CSE211: Compiler Design

Nov. 19, 2021

- **Topic**: Halide continued

- **Discussion questions**:
  - Anyone have any extra thoughts on DSLs?

Announcements

- Homework 2 and midterm are graded
  - Let me know if there are issues or if you have questions (Office hours on Thursday)
  - Last day to raise concerns is Friday
- Homework 4 is out
  - Due on last day of class (Dec. 3)
- Guest lecture for Nov. 22
  - Aviral Goel will talk about laziness in R

Announcements

- **Paper assignment**:
  - everyone is registered, thanks!
  - also due on Dec. 3

- **Project**:
  - we have 9 signed up
  - everyone wants to present on the Dec. 3
  - thought: cancel class on the the 1st and have a longer class on the 3rd?

CSE211: Compiler Design

Nov. 19, 2021

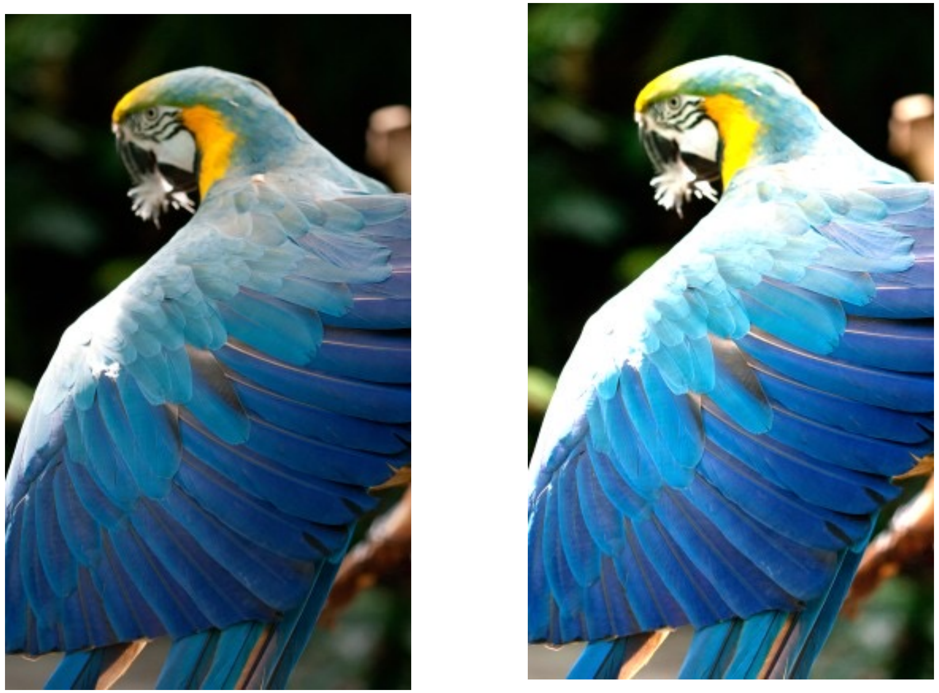
- **Topic**: Halide continued

- **Discussion questions**:
  - Anyone have any extra thoughts on DSLs?

Halide

- A discussion and overview of **Halide:**
  - Huge influence on modern DSL design
  - Great tooling
  - Great paper

- Originally: A DSL for image pipelining:



Brighten example

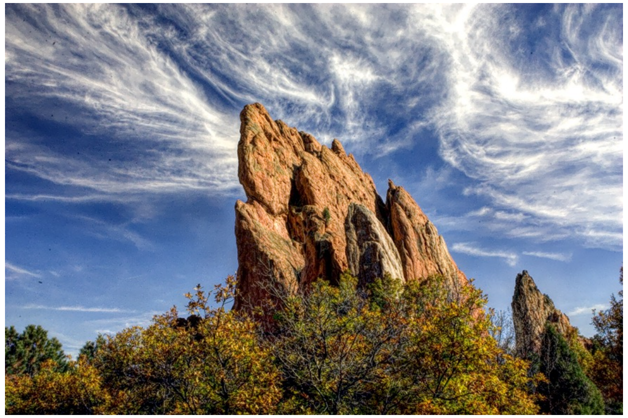
from: https://halide-lang.org/tutorials/tutorial_lesson_02_input_image.html

Motivation:

pretty straight
forward computation
for brightening

(1 pass over all pixels)

This computation is known as the "Local Laplacian Filter". Requires visiting all pixels 99 times



We want to be able to do this
fast and efficiently!

Main results in from Halide show
a 1.7x speedup with 1/5 the LoC
over hand optimized versions at Adobe

from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/

Decoupling computation from optimization

- We love Halide not only because it can make pretty pictures very fast
- We love it because it changed the level of abstraction for thinking about computation and optimization
- (Halide has been applied in many other domains now, turns out everything is just linear algebra)

Example

- in C++

```
for (int x = 0; x < x_size; x++) {
 for (int y = 0; y < y_size; y++) {
    a[x,y] = b[x,y] + c[x,y];
 }
}
```

Which one would you write?

```
for (int y = 0; y < y_size; y++) {
 for (int x = 0; x < x_size; x++) {
    a[x,y] = b[x,y] + c[x,y];
 }
}
```

Optimizations are a black box

- What are the options?
  - -O0, -O1, -O2, -O3
  - Is that all of them?
  - What do they actually do?

https://stackoverflow.com/questions/15548023/clang-optimization-levels

Optimizations are a black box

- What are the options?
  - -O0, -O1, -O2, -O3
  - Is that all of them?
  - What do they actually do?

- ***Answer***: they do their best for a wide range of programs. The common case is that you should not have to think too hard about them.

- ***In practice***, to write high-performing code, you are juggling computation and optimization in your mind!

Halides approach

- Decouple
  - what to compute (the program)
  - with how to compute (the optimizations, also called the schedule)

Halides approach

- Decouple
  - what to compute (the program)
  - with how to compute (the optimizations, also called the schedule)

```
for (int y = 0; y < y_size; y++) {
  for (int x = 0; x < x_size; x++) {
        a[x,y] = b[x,y] + c[x,y];
   }
}
```

C++:

program

```
add(x,y) = b(x,y) + c(x,y)
```

schedule

```
add.order(x,y)
```

Halide (high-level)

Halides approach

- Decouple
  - what to compute (the program)
  - with how to compute (the optimizations, also called the schedule)

program

```
add(x,y) = b(x,y) + c(x,y)
```

Pros and Cons?

schedule

```
add.order(x,y)
```

Halide (high-level)

Halide optimizations

- Now all of a sudden, the programmer has to worry about how to optimize the program. Previously the compiler compiler made those decisions and we just “helped”.

- What can we do here?

Halide optimizations

- Auto-tuning
    - automatically select a schedule
    - compile and run/time the program.
    - Keep track of the schedule that performs the best

- Why don’t all compilers do this?

Halide optimizations

- Auto-tuning
  - automatically select a schedule
  - compile and run/time the program.
  - Keep track of the schedule that performs the best
- Why don’t all compilers do this?
- Image processing is especially well-suited for this:
  - Images in different contexts might have similar sizes (e.g. per phone, on twitter, on facebook)

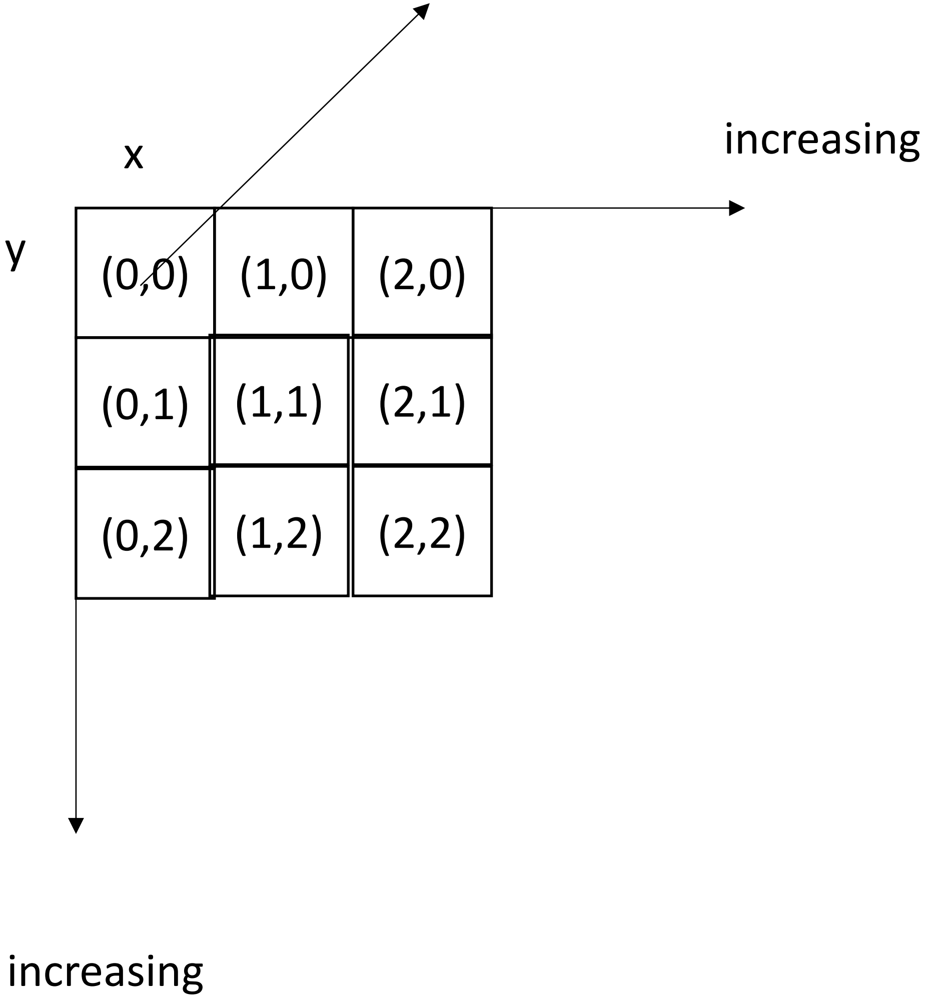
Halide programs

- Halide programs:
  - built into C++, contained within a header

```
#include "Halide.h"


Halide::Func gradient;        // a pure function declaration

Halide::Var x, y;             // variables to use in the definition of the function (types?)

gradient(x, y) = x + y;       // the function takes two variables (coordinates in the image) and adds them
```

from: https://halide-lang.org/tutorials/tutorial_lesson_01_basics.html

```
gradient(x, y) = x + y;
```



