Recursive filtering in Halide

Dillon Sharlet, Google

Description

- Recursive filter for a 1D signal

$$\mathbf{y}_n = (1 - A)\,\mathbf{y}_{n-1} + A\,\mathbf{x}_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

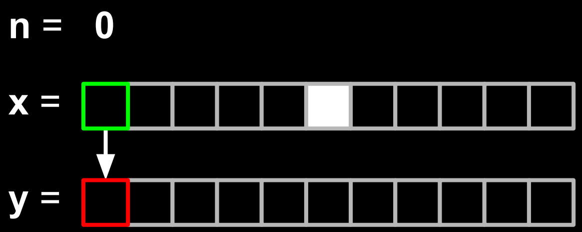
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\ y_{n-1} + A\ x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



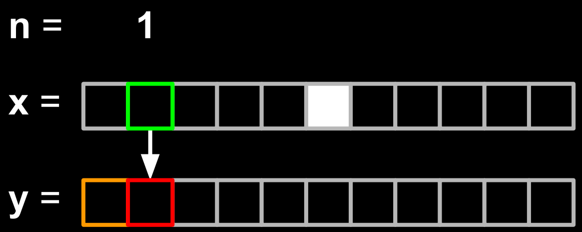
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



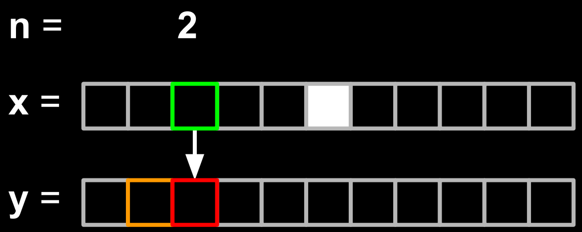
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



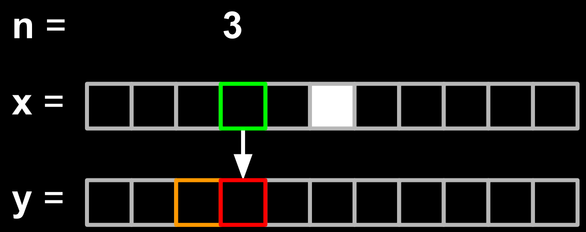
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



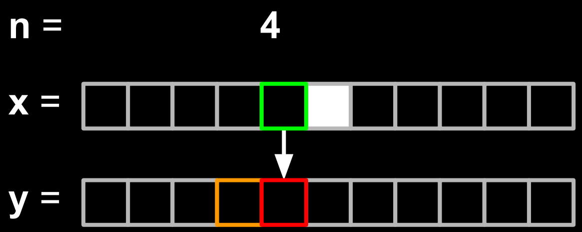
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



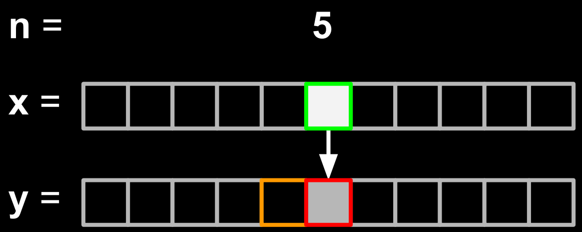
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



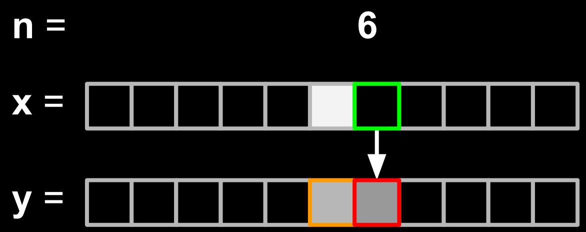
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



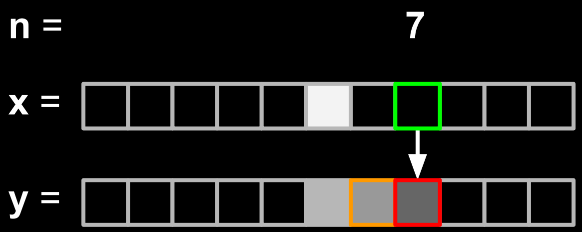
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



