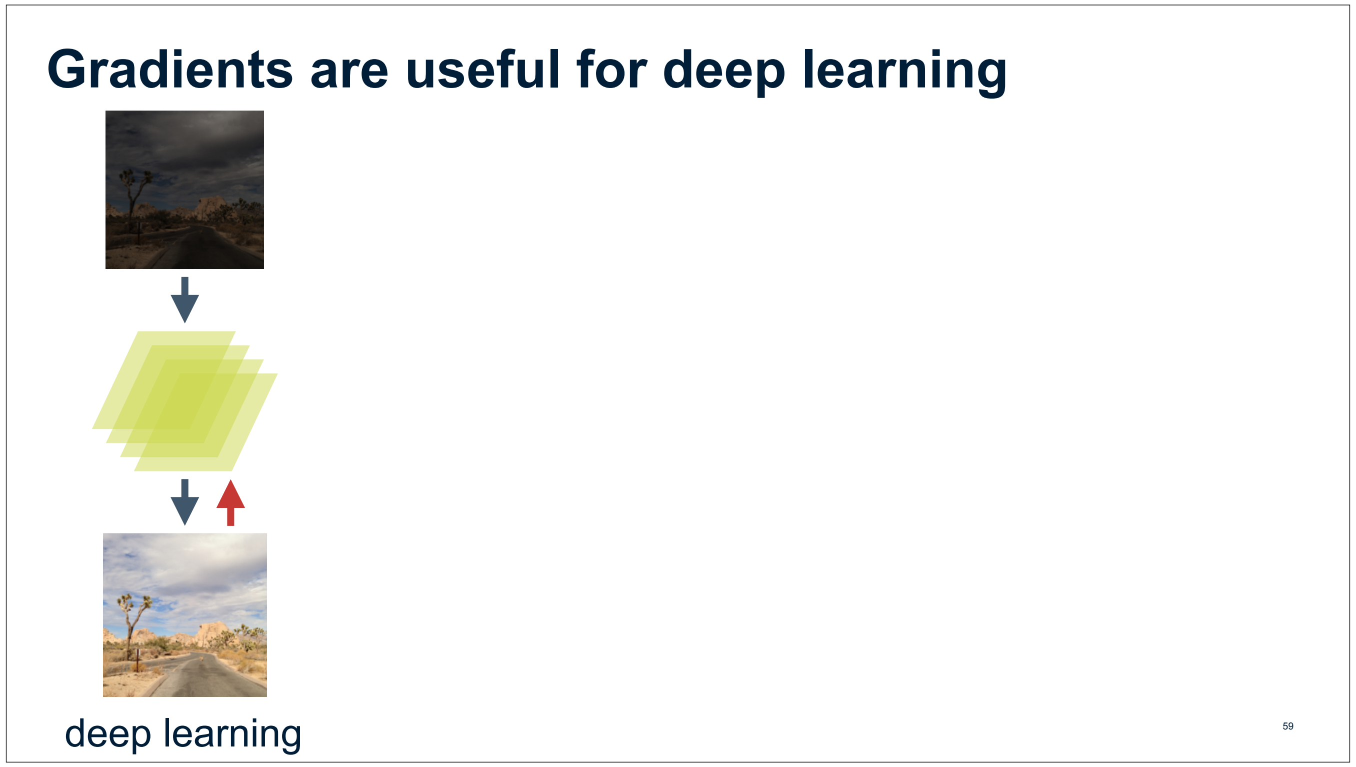


one frame demosaicked
(AHD)

