CSE211: Compiler Design

Nov. 10, 2022

- **Topic**: DSLs for data processing
  - Halide
  - Graphs

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

Announcements

- Lots of grades got put in on Tuesday
  - attendance
  - Midterm

- Let us know ASAP (within 1 week) if there are any issues
  - For attendance, you might need to remind me if you had an excused day

Announcements

- Homework 3 is out
  - Please find a partner ASAP (the spreadsheet is live)
    - Put your name down on looking for a partner
    - Either talk between yourselves (e.g. on the class discord or piazza)
    - I really don't want to chase people down for this
  - It covers two topics:
    - A microbenchmark generator for ILP
    - Checking if loops are safe to do in parallel
  - Due Nov. 21

Announcements

- Start thinking about paper review and project
  - I made these canvas assignments so that people can register their papers and projects
- Paper: required by everyone
  - Get paper proposed by Nov. 15 (Next week)
  - Get paper approved by Nov. 17
  - I’m not going to chase you down for this, late policy still applies
- Project: You can do this or take the final
  - Project proposed by Nov. 15 (Next week)
  - Project approved by Nov. 17
  - You cannot switch after Nov. 15

Announcements

- Grading Plan:
  - Grade HW 2 next week
  - Strike makes everything a little uncertain

Review loop restructuring

Transforming Loops

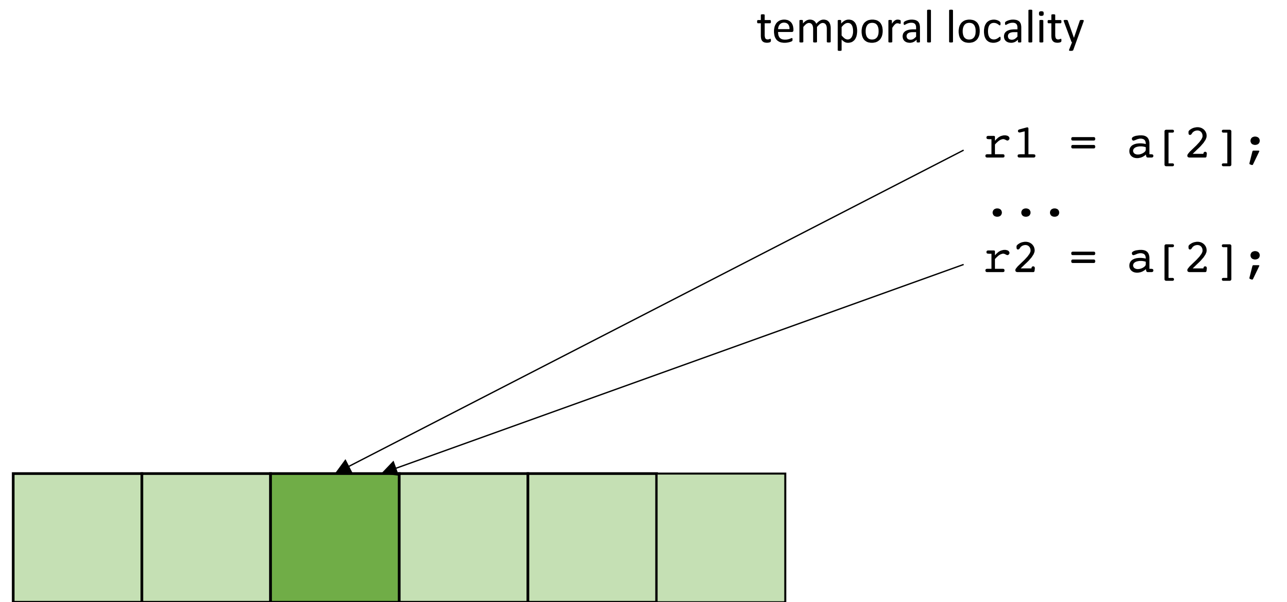
- Locality is key for good (parallel) performance:

- What kind of locality are we talking about?

Transforming Loops

- Locality is key for good parallel performance:

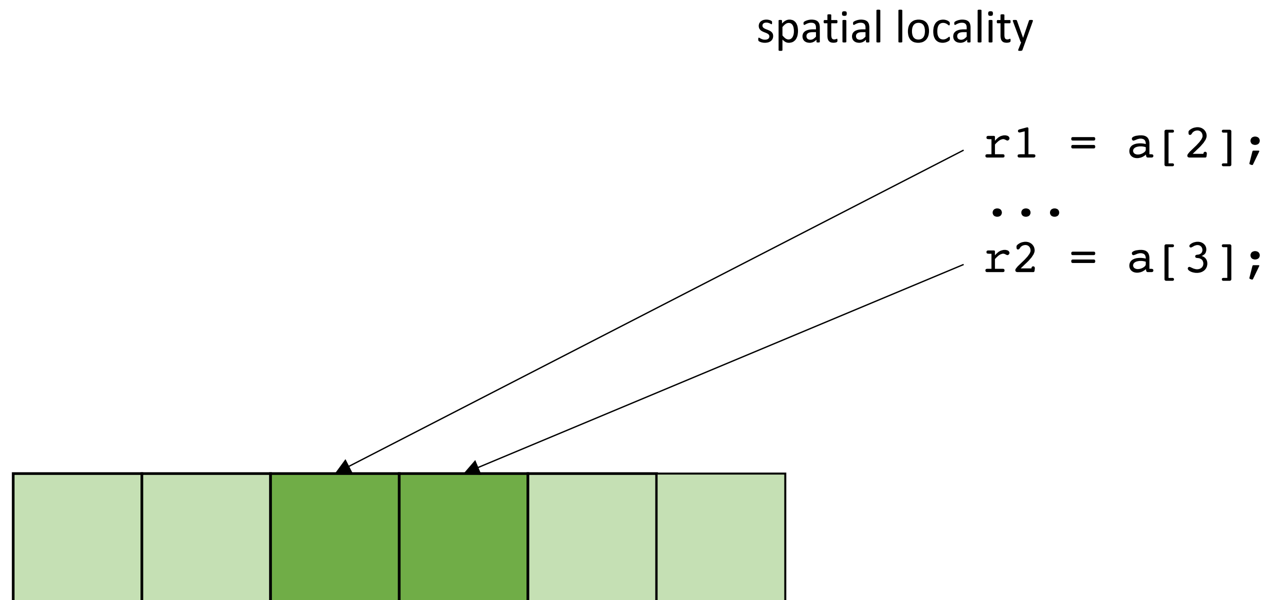
- Two types of locality:
  - Temporal locality
  - Spatial locality



Transforming Loops

- Locality is key for good parallel performance:

- Two types of locality:
  - Temporal locality
  - Spatial locality



How much does this matter?

```
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];
   }
}
```

```
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];
   }
}
```

which will be faster?
by how much?

How to reorder loop nestings?

- For a loop when can we reorder loop nestings?
  - If loop iterations are independent
  - If loop bounds are independent

- If the loop bounds are dependent...

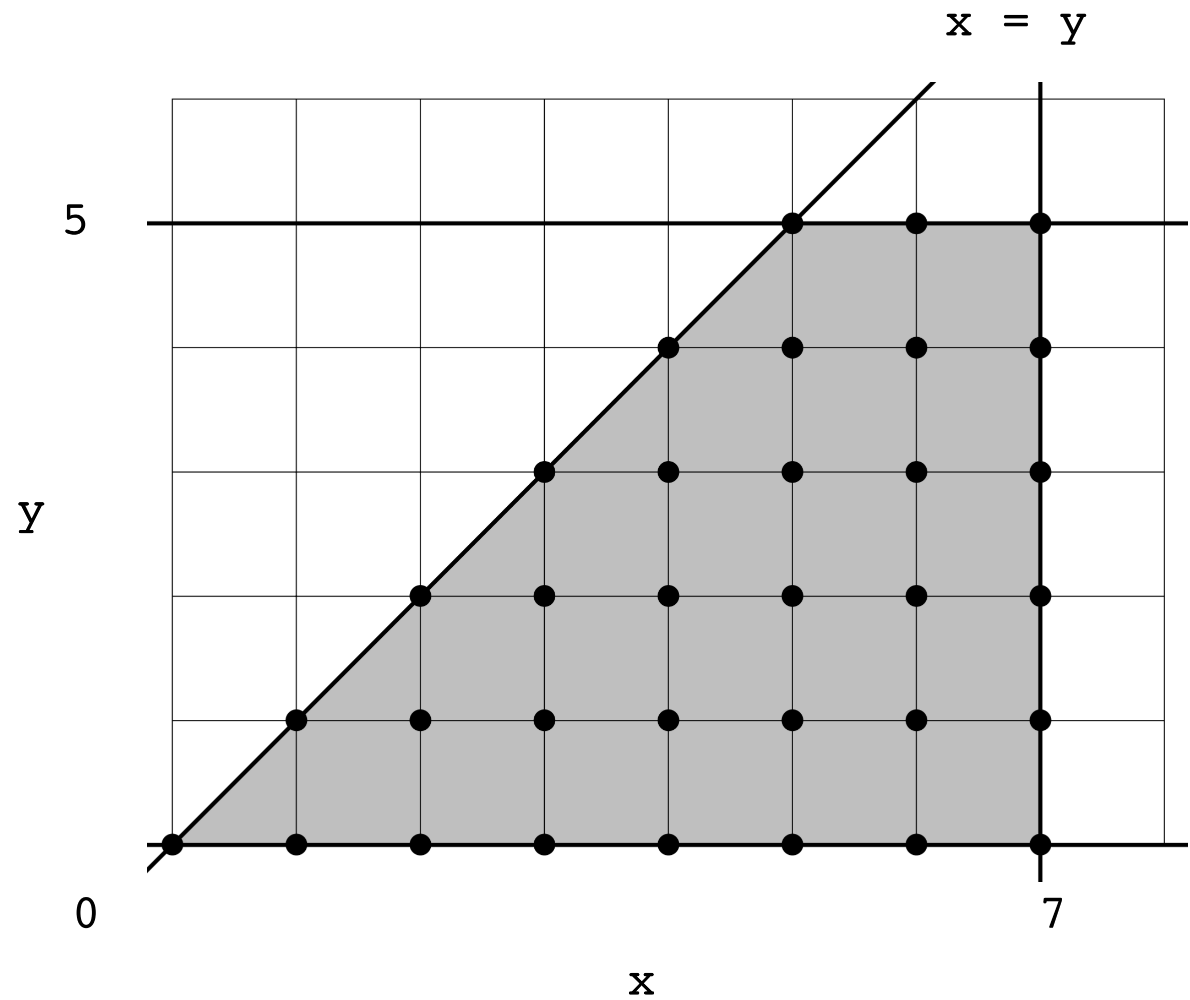
Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