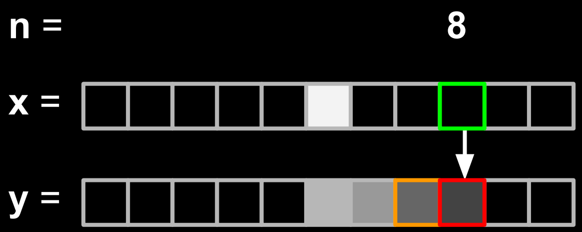
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



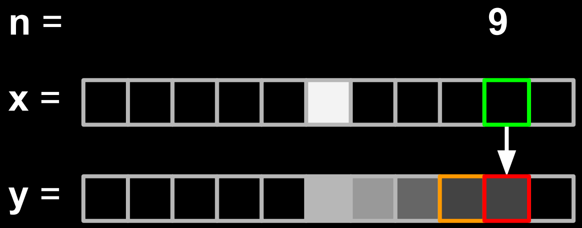
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



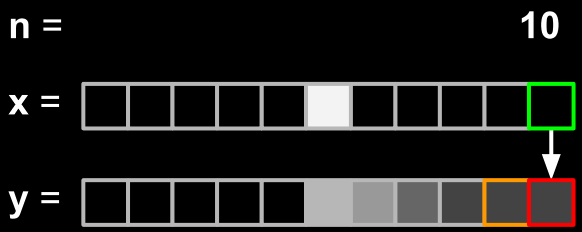
Description

- Recursive filter for a 1D signal

$$y_n = (1 - A)\, y_{n-1} + A\, x_n$$

where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



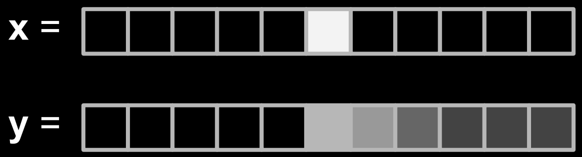
Description

- Recursive filter for a 1D signal

$$\mathbf{y}_n = (1 - A)\,\mathbf{y}_{n-1} + A\,\mathbf{x}_n$$

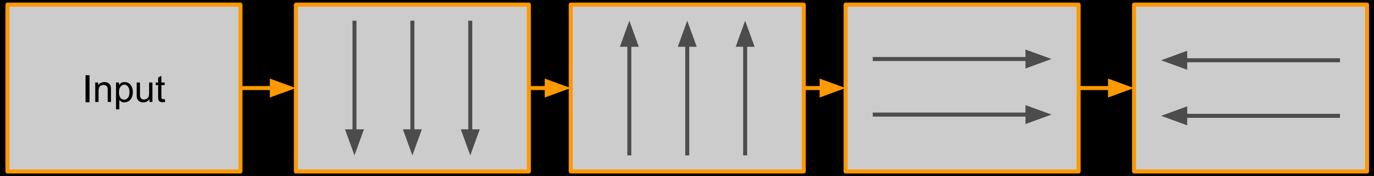
where **x** is input, **y** is output, **A** is the filter coefficient

- Example applied to a delta function



Description

- To apply this recursive filter to an image, apply it four times:
  a. Up and down the columns
  b. Right and left across the rows



Reductions

- To implement this algorithm:
  - Need to reference output at previous pixel to compute current output
- This cannot be done with a pure definition
- We can do this with **update stages** and **`RDom`s**
  - `RDom` (**R**eduction **Dom**ain) provides a serial loop
  - Can have dependencies between loop iterations

Multi-stage Funcs

```
f(x, y) = x + y;
f(x, 0) += 5;
```

Funcs can have multiple stages

We call the additional ones “update” stages

They run in sequence

```
f(x, y) = x + y;
f(x, 0) += 5;
```

They can use arbitrary index expressions on the left-hand-side

```
f(x, y) = x + y;
f(x, 0) += 5;
// f(x, 0) = f(x, 0) + 5;
```

They can recursively load values defined by the previous stage

```
f(x, y) = x + y;
f(x, 0) += 5;
f.vectorize(x, 8);
```

They are scheduled independently

```
f(x, y) = x + y;
f(x, 0) += 5;
f.vectorize(x, 8);
f.update(0)
 .unroll(x, 2);
```

They are scheduled independently

```
f(x, y) = x + y;
f(x, 0) += 5;
```

```
for y:
 for x:
  f[x,y] = x + y
```

```
f(x, y) = x + y;
f(x, 0) += 5;
```

```
for y:
 for x:
  f[x,y] = x + y
for x:
 f[x,0] = f[x,0] + 5;
```

```
f(x, y) = x + y;
RDom r(1, 10);
f(x, 0) += f(x, r);
```