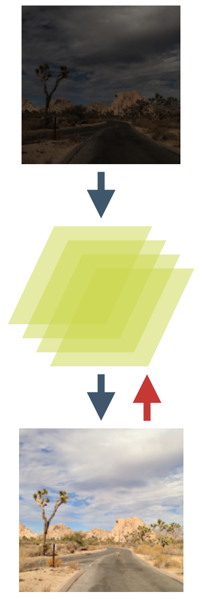


ours

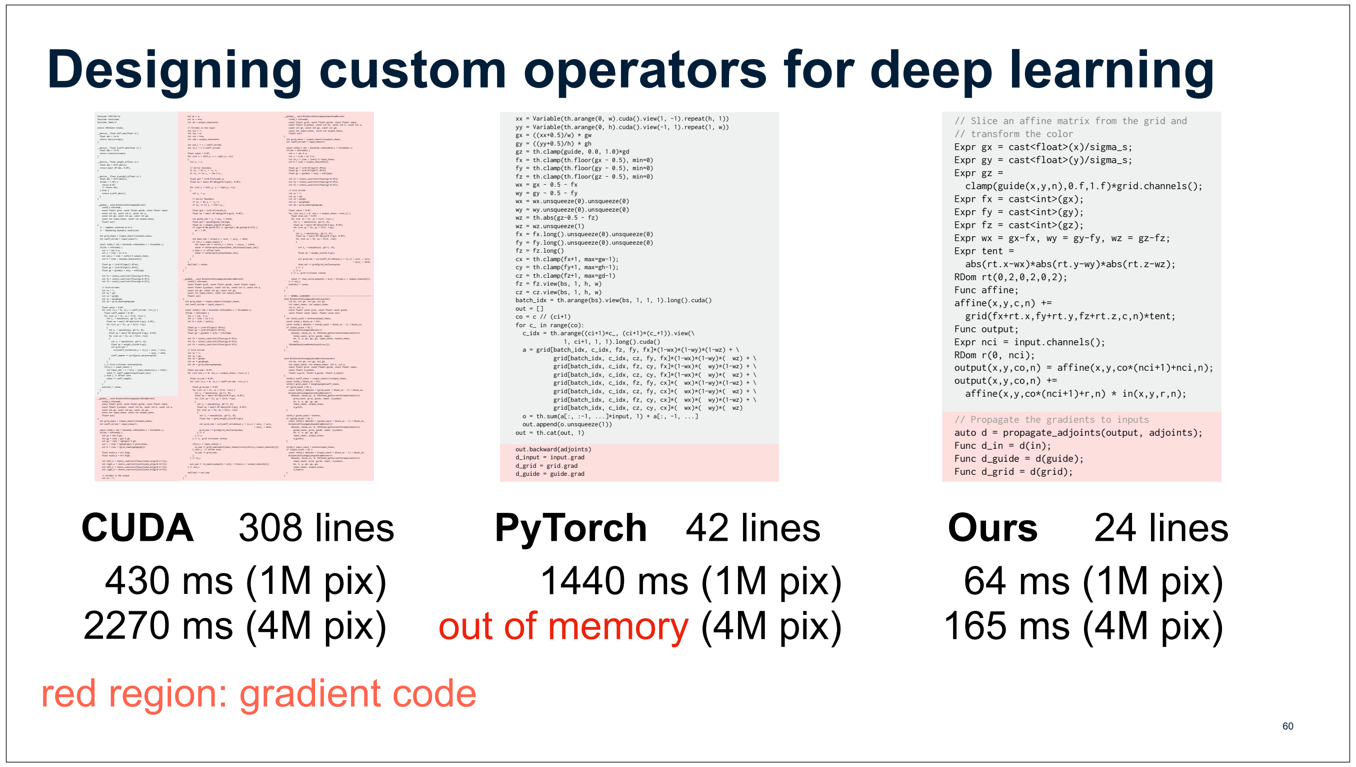
Here’s an result with comparison to demosaicking only using one frame with AHD.
If you use only one frame you would see some moire patterns due to undersampling.
On the other hand if we have multiple observations we essentially have more samples to resolve the moire.



Gradients are useful for deep learning



We can also use our framework for deep learning.