x

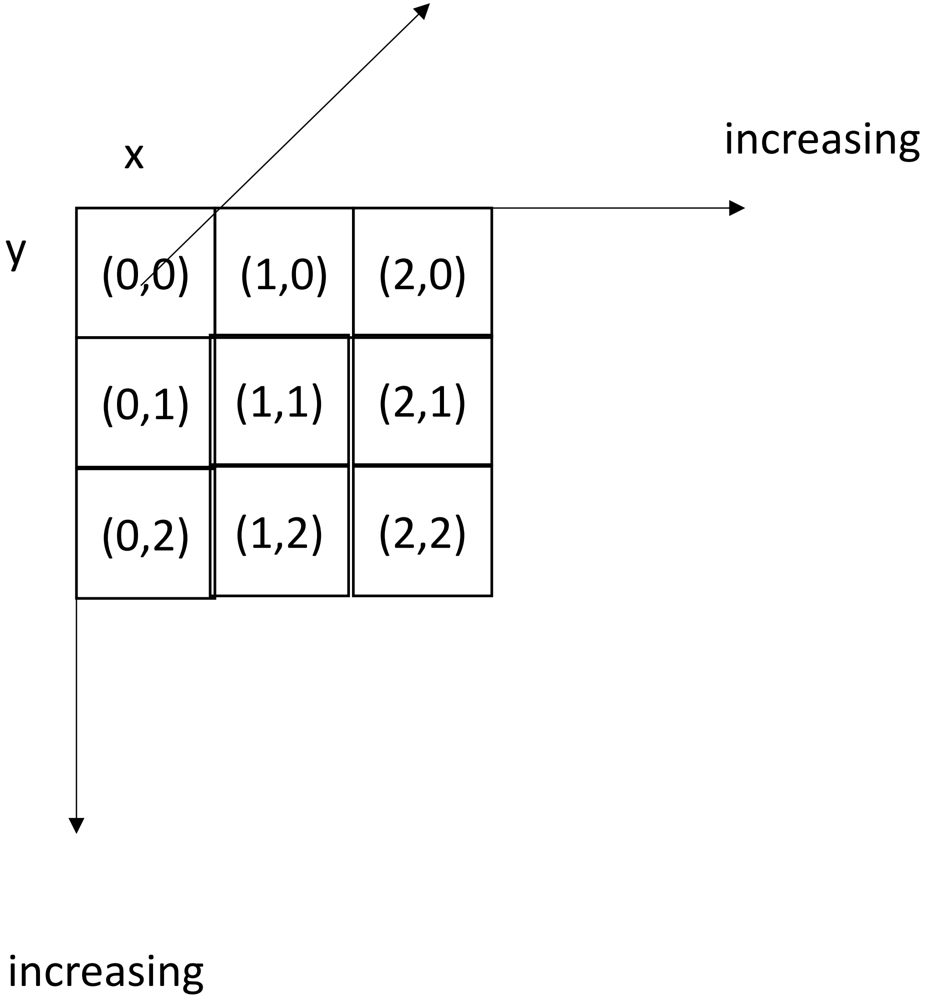
increasing

y

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,2)	(1,2)	(2,2)

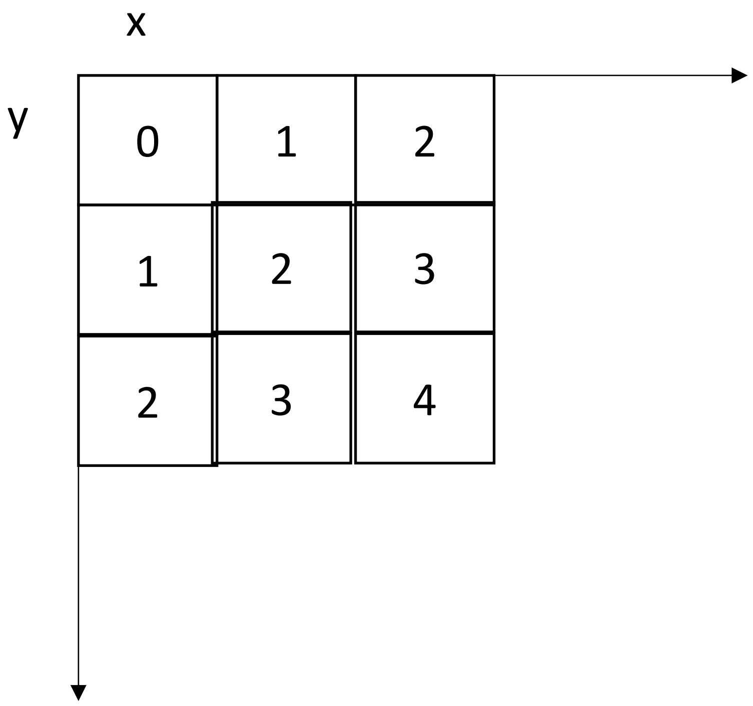
increasing

```
gradient(x, y) = x + y;
```



(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,2)	(1,2)	(2,2)

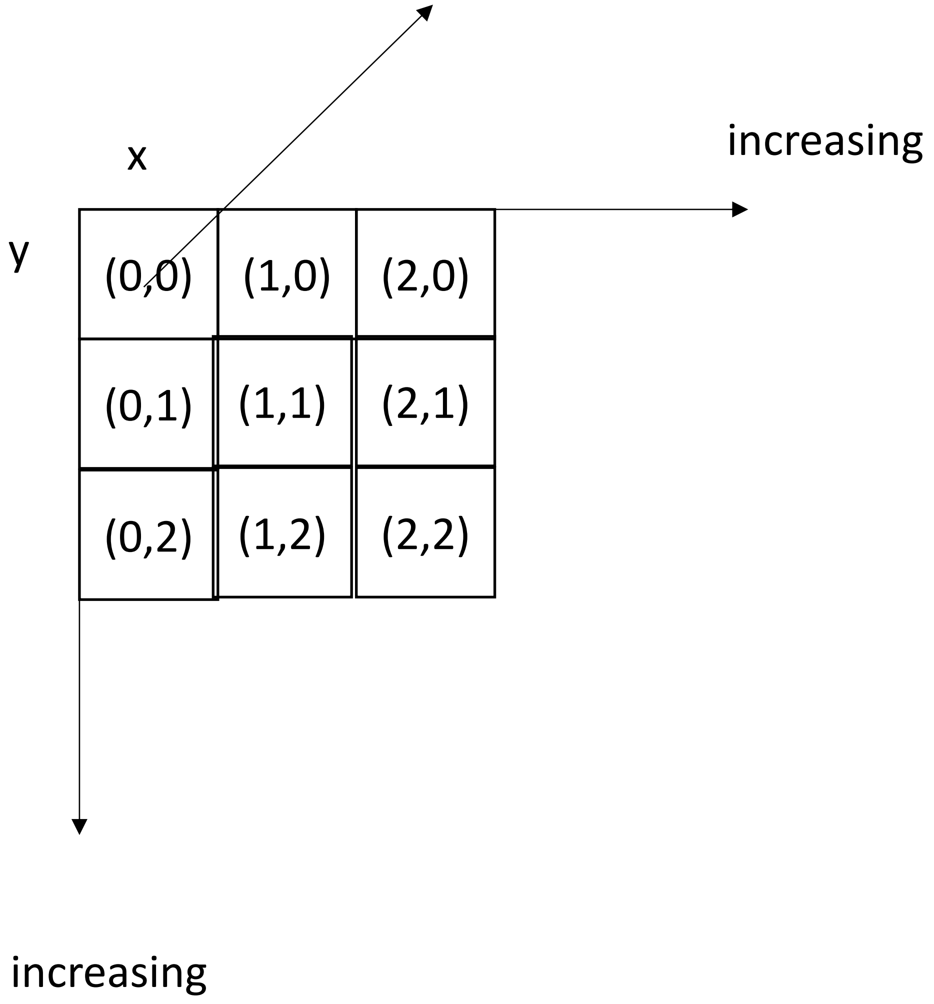
after applying the gradient function



0	1	2
1	2	3
2	3	4

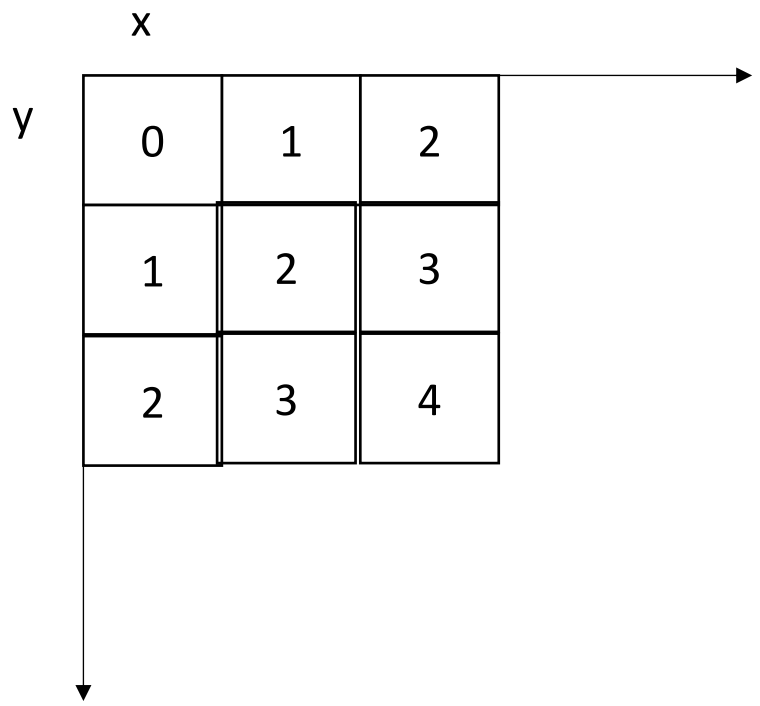
what are some properties of this computation?

```
gradient(x, y) = x + y;
```



(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,2)	(1,2)	(2,2)

after applying the gradient function



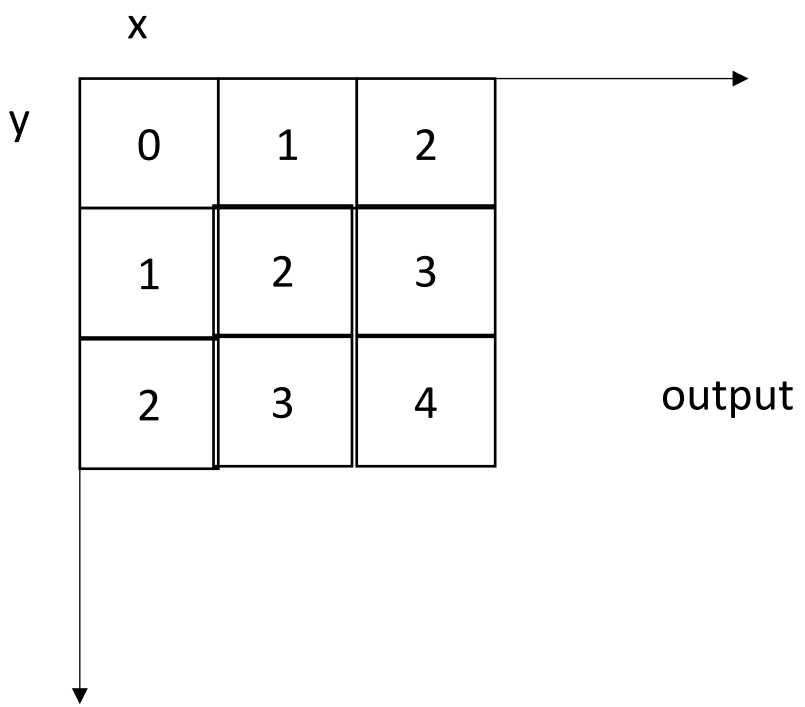
0	1	2
1	2	3
2	3	4

what are some properties of this computation?
Data races?
Loop indices and increments?
The order to compute each pixel?