An update stage can be a reduction over some domain "RDom"

```
f(x, y) = x + y;
RDom r(1, 10);
f(x, 0) += f(x, r);
```

This just throws an extra loop around the loop nest for that stage:

```
for r from 1 to 10:
 for x:
  f[x,0] = f[x,0] + f[x,r];
```

```
f(x, y) = x + y;
RDom r(1, 10);
f(x, 0) += f(x, r);
f.update(0)
 .unroll(r);
```

You can schedule RDom variables

```
f(x, y) = x + y;
RDom r(1, 10);
f(x, 0) += f(x, r);
f.update(0)
  .reorder(r, x);
```

You can schedule RDom variables

```
f(x, y) = x + y;
RDom r(1, 10);
f(x, 0) += f(x, r);
f.update(0)
  .parallel(r);
```

ERROR: Potential Race Condition

But only when we can prove there's no race condition or change in meaning.

Halide's promise:

Scheduling never changes the results!

Generators

- Two ways to call Halide code
  - **JIT**: Halide pipelines executed in the same process they are defined in
  - **AOT**: Halide pipelines compiled to object files (.o, .obj) and linked into/called from another program via C ABI (i.e. extern “C”)

Generators

- Generators are C++ programs that, when run, produce objects (.o, .obj) and C headers (.h) containing compiled pipelines
- Applications `#include` generated header files declaring the functions, link to generated objects
- Pipeline functions are declared with arguments corresponding to `Param` objects, including `ImageParam`s in `buffer_t` objects.
    - Holds pointer, element size and strides of each dimension of an image
    - Halide never assumes ownership of the memory a buffer_t points to

Using Generators with Matlab

- Generators can also be used within Matlab (or Octave) via the mex library interface
- Halide pipeline compiled with `matlab` target feature defines a suitable `mexFunction` wrapper
  - Validates and converts `mxArray` to `buffer_t` (or scalar params)
- `mex_halide` Matlab function performs all the required steps to build a mex library from a source file containing a generator

Code!

Scheduling for locality

- So far, we've talked about some scheduling operators
  - vectorize, unroll, etc.
- We've also briefly discussed **compute_at**
- To significantly improve performance, we need to use **compute_at** to improve locality

compute_root

```
f(x, y) = x + y;
g(x, y) = 2*f(x, y);
```

Here is a simple two stage pipeline

compute_root

```
f(x, y) = x + y;
g(x, y) = 2*f(x, y);
f.compute_root();
g.compute_root();
```

This means compute all of f, followed by all of g

Poor locality!

compute_root

```
f(x, y) = x + y;
g(x, y) = 2*f(x, y);
f.compute_root();
g.compute_root();
```

```
for f.y:
 for f.x:
  f[f.x,f.y] = f.x + f.y
for g.y:
 for g.x:
  g[g.x,g.y] = 2*f[g.x,g.y]
```

compute_at

```
f(x, y) = x + y;
g(x, y) = 2*f(x, y);
f.compute_at(g, y);
g.compute_root();
```

“Compute f at each iteration of y when computing g”

All stages of a Func share the same compute_at location

compute_at

```
f(x, y) = x + y;
g(x, y) = 2*f(x, y);
f.compute_at(g, y);
g.compute_root();
```

```
for g.y:
  for g.x:
    g[g.x,g.y] = 2*f[g.x,g.y]
```

compute_at

```
f(x, y) = x + y;
g(x, y) = 2*f(x, y);
f.compute_at(g, y);
g.compute_root();
```

```
for g.y:
  for f.x:
    f[f.x,g.y] = f.x + g.y
  for g.x:
    g[g.x,g.y] = 2*f[g.x,g.y]
```

compute_at

```
f(x, y) = x + y;
g(x, y) = 2*f(x, y);
f.compute_at(g, x);
g.compute_root();
```

