

# Dynamic Scheduling in Halide

*cache rules everything around me*

Ben Blum ([bblum@andrew.cmu.edu](mailto:bblum@andrew.cmu.edu))

Carnegie Mellon University

2013, December 16

```
compute_foo();
```

```
if (!cache_contains(foo)) {  
    compute_foo();  
    cache_insert(foo);  
}
```

# Outline

## Halide Review

- ▶ Function Definitions
- ▶ Function Scheduling

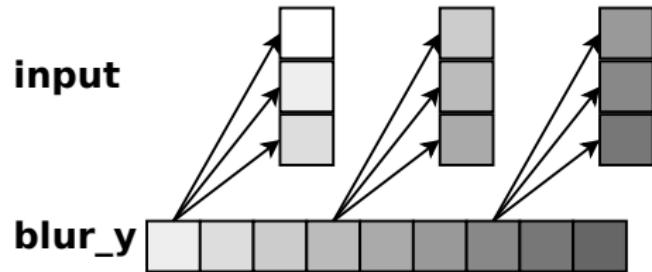
## Dynamic Scheduling

- ▶ Problem: Dynamic Vector Fields
- ▶ Solution: Cache Computed Values

## Evaluation

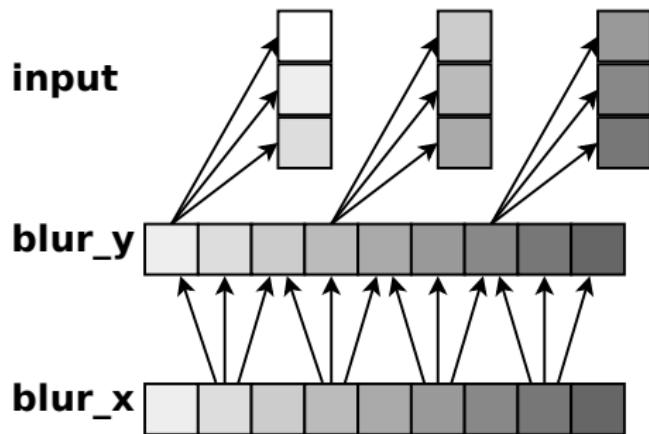
# Image Processing is Functional Programming

```
blur_y(x,y) =  
    (input(x, y-1) +  
     input(x, y ) +  
     input(x, y+1))/3;
```



# Image Processing is Functional Programming

```
blur_y(x,y) =  
    (input(x, y-1) +  
     input(x, y ) +  
     input(x, y+1))/3;  
  
blur_x(x,y) =  
    (blur_y(x-1, y) +  
     blur_y(x, y) +  
     blur_y(x+1, y))/3;
```



# ...embedded in C++

```
void main(int argc, char **argv)
    Image<float> input = load(argv[1]);
    Image<float> output(input.width(), input.height());
    Func blur_y, blur_x;
    Var x,y;
    blur_y(x,y) = (input(x, y-1) + ...)/3;
    blur_x(x,y) = (blur_y(x-1, y) + ...)/3;

    blur_x.realize(output); // JITs and runs the blur
    save(output, "output.png");
}
```

## ...embedded in C++

```
class Expr { ... };
class Func {
    operator()(Expr x, Expr y, Expr z, ...);
    ...
};

void main(int argc, char **argv)
    Image<float> input = load(argv[1]);
    Image<float> output(input.width(), input.height());
    Func blur_y, blur_x;
    Var x,y;
    blur_y(x,y) = (input(x, y-1) + ...)/3;
    blur_x(x,y) = (blur_y(x-1, y) + ...)/3;

    blur_x.realize(output); // JITs and runs the blur
    save(output, "output.png");
}
```

# Functions are Automatically “Scheduled”

```
blur_y(x,y) =          float blur_y[HEIGHT][WIDTH];  
    (input(x, y-1) +      for (row = 0 to HEIGHT) {  
        input(x, y, c) +      for (col = 0 to WIDTH) {  
        input(x, y+1))/3;      blur_y[row][col] = ...;  
                            }  
                            }  
  
blur_x(x,y) =          float blur_x[HEIGHT][WIDTH];  
    (blur_y(x-1, y) +      for (row = 0 to HEIGHT) {  
        blur_y(x, y) +      for (col = 0 to WIDTH) {  
        blur_y(x+1, y))/3;      blur_x[row][col] = ...;  
                            }  
                            }
```