Executing the function

```
Halide::Buffer<int32_t> output = gradient.realize({3, 3});
```

Not compiled until this point
Needs values for x and y



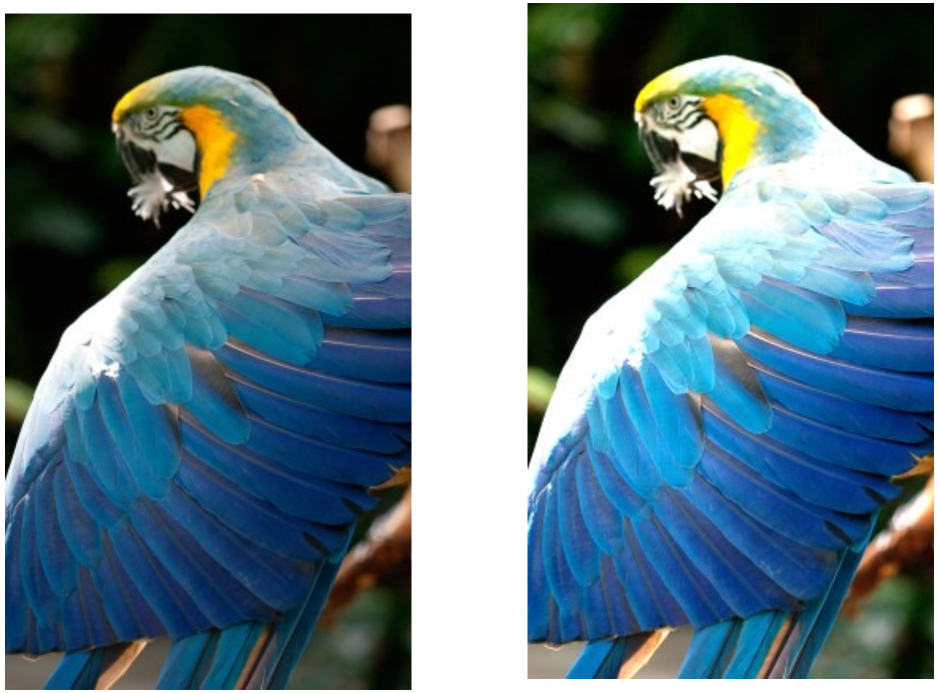
x

y

0	1	2
1	2	3
2	3	4

output

Example: brightening



Brighten example

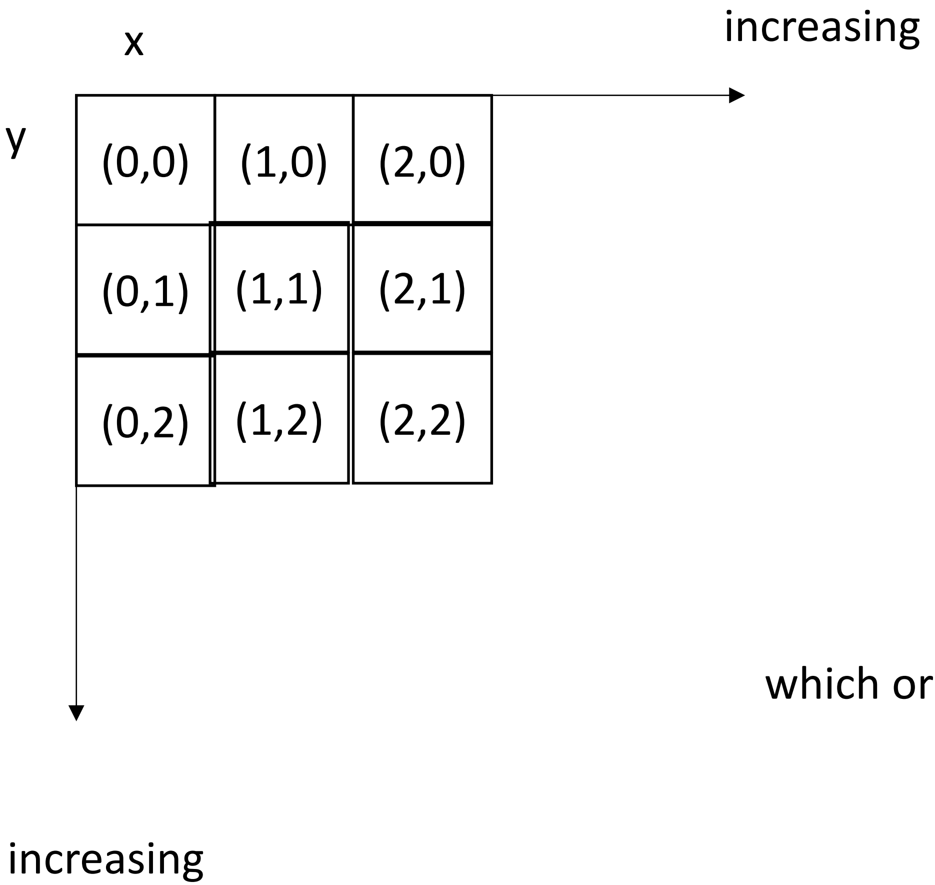
```
Halide::Buffer<uint8_t> input = load_image("parrot.png");

Halide::Func brighter;

Halide::Expr value = input(x, y, c);

value = Halide::cast<float>(value);

value = value * 1.5f;

value = Halide::min(value, 255.0f);

value = Halide::cast<uint8_t>(value);

brighter(x, y, c) = value;

Halide::Buffer<uint8_t> output =
               brighter.realize({input.width(), input.height(), input.channels()});
```

```
Halide::Buffer<uint8_t> input = load_image("parrot.png");

Halide::Func brighter;

Halide::Expr value = input(x, y, c);

value = Halide::cast<float>(value);

value = value * 1.5f;

value = Halide::min(value, 255.0f);

value = Halide::cast<uint8_t>(value);

brighter(x, y, c) = value;

Halide::Buffer<uint8_t> output =
               brighter.realize({input.width(), input.height(), input.channels()});
```

```
brighter(x, y, c) = Halide::cast<uint8_t>(min(input(x, y, c) * 1.5f, 255));
```

Schedules

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({3, 3});
```



x

increasing

y

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,2)	(1,2)	(2,2)

increasing

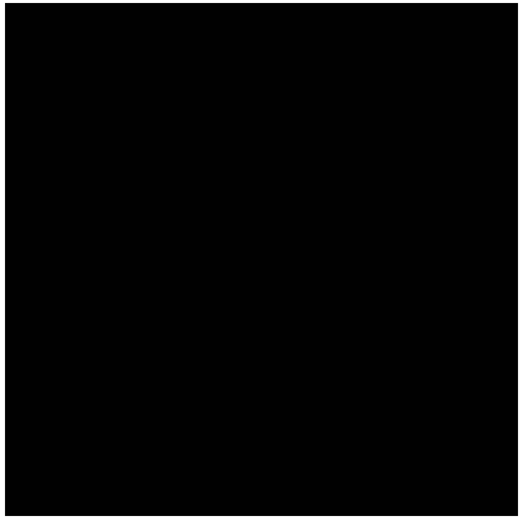
which order to traverse these elements?

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

```
for (int y = 0; y < 4; y++) {
    for (int x = 0; x < 4; x++) {
        output[y,x] = x + y;
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```



```
for (int y = 0; y < 4; y++) {
    for (int x = 0; x < 4; x++) {
        output[y,x] = x + y;
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

Schedule

```
gradient.reorder(y, x);
```

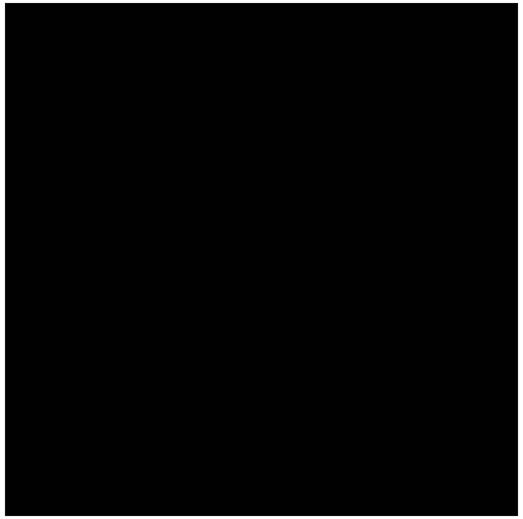
```
for (int x = 0; x < 4; x++) {
    for (int y = 0; y < 4; y++) {
        output[y,x] = x + y;
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

Schedule

```
gradient.reorder(y, x);
```



```
for (int x = 0; x < 4; x++) {
    for (int y = 0; y < 4; y++) {
        output[y,x] = x + y;
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

Schedule

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
```

```
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {
        for (int x_inner = 0; x_inner < 2; x_inner++) {
            x = x_inner + x_outer * 2;
            output[y,x] = x + y;
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

Schedule

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
```

```
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {
        for (int x_inner = 0; x_inner < 2; x_inner++) {
            x = x_outer*2 + x_inner;
            output[y,x] = x + y;
        }
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

Schedule