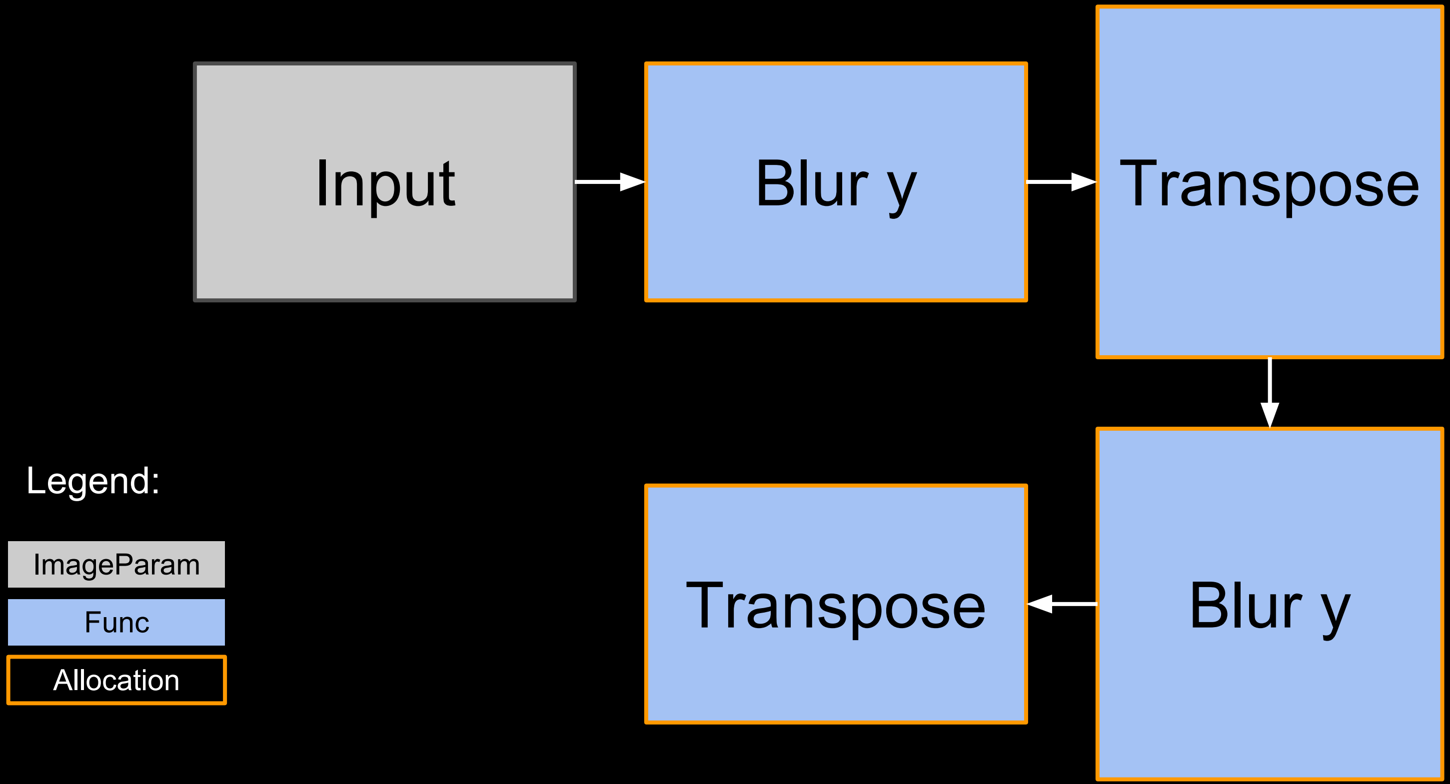
```
for g.y:
 for g.x:
  g[g.x,g.y] = 2*f[g.x,g.y]
```