Example:

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

System with N variables can be viewed as an N dimensional polyhedron



Fourier-Motzkin elimination:

- Given a system of inequalities with N variables, reduce it to a system with N - 1 variables.
- A system of inequalities describes an N-dimensional polyhedron. Produce a system of equations that projects the polyhedron onto an N-1 dimensional space

Example: remove y from the constraints

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

All pairs of upper/lower bounds on y:

```
0 <= y <= 5
0 <= y <= x
```

Then eliminate y:

```
0 <= 5
0 <= x
```

loop constraints without y:

```
x >= 0
x <= 7
```

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

new order: [x,y]

for x: eliminate y using FM elimination:

x loop constraints without y:

```
x >= 0
x <= 7
```

```
y loop constraints:
y >= 0
y <= 5
y <= x
```

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

new order: [x,y]

for x: eliminate y using FM elimination:

x loop constraints without y:

```
x >= 0
x <= 7
```

```
y loop constraints:
y >= 0
y <= 5
y <= x
```

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

new order: [x,y]

for x: eliminate y using FM elimination:

x loop constraints without y:

```
x >= 0
x <= 7
```

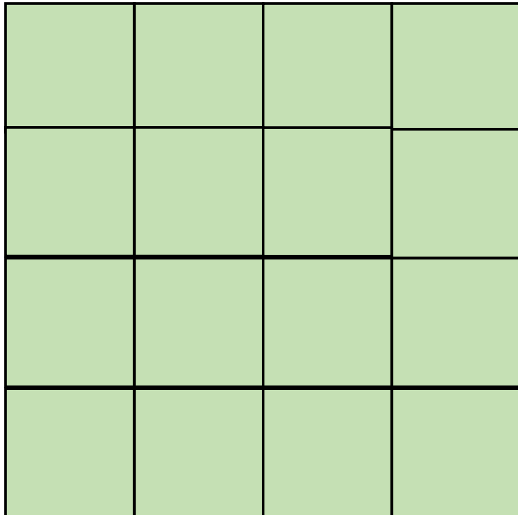
```
y loop constraints:
y >= 0
y <= min(x,5)
```

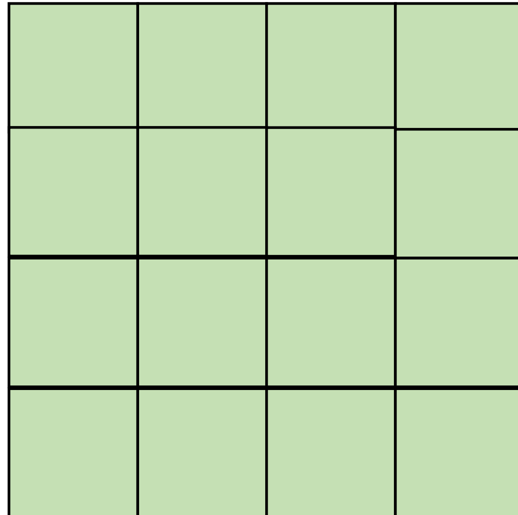
Adding loop nestings

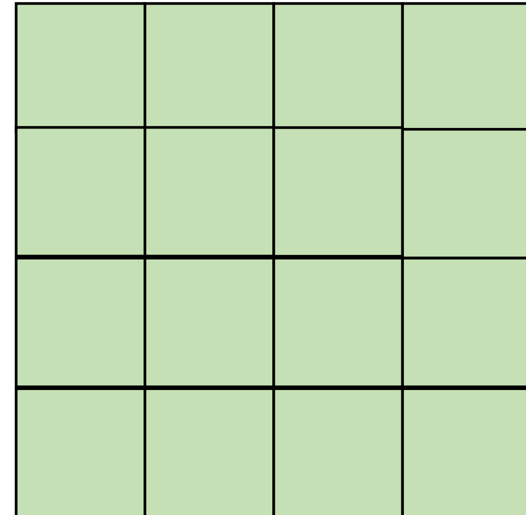
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

A B C







CSE211: Compiler Design

Nov. 10, 2022

- **Topic**: DSLs for data processing
  - Halide
  - Graphs

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

Discussion

Discussion questions:

What is a DSL?

What are the benefits and drawbacks of a DSL?

What DSLs have you used?

What is a DSL

- Objects in an object oriented language?
  - operator overloading (C++ vs. Java)
- Libraries?
  - Numpy
- Does it need syntax?
  - Pytorch/Tensorflow

What is a DSL

- Not designed for general computation, instead designed for a domain

- How wide or narrow can this be?
    - Numpy vs TensorFlow
    - Pros and cons of this design?

- Domain specific optimizations
    - Optimizations do not have to work well in all cases

The rest of the lecture

- 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

- Wait, now the programmer only needs 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

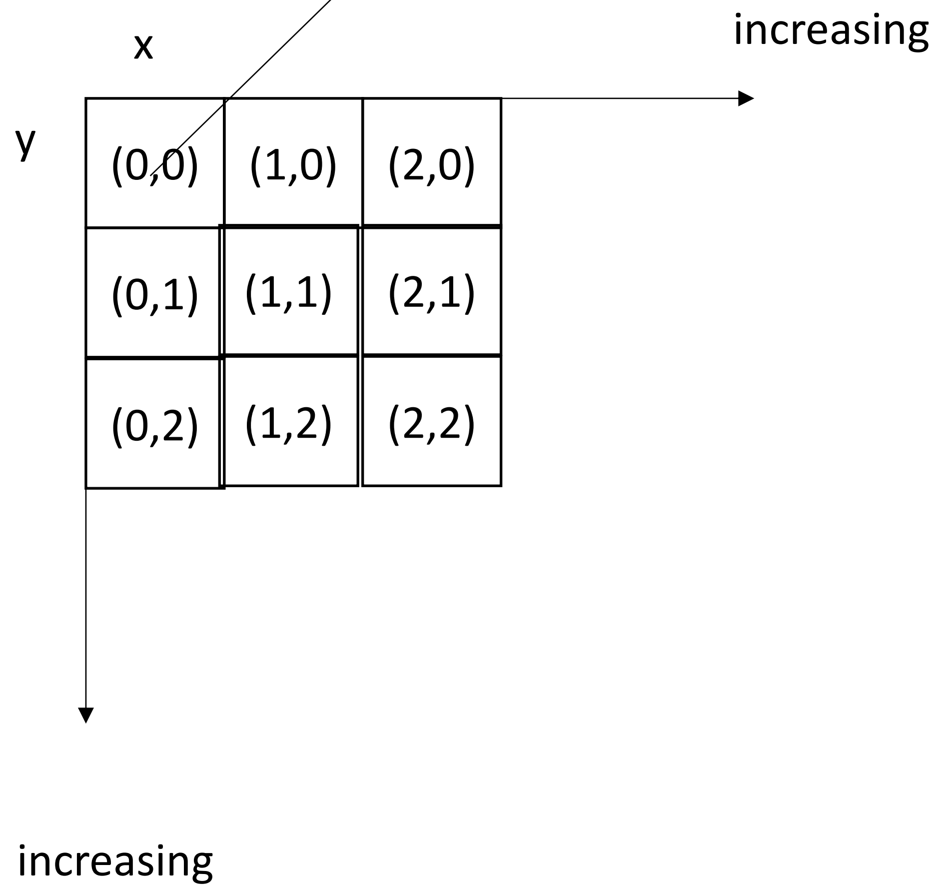
- Halide programs:
  - built into C++, contained within a header

```
#include "Halide.h"
```

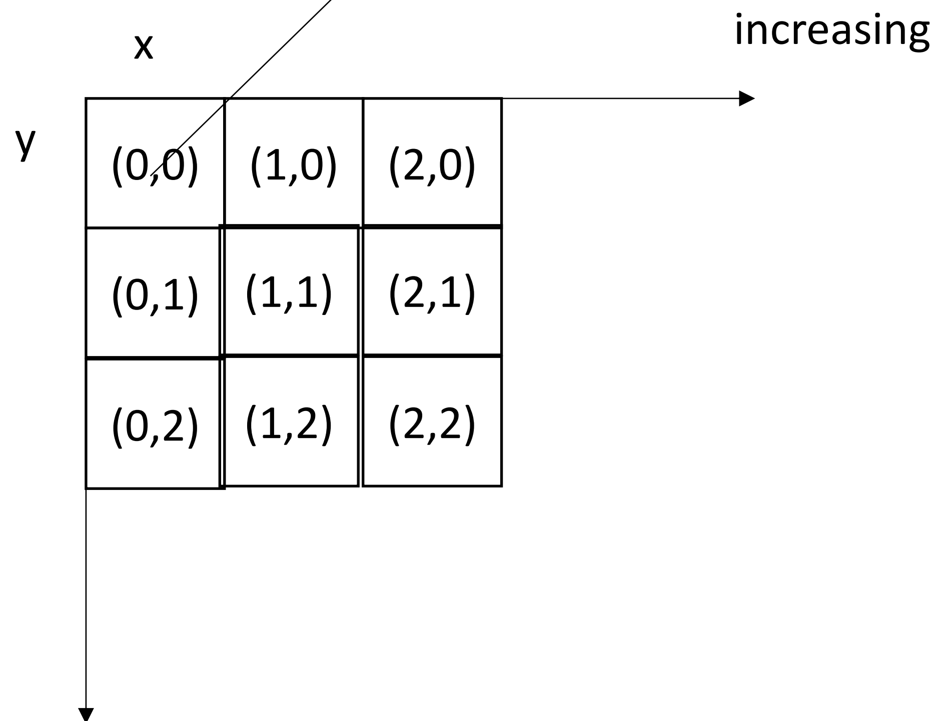
```
Halide::Func gradient;      // a function declaration

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;
```