```
Var xy;
gradient.fuse(x, y, xy);
```

```
for (int xy = 0; xy < 4*4; xy++) {
    x = xy/4;
    y = xy%4
    output[y,x] = x + y;

}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

Schedule

```
gradient.fuse(x, y);
```

```
for (int xy = 0; xy < 4*4; xy++) {
    y = xy / 4;
    x = xy % 4;
    output[y,x] = x + y;
}
```

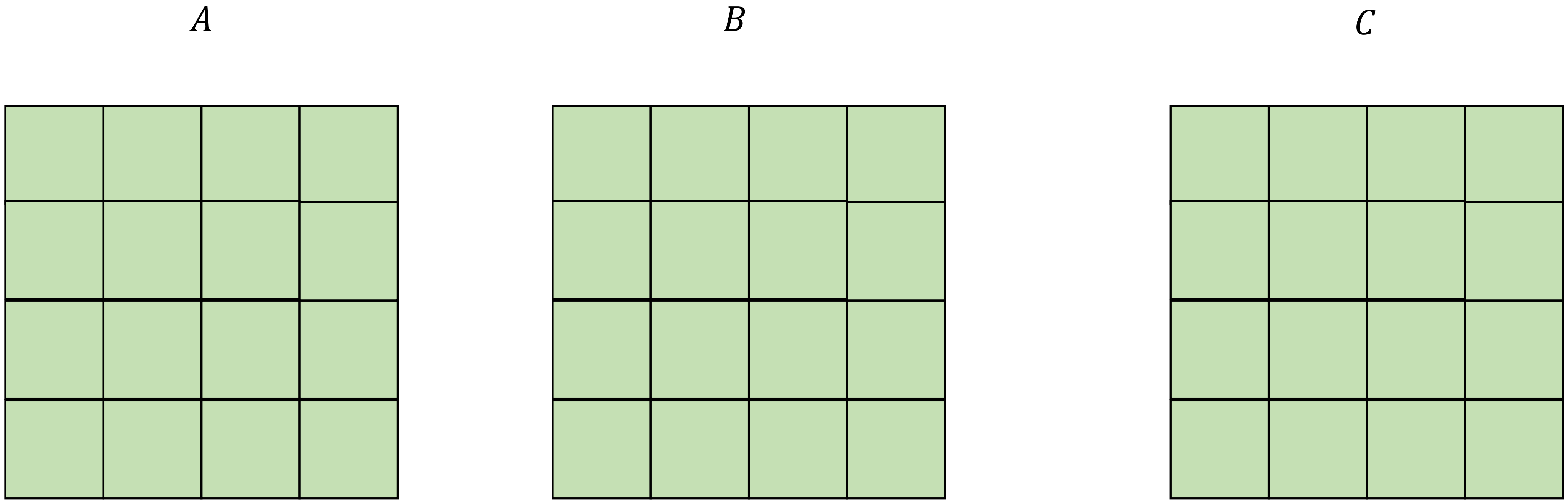
from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Tiling

Adding loop nestings

- In some cases, there might not be a good nesting order for all accesses:

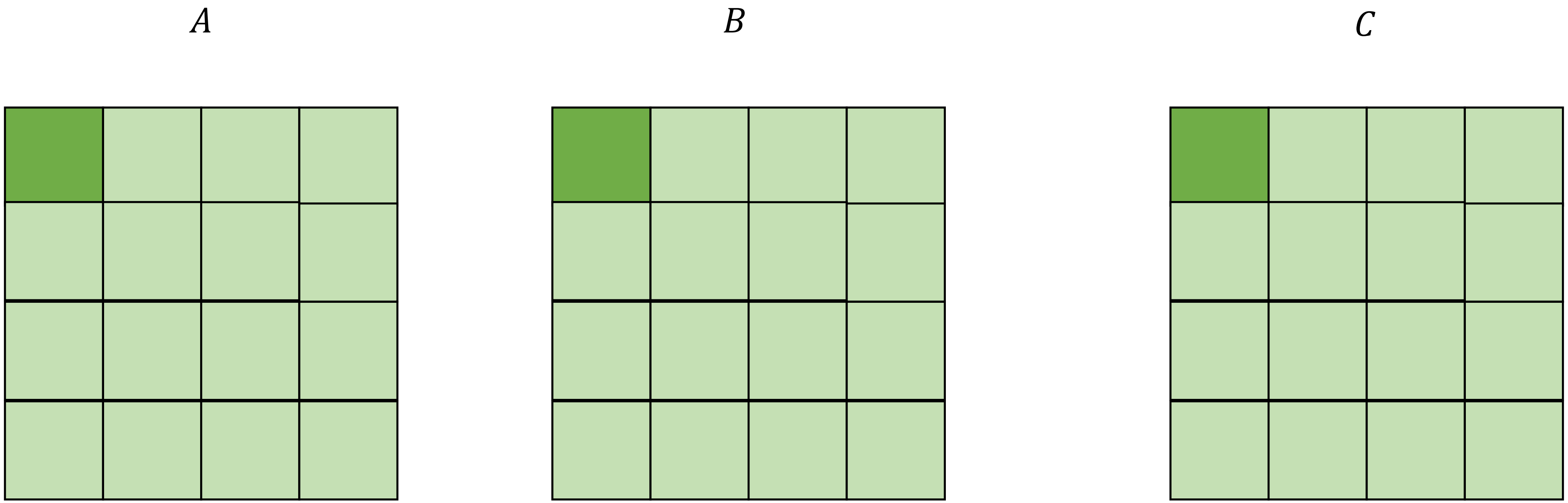
$$A = B + C^T$$



Adding loop nestings

- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

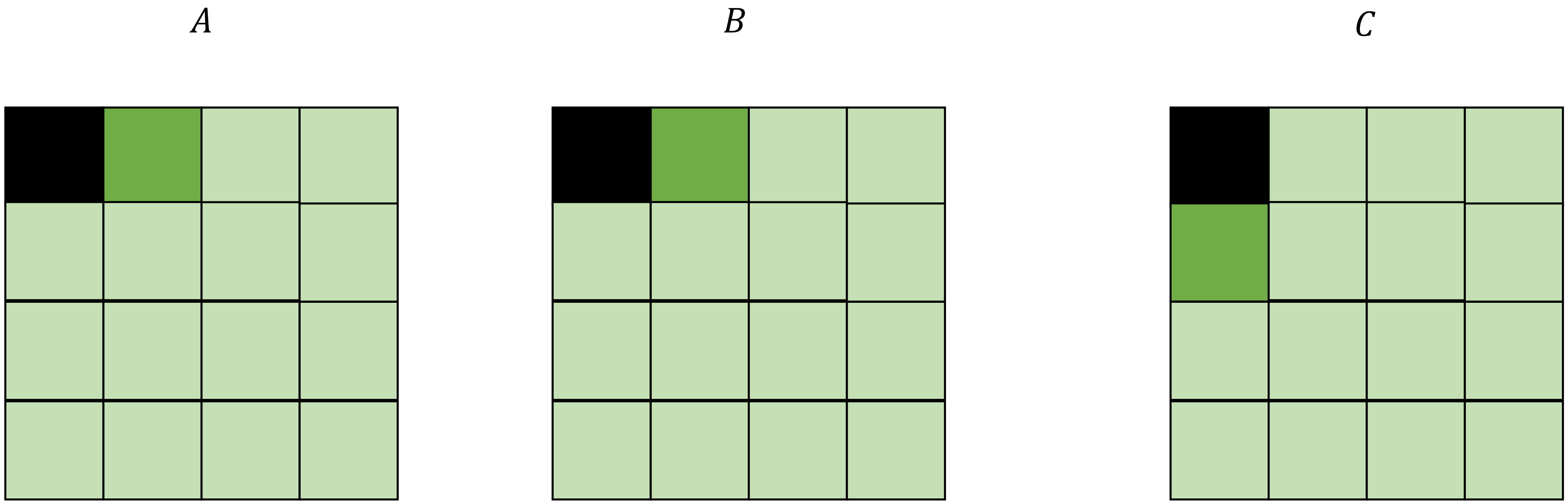


cold miss for all of them

Adding loop nestings

- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

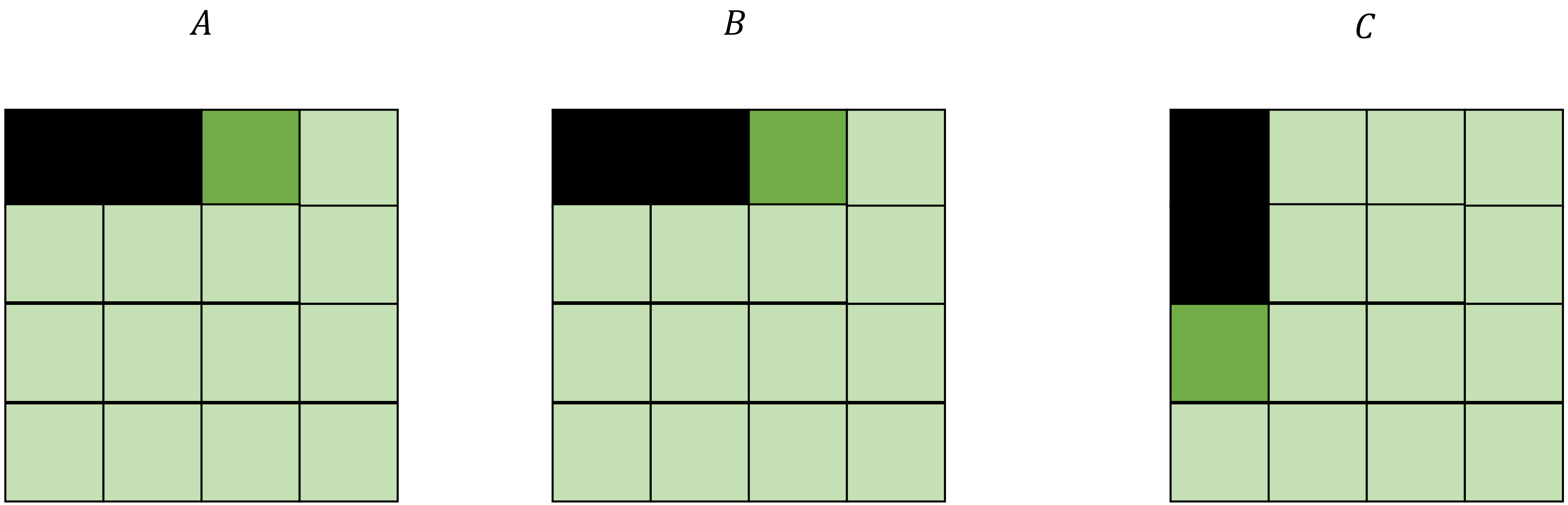


Hit on A and B. Miss on C

Adding loop nestings

- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$



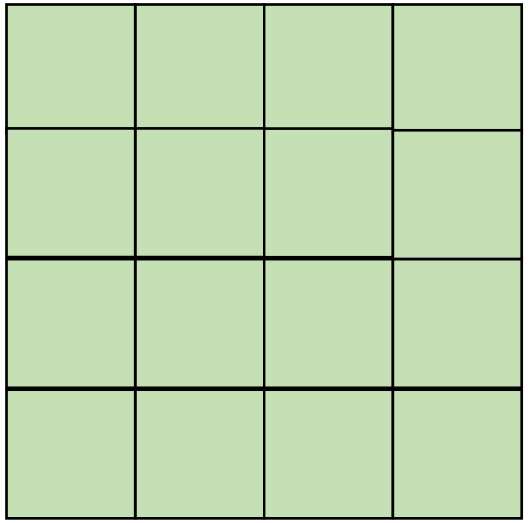
Hit on A and B. Miss on C

Adding loop nestings

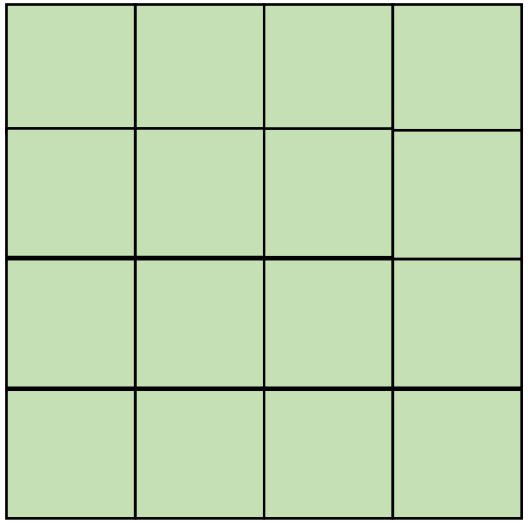
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

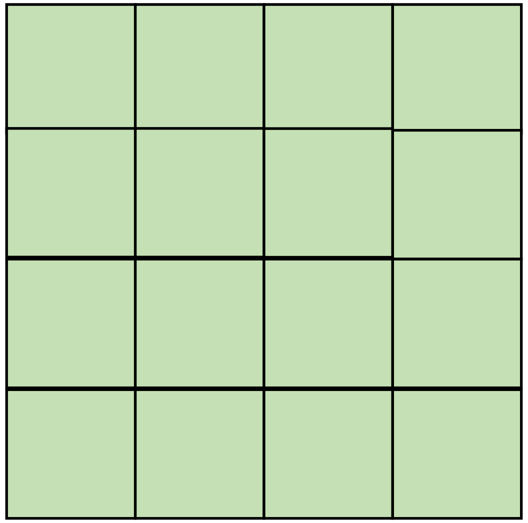
A



B



C

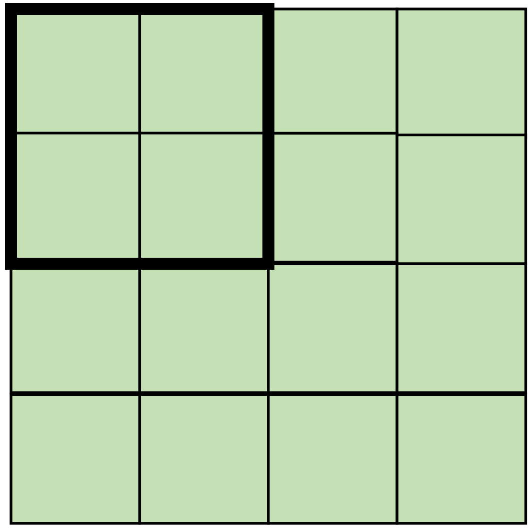


Adding loop nestings

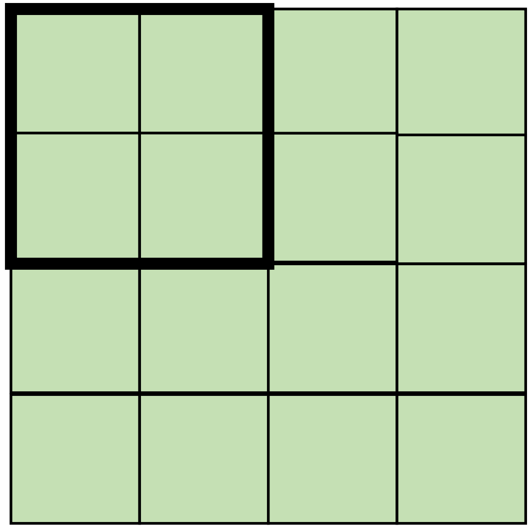
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