Designing custom operators for deep learning

[illegible]

CUDA 308 lines

430 ms (1M pix)
2270 ms (4M pix)

```
xx = Variable(th.arange(0, w).cuda().view(1, -1).repeat(h, 1))
yy = Variable(th.arange(0, h).cuda().view(-1, 1).repeat(1, w))
gx = ((xx+0.5)/w) * gw
gy = ((yy+0.5)/h) * gh
gz = th.clamp(guide, 0.0, 1.0)*gd
fx = th.clamp(th.floor(gx - 0.5), min=0)
fy = th.clamp(th.floor(gy - 0.5), min=0)
fz = th.clamp(th.floor(gz - 0.5), min=0)
wx = gx - 0.5 - fx
wy = gy - 0.5 - fy
wx = wx.unsqueeze(0).unsqueeze(0)
wy = wy.unsqueeze(0).unsqueeze(0)
wz = th.abs(gz-0.5 - fz)
wz = wz.unsqueeze(1)
fx = fx.long().unsqueeze(0).unsqueeze(0)
fy = fy.long().unsqueeze(0).unsqueeze(0)
fz = fz.long()
cx = th.clamp(fx+1, max=gw-1);
cy = th.clamp(fy+1, max=gh-1);
cz = th.clamp(fz+1, max=gd-1)
fz = fz.view(bs, 1, h, w)
cz = cz.view(bs, 1, h, w)
batch_idx = th.arange(bs).view(bs, 1, 1, 1).long().cuda()
out = []
co = c // (ci+1)
for c_ in range(co):
  c_idx = th.arange((ci+1)*c_, (ci+1)*(c_+1)).view(\
              1, ci+1, 1, 1).long().cuda()
  a = grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, fy, fx]*(1-wx)*(1-wy)*(  wz) + \
           grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, cy, fx]*(1-wx)*(  wy)*(  wz) + \
           grid[batch_idx, c_idx, fz, fy, cx]*(  wx)*(1-wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, fy, cx]*(  wx)*(1-wy)*(  wz) + \
           grid[batch_idx, c_idx, fz, cy, cx]*(  wx)*(  wy)*(1-wz) + \
           grid[batch_idx, c_idx, cz, cy, cx]*(  wx)*(  wy)*(  wz)
  o = th.sum(a[:, :-1, ...]*input, 1) + a[:, -1, ...]
  out.append(o.unsqueeze(1))
out = th.cat(out, 1)

out.backward(adjoints)
d_input = input.grad
d_grid = grid.grad
d_guide = guide.grad
```

PyTorch 42 lines

1440 ms (1M pix)
out of memory (4M pix)

