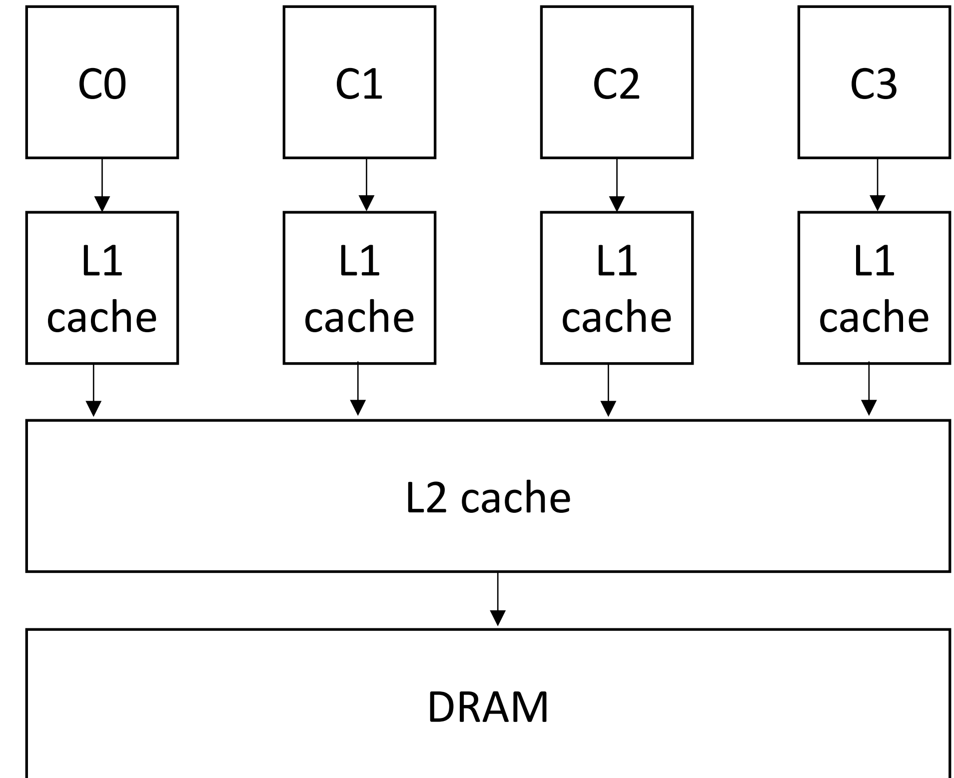
CSE211: Compiler Design

Nov. 8, 2022

- **Topic**: Loop structure and DSLs

- **Discussion questions**:
  - *Lots of discussions throughout about loops and DSLs*



Announcements

- Midterm was due on Friday
  - Please let me know if there was any issues ASAP

Announcements

- Homework 3 is out
  - Released yesterday
  - Please find a partner ASAP (the spreadsheet is live)
  - 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

- 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

- Homework 1 is graded
  - Thanks to Kyle and Arrian for all the help
  - Please let us know if there are any issues within 1 week
  - The test cases were released on Piazza
    - It was a combination of the provided tests and the tests you wrote.
- Plan:
  - Grade midterm this week
  - Grade HW 2 next week

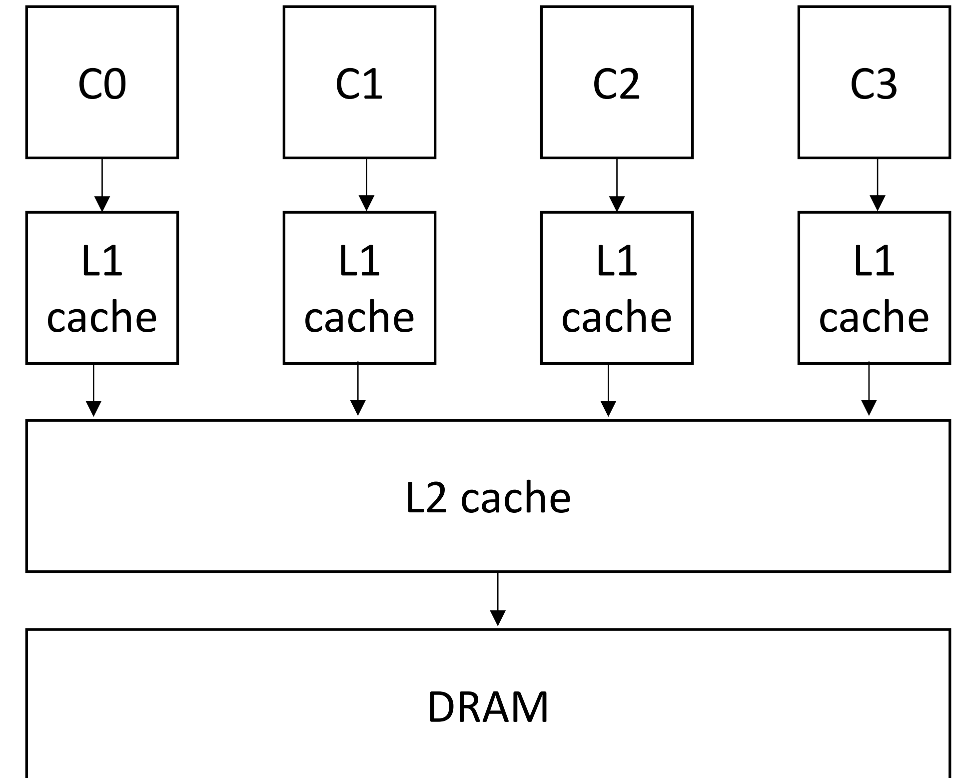
Review implementing parallel loops

CSE211: Compiler Design

Nov. 8, 2022

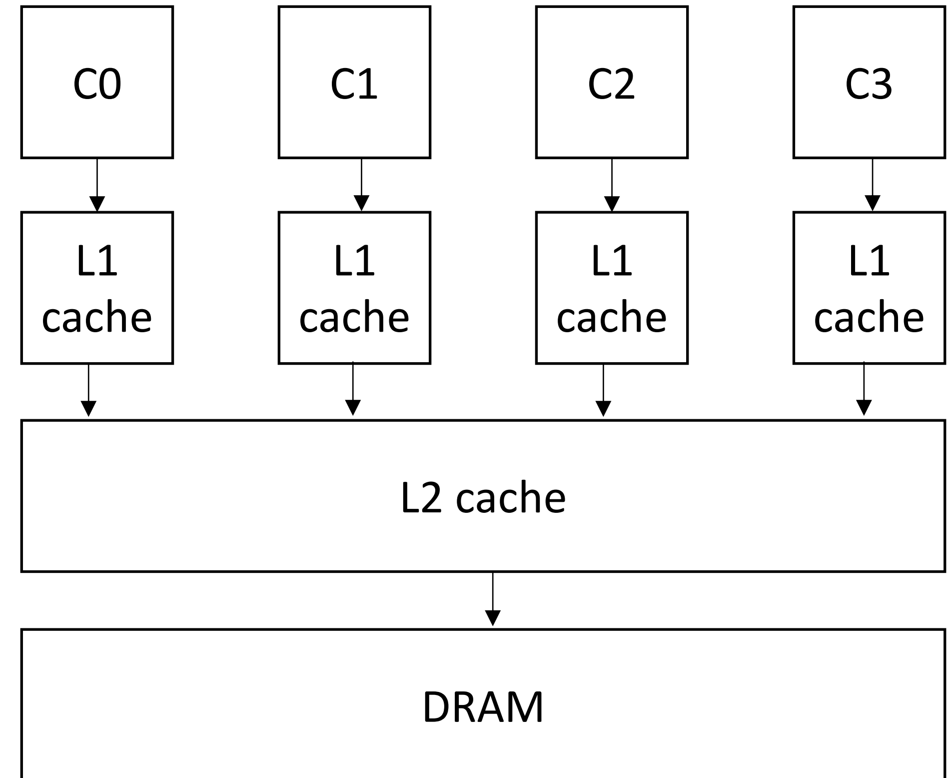
- **Topic**: Loop structure and DSLs

- **Discussion questions**:
  - *Lots of discussions throughout about loops and DSLs*



Shifting our focus back to a single core

- Why?



Shifting our focus back to a single core

- Why?

Scalability! But at what COST?

Frank McSherry
Unaffiliated

Michael Isard
Unaffiliated*

Derek G. Murray
Unaffiliated†

Abstract

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

We survey measurements of data-parallel systems recently reported in SOSP and OSDI, and find that many systems have either a surprisingly large COST, often ... of cores, or simply underperform one thread ... configurations.

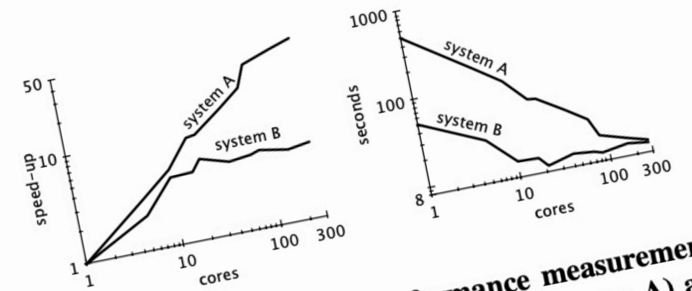


Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.

While this may appear to be a contrived example, we will argue that many published big data systems more closely

Shifting our focus back to a single core

- Why?

1 Introduction

"You can have a second computer once you've shown you know how to use the first one."

–Paul Barham

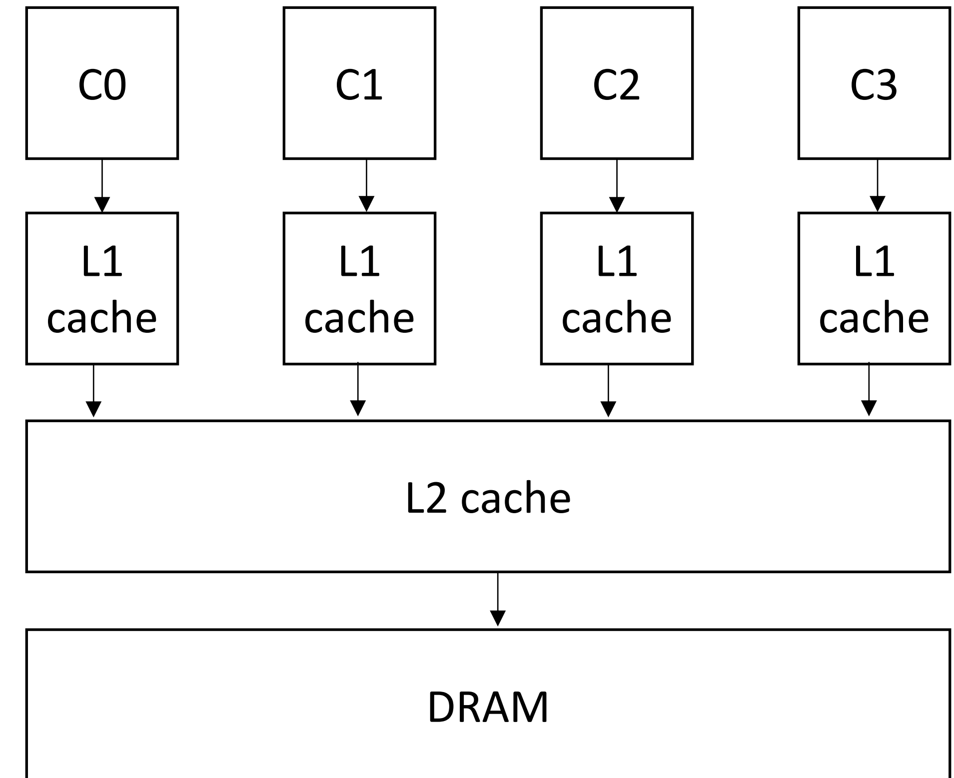
scalable system	cores	twitter	uk-2007-05
GraphChi [12]	2	3160s	6972s
Stratosphere [8]	16	2250s	-
X-Stream [21]	16	1488s	-
Spark [10]	128	857s	1759s
Giraph [10]	128	596s	1235s
GraphLab [10]	128	249s	833s
GraphX [10]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Table 2: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. GraphChi and X-Stream report times for 5 PageRank iterations, which we multiplied by four.

Shifting our focus back to a single core

- We need to consider single threaded performance

- Good single threaded performance can enable better parallel performance
    - **Memory locality** is key to good parallel performance.



Transforming Loops

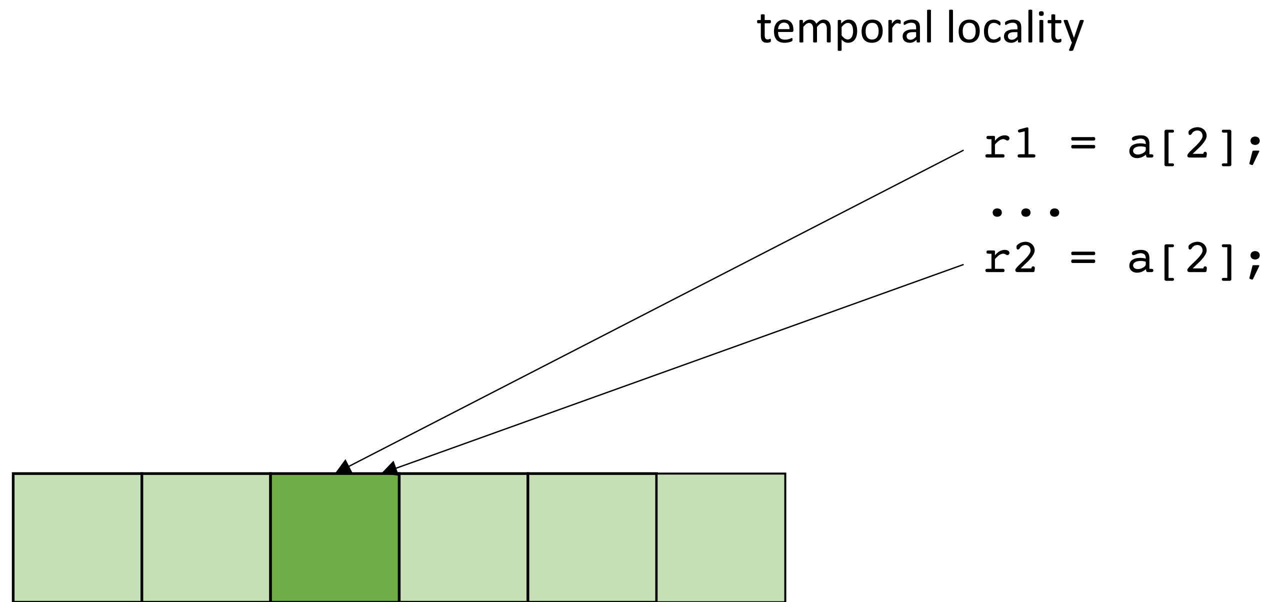
- Locality is key for good (parallel) performance:

- What kind of locality are we talking about?

Transforming Loops

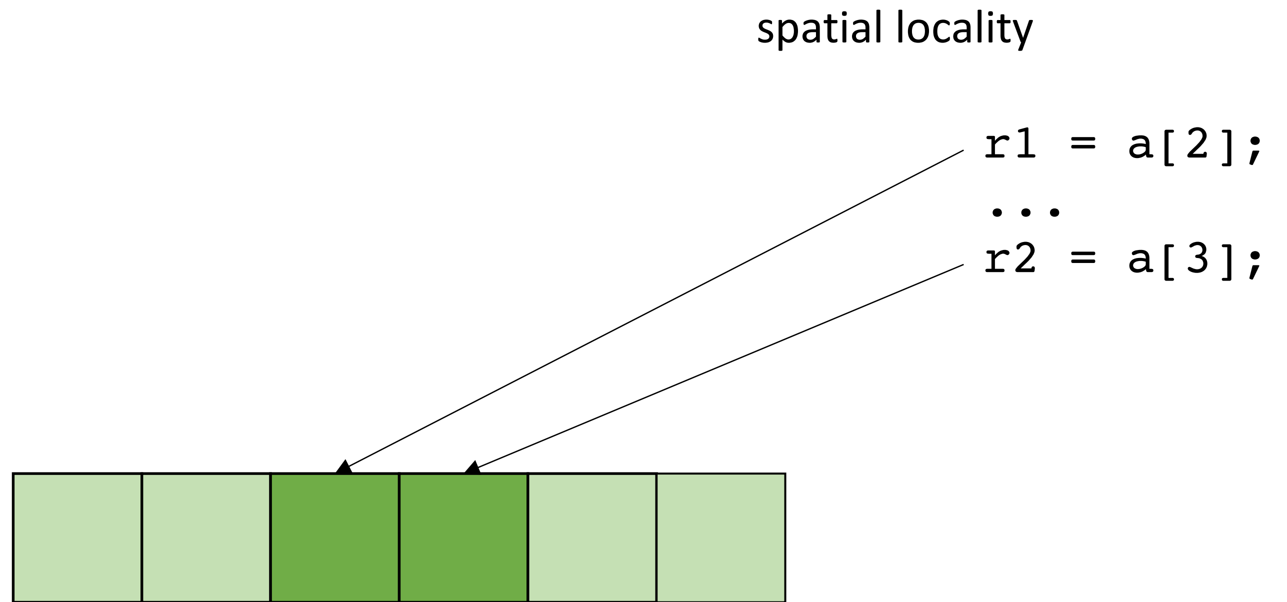
- Locality is key for good parallel performance:

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



Transforming Loops

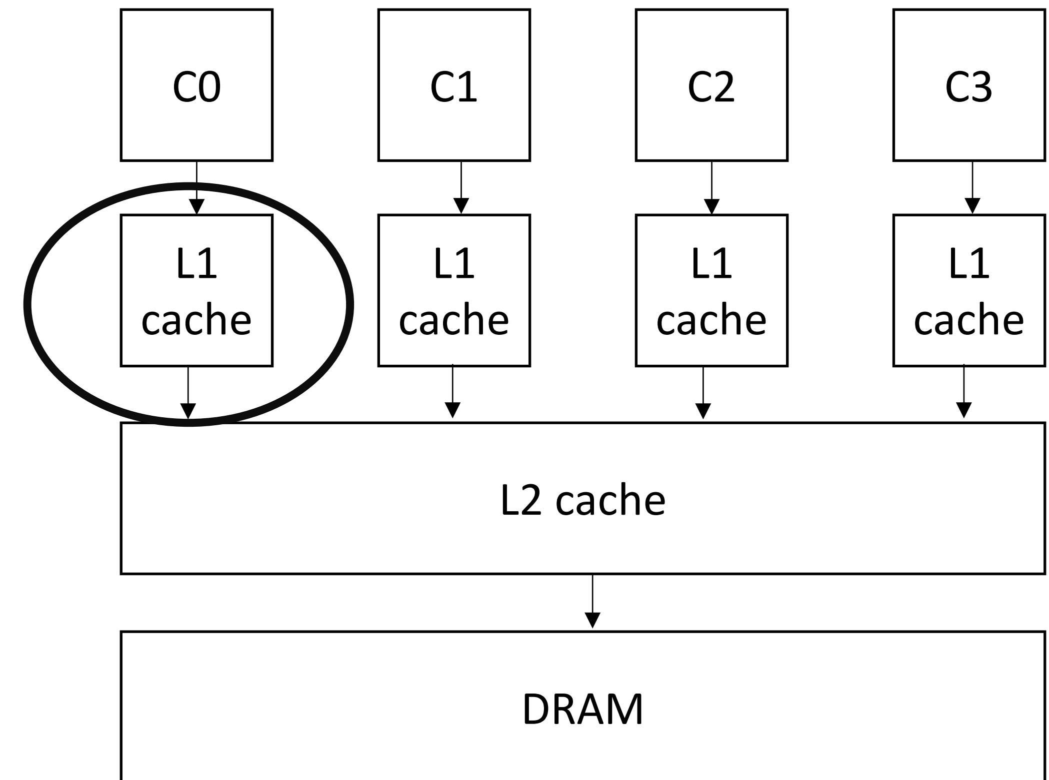
- Locality is key for good parallel performance:
- Two types of locality:
  - Temporal locality
  - Spatial locality



Transforming Loops

- Locality is key for good (parallel) performance:

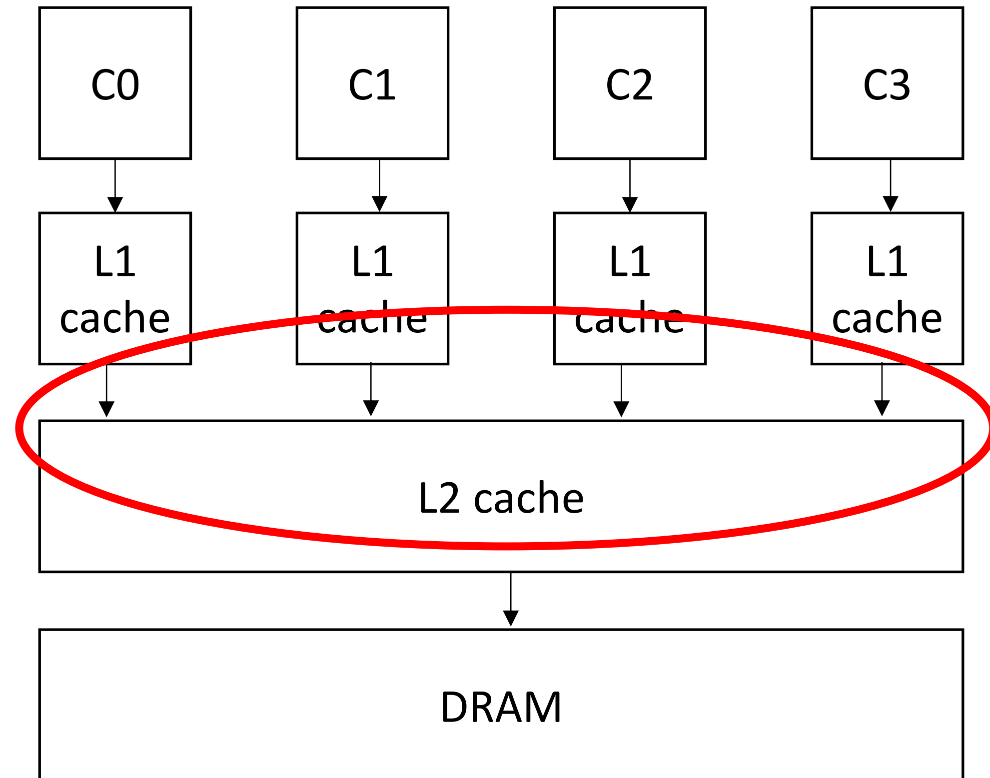
good data locality: cores will
spend most of their time accessing
private caches



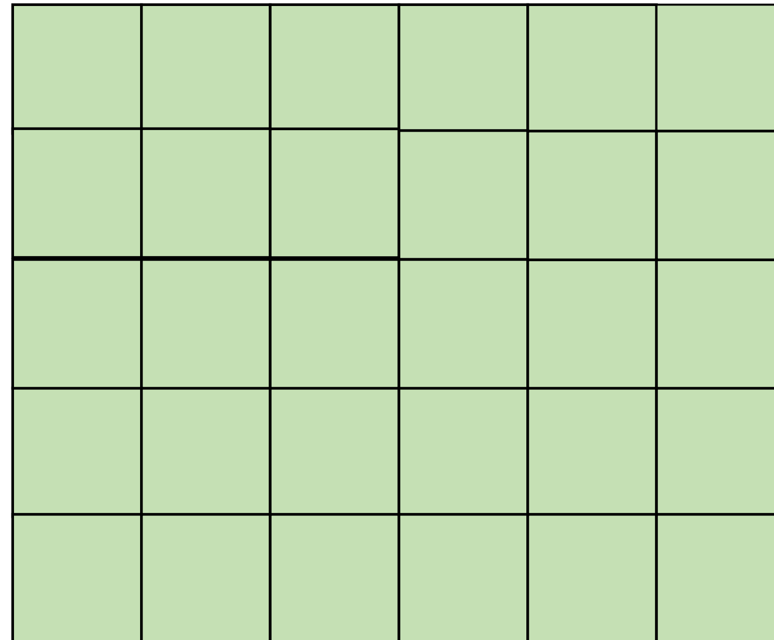
Transforming Loops

- Locality is key for good (parallel) performance:

Bad data locality: cores will pressure and thrash shared memory resources

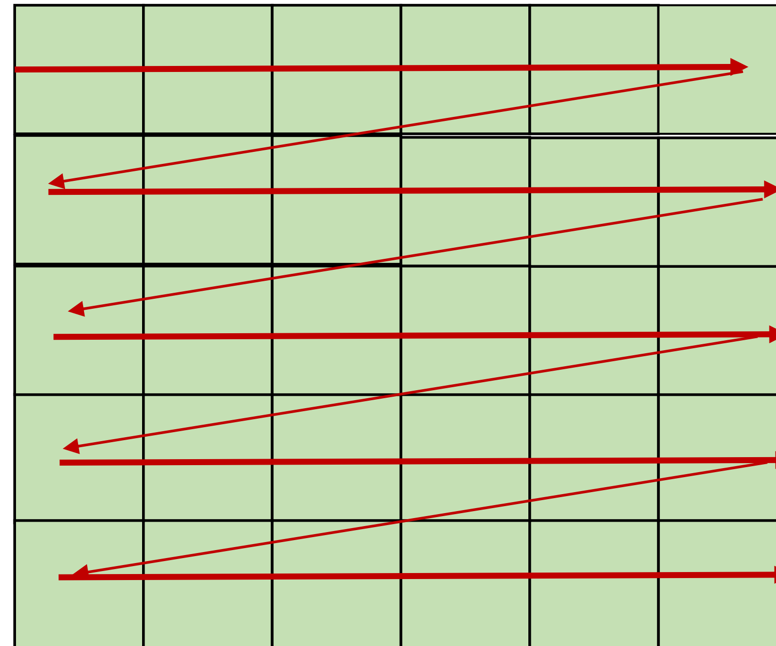


How multi dimensional arrays are stored:



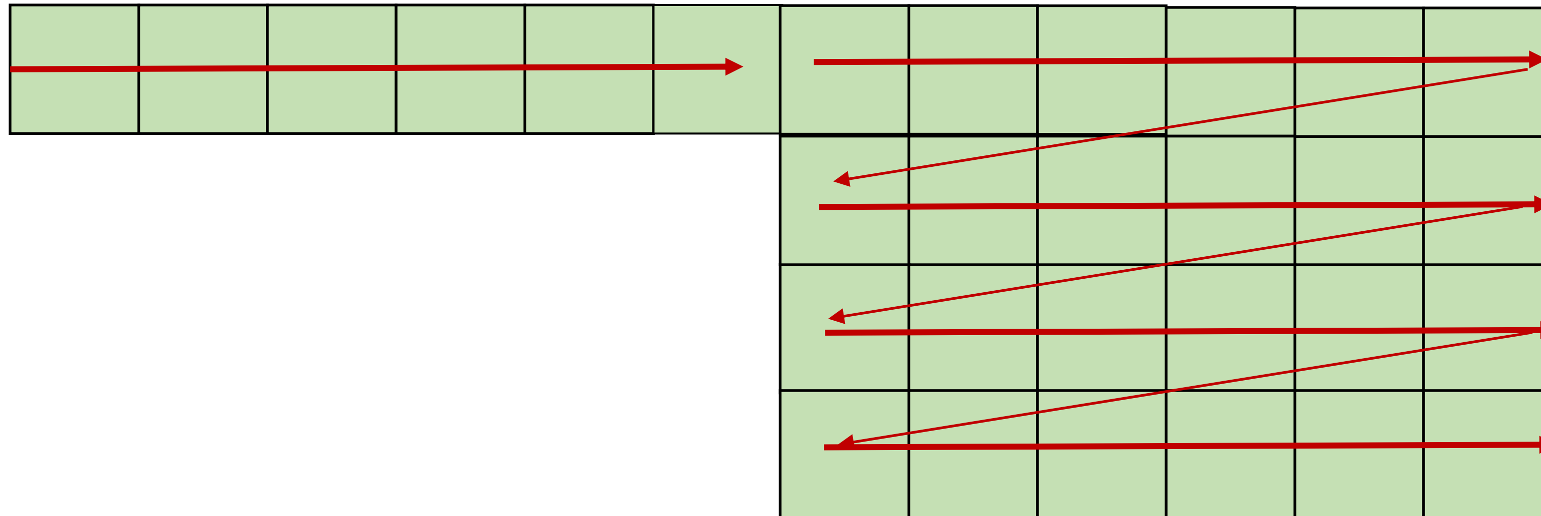
How multi dimensional arrays are stored:

Row major



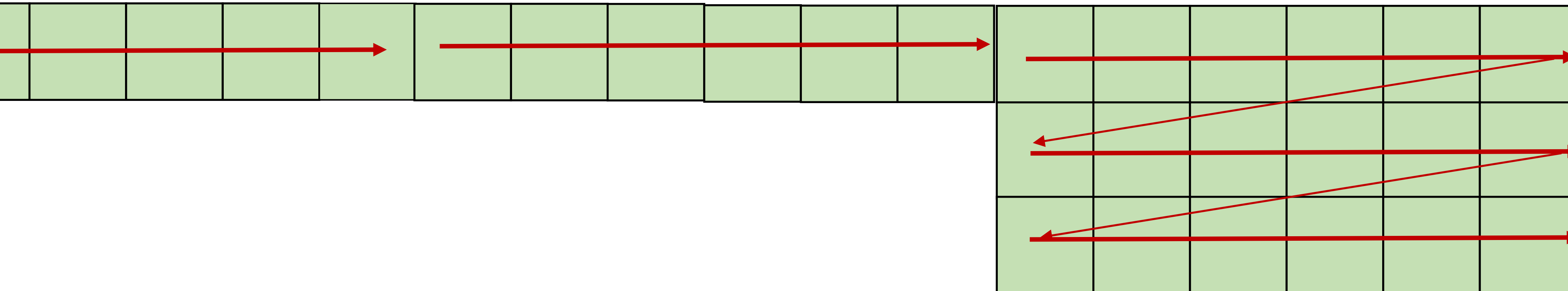
How multi dimensional arrays are stored:

Row major



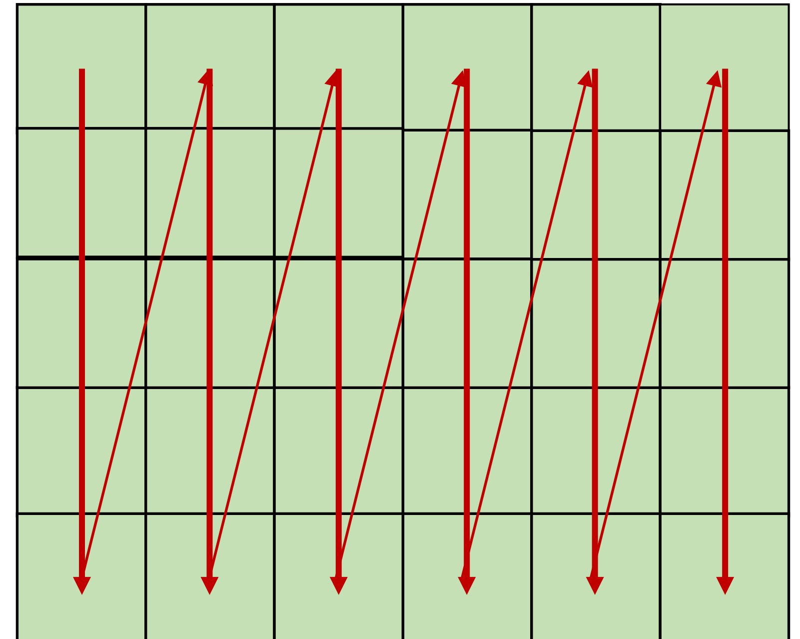
How multi dimensional arrays are stored:

Row major



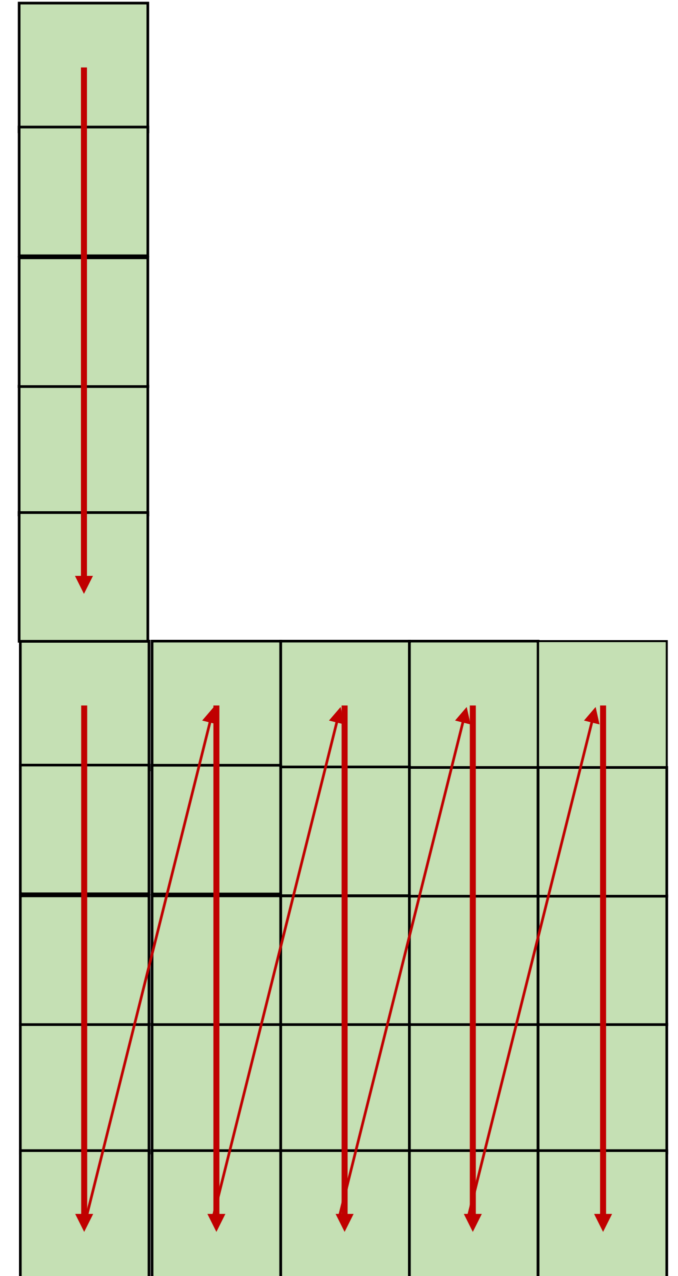
How multi dimensional arrays are stored:

Column major?
Fortran
Matlab
R



How multi dimensional arrays are stored:

Column major?
Fortran
Matlab
R

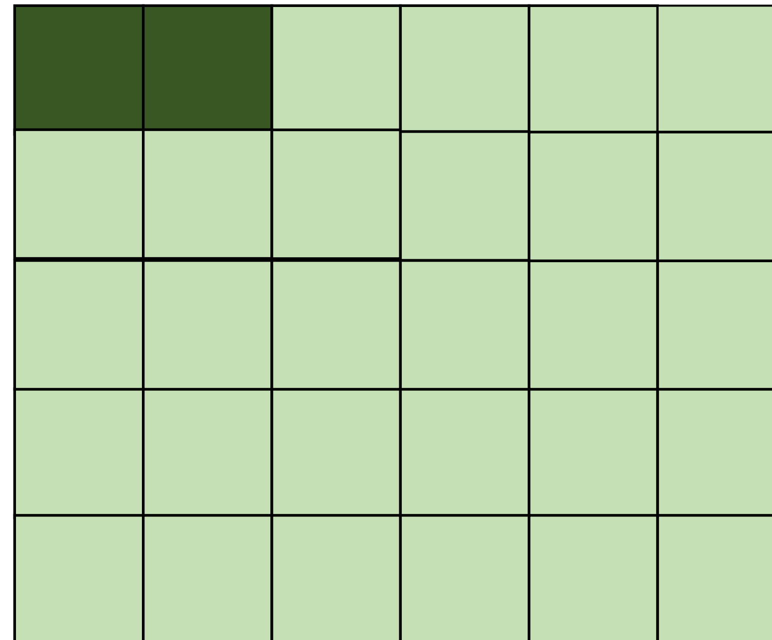


How multi dimensional arrays are stored:

say x == y == 0

```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major



How multi dimensional arrays are stored:

unrolled row major: still has locality

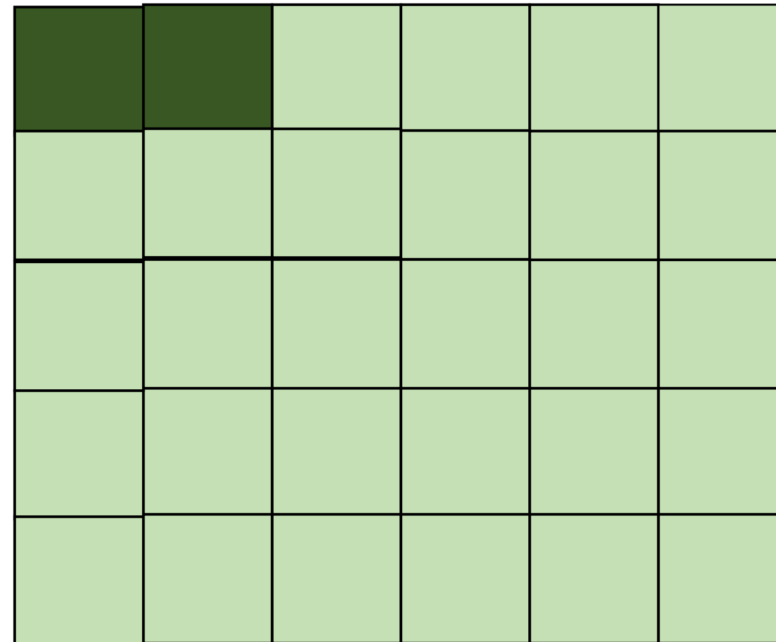
```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major

How multi dimensional arrays are stored:

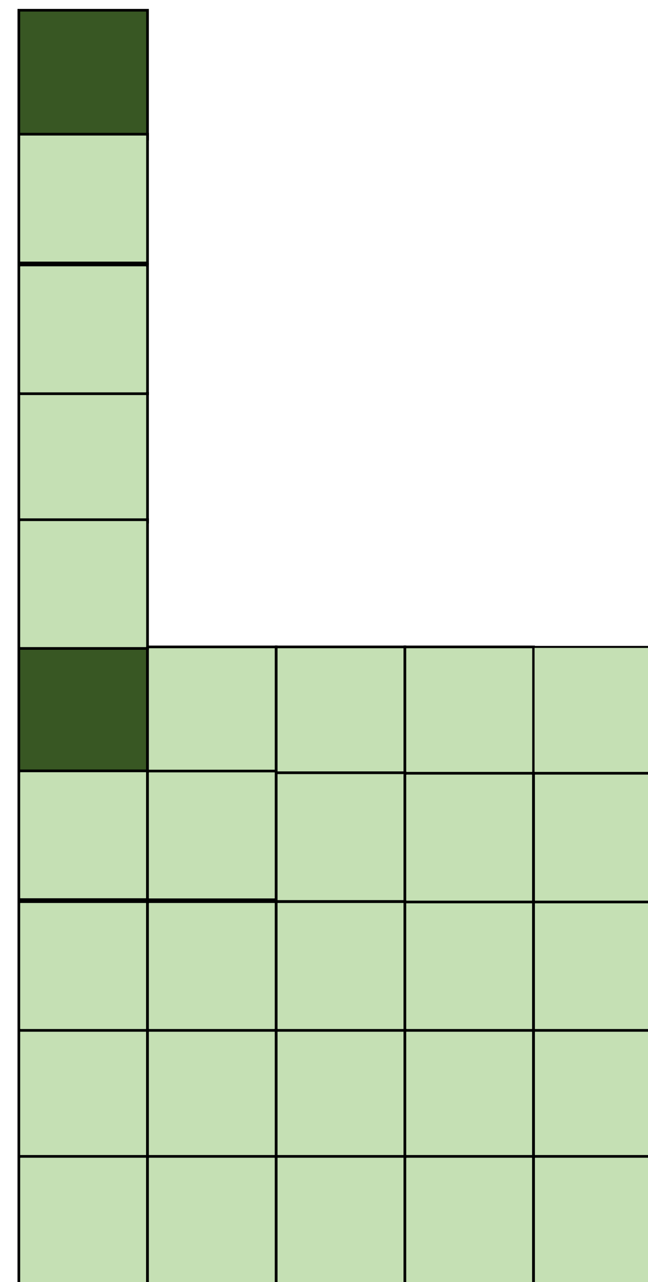
```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major



How multi dimensional arrays are stored:

unrolled
column
major:
Bad locality



```
x1 = a[x,y];
x2 = a[x, y+1];
```

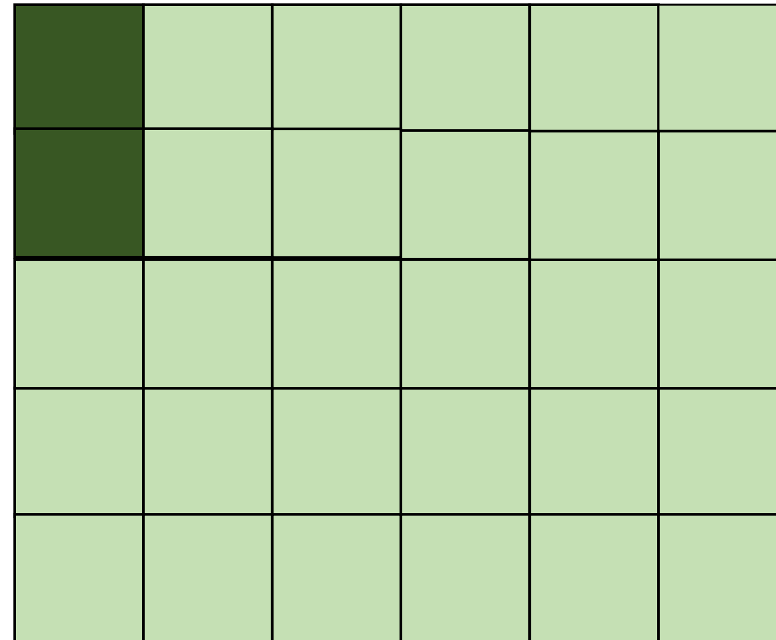
good pattern for row major
bad pattern for column major

How multi dimensional arrays are stored:

say x == y == 0

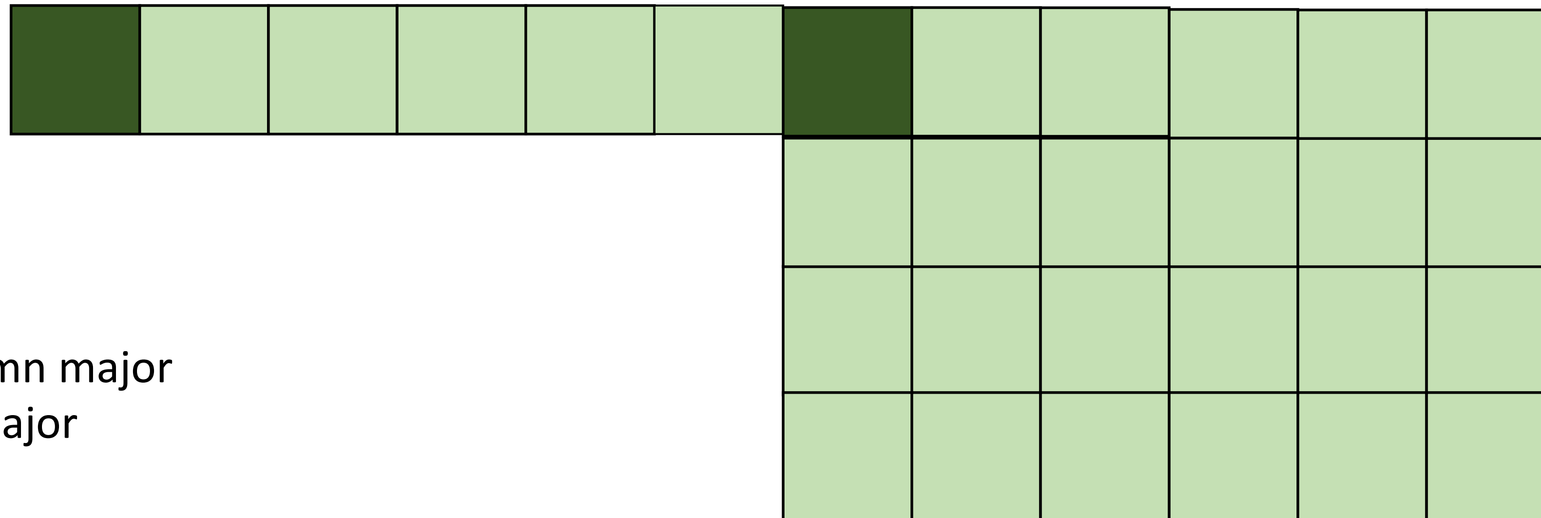
```
x1 = a[x,y];
x2 = a[x+1, y];
```

good pattern for column major
bad pattern for row major



How multi dimensional arrays are stored:

row major unrolled: bad spatial locality

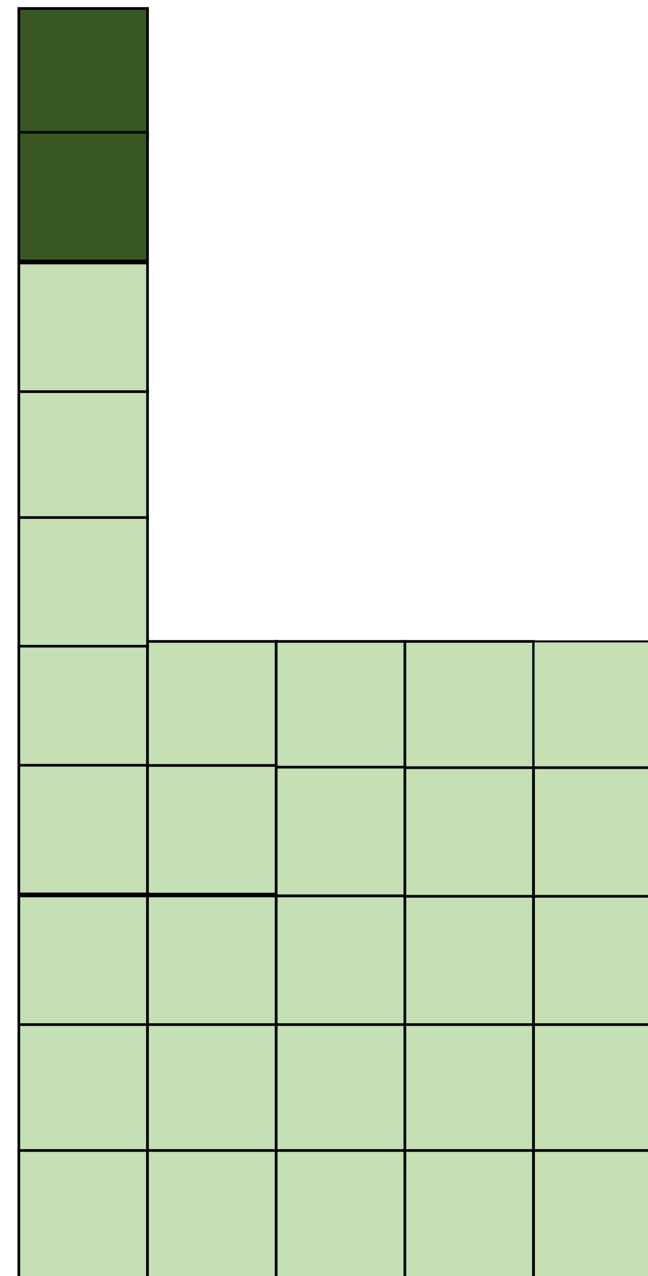


```
x1 = a[x,y];
x2 = a[x+1, y];
```

good pattern for column major
bad pattern for row major

How multi dimensional arrays are stored:

unrolled
column
major:
good locality



```
x1 = a[x,y];
x2 = a[x+1, y];
```

good pattern for column major
bad pattern for row major

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?

Demo

How to reorder loop nestings?

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

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:

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

bad nesting order for row-major!

Example:

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

bad nesting order for row-major!

but iteration variables are dependent

Example:

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

bad nesting order for row-major!

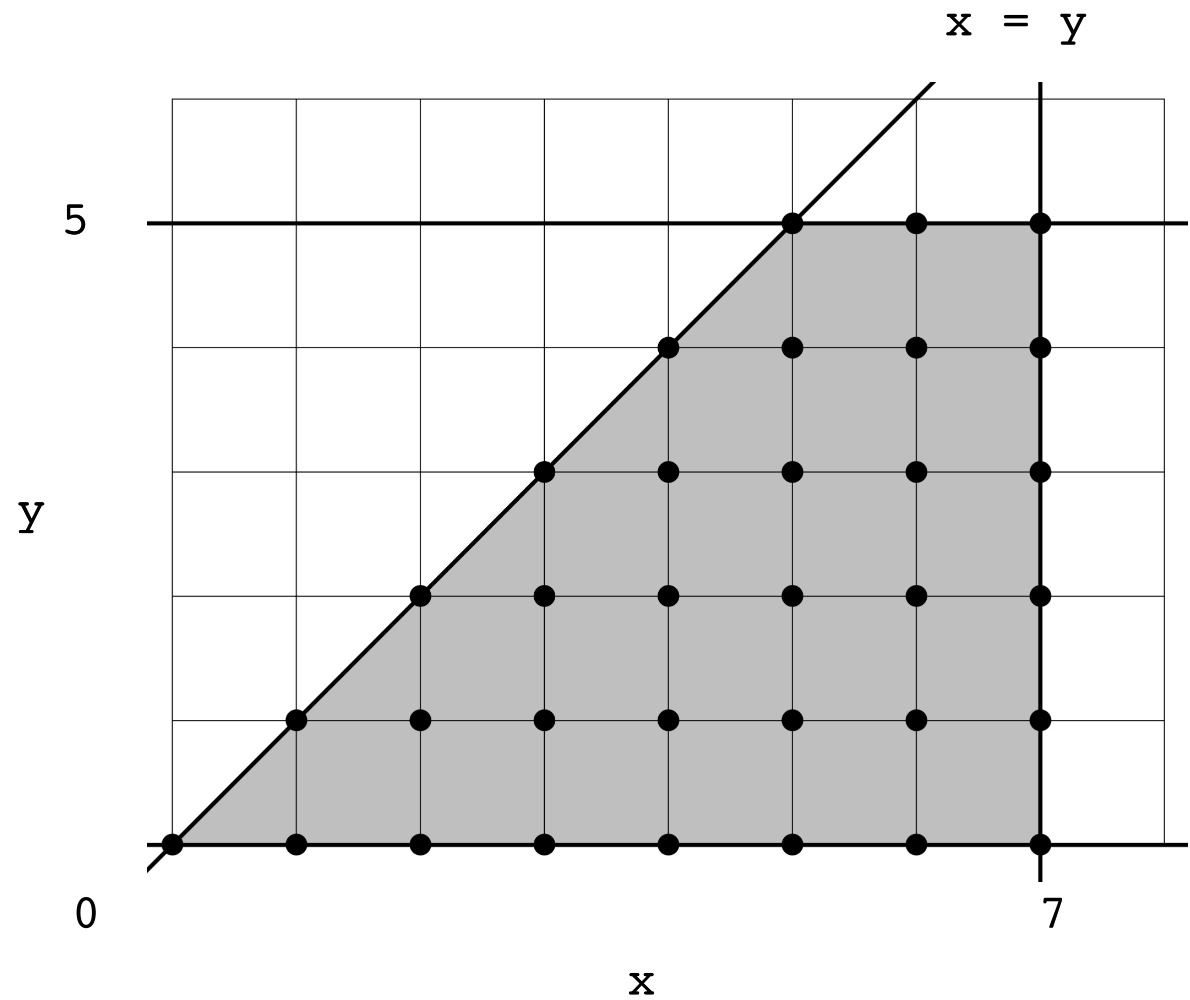
but iteration variables are dependent

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

Example:

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

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



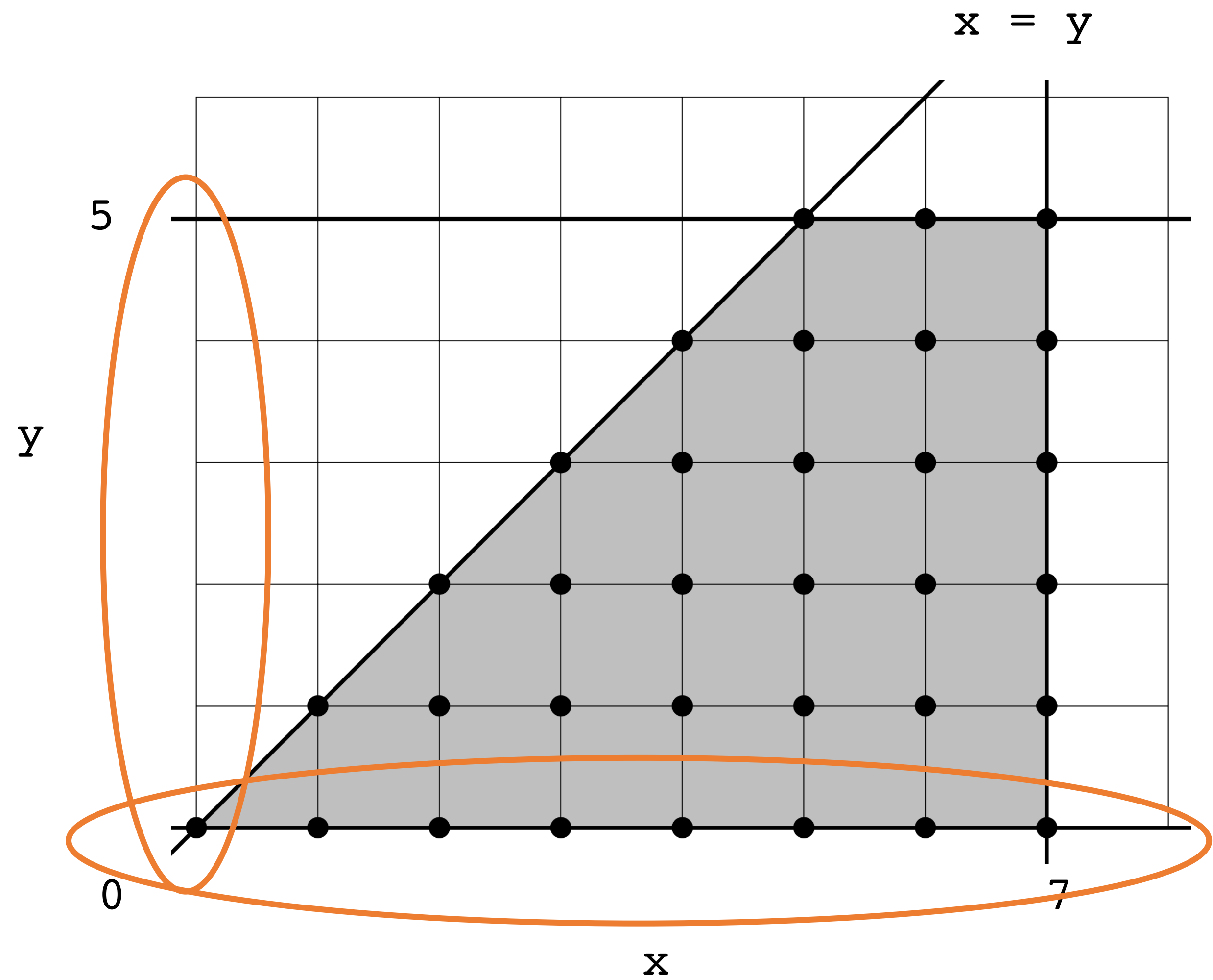
Fourier-Motzkin elimination:

- 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:

```
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:

- To eliminate variable x_i:
  For every pair of lower bound L_i and upper bound U_i on x_i, create:

$$L_i \leq x_i \leq U_i$$

  Then simply remove x_i :

$$L_i \leq U_i$$

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];
  }
}
```

All pairs of upper/lower bounds on y:

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

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
```

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
```

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
```

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 <= x
```

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 <= x
```

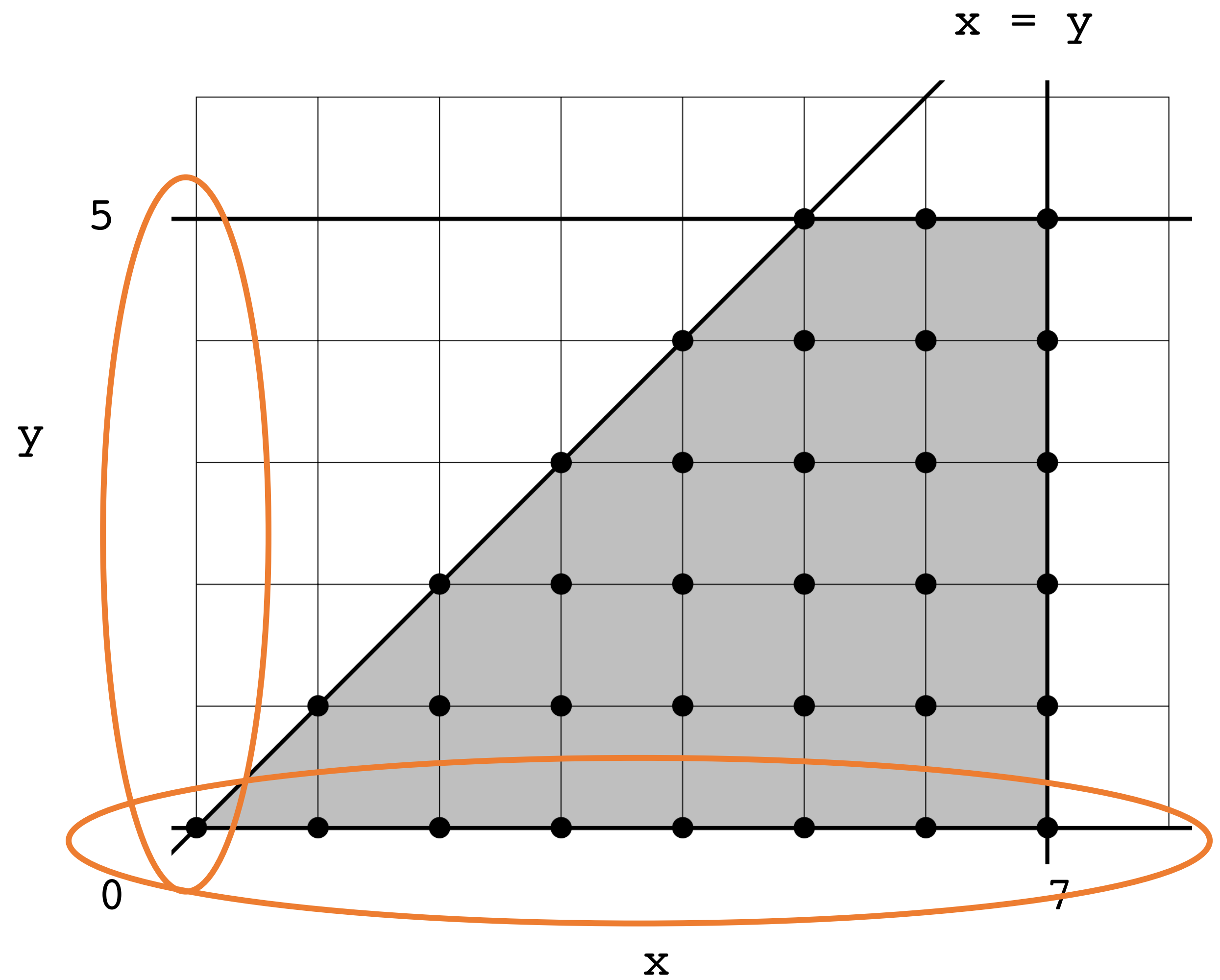
loop constraints without y:

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

Example:

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

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



Reording Loop bounds:

- Given a new order: $[x_0, x_1, x_2, \dots x_n]$

- For each variable x_i : perform Fourier-Motzkin elimination to eliminate any variables that come after x_i in the new order.

- Instantiate loop conditions for x_i, potentially using `max/min` operators

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
```

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:

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
```

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:

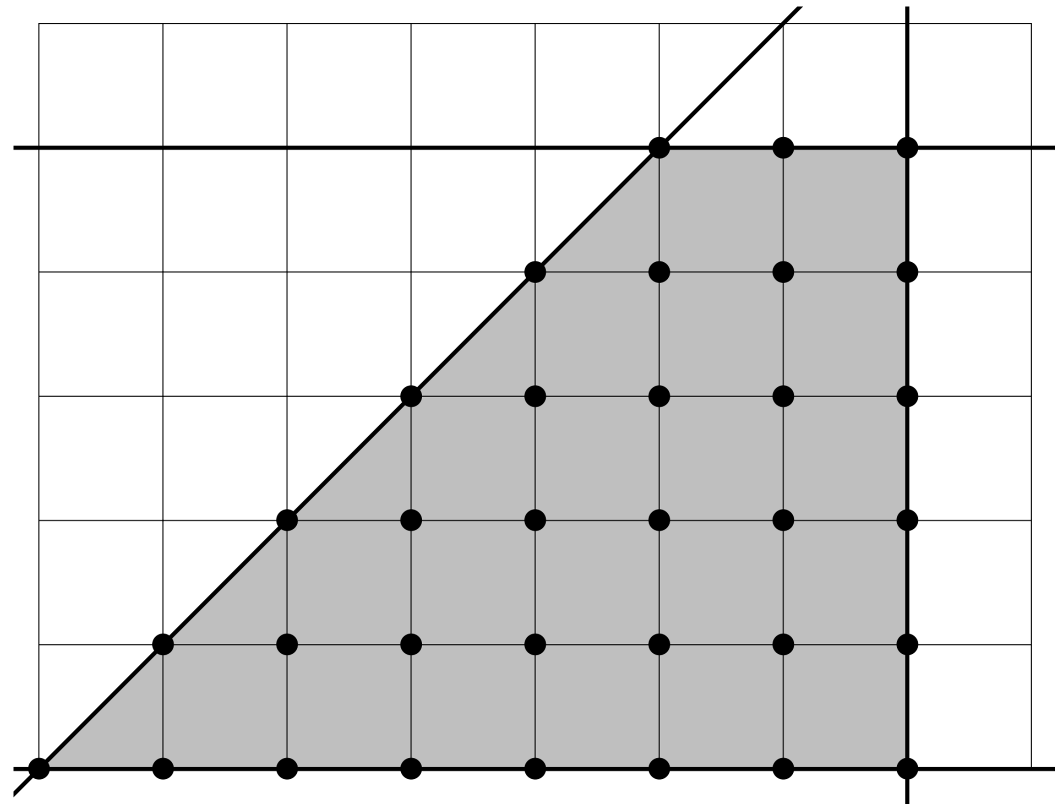
x loop constraints without y:

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

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

Example:

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



x loop constraints without y:

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

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

Reordering loop bounds

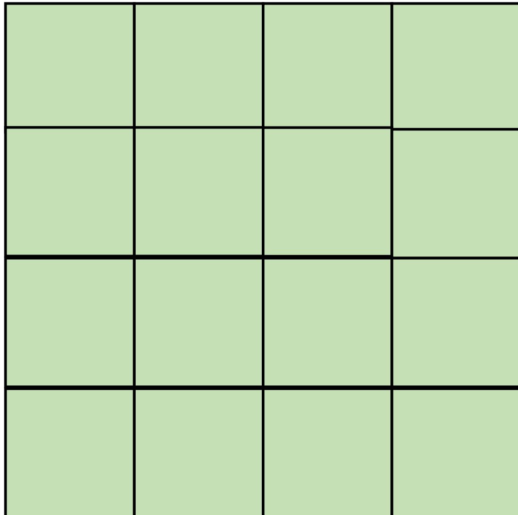
- only works if loop increments by 1; assumes a closed polyhedron
- best performance when array indexes are simple:
  - e.g.: `a[x,y]`
  - harder with, e.g.: `a[x*5+127, y+x*37]`
  - There exists schemes to automatically detect locality. Reach chapter 10 of the Dragon book
- compiler implementation allows exploration and auto-tuning

Adding loop nestings

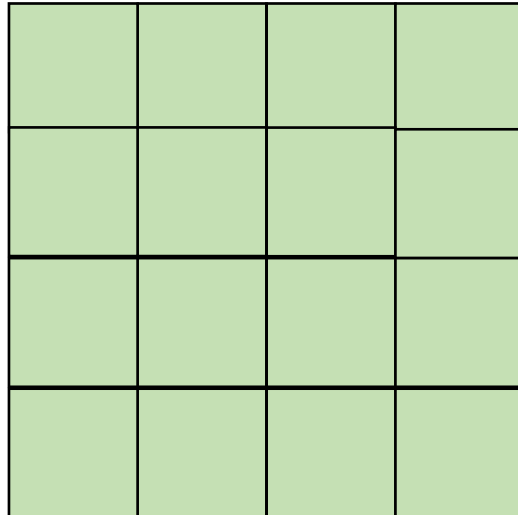
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

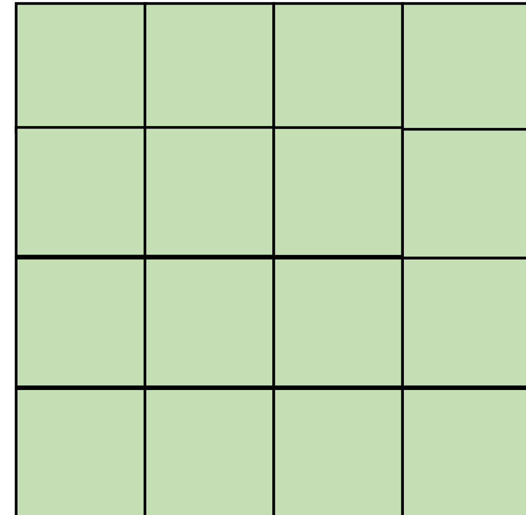
A



B



C

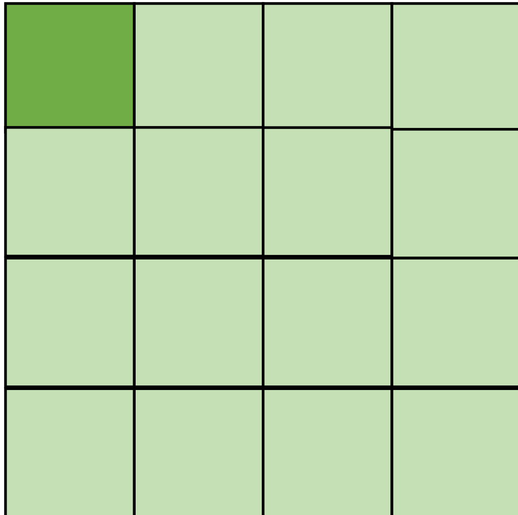


Adding loop nestings

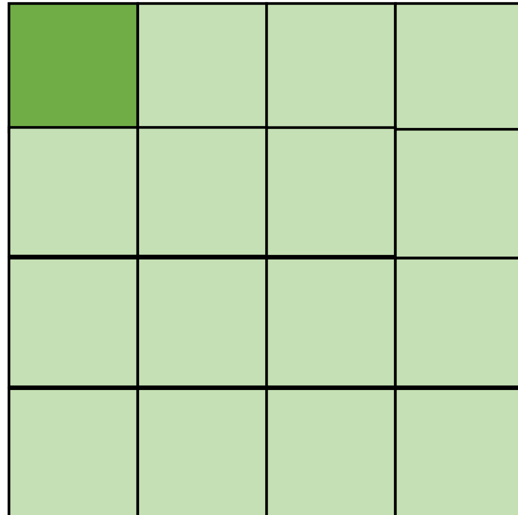
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

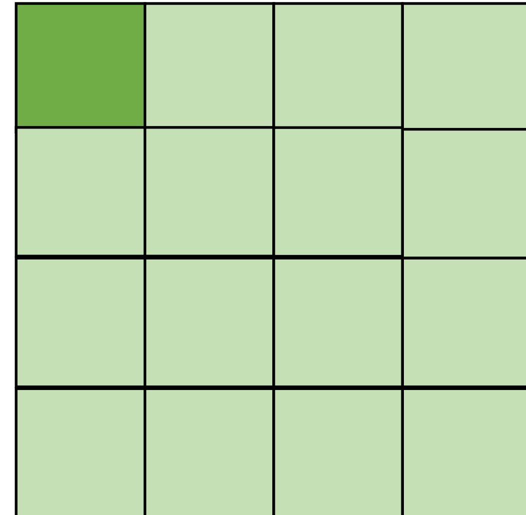
A



B



C



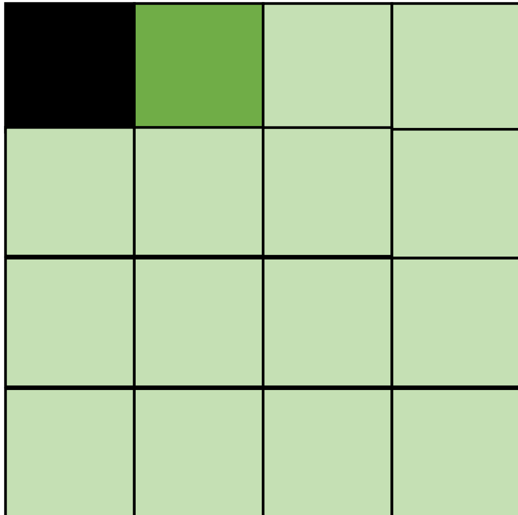
cold miss for all of them

Adding loop nestings

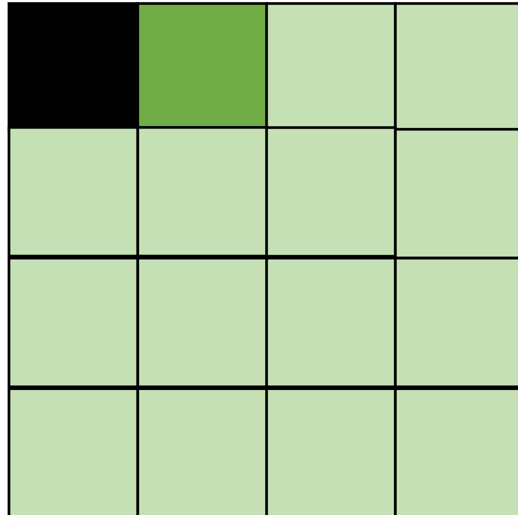
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

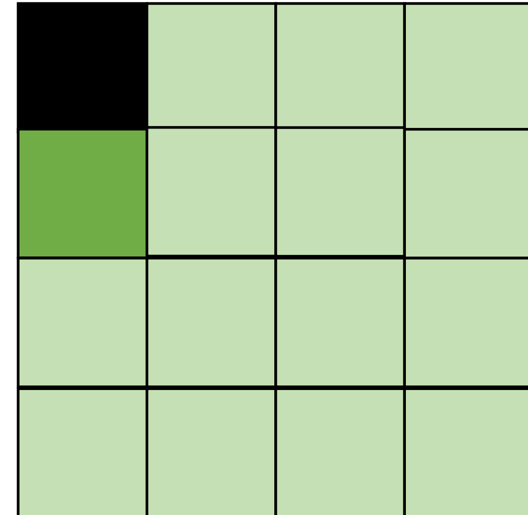
A



B



C



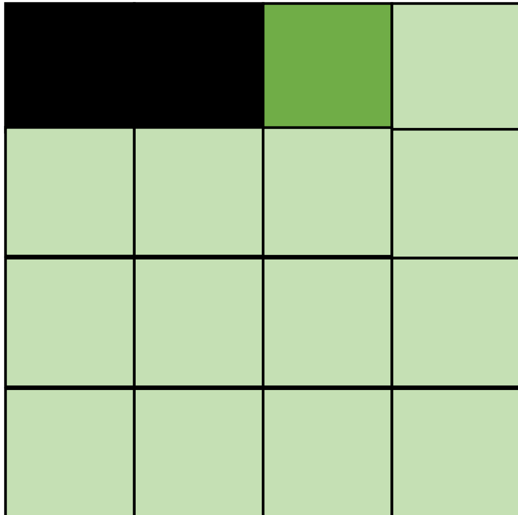
Hit on A and B. Miss on C

Adding loop nestings

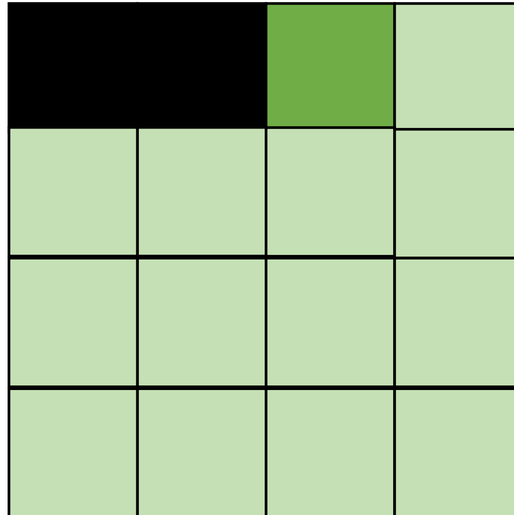
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

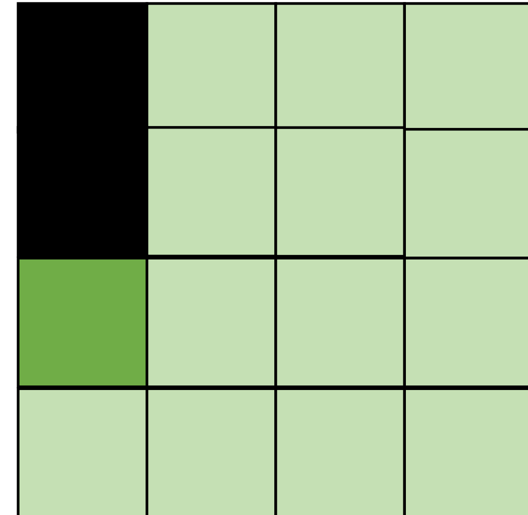
A



B



C



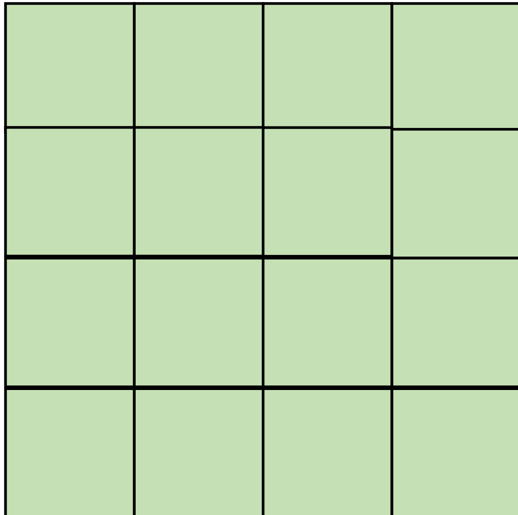
Hit on A and B. Miss on C

Adding loop nestings

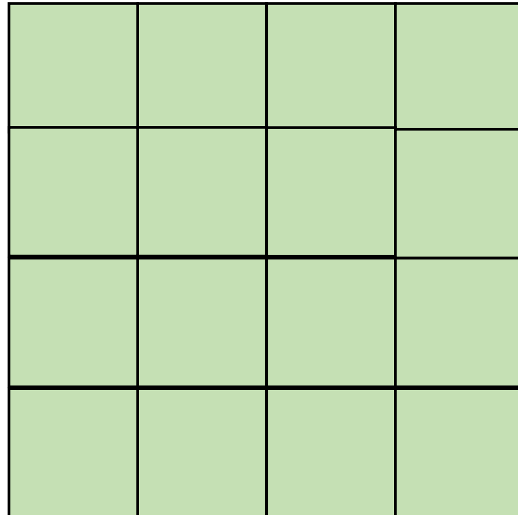
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

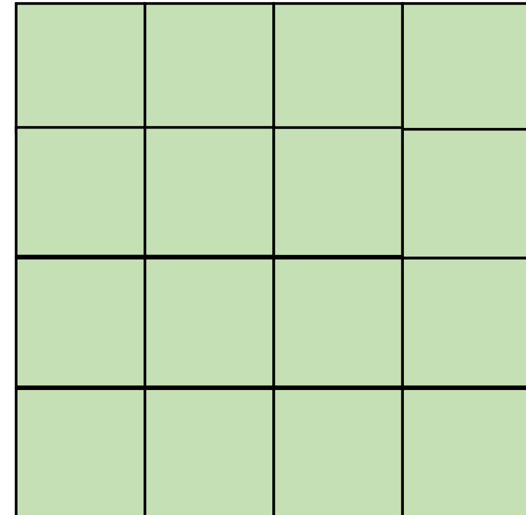
A



B



C

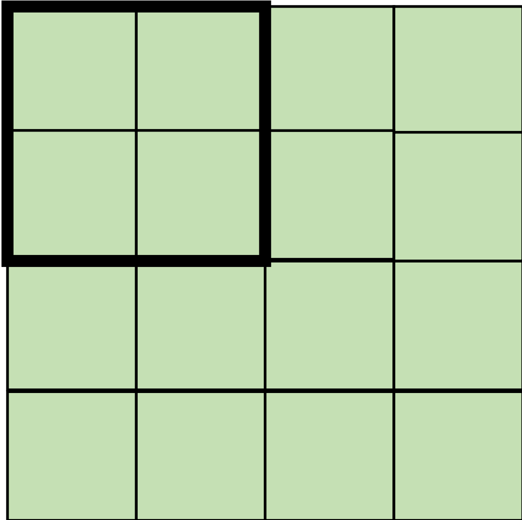


Adding loop nestings

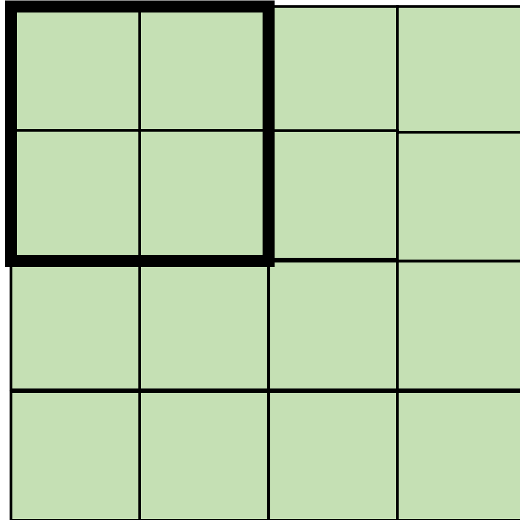
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

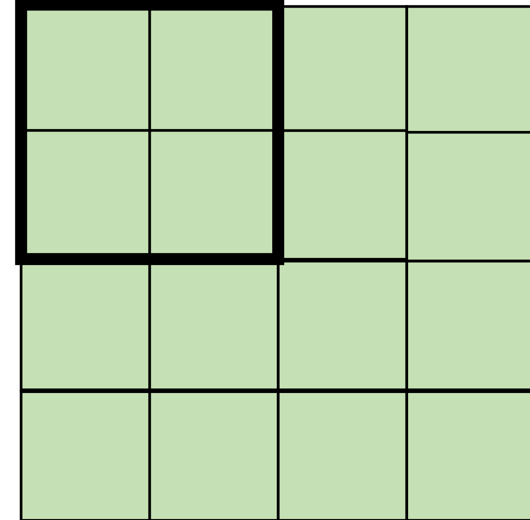
A



B



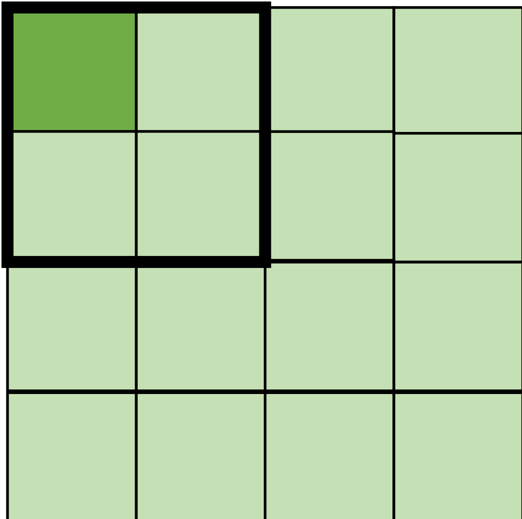
C

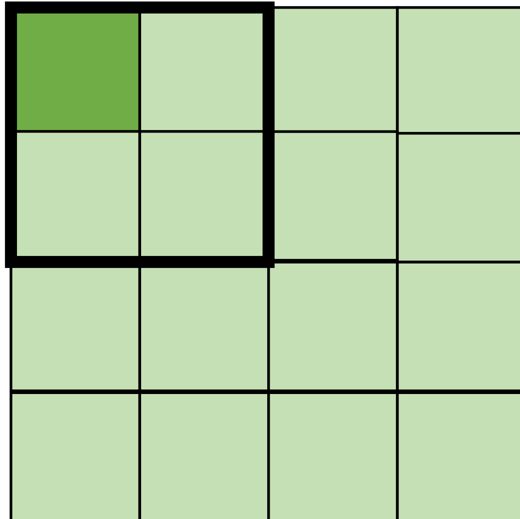


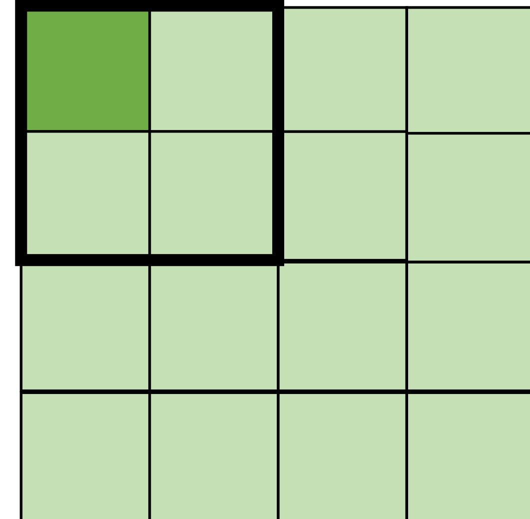
Adding loop nestings

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

$$A = B + C^T$$





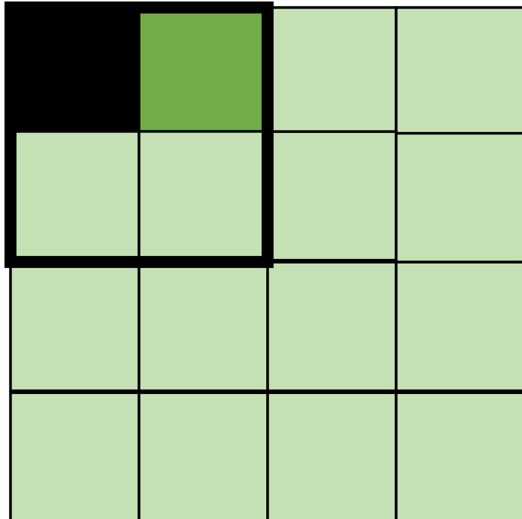


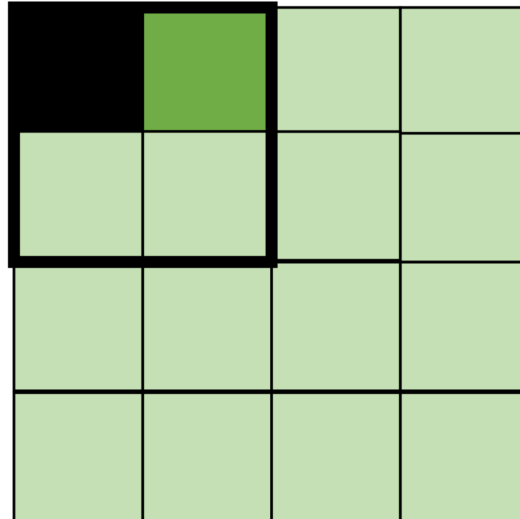
cold miss for all of them

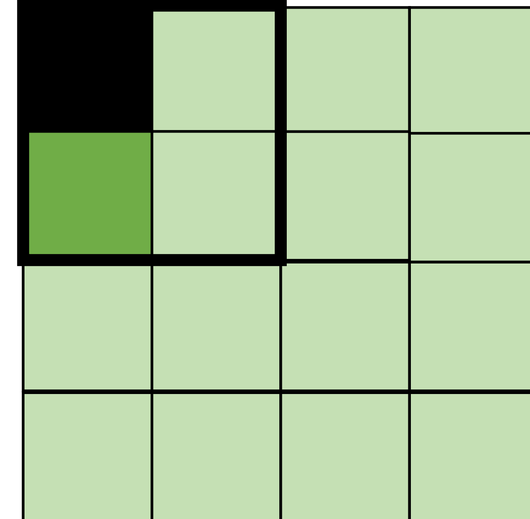
Adding loop nestings

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

$$A = B + C^T$$







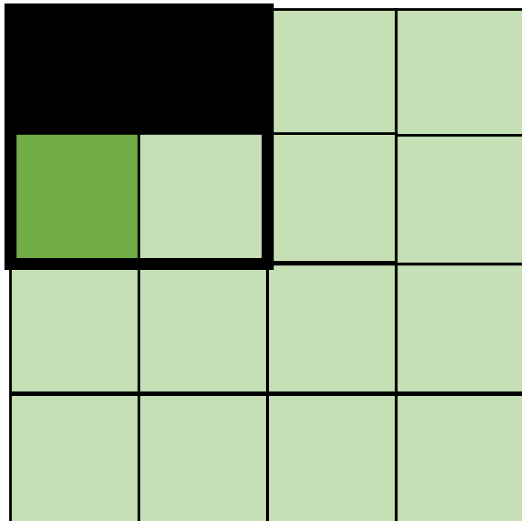
Miss on C

Adding loop nestings

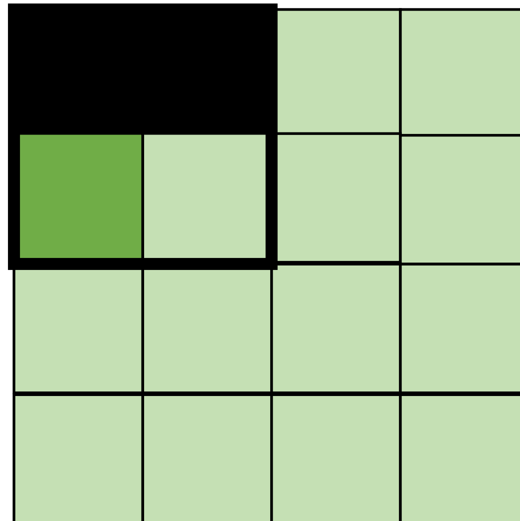
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

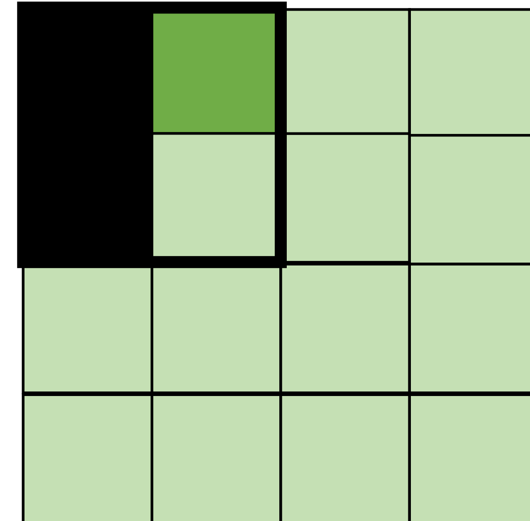
A



B



C



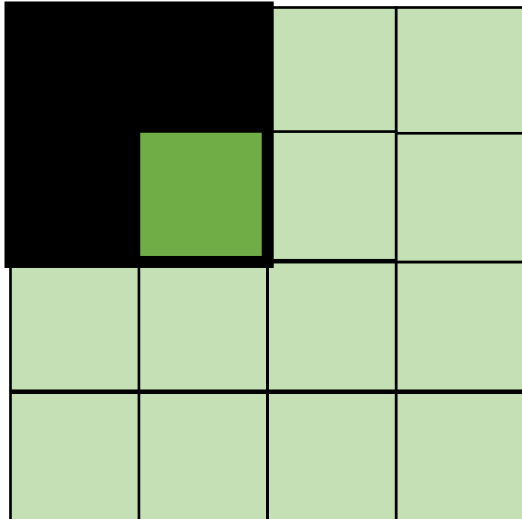
Miss on A,B, hit on C

Adding loop nestings

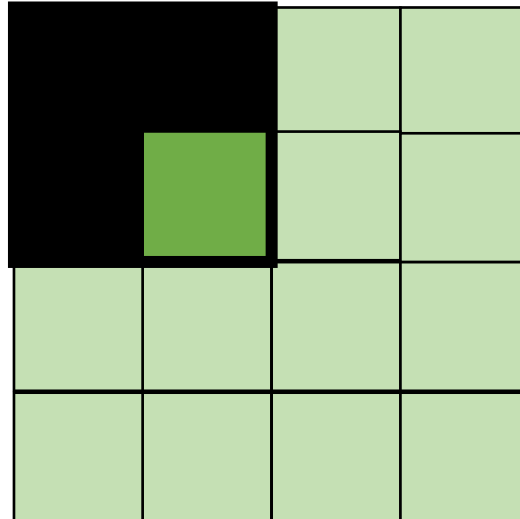
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

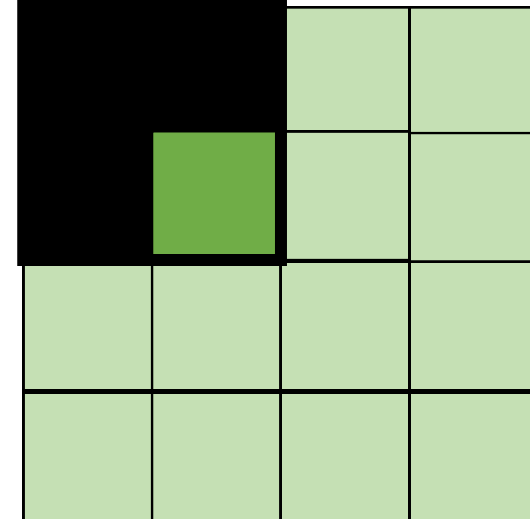
A



B



C



Hit on all!

Adding loop nestings

- Add two outer loops for both x and y

```
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];
    }
  }
```

Adding loop nestings

- Add two outer loops for both x and y

```
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];
      }
    }
  }
}
```

Adding loop nestings

- Add two outer loops for both x and y

```
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];
      }
    }
  }
}
```

Adding loop nestings

- Add two outer loops for both x and y

```
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];
      }
    }
  }
}
```

Adding loop nestings

- Add two outer loops for both x and y

```
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];
      }
    }
  }
}
```

Demo

Recap what we’ve covered with loops

- Are the loop iterations independent?
  - The property holding all of these optimizations together
- mainstream compilers don’t do much to help us out here
  - why not?
- But DSLs can!

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

DSL designs

- Ease of expressiveness

```
sed ‘s/Utah/California’ address.txt
```

gnuplot

```
set title ”Parallel timing experiments"
set xlabel ”Threads”
set ylabel ”Speedup"
plot ”data.dat" with lines
```

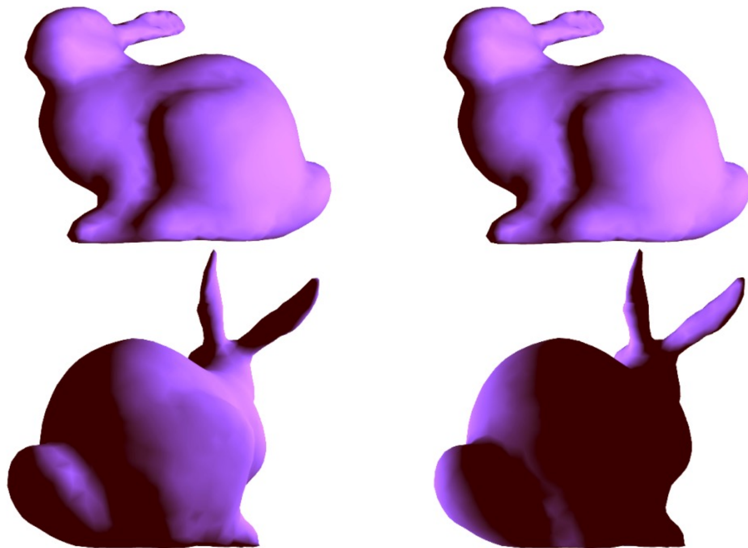
Other examples?

These require their own front end. What about Matplotlib?

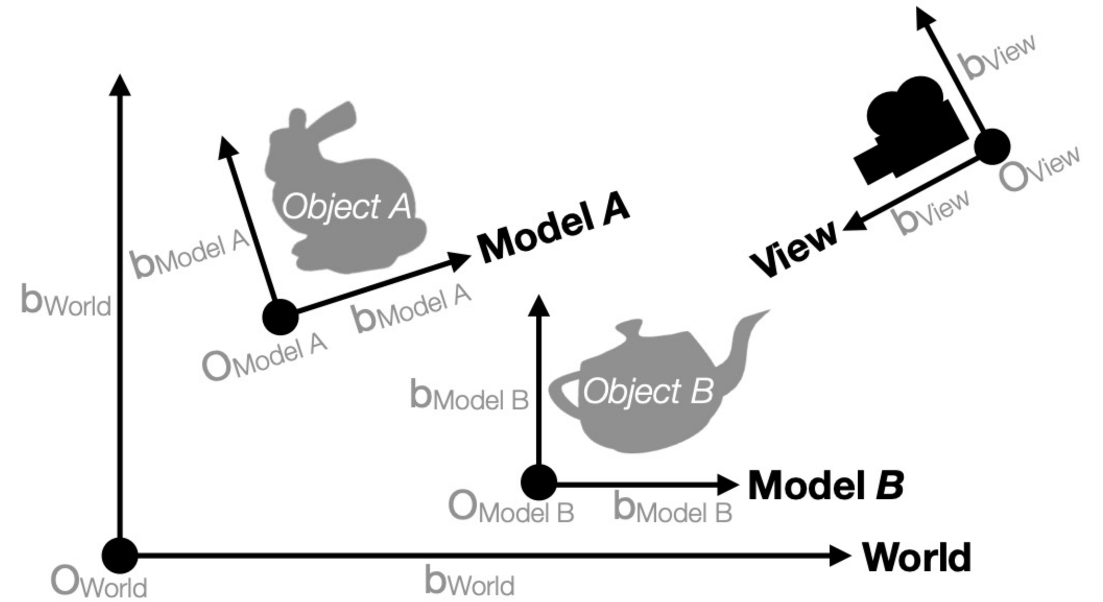
DSL designs

- Ease of expressiveness

make it harder to write bugs!



(a) Correct implementation. (b) With geometry bug.



Add reference tags to types: World or View

From: Geometry Types for Graphics Programming, OOPSLA 2020

DSL designs

- Ease of optimizations

Examples?

From homework 3:

What does this assume?
Optional in C++
Non-optional in Tensorflow

- reduction loops:
  - Entire computation is dependent
  - Typically short bodies (addition, multiplication, max, min)

1	2	3	4	5	6

addition: 21

max: 6

min: 1

Typically faster than implementations in general languages.

DSL designs

- Easier to reason about

Typically much fewer lines of code than implementations in general languages.

gnuplot example again

```
set title ”Parallel timing experiments"
set xlabel ”Threads”
set ylabel ”Speedup"
plot ”data.dat" with lines
```

tensorflow

```
tf.matmul(a, b)
```

What does an optimized matrix multiplication look like?

https://github.com/flame/blis/tree/master/kernels

DSL designs

- Easier to maintain

- *Optimizations and transforms are less general (more targeted).*
- *Less syntax (sometimes no syntax).*
- *Fewer corner cases.*

DSL design

- Recipe for a DSL talk:
  - Introduce your domain
  - Show scary looking optimized code
  - Show clean DLS code
  - Show performance improvement
  - Have a correctness argument

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)

Pros and Cons?

program

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

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)

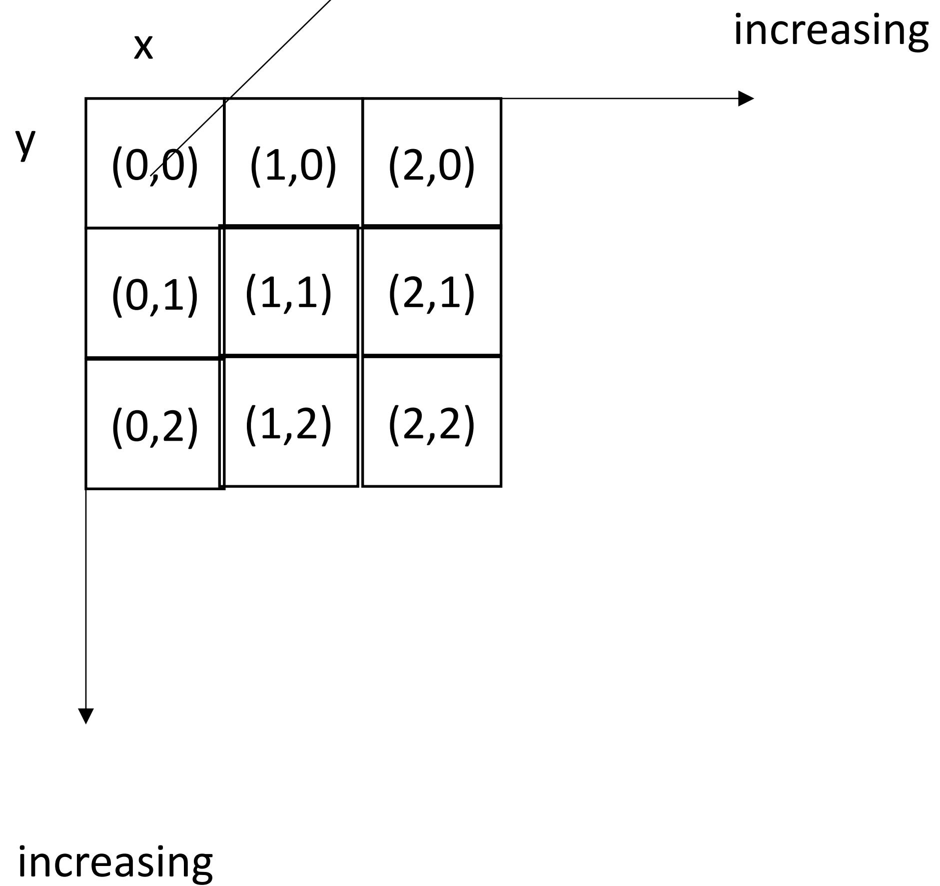
Halide programs

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

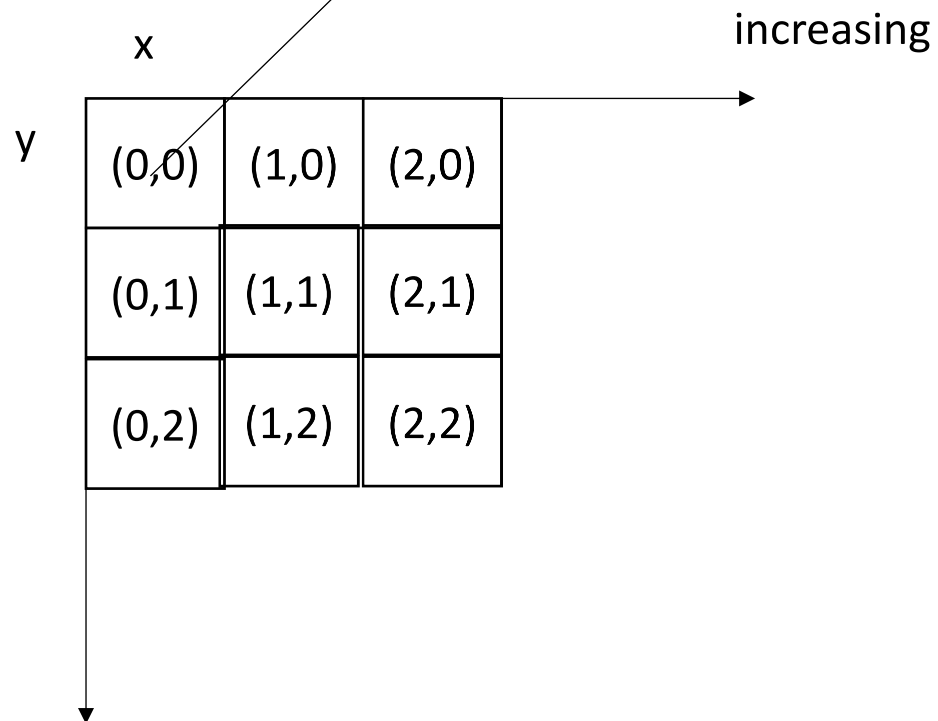
```
#include "Halide.h"

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;
```

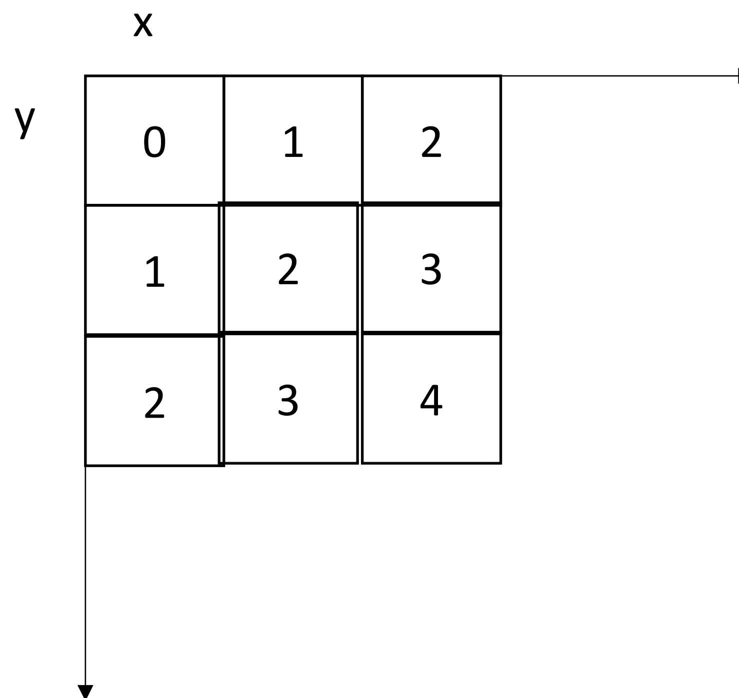


```
gradient(x, y) = x + y;
```



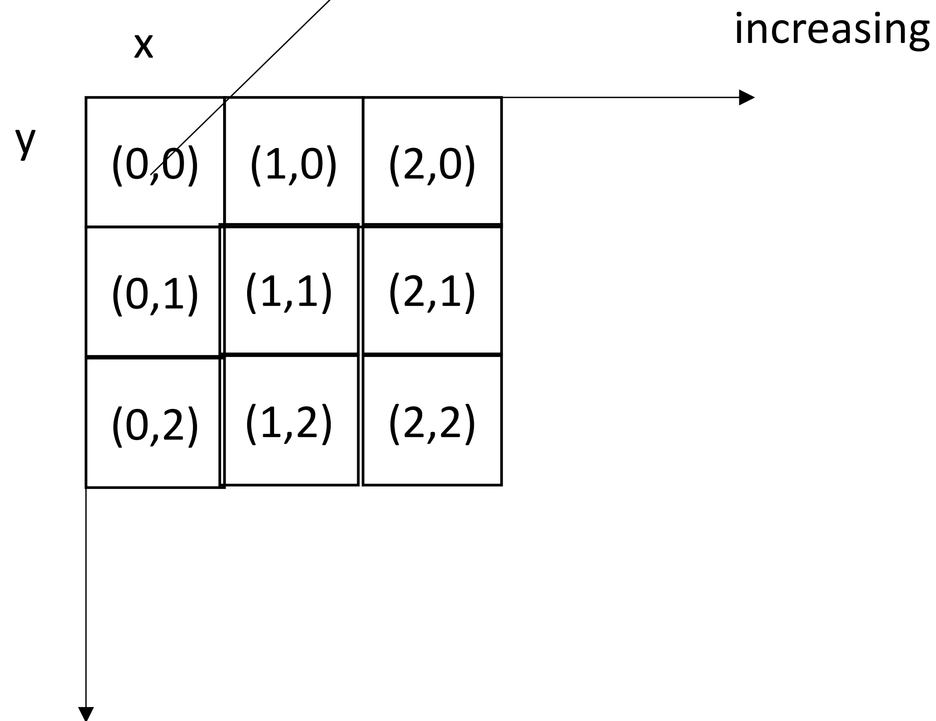
increasing

after applying the gradient function

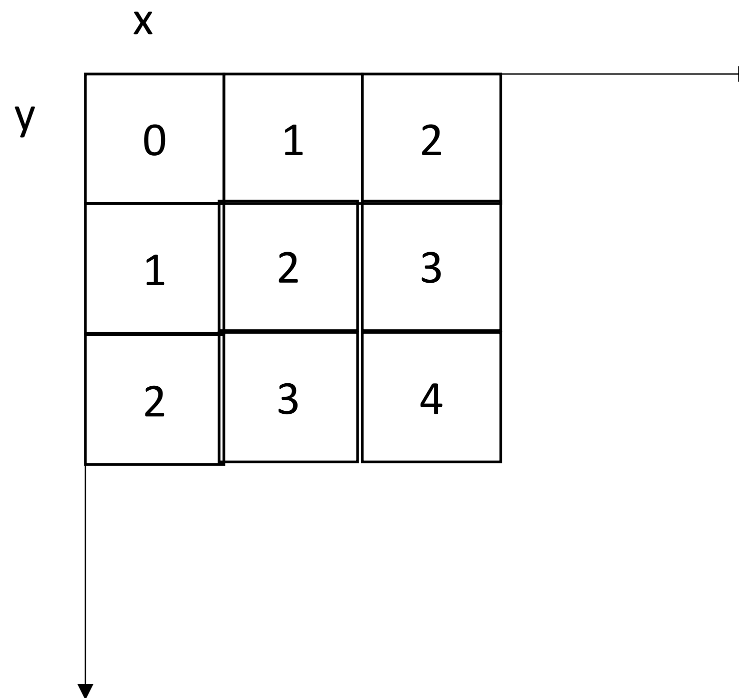


what are some properties of this computation?

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



after applying the gradient function



what are some properties of this computation?

Data races?

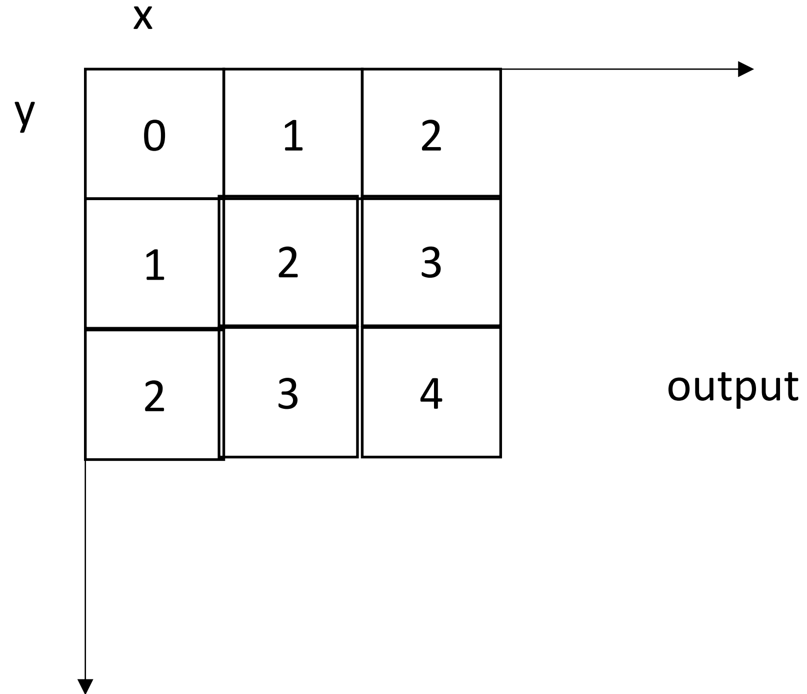
Loop indices and increments?

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



output

Example: brightening

Brighten example