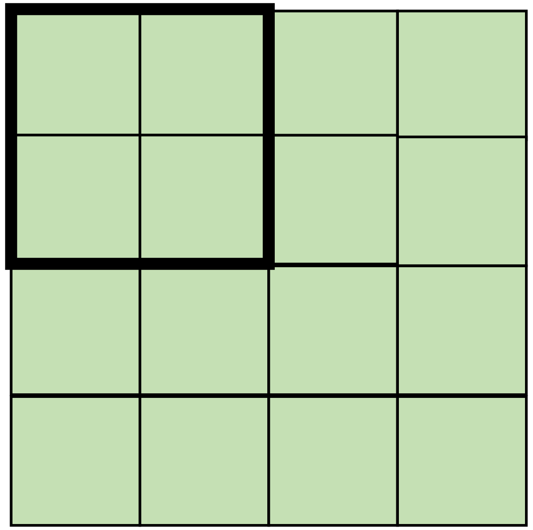
A



B



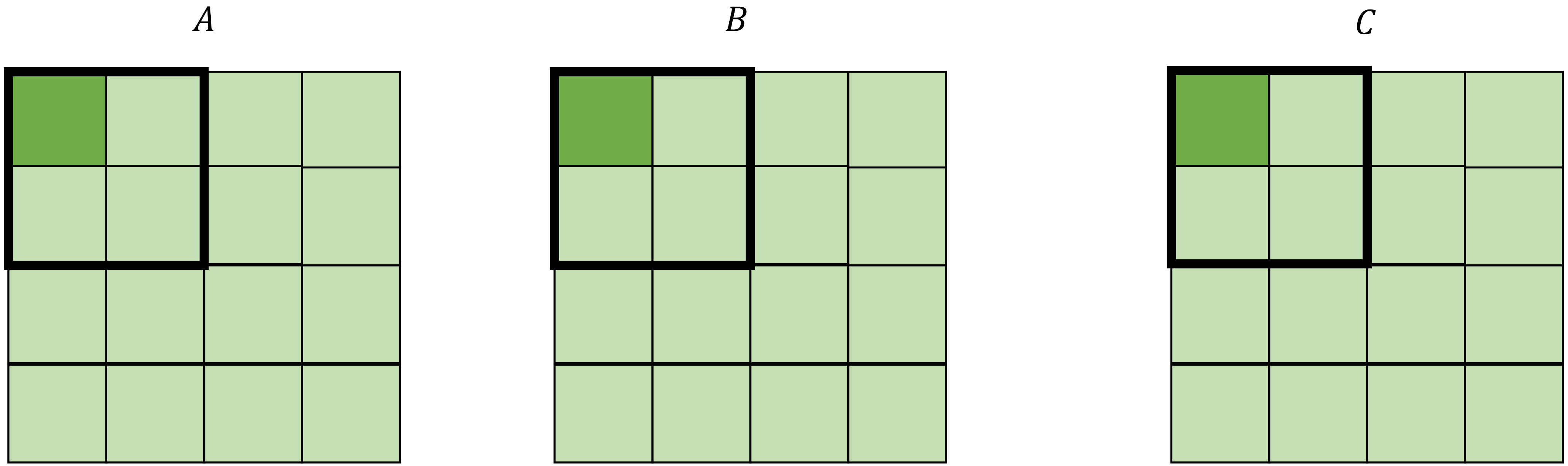
C



Adding loop nestings

- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

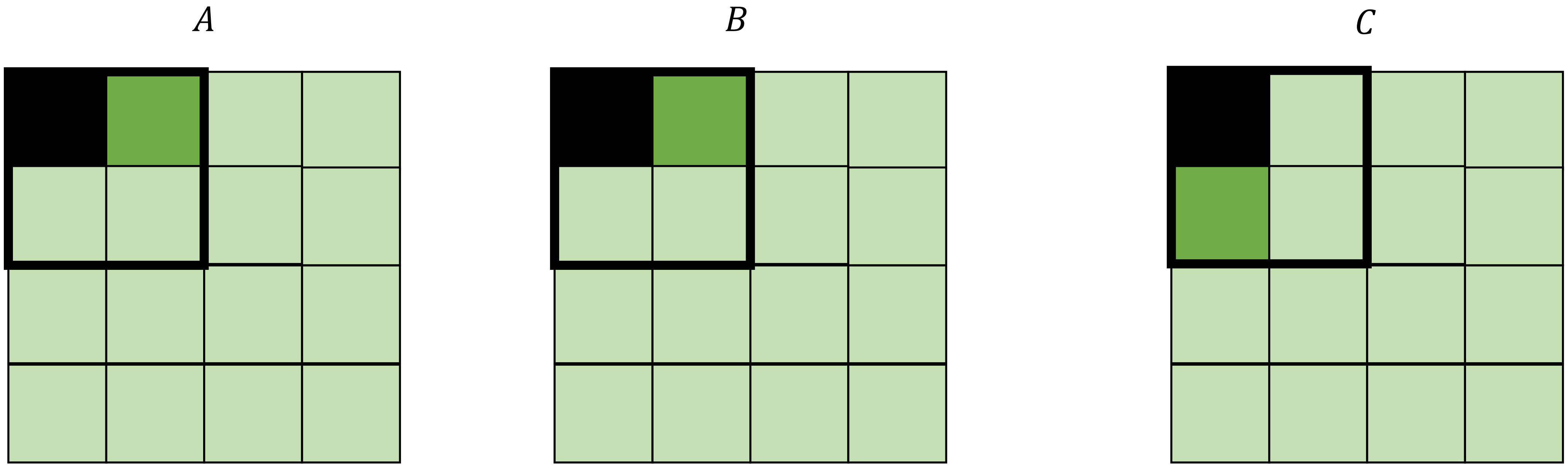


cold miss for all of them

Adding loop nestings

- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

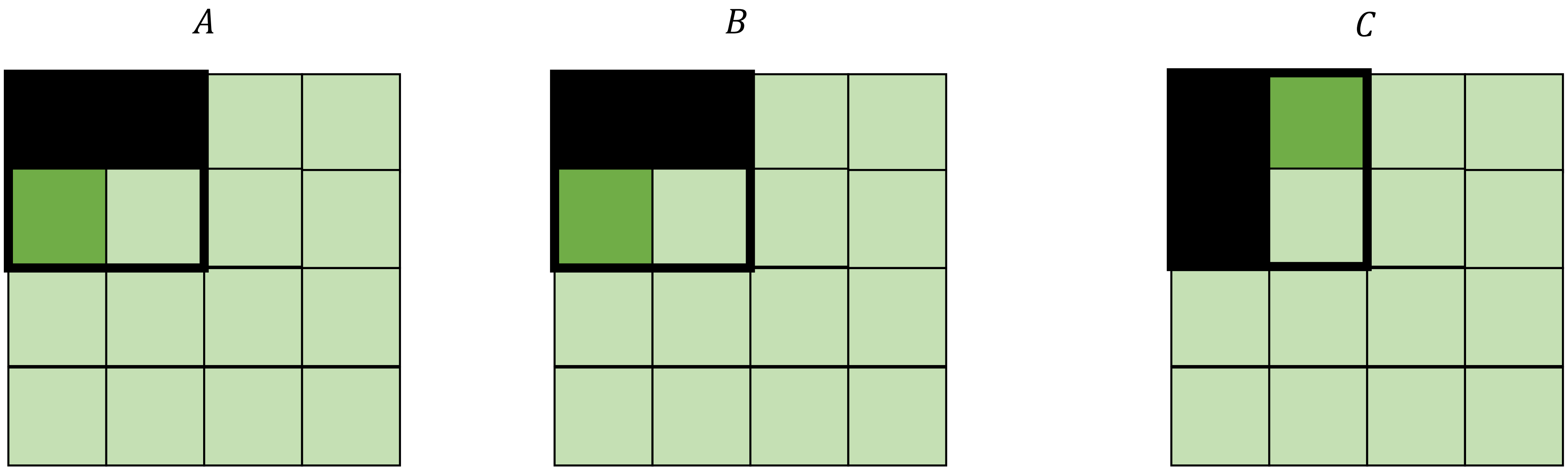


Miss on C

Adding loop nestings

- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

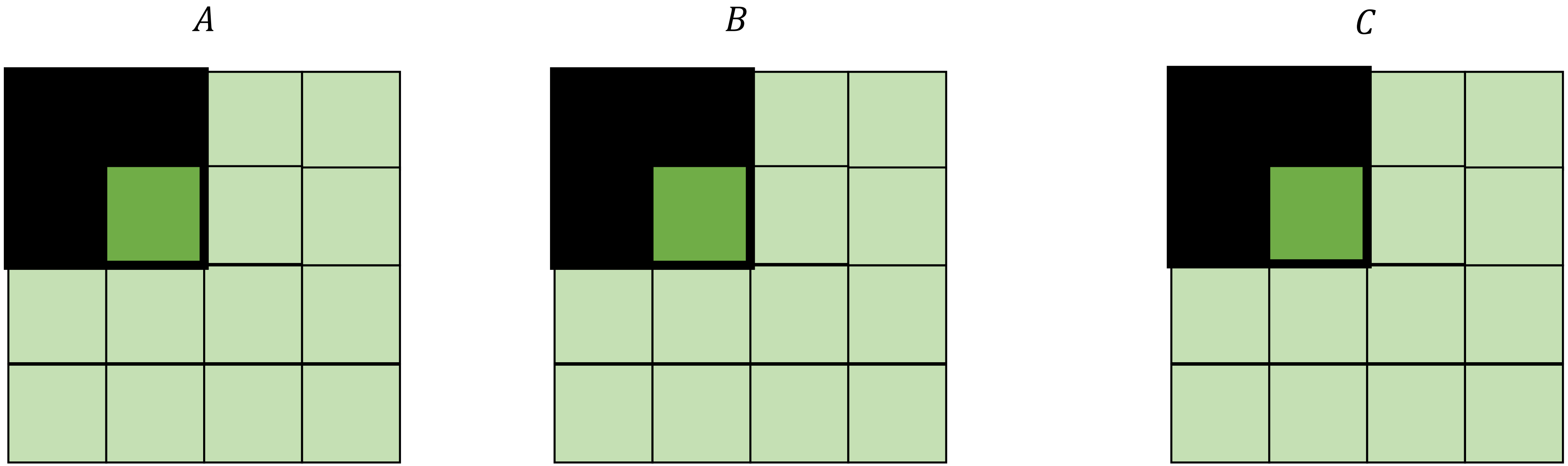


Miss on A,B, hit on C

Adding loop nestings

- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$



Hit on all!

```
for (int x = 0; x < SIZE; x++) {
  for (int y = 0; y < SIZE; y++) {
   a[x*SIZE + y] = b[x*SIZE + y] + c[y*SIZE + x];
  }
 }
```

transforms into:

```
for (int xx = 0; xx < SIZE; xx += B) {
 for (int yy = 0; yy < SIZE; yy += B) {
  for (int x = xx; x < xx+B; x++) {
   for (int y = yy; y < yy+B; y++) {
    a[x*SIZE + y] = b[x*SIZE + y] + c[y*SIZE + x];
   }
  }
 }
}
```

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({16, 16});
```

Schedule

```
gradient.split(x, x_inner, x_outer, 4)
gradient.split(y, y_inner, y_outer, 4)
gradient.reorder(x_outer, y_outer, x_inner,
```

```
for (int y = 0; y < 16; y++) {
    for (int x = 0; x < 16; x++) {
        output[y,x] = x + y;
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({16, 16});
```

Schedule

```
Var x_outer, x_inner, y_outer, y_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.split(y, y_outer, y_inner, 4);
gradient.reorder(x_inner, y_inner, x_outer, y_outer);
```

```
for (int y = 0; y < 4; y++) {
    for (int x = 0; x < 4; x++) {
        output[y,x] = x + y;
    }
}
```

```
gradient.tile(x, y,
              x_outer, y_outer,
              x_inner, y_inner, 4, 4);
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

how would we make a program
that would benefit from tiling?

```
Halide::Buffer<uint8_t> a = // big matrx
Halide::Buffer<uint8_t> b = // big matrx

Halide::Func our_function;
Halide::Var x, y;
out_function(x,y) = a(x,y) + b(y,x)
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

how would we make a program
that would benefit from tiling?

```
Halide::Buffer<uint8_t> a = // big matrx
Halide::Buffer<uint8_t> b = // big matrx
```

```
Halide::Func our_function;
Halide::Var x, y;
our_function(x,y) = a(x,y) + b(y,x)
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({8, 4});
```

Schedule

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.vectorize(x_inner);
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({8, 4});
```

Schedule

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.vectorize(x_inner);
```

```
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {

        int x_vec[] = {x_outer * 4 + 0,
                       x_outer * 4 + 1,
                       x_outer * 4 + 2,
                       x_outer * 4 + 3};

        int val[] =   {x_vec[0] + y,
                       x_vec[1] + y,
                       x_vec[2] + y,
                       x_vec[3] + y};
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({8, 4});
```

Schedule

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.vectorize(x_inner);
```

```
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {

        int x_vec[] = {x_outer * 4 + 0,
                       x_outer * 4 + 1,
                       x_outer * 4 + 2,
                       x_outer * 4 + 3};

        int val[] =   {x_vec[0] + y,
                       x_vec[1] + y,
                       x_vec[2] + y,
                       x_vec[3] + y};
    }
}
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({8, 4});
```

Schedule

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
gradient.unroll(x_inner);
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
             gradient.realize({8, 4});
```

Schedule

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
gradient.unroll(x_inner);
```

```
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {
    {
        int x_inner = 0;
        int x = x_outer * 2 + x_inner;
        output(x,y) = x + y;
    }
    {
        int x_inner = 1;
        int x = x_outer * 2 + x_inner;
        output(x,y) = x + y;
    }
    }
}
```

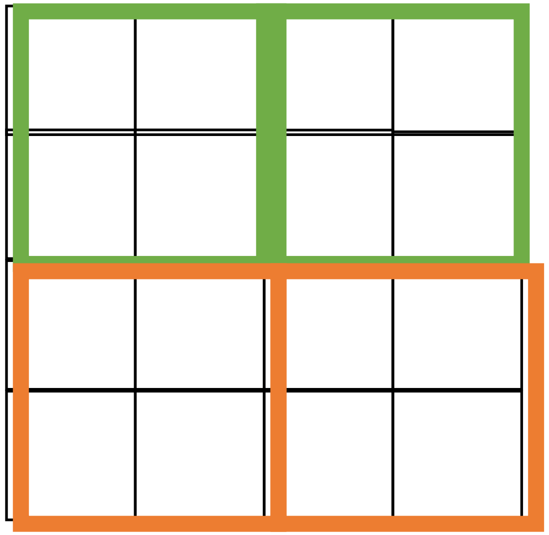
from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({2, 2});
```

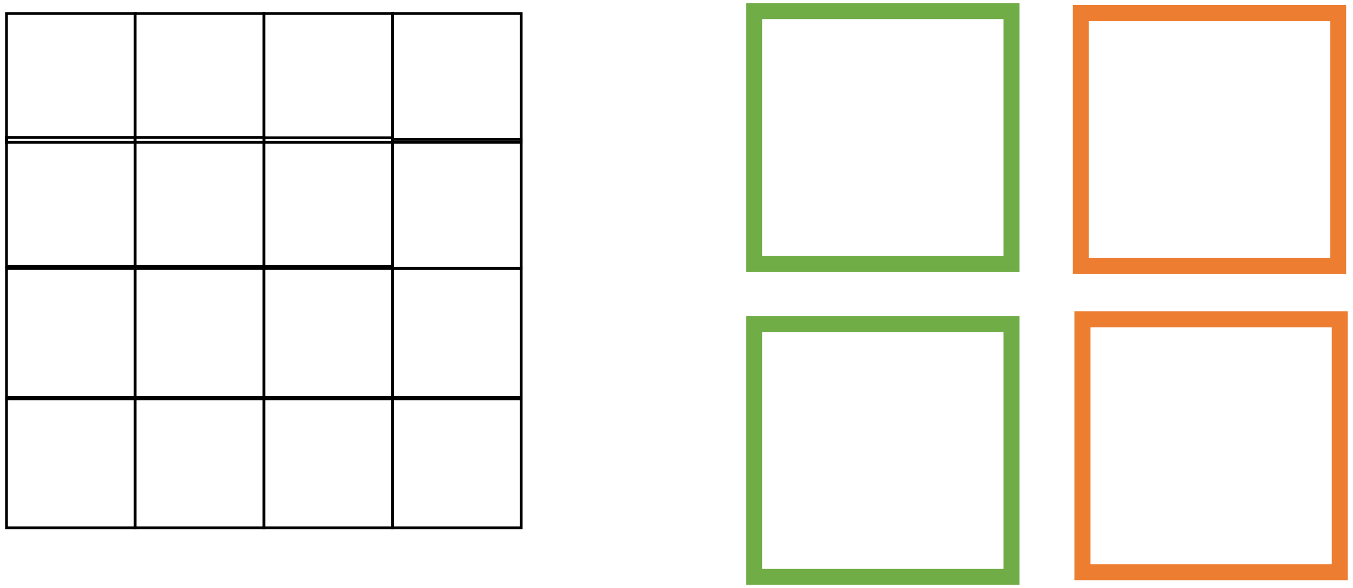
Schedule