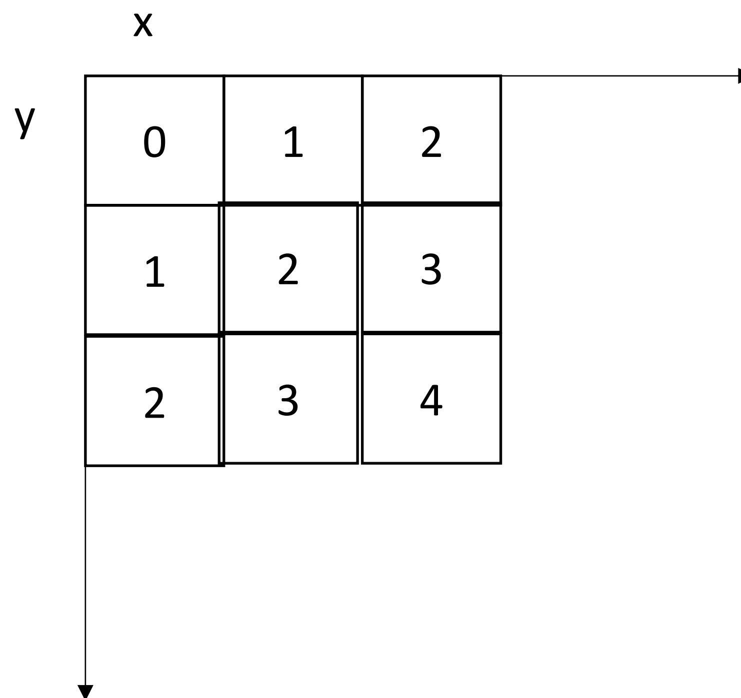


```
gradient(x, y) = x + y;
```



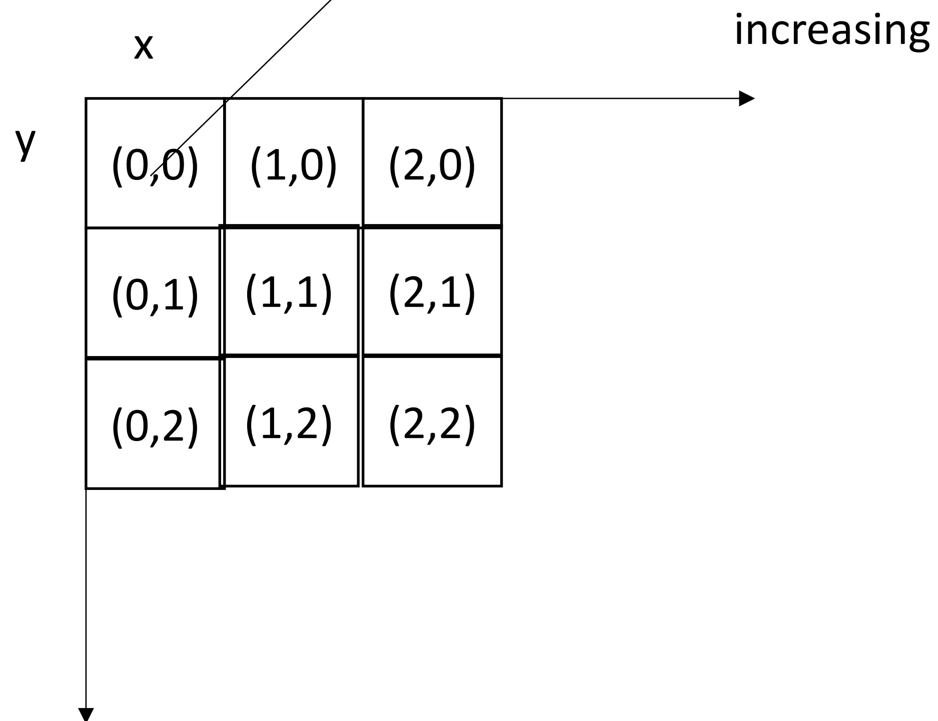
increasing

after applying the gradient function



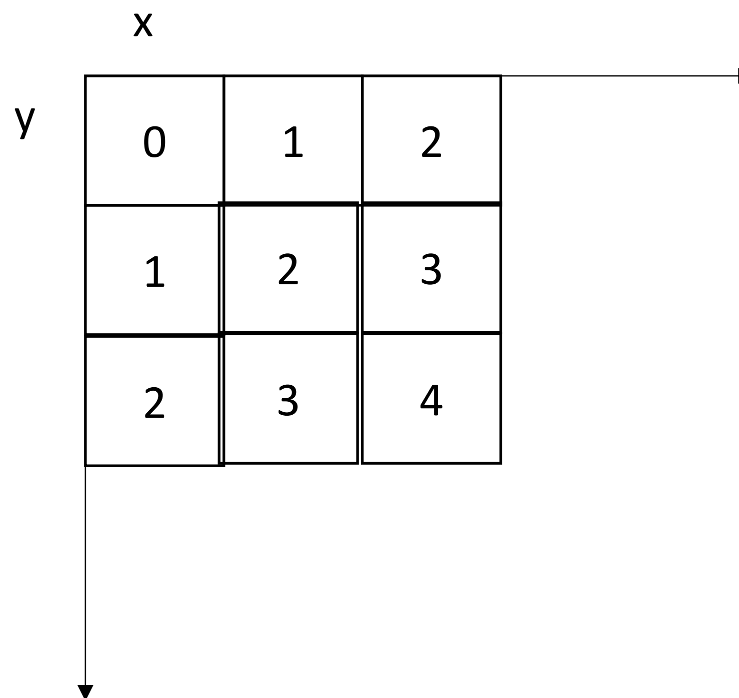
what are some properties of this computation?

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



increasing

after applying the gradient function

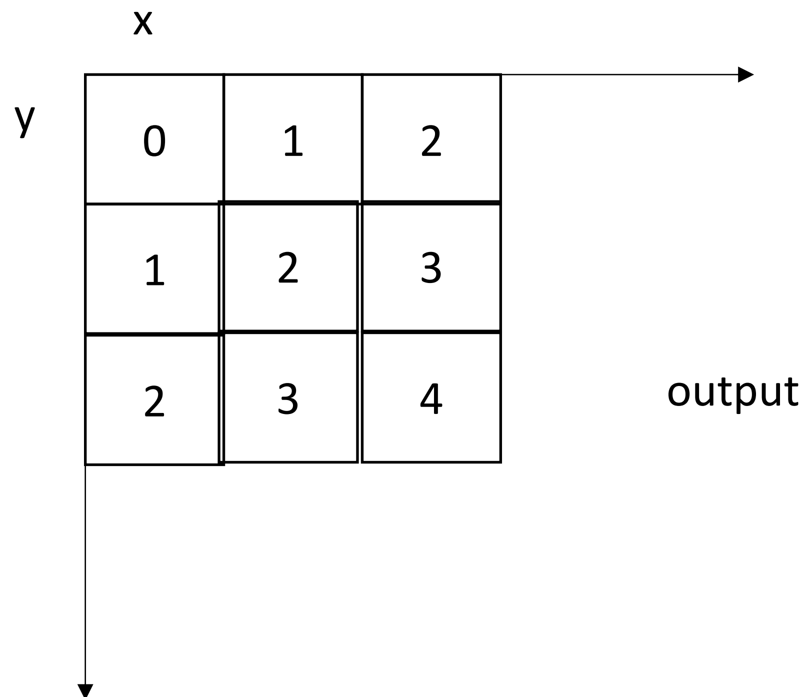


what are some properties of this computation?
Data races?
Loop indices and increments?
The order to compute each pixel?

Executing the function

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

Not compiled until this point
Needs values for x and y



Example: brightening

Brighten example

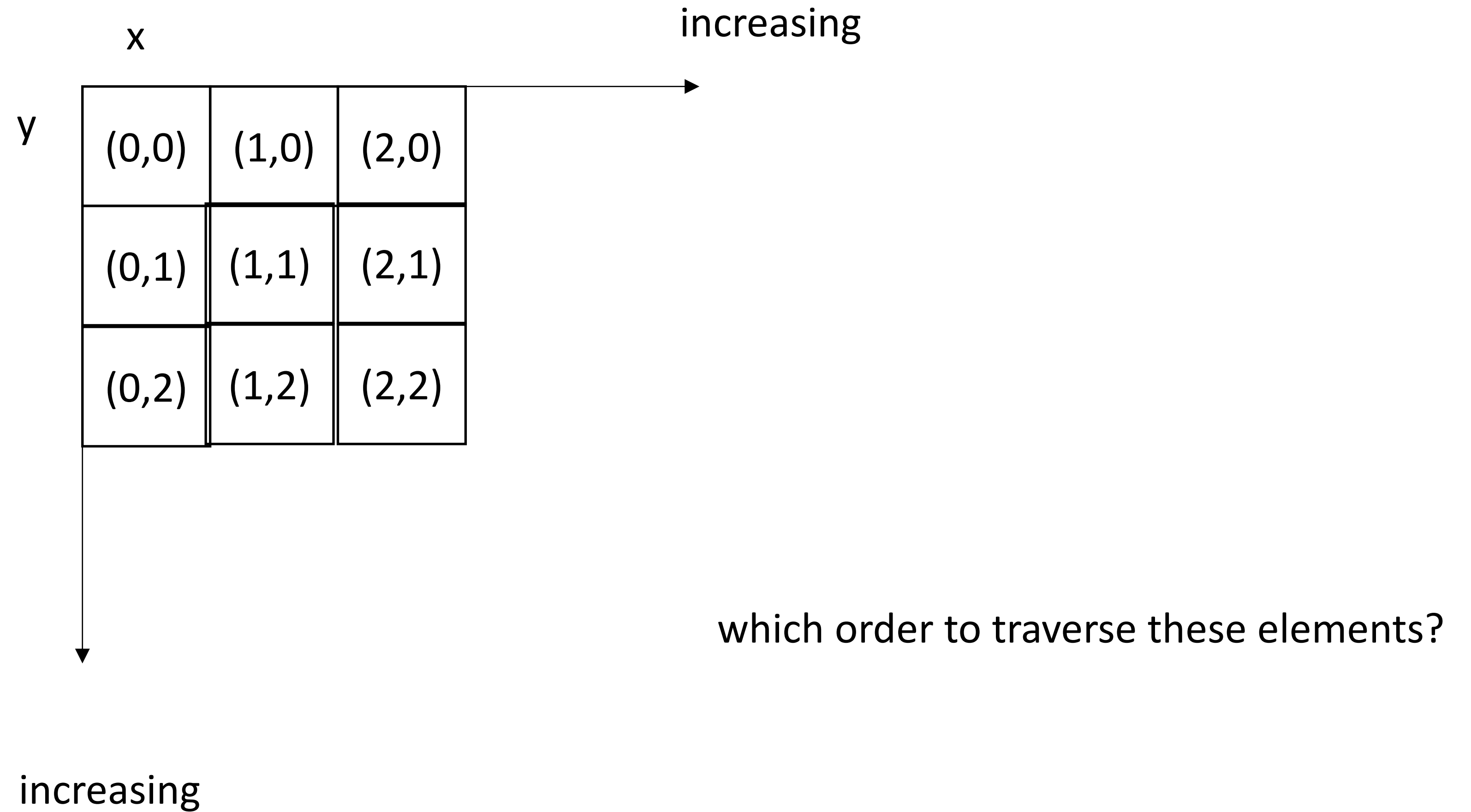
```cpp
Halide::Buffer<uint8_t> input = load_image("parrot.png");

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()});
```

```cpp
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()});
```

```cpp
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});
```