compute_at

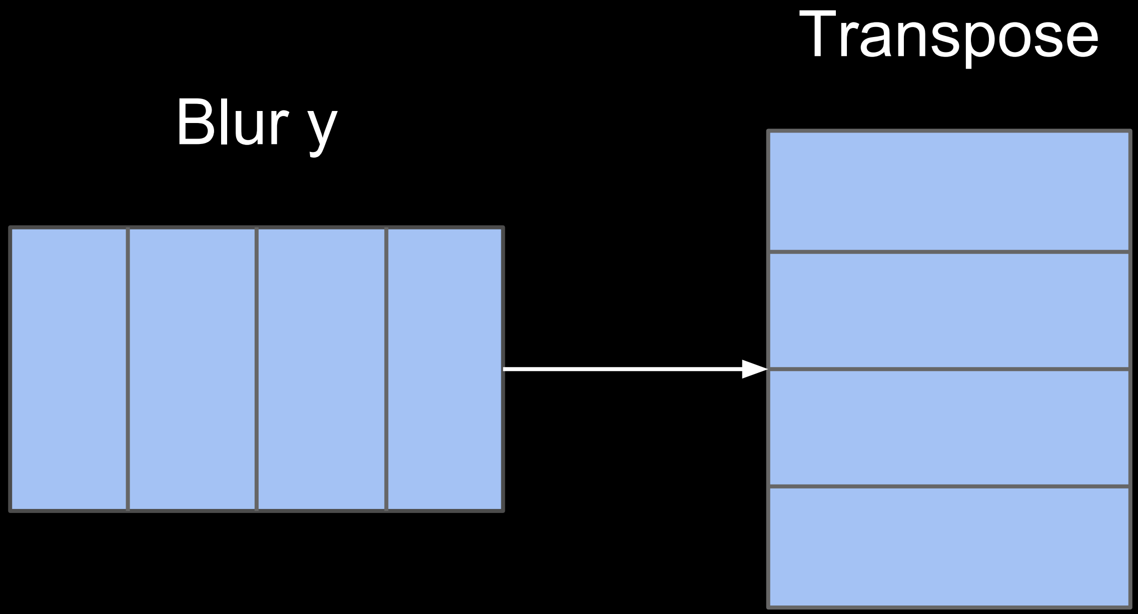
```
f(x, y) = x + y;
g(x, y) = 2*f(x, y);
f.compute_at(g, x);
g.compute_root();
```

```
for g.y:
 for g.x:
  f[g.x,g.y] = g.x + g.y
  g[g.x,g.y] = 2*f[g.x,g.y]
```