```
Var x_outer, y_outer, x_inner,  y_inner, tile_index;

gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2);

gradient.fuse(x_outer, y_outer, tile_index);

gradient.parallel(tile_index);
```

```
for (int y_outer = 0; y_outer < 2; y_outer++) {
  for (int x_outer = 0; x_outer < 2; x_outer++) {
    for (int y_innder = 0; y_inner < 2; y_inner++) {
      for (int x_inner = 0; x_inner < 2; x_inner++) {
        ...
      }
    }
  }
}
```

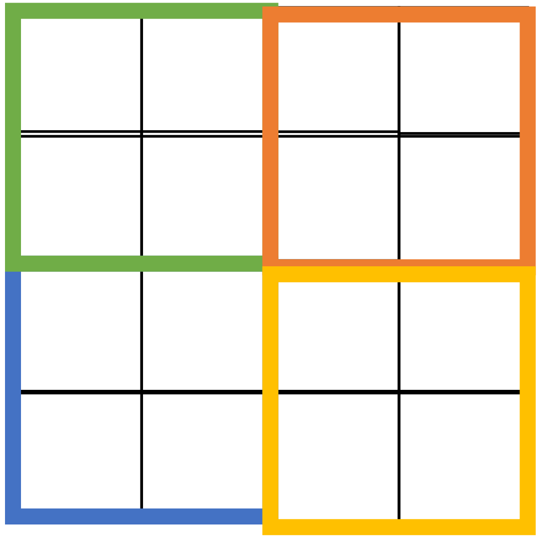


```
for (int y_outer = 0; y_outer < 2; y_outer++) {
  for (int x_outer = 0; x_outer < 2; x_outer++) {
    for (int y_innder = 0; y_inner < 2; y_inner++) {
      for (int x_inner = 0; x_inner < 2; x_inner++) {
        ...
      }
    }
  }
}
```



How to make 4 threads?

```
for (int fused = 0; fused < 4; fused++) {
    y_outer = fused/2;
    x_outer = fused%2;
    for (int y_innder = 0; y_inner < 2; y_inner++) {
      for (int x_inner = 0; x_inner < 2; x_inner++) {
        ...
      }
    }
  }
}
```



How to make 4 threads?

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({2, 2});
```

Schedule

```
Var x_outer, y_outer, x_inner,  y_inner, tile_index;

gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2);

gradient.fuse(x_outer, y_outer, tile_index);

gradient.parallel(tile_index);
```

Finally: a fast schedule that they found:

```
Halide::Func gradient_fast;
Halide::Var x, y;
gradient_fast(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({2, 2});


Var x_outer, y_outer, x_inner, y_inner, tile_index; =
gradient_fast
              .tile(x, y, x_outer, y_outer, x_inner, y_inner, 64, 64)
              .fuse(x_outer, y_outer, tile_index)
              .parallel(tile_index);

Var x_inner_outer, y_inner_outer, x_vectors, y_pairs;
gradient_fast
       .tile(x_inner, y_inner, x_inner_outer, y_inner_outer, x_vectors, y_pairs, 4, 2)
       .vectorize(x_vectors)
       .unroll(y_pairs);
```

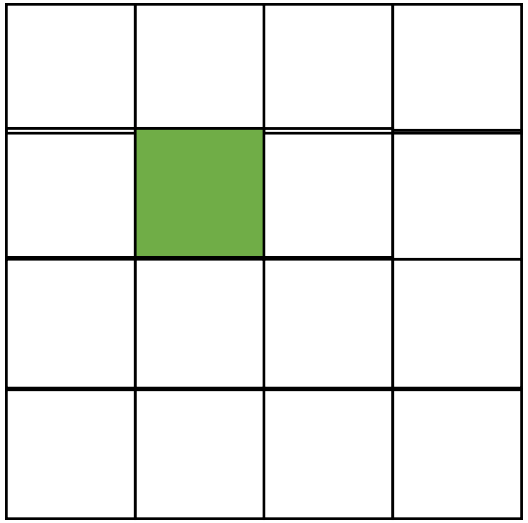
Now for function fusing...

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

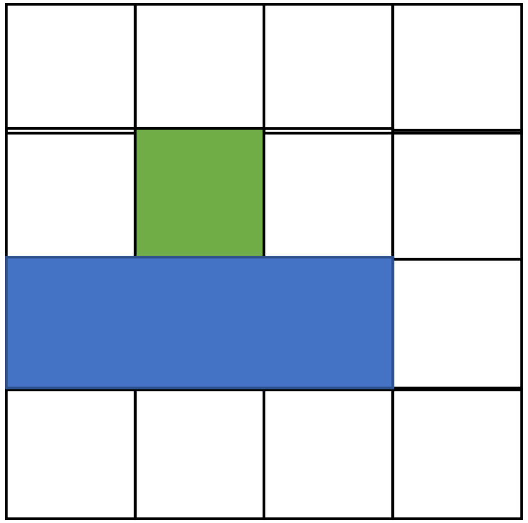
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



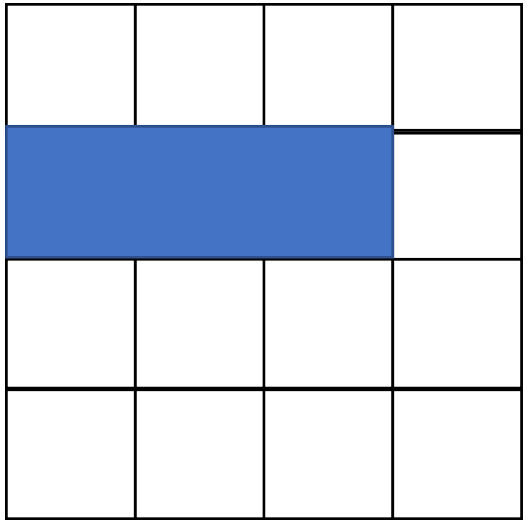
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



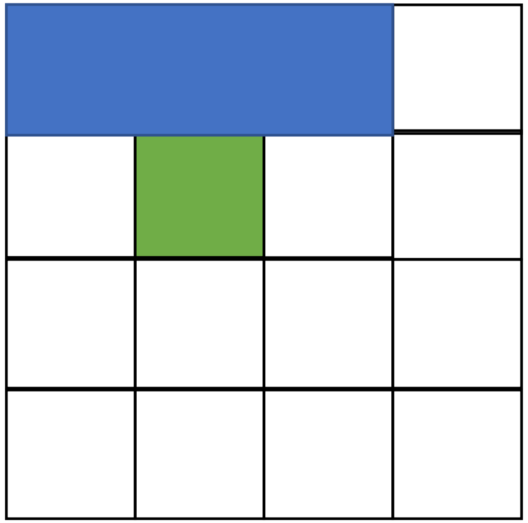
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



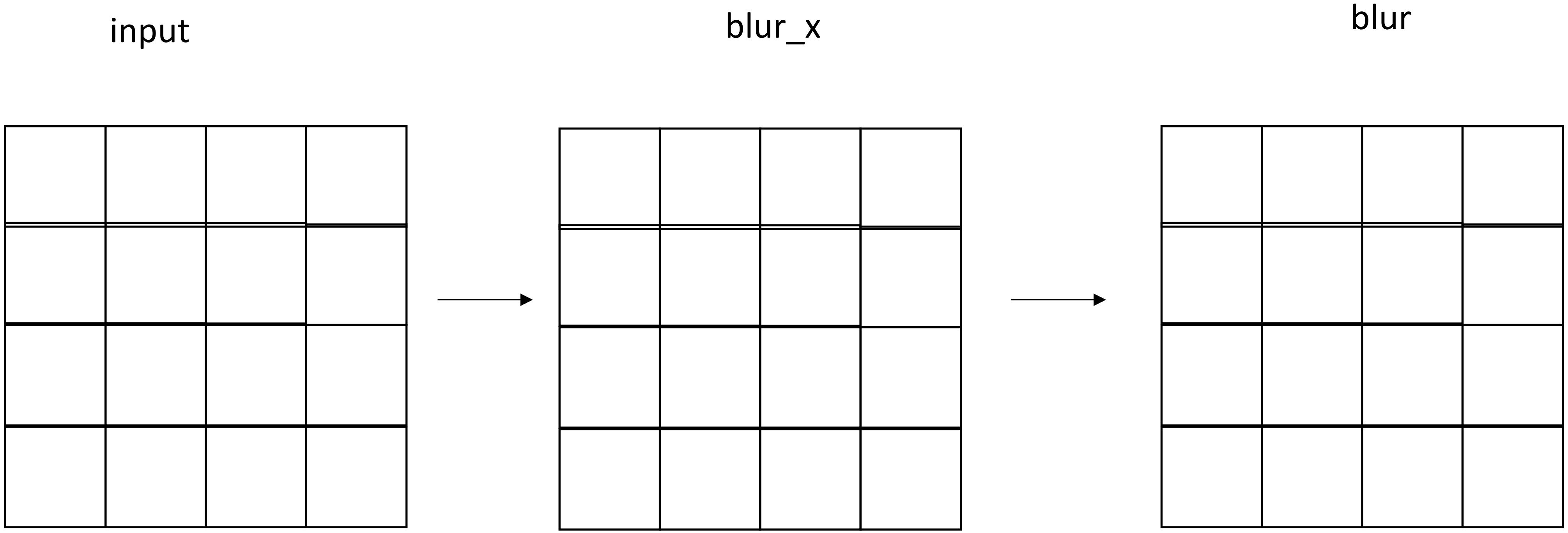
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

how to compute?