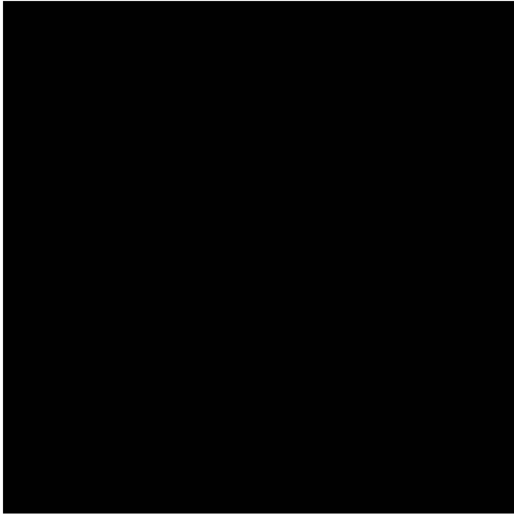


```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

```
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);
```

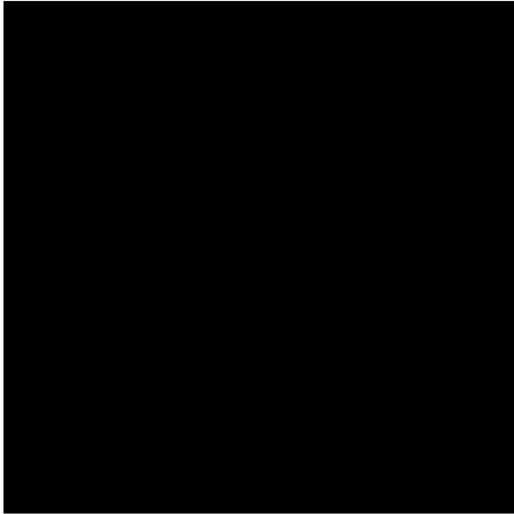
```
for (int x = 0; x < 4; x++) {
    for (int y = 0; y < 4; y++) {
        output[y,x] = x + y;
    }
}
```

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_outer*2 + x_inner;
            output[y,x] = x + y;
        }
    }
}
```

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

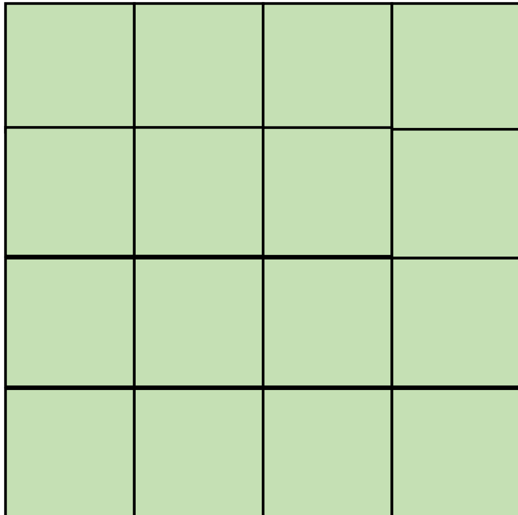
Tiling

Adding loop nestings

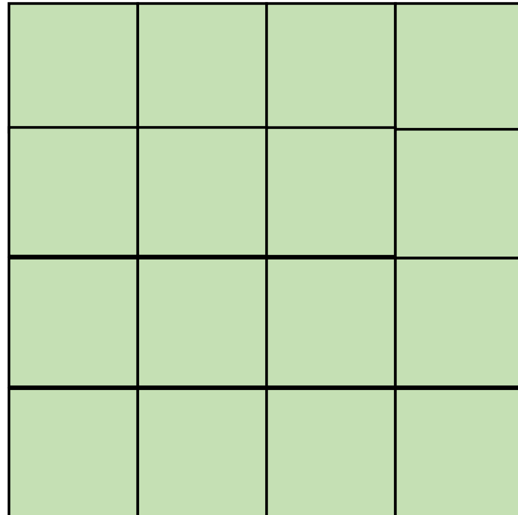
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

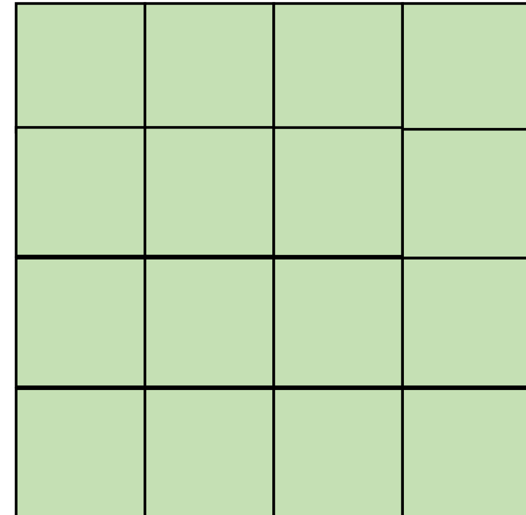
A



B



C

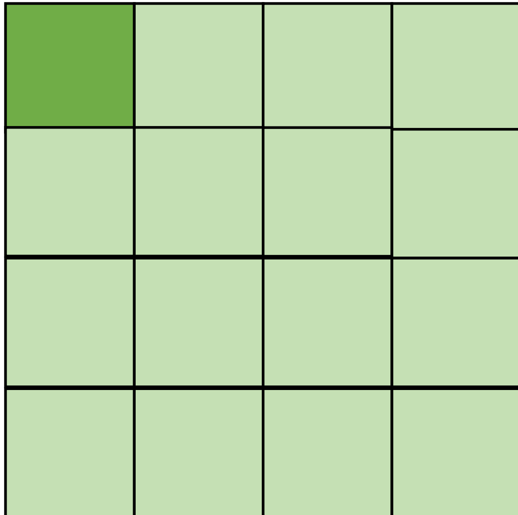


Adding loop nestings

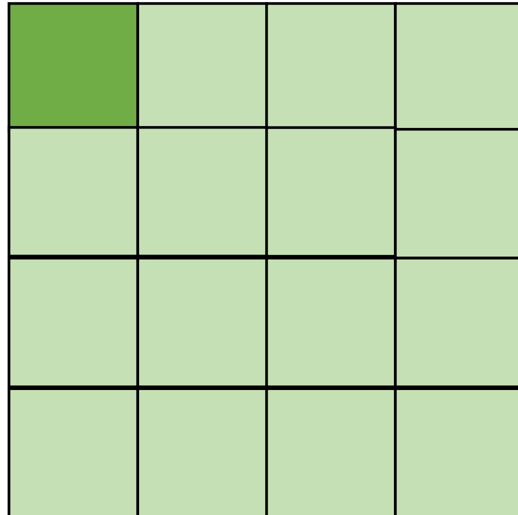
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

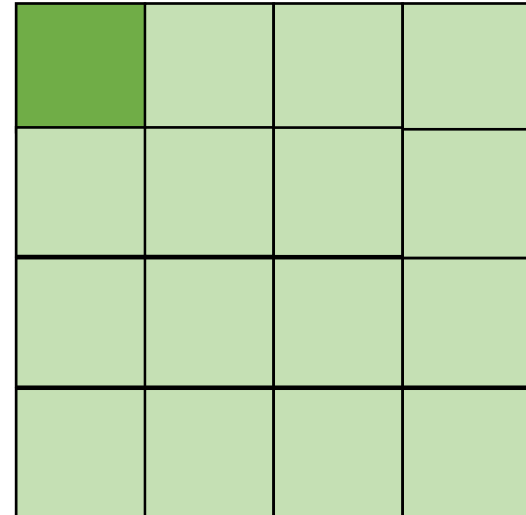
A



B



C



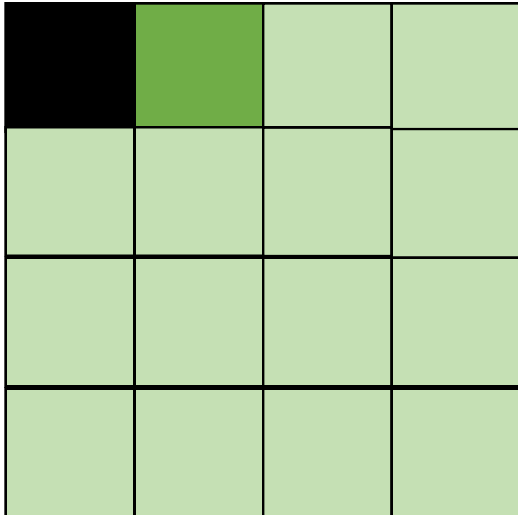
cold miss for all of them

Adding loop nestings

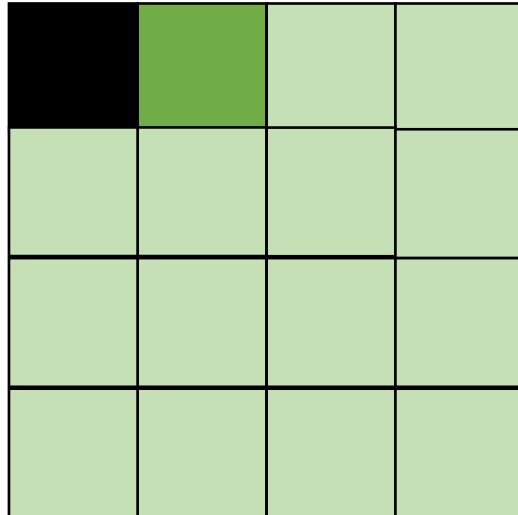
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