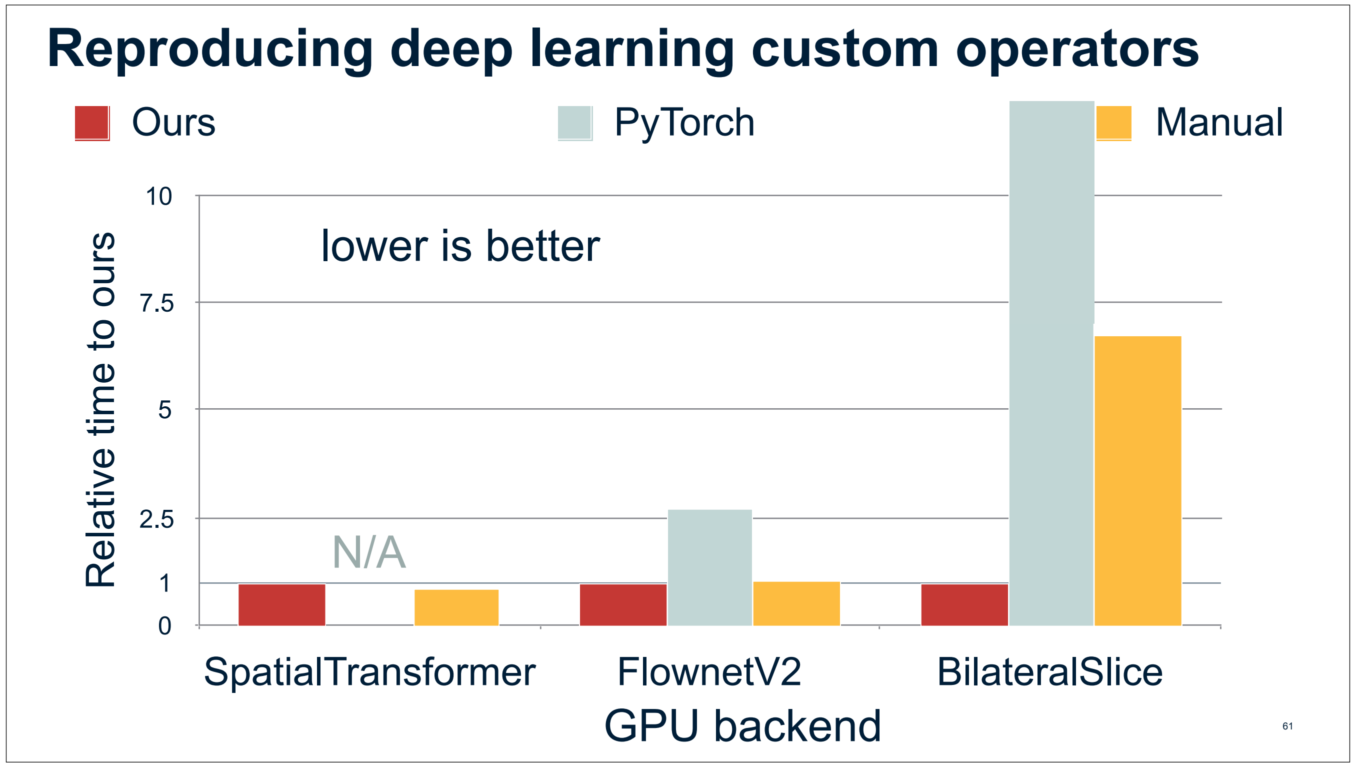
```
// Slice an affine matrix from the grid and
// transform the color
Expr gx = cast<float>(x)/sigma_s;
Expr gy = cast<float>(y)/sigma_s;
Expr gz =
  clamp(guide(x,y,n),0.f,1.f)*grid.channels();
Expr fx = cast<int>(gx);
Expr fy = cast<int>(gy);
Expr fz = cast<int>(gz);
Expr wx = gx-fx, wy = gy-fy, wz = gz-fz;
Expr tent =
  abs(rt.x-wx)*abs(rt.y-wy)*abs(rt.z-wz);
RDom rt(0,2,0,2,0,2);
Func affine;
affine(x,y,c,n) +=
  grid(fx+rt.x,fy+rt.y,fz+rt.z,c,n)*tent;
Func output;
Expr nci = input.channels();
RDom r(0, nci);
output(x,y,co,n) = affine(x,y,co*(nci+1)+nci,n);
output(x,y,co,n) +=
  affine(x,y,co*(nci+1)+r,n) * in(x,y,r,n);

// Propagate the gradients to inputs
auto d = propagate_adjoints(output, adjoints);
Func d_in = d(in);
Func d_guide = d(guide);
Func d_grid = d(grid);
```

Ours 24 lines

64 ms (1M pix)
165 ms (4M pix)