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



Example: unnormalized blur

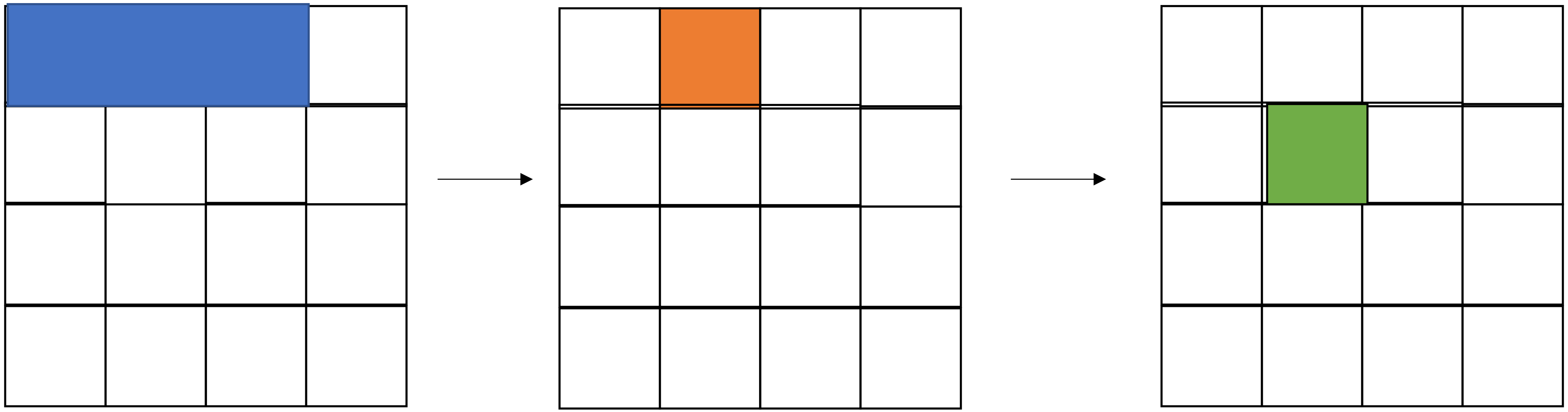
```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

```
alloc blurx[2048][3072]
foreach y in 0..2048:
    foreach x in 0..3072:
      blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]


alloc out[2046][3072]
foreach y in 1..2047:
    foreach x in 0..3072:
      out[y][x] = blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

pros?
cons?

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



input

blur_x

stored to memory!

blur

no locality

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

Other options?

Example: unnormalized blur

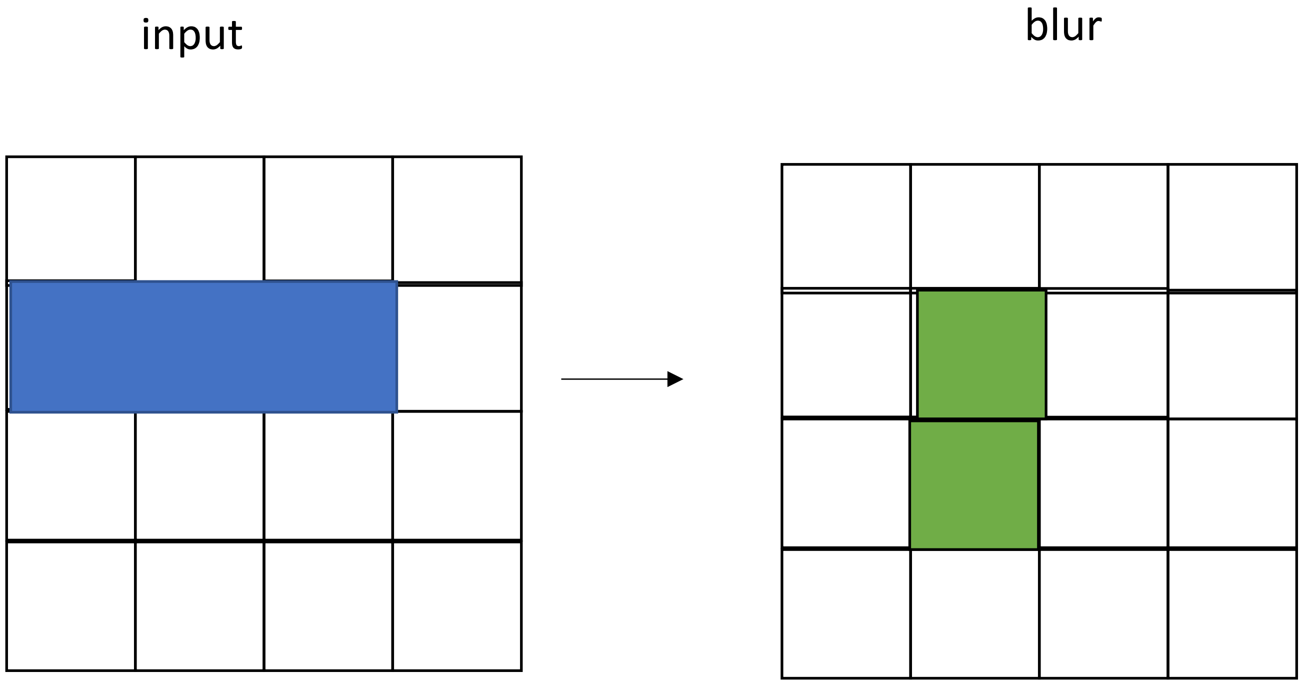
```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

completely inline

```
alloc out[2046][3072]
foreach y in 1..2047:
    foreach x in 0..3072:
      out[y][x] = in[y-1][x] + in[y][x] + in[y+1][x] +
                  in[y-1][x-1] + in[y][x-1] + in[y+1][x-1]
                  in[y-1][x+1] + in[y][x+1] + in[y+1][x+1]
```

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



These two squares will both sum up the same values in blue

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

other ideas?

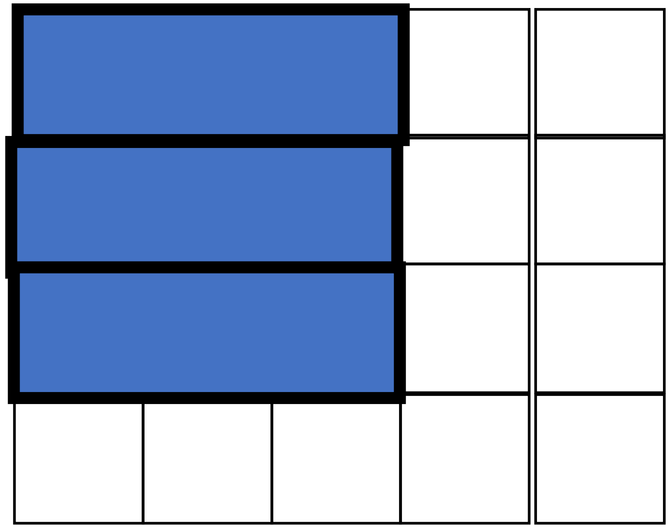
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

first iteration, only compute blur_x

blur

sliding window



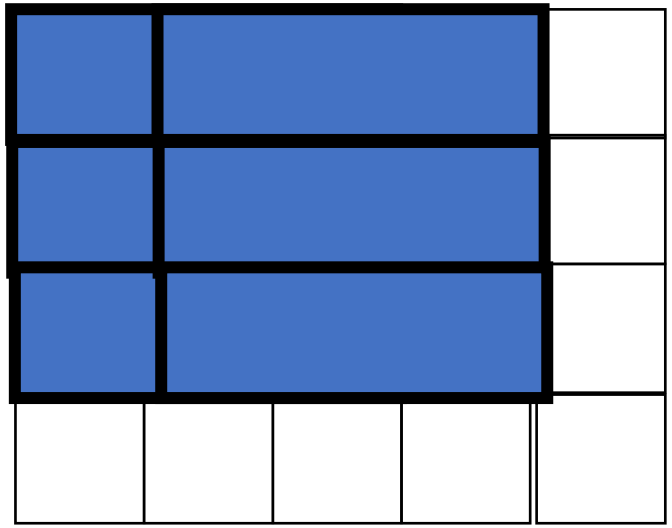
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

first iteration, only compute blur_x
second iteration, compute blur_x again:

blur

sliding window

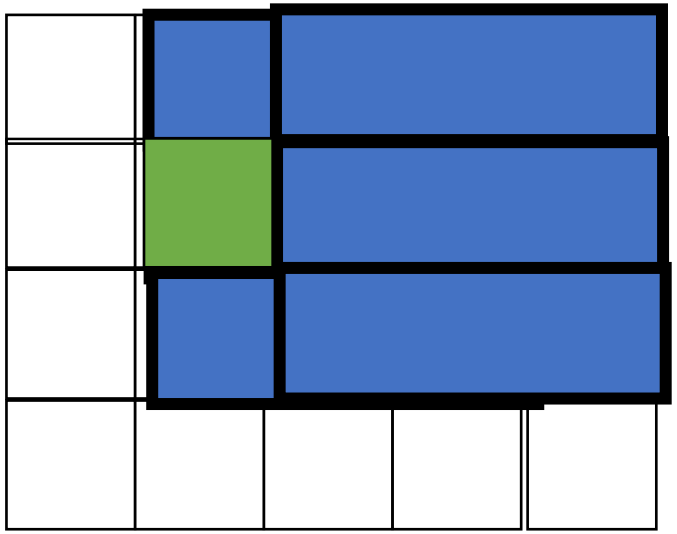


Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

sliding window

blur



first iteration, only compute blur_x
second iteration, compute blur_x again:
third iteration, compute_blur_x again, but
also compute blur,

blur_x should be available,

pros? cons?

Pros cons of each?

- Completely different buffers?
- Completely inlined functions?
- Sliding window?

- Control through a “schedule” and search spaces.
- Fused functions can take advantage of all function schedules (e.g. tiling)

Monday

- Finish up on Halide (results)
- Move to a graph DSL