# Functions are Automatically “Scheduled”

```

blur_y(x,y) =
  (input(x, y-1) +
   input(x, y,    c) +
   input(x, y+1))/3;

blur_x(x,y) =
  (blur_y(x-1, y) +
   blur_y(x,    y) +
   blur_y(x+1, y))/3;

```

```

float blur_y[HEIGHT][WIDTH];
for (row = 0 to HEIGHT) {
    for (col = 0 to WIDTH) {
        blur_y[row][col] = ...;
    }
}
float blur_x[HEIGHT][WIDTH];
for (row = 0 to HEIGHT) {
    for (col = 0 to WIDTH) {
        blur_x[row][col] = ...;
    }
}

```

- ▶ Bad locality: Traversing the whole image twice
- ▶ Bad parallelism: One thread per row/column/pixel??

# Function Schedules Can Be Configured

```
blur_x.tile(x, y, x2, y2, 8, 8).parallel(x, y);  
blur_y.store_at(blur_x, x).compute_at(blur_x, y2);
```

# Function Schedules Can Be Configured

```
blur_x.tile(x, y, x2, y2, 8, 8).parallel(x, y);
blur_y.store_at(blur_x, x).compute_at(blur_x, y2);

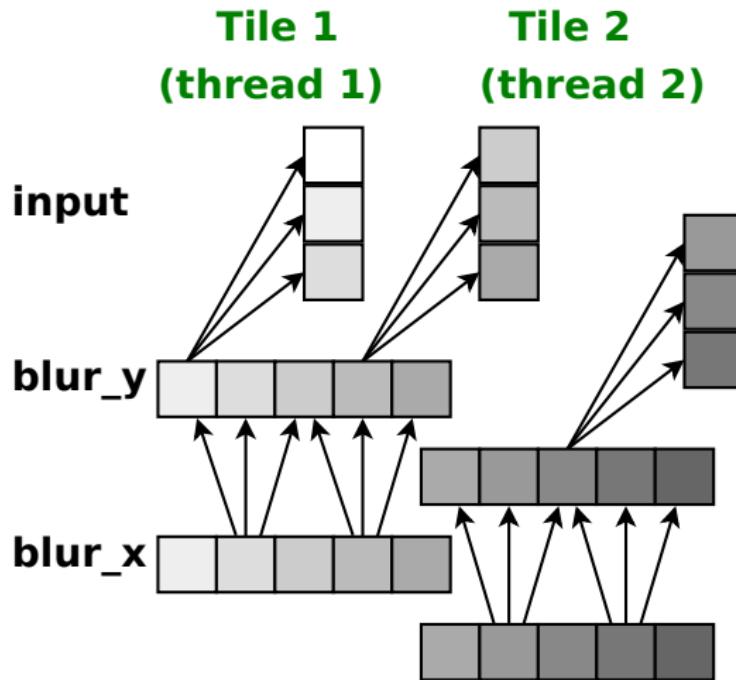
float blur_x[HEIGHT][WIDTH];
PARALLEL for (row = 0 to 8) {
    PARALLEL for (col = 0 to 8) {
        float blur_y[HEIGHT/8][WIDTH/8];
        for (row2 = 0 to WIDTH/8) {
            for (col2 = 0 to WIDTH/8) {
                blur_y[row*8 + row2][col*8 + col2] = ...;
            }
            for (col2 = 0 to WIDTH/8) {
                blur_x[row*8 + row2][col*8 + col2] = ...;
            }
        }
    }
}
```