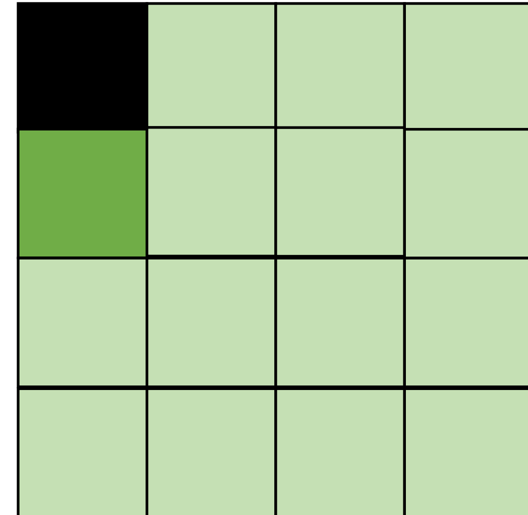
A



B



C



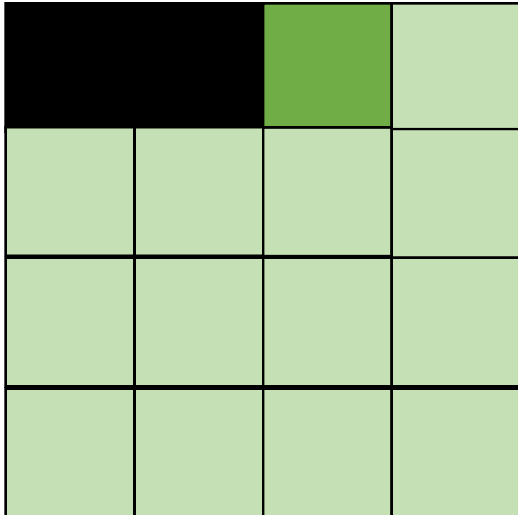
Hit on A and B. Miss on C

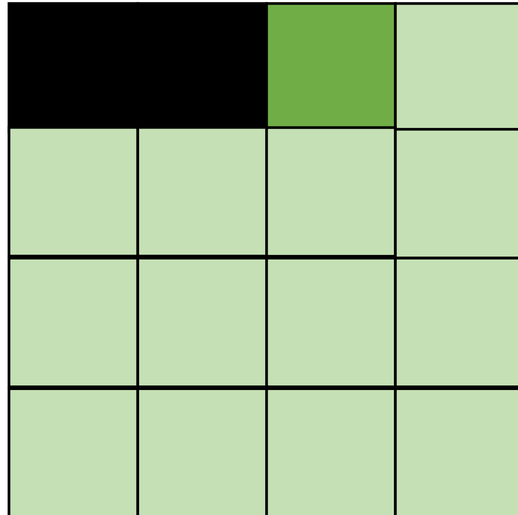
Adding loop nestings

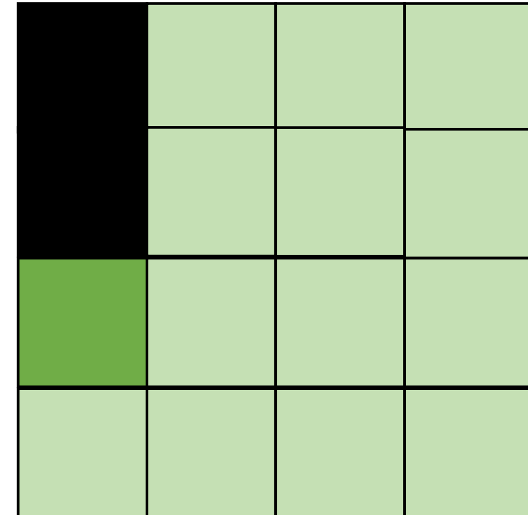
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

A B C







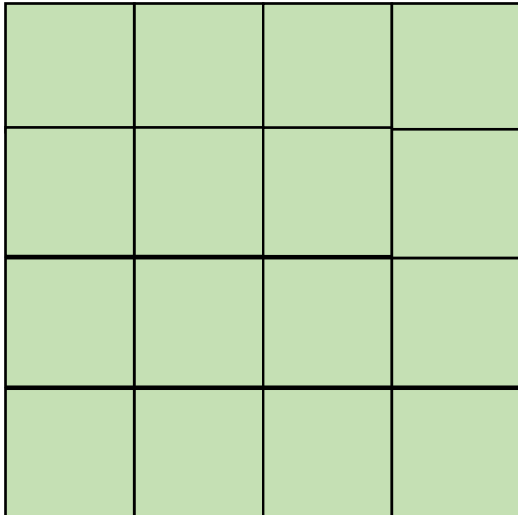
Hit on A and B. Miss on C

Adding loop nestings

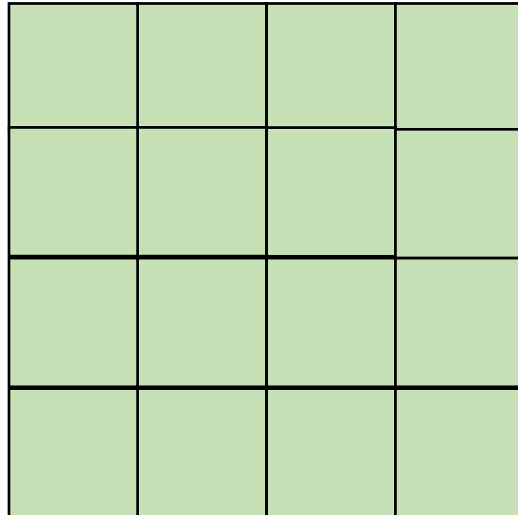
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

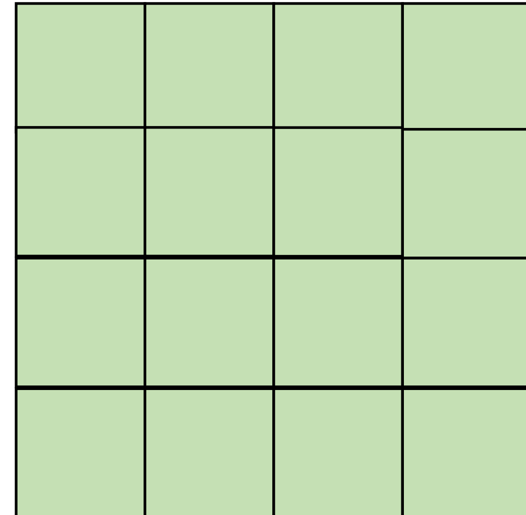
A



B



C

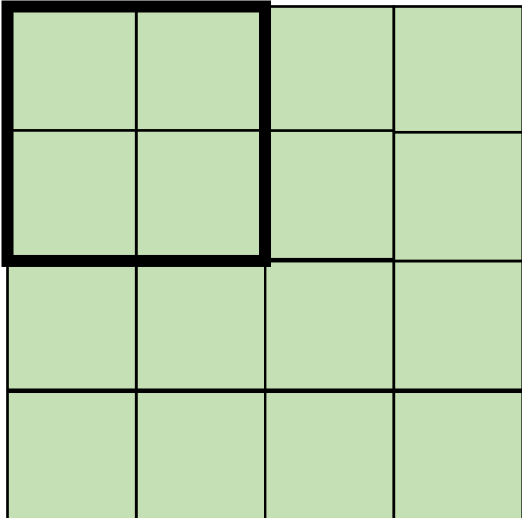


Adding loop nestings

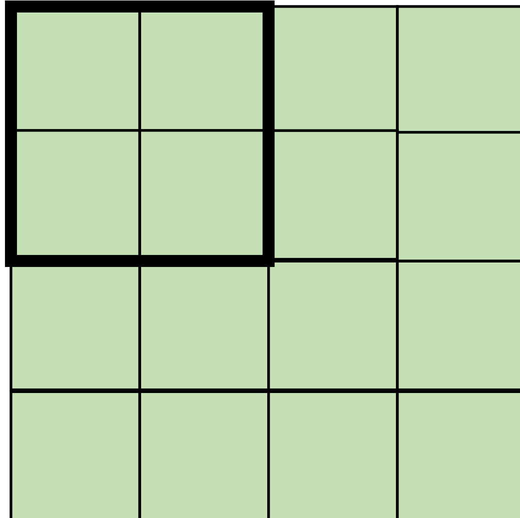
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