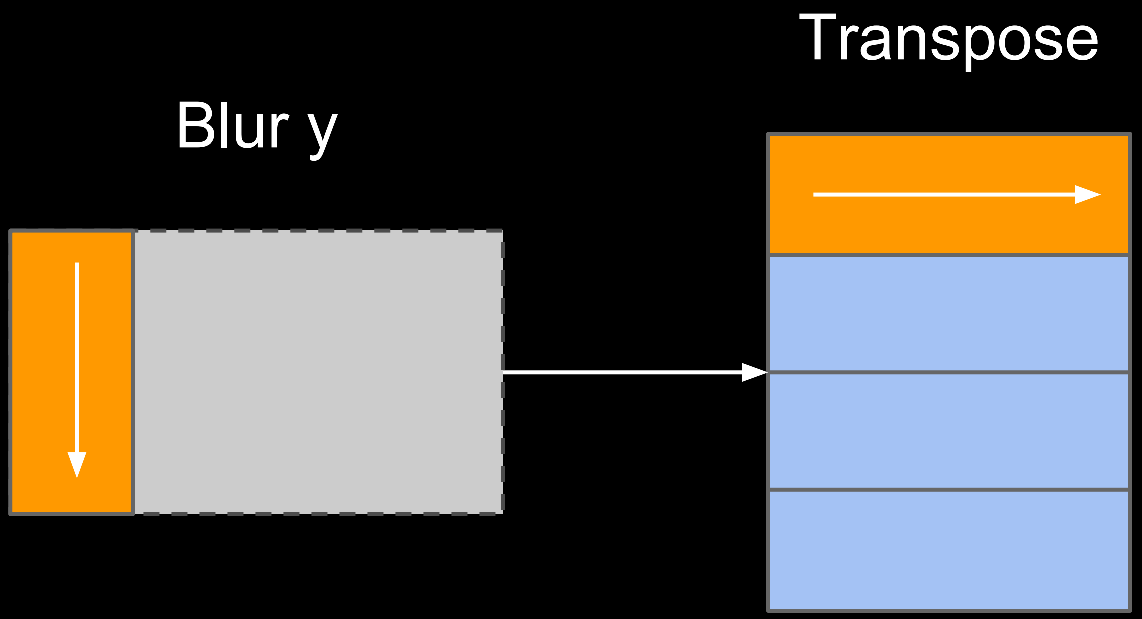
IIR blur compute_root visualization



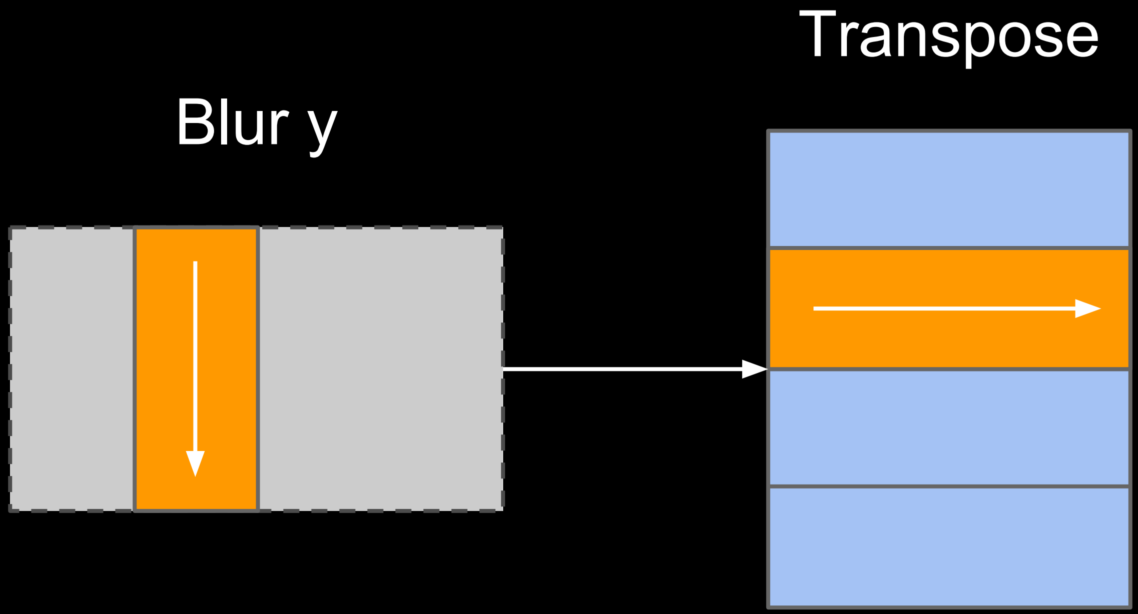
IIR blur locality schedule visualization



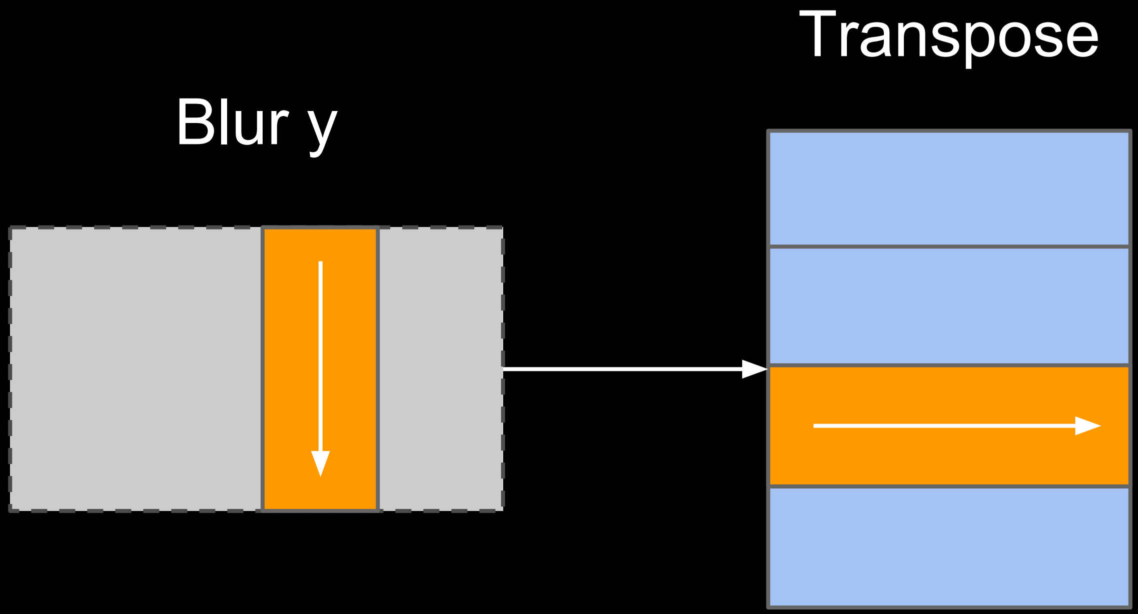
IIR blur locality schedule visualization



IIR blur locality schedule visualization



IIR blur locality schedule visualization



Code!

https://github.com/halide/CVPR2015/tree/master/RecursiveFilter