# Function Schedules Can Be Configured

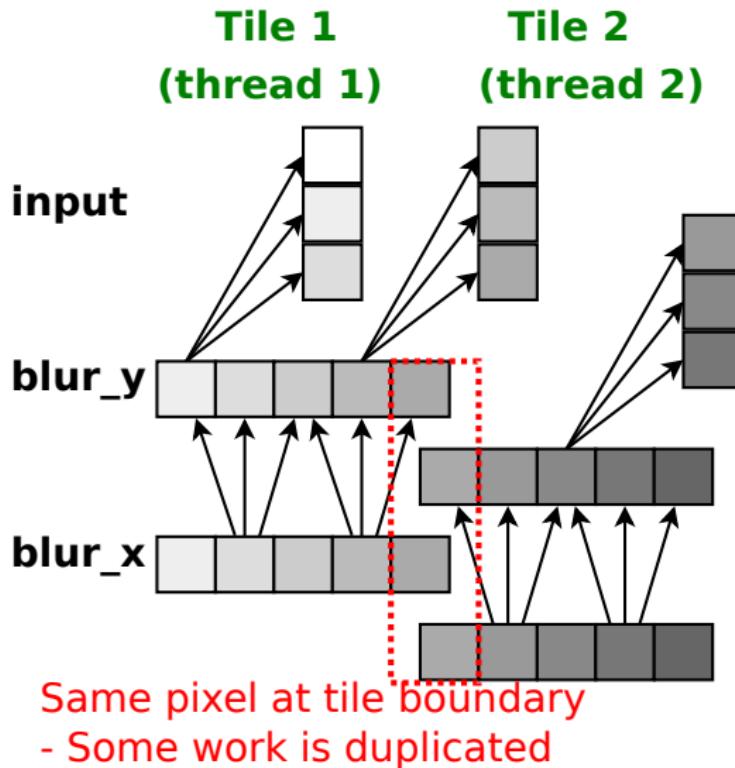
```
blur_x.tile(x, y, x2, y2, 8, 8).parallel(x, y);
blur_y.store_at(blur_x, x).compute_at(blur_x, y2);

float blur_x[HEIGHT][WIDTH];
PARALLEL for (row = 0 to 8) { // Good parallelism
    PARALLEL for (col = 0 to 8) {
        float blur_y[HEIGHT/8][WIDTH/8]; // Low memory usage
        for (row2 = 0 to WIDTH/8) {
            for (col2 = 0 to WIDTH/8) { // Good locality
                blur_y[row*8 + row2][col*8 + col2] = ...;
            }
            for (col2 = 0 to WIDTH/8) {
                blur_x[row*8 + row2][col*8 + col2] = ...;
            }
        }
    }
}
```

# Tiling Helps Parallelism and Locality



# Tiling Helps Parallelism and Locality



# Dynamic Scheduling

# Static vs Dynamic Data Dependencies

Avoiding duplicate work when tiling relies on **static data dependencies**.

- ▶ In 3x3 blur, max distance from output to input location was 1.

However, some algorithms compute which pixels to read from dynamically.

- ▶ Patch-match
- ▶ Edge detection

Which input pixels each output pixel will need to compute (the **vector field**) is not known until runtime.