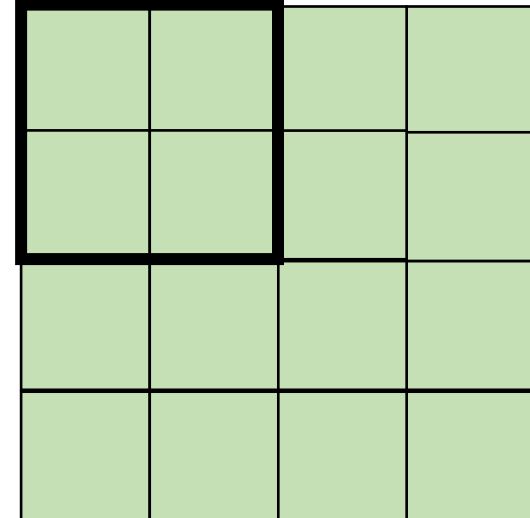
A



B



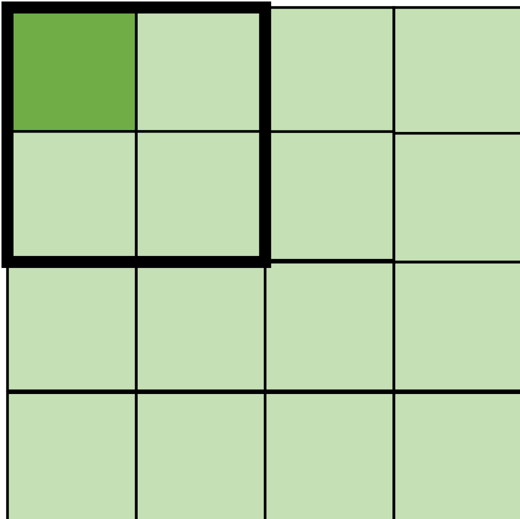
C

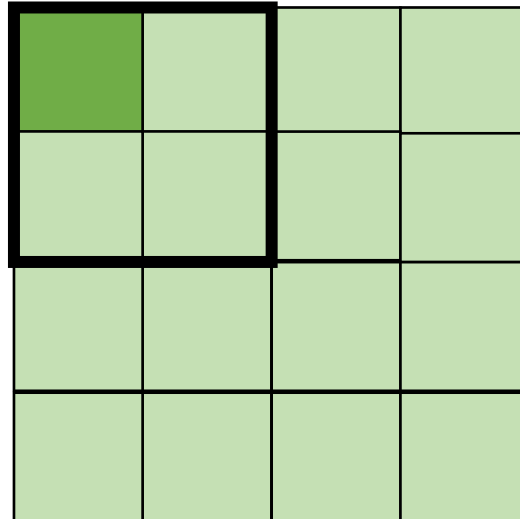


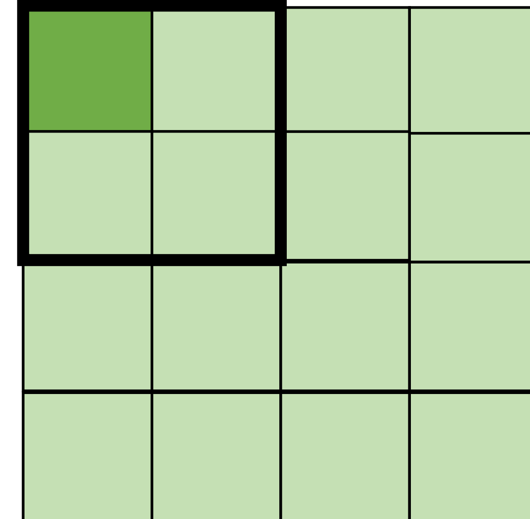
Adding loop nestings

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

$$A = B + C^T$$





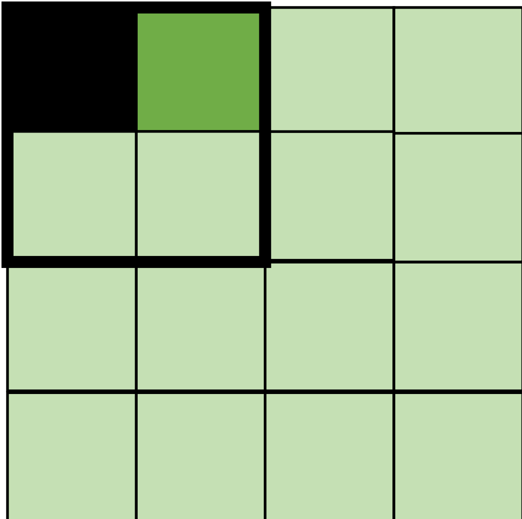


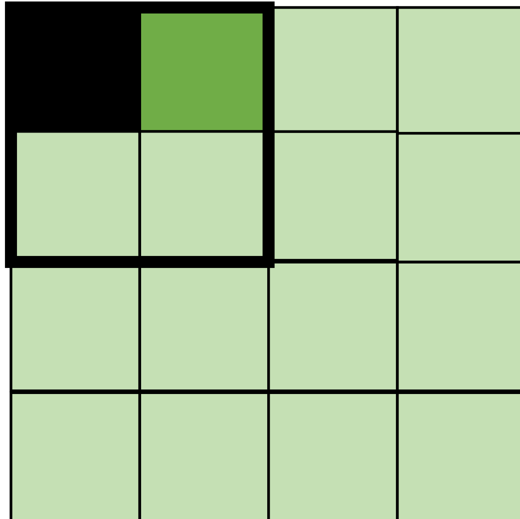
cold miss for all of them

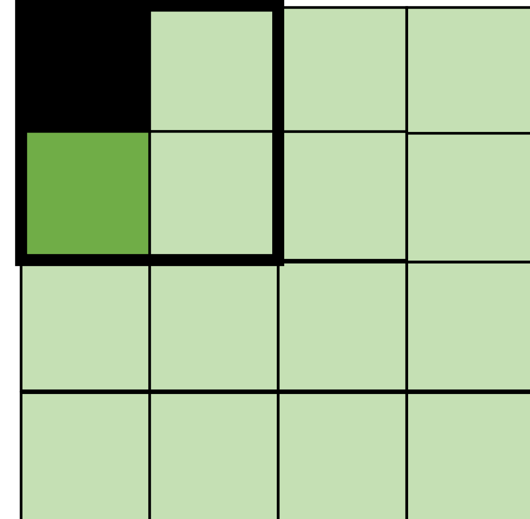
Adding loop nestings

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

$$A = B + C^T$$





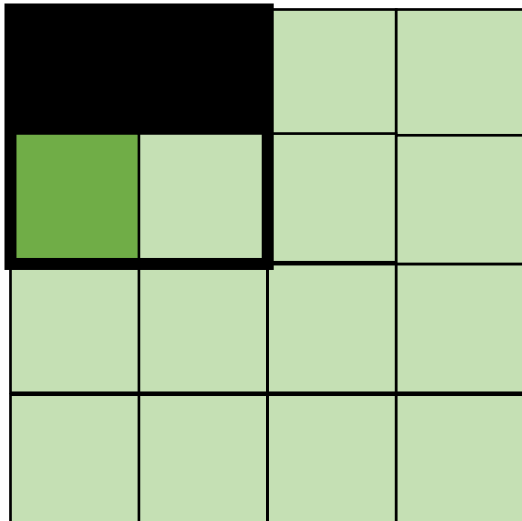


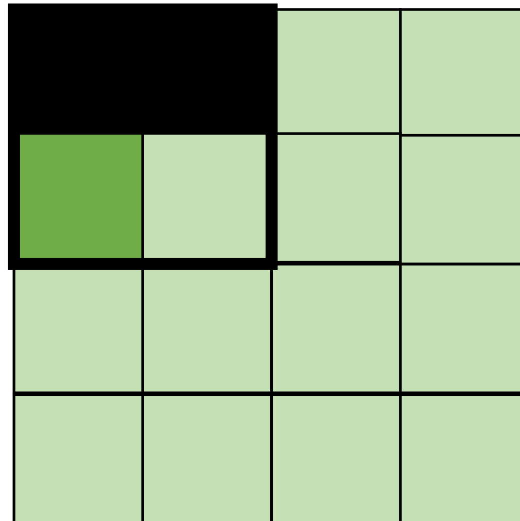
Miss on C

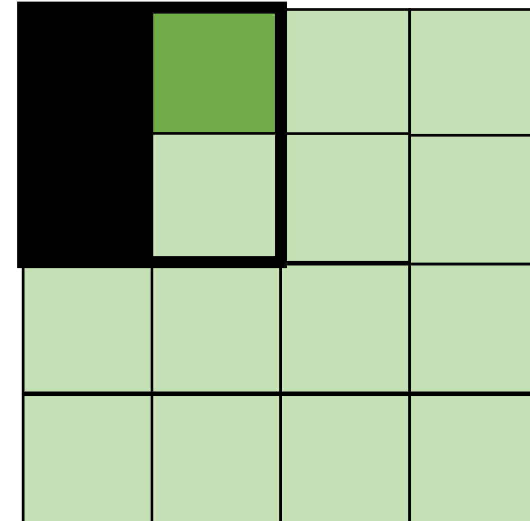
Adding loop nestings

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

$$A = B + C^T$$





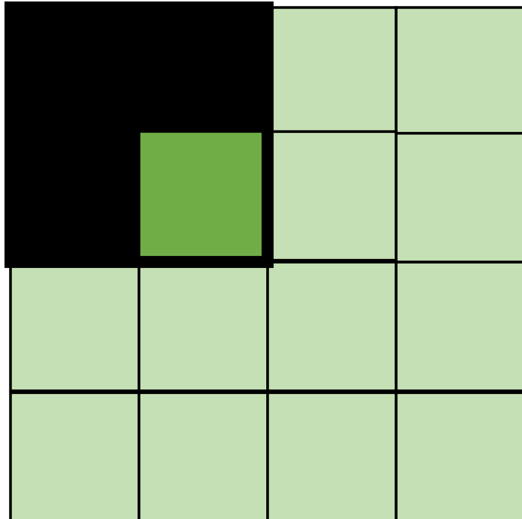


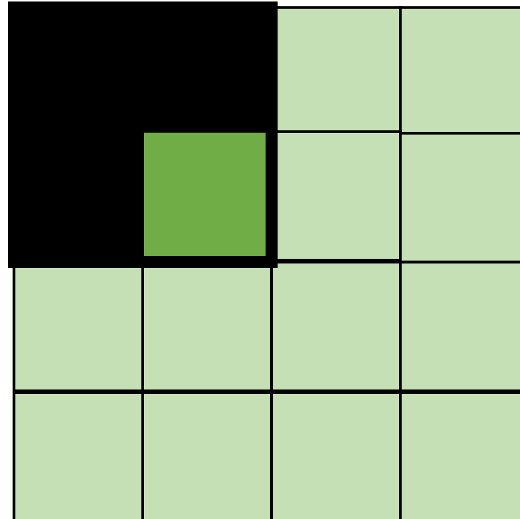
Miss on A,B, hit on C

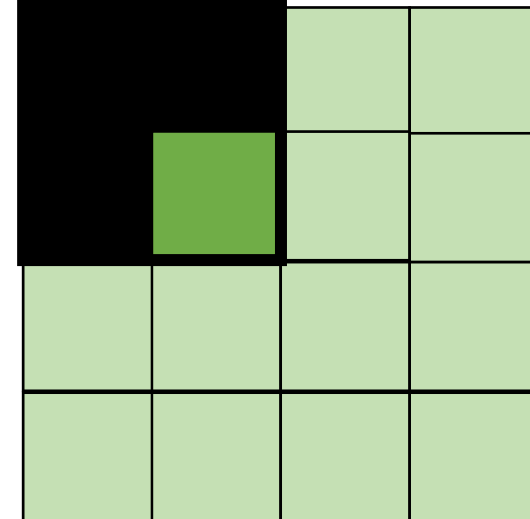
Adding loop nestings

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

$$A = B + C^T$$







Hit on all!

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

transforms into:

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

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

How to do this with Halide transformations?

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

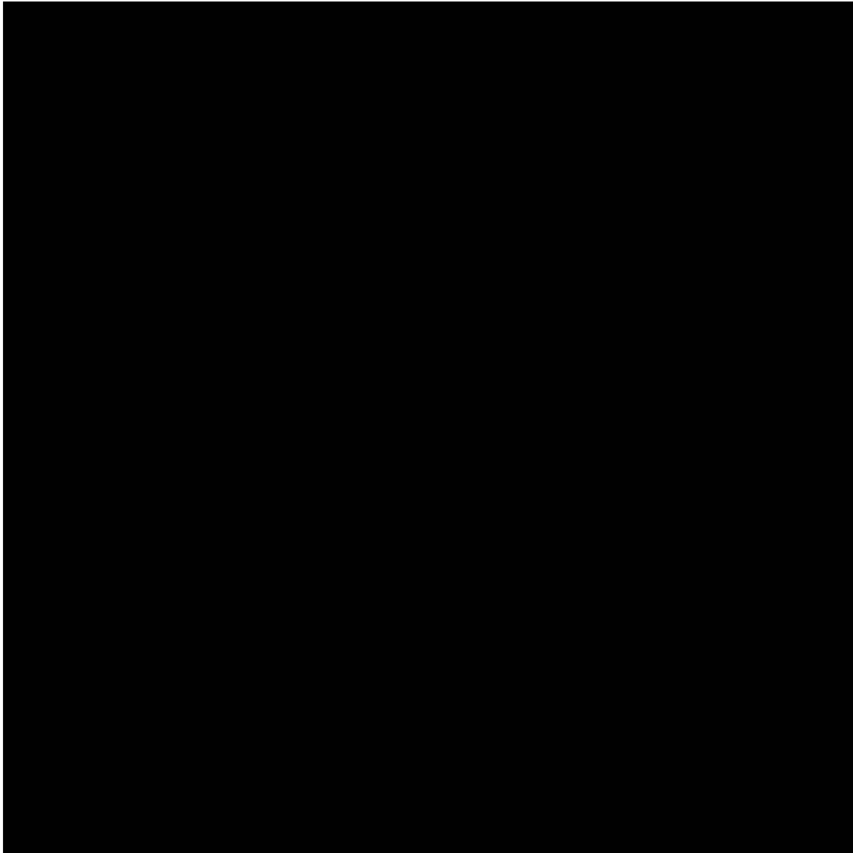
Schedule

```
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 < 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 < 16; y++) {
    for (int x = 0; x < 16; 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;
```

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

Parallelism?

```
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

What about parallelism?

```
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

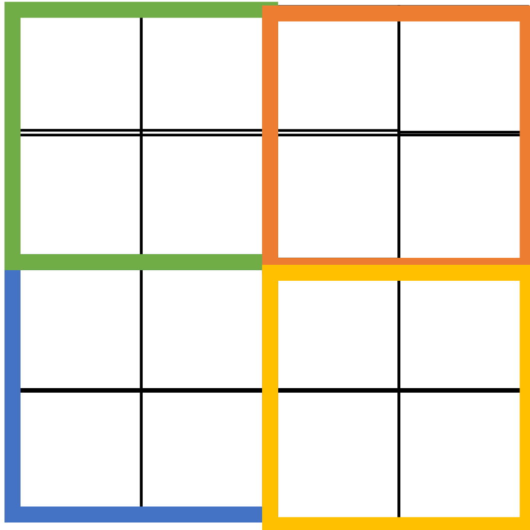
```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.vectorize(x_inner);
```

```
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

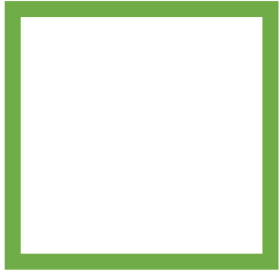
SMP Parallelism

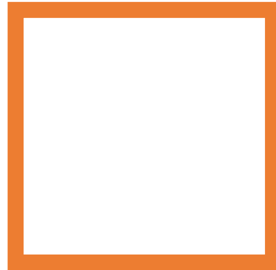
```
for (int y_outer = 0; y_outer < 2; y_outer++) {
  for (int x_outer = 0; x_outer < 2; x_outer++) {
    for (int y_innder = 0; y_inner < 2; y_inner++) {
      for (int x_inner = 0; x_inner < 2; x_inner++) {
        ...
      }
    }
  }
}
```

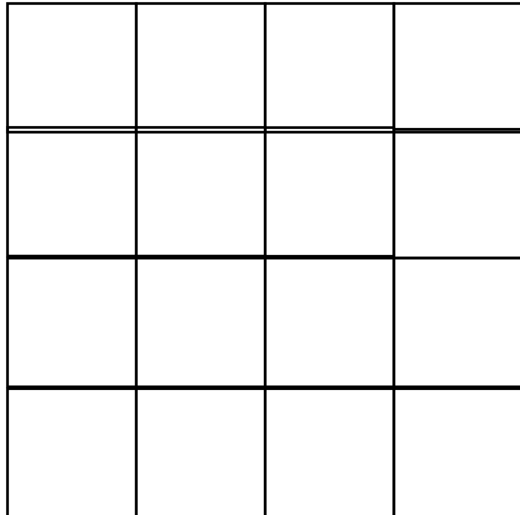


How can we make parallel across tiles?

```
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++) {
        ...
      }
    }
  }
}
```

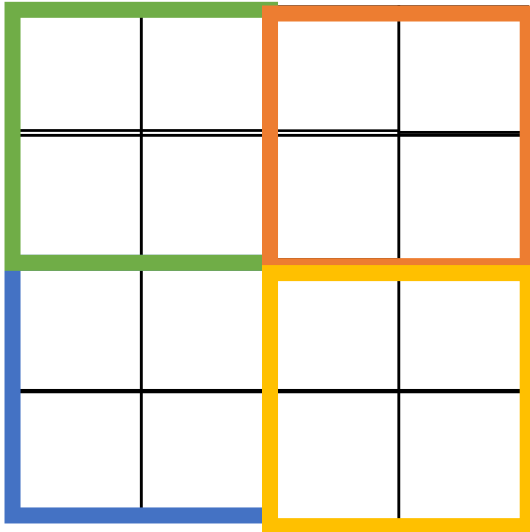






if you make parallel across the outer loop
what about inner loop?
Ideas on how to fix?

```
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++) {
        ...
      }
    }
  }
}
```



```
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({SIZE, SIZE});


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);
```

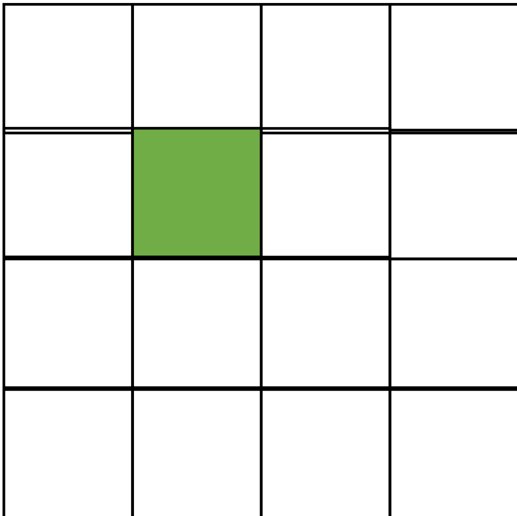
Now for function fusing...

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);
```

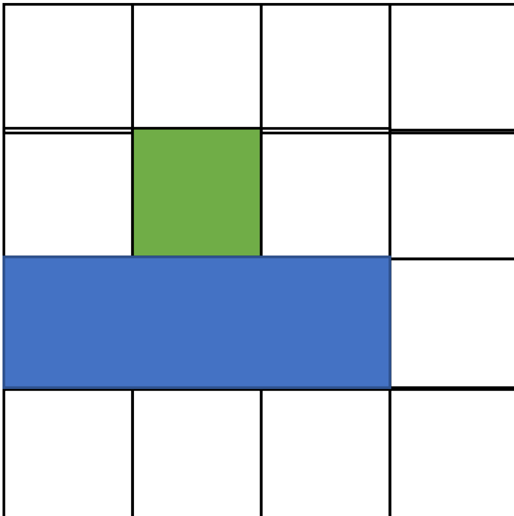
Example: unnormalized blur

```
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);
```



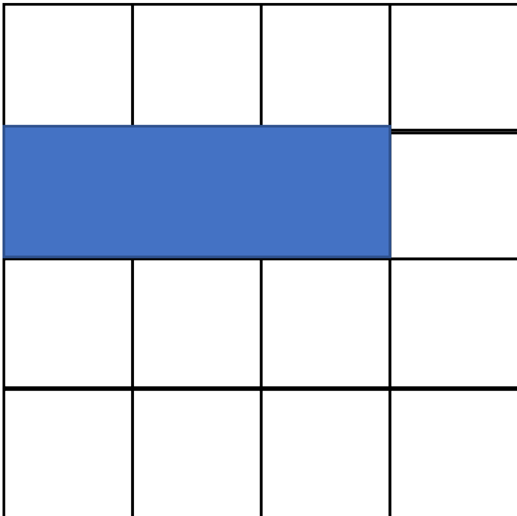
Example: unnormalized blur

```
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);
```



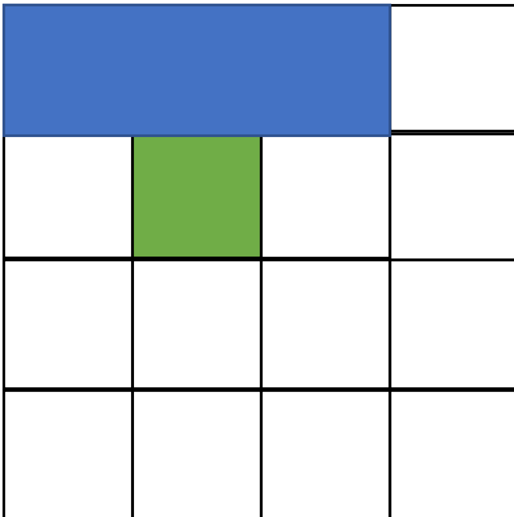
Example: unnormalized blur

```
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);
```



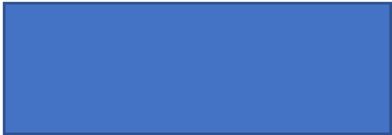
Example: unnormalized blur

```
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

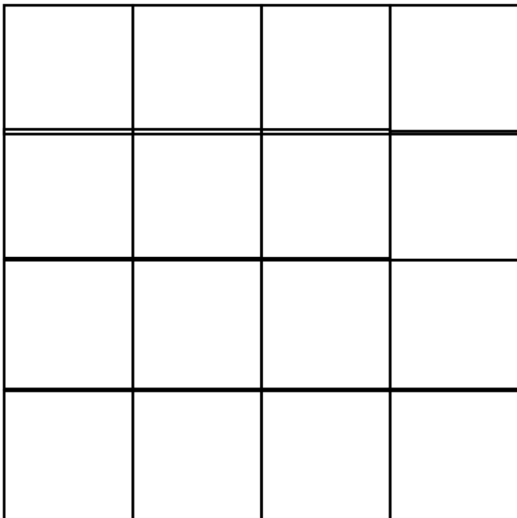
```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

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

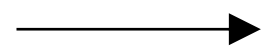


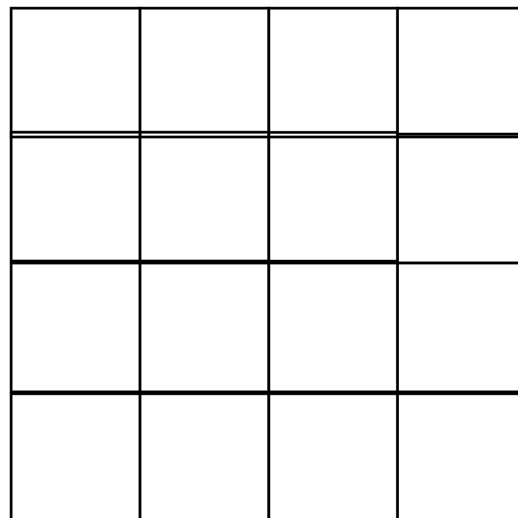
input

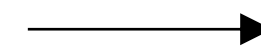
blur_x

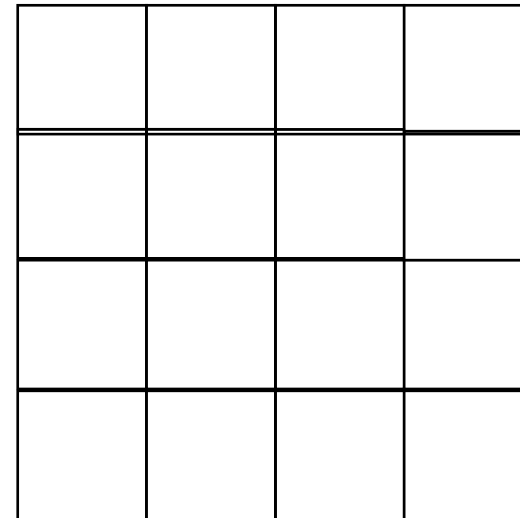
blur











Example: unnormalized blur

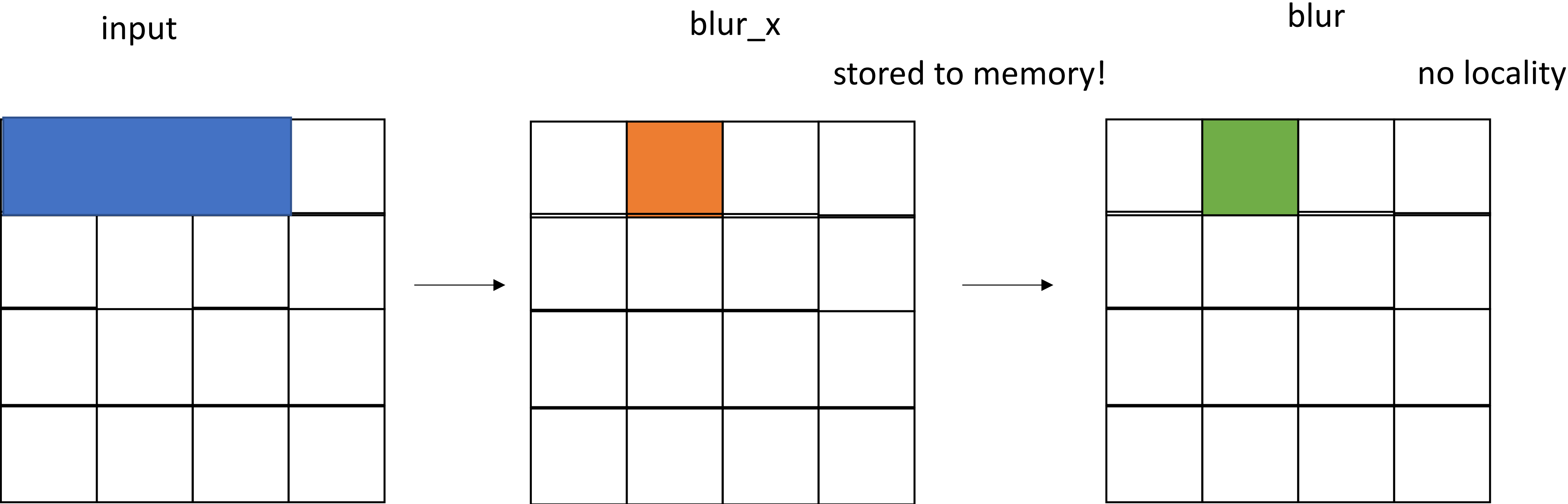
```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

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);
```



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);
```

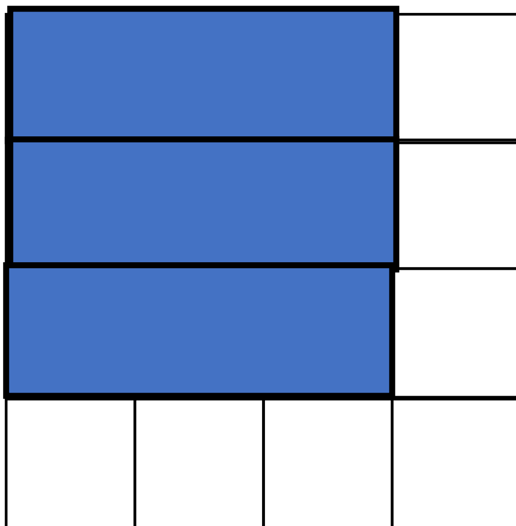
completely inline

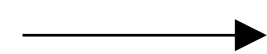
```
alloc out[2046][3072]
foreach y in 1..2047:
    foreach x in 0..3072:
      out[y][x] = in[y-1][x] + in[y][x] + in[y+1][x] +
                  in[y-1][x-1] + in[y][x-1] + in[y+1][x-1]
                  in[y-1][x+1] + in[y][x+1] + in[y+1][x+1]
```

Example: unnormalized blur

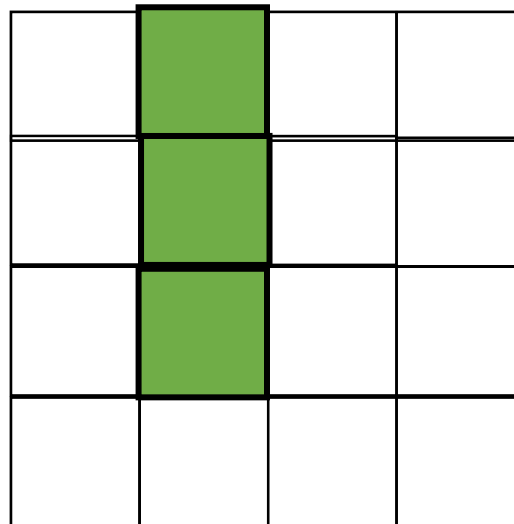
```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

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

input





blur_x



No need to store these back to memory!

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?

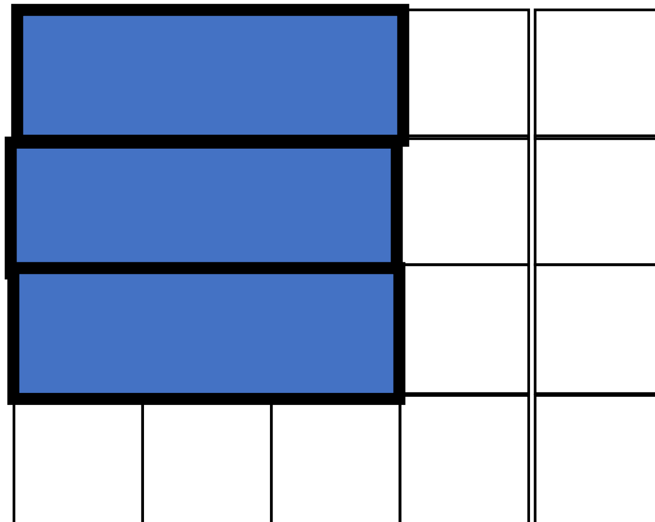
Example: unnormalized blur

```
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



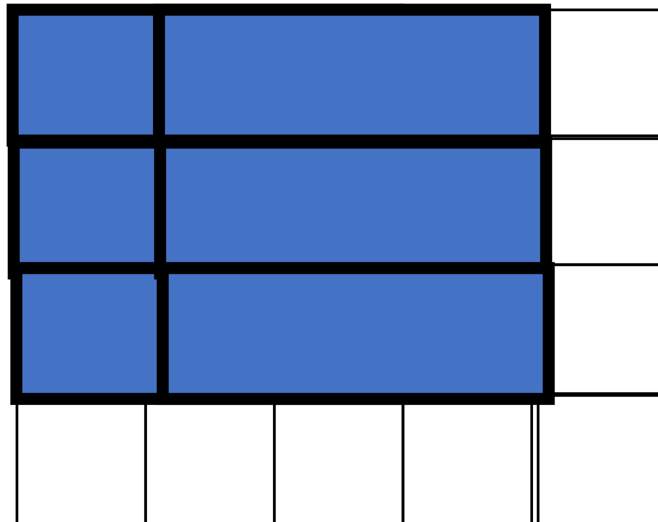
Example: unnormalized blur

```
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);
```

sliding window

blur

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

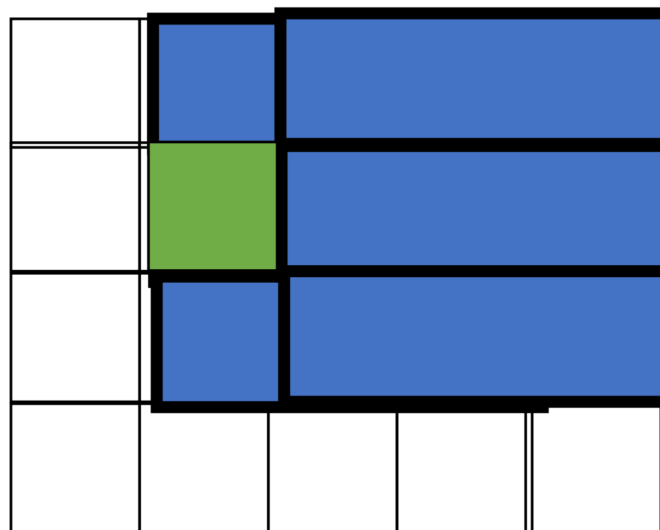


Example: unnormalized blur

```
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)

Lots of Halide optimizations appear in DNN DSLs (fusion, reordering, etc)

Tons of future work on mapping to new architectures!

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/

Moving on to another DSL

- For Graph computation