red region: gradient code

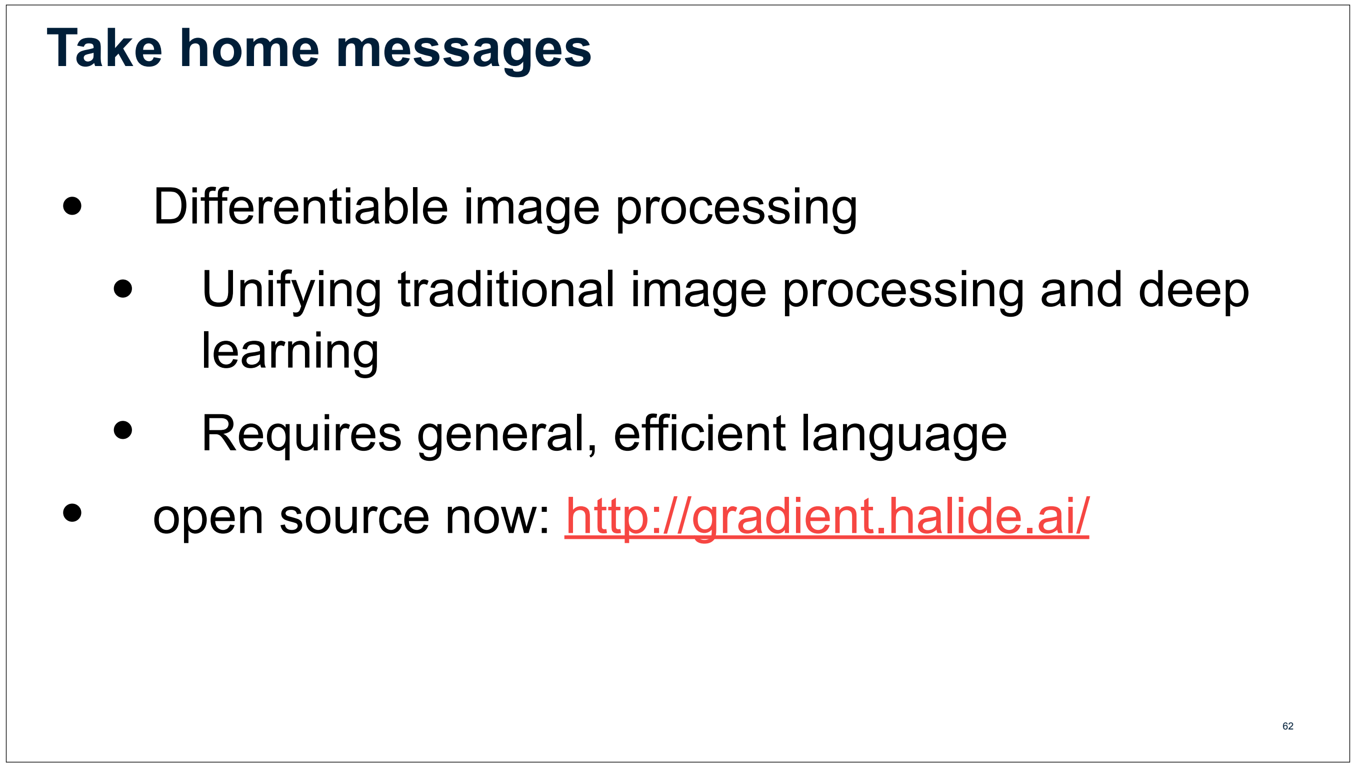
Like I mentioned before you can use our system to design custom operators for deep learning



Reproducing deep learning custom operators

We reproduce a few other custom deep learning operators and compare to their PyTorch equivalent.
At the time the authors of all these operators had to implement the derivatives by hand.
In general our method is as fast as highly optimized manual code, and much faster than PyTorch.

(pause for a second)



Take home messages

- Differentiable image processing
  - Unifying traditional image processing and deep learning
  - Requires general, efficient language
- open source now: http://gradient.halide.ai/

I would like to end the talk with some take home messages.
I think we are in witness of an emerging new paradigm of differentiable image processing, unifying the more traditional image processing and deep learning.
To do this we need to make our programming language differentiable.
It is important to make these programming languages general and efficient for fast iteration.
We shouldn't draw a line between deep learning and traditional image processing.
By taking the training scheme from deep learning we can improve image processing.
By studying the nonlinearities in image processing we can improve deep learning.
The code is open source now as a branch of Halide and we're working on upstreaming the code to master.
Apparently we have registered the halide.ai domain, and the link on the slide will take you to our project page.