# Dynamic Vector Field Example

Original image



# Dynamic Vector Field Example

Original image



Invert colours



# Dynamic Vector Field Example

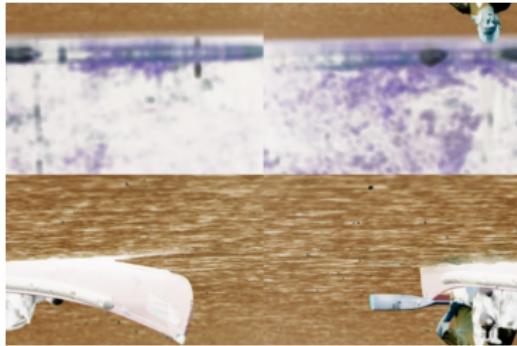
Original image



Invert colours



Flip & Blur



# Dynamic Vector Field Example

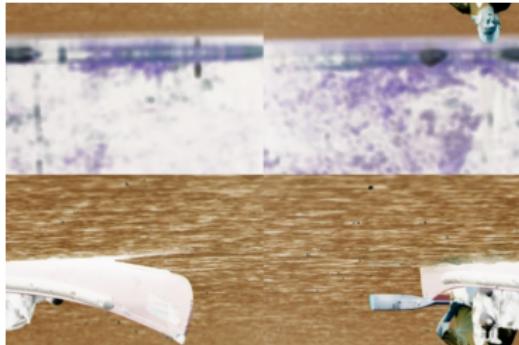
Original image



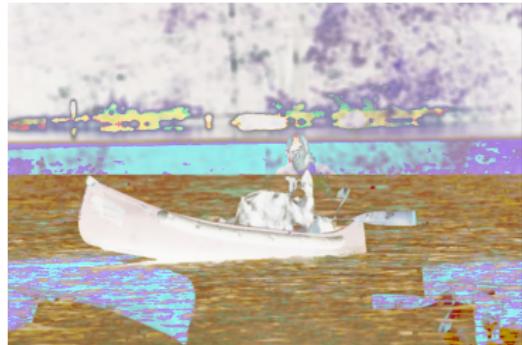
Invert colours



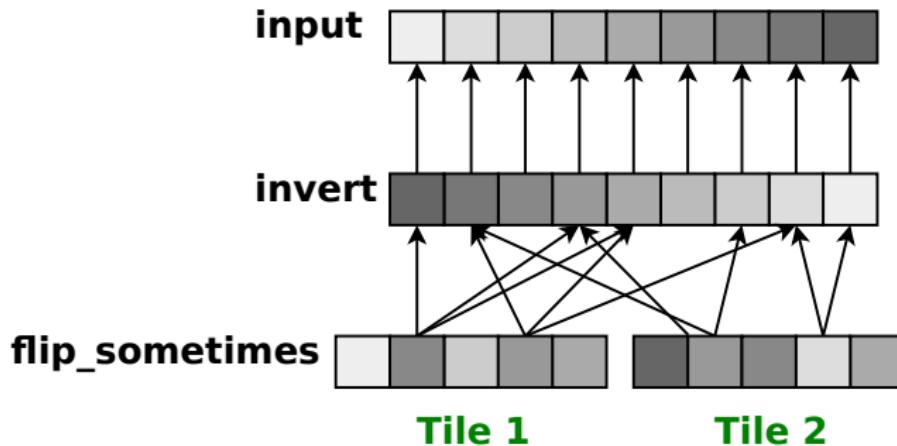
Flip & Blur



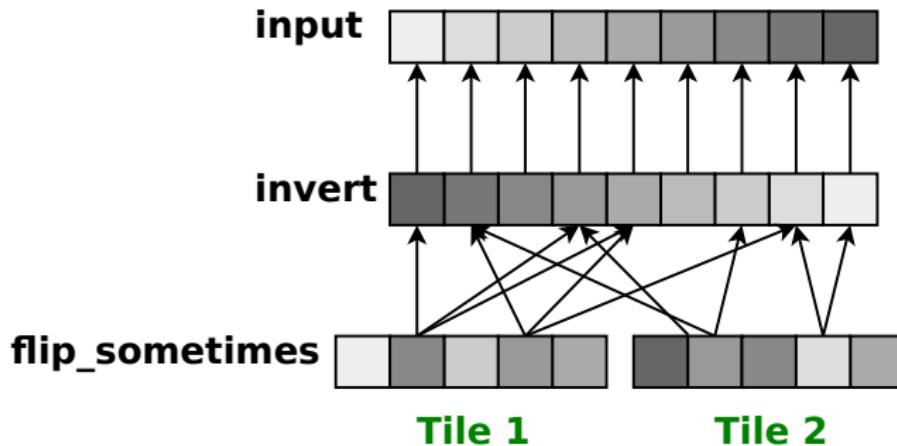
Flip conditionally by colour of pixel



# Dynamic Vector Fields Cause Duplicate Work



# Dynamic Vector Fields Cause Duplicate Work



Goal: When tile 1 computes `invert()` on a pixel, **remember it** for tile 2.

# Solution: Caching Computed Results

Generated code, before:

```
for (tile = ...) {    // Work duplicated across iterations
    for (row = ...)  {
        for (col = ...)  {

            inverted[row][col] = ...;

        }
    }
    (...  compute flipped ...)
}
```

# Solution: Caching Computed Results

Generated code, after:

```
bool result_computed[...];
for (tile = ...) { // Work cached across iterations
    for (row = ...) {
        for (col = ...) {
            if (!result_computed[row][col]) {
                inverted[row][col] = ...;
                result_computed[row][col] = true;
            }
        }
    }
    (... compute flipped ...)
}
```

# Evaluation

# Evaluation

## Evaluation Questions

- ▶ How much duplicate work does this let us avoid?
- ▶ How do different tilings, etc. affect savings?

# Evaluation

## Evaluation Questions

- ▶ How much duplicate work does this let us avoid?
- ▶ How do different tilings, etc. affect savings?

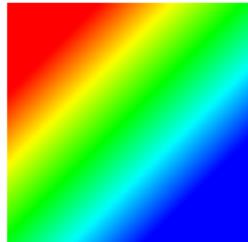
## Approach

- ▶ Used the transformation from the example
  - ▶ (stage 1 invert, stage 2 flip\_and\_blur)
- ▶ Measured # of calls to invert and execution time
- ▶ Tested with several different function schedules

# Experimental Setup

Ran experiments using small, medium, and large images

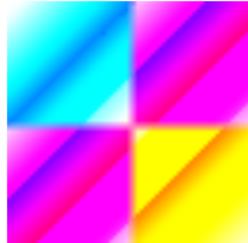
$64 \times 64$



$400 \times 261$

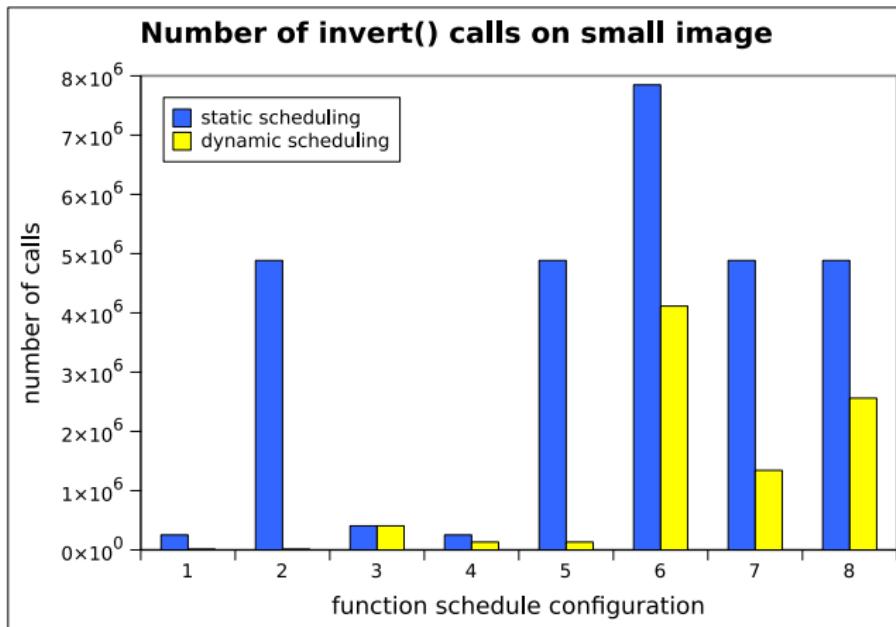


$963 \times 642$



# Result: Dynamic Scheduling Saves Computation

For small image, measured number of `invert()` calls during execution  
▶ (think: number of “cache misses”)



# Conclusion

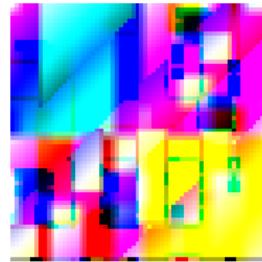
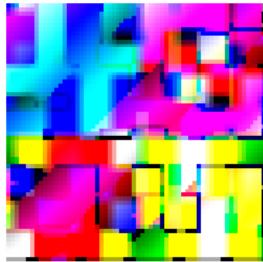
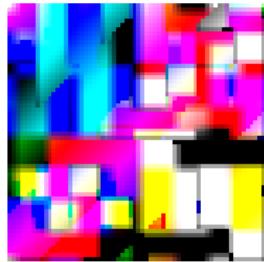
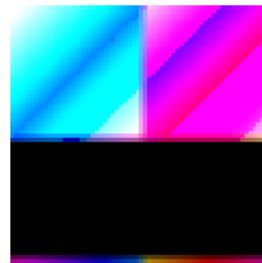
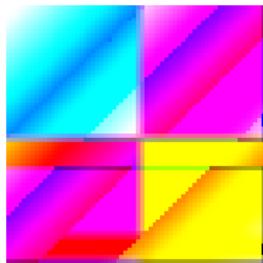
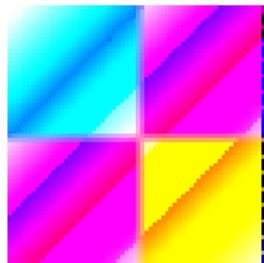
# Summary

Halide separates **algorithm logic** from **function scheduling**.

Schedules are influenced by **data dependencies** between algorithm stages.

**Dynamically caching results** can succeed where static reasoning fails.

# Questions?



(“bloopers” produced during debugging)

## Bonus Slides

# Open Questions

Can we cache results at **tile granularity** instead of **pixel granularity**?

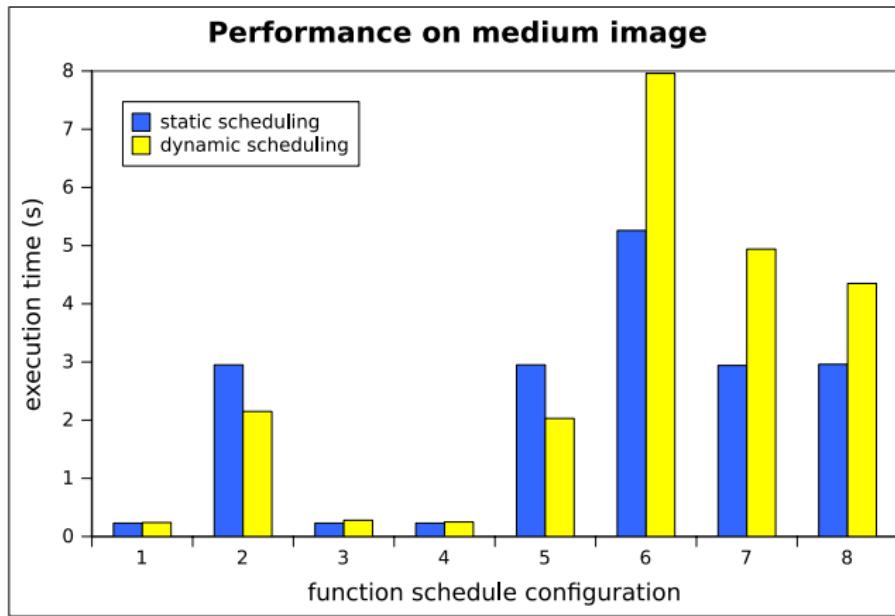
- ▶ (Not straightforwardly. Sizes and shapes of tiles change across iterations of outer loops. Need more runtime support.)

With **parallel loops**, where do we need memory barriers?

- ▶ (And how does multithreading affect work deduplication?)

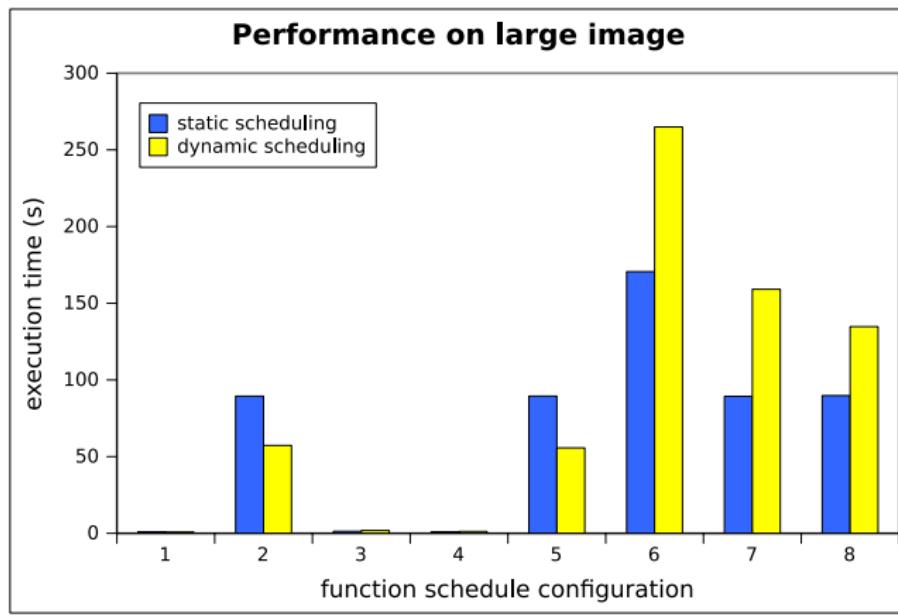
# Result: Dynamic Scheduling Saves Time (sometimes...)

For medium and large images, measured execution time



# Result: Dynamic Scheduling Saves Time (sometimes...)

For medium and large images, measured execution time



# Which function schedules did I use?

```
flipped.tile(x, y, xi, yi, 2, 2);
```

# Which function schedules did I use?

```
flipped.tile(x, y, xi, yi, 2, 2);
```

- ▶ `inverted.store_at(c)`
  - ▶ `.compute_at(y);` (not included)
  - ▶ `.compute_at(x);` (not included)
  - ▶ `.compute_at(yi); (#1)`
  - ▶ `.compute_at(xi); (#2)`
- ▶ `inverted.store_at(y)`
  - ▶ `.compute_at(x); (#3)`
  - ▶ `.compute_at(yi); (#4)`
  - ▶ `.compute_at(xi); (#5)`
- ▶ `inverted.store_at(x)`
  - ▶ `.compute_at(yi); (#6)`
  - ▶ `.compute_at(xi); (#7)`
- ▶ `inverted.store_at(yi)`
  - ▶ `.compute_at(xi); (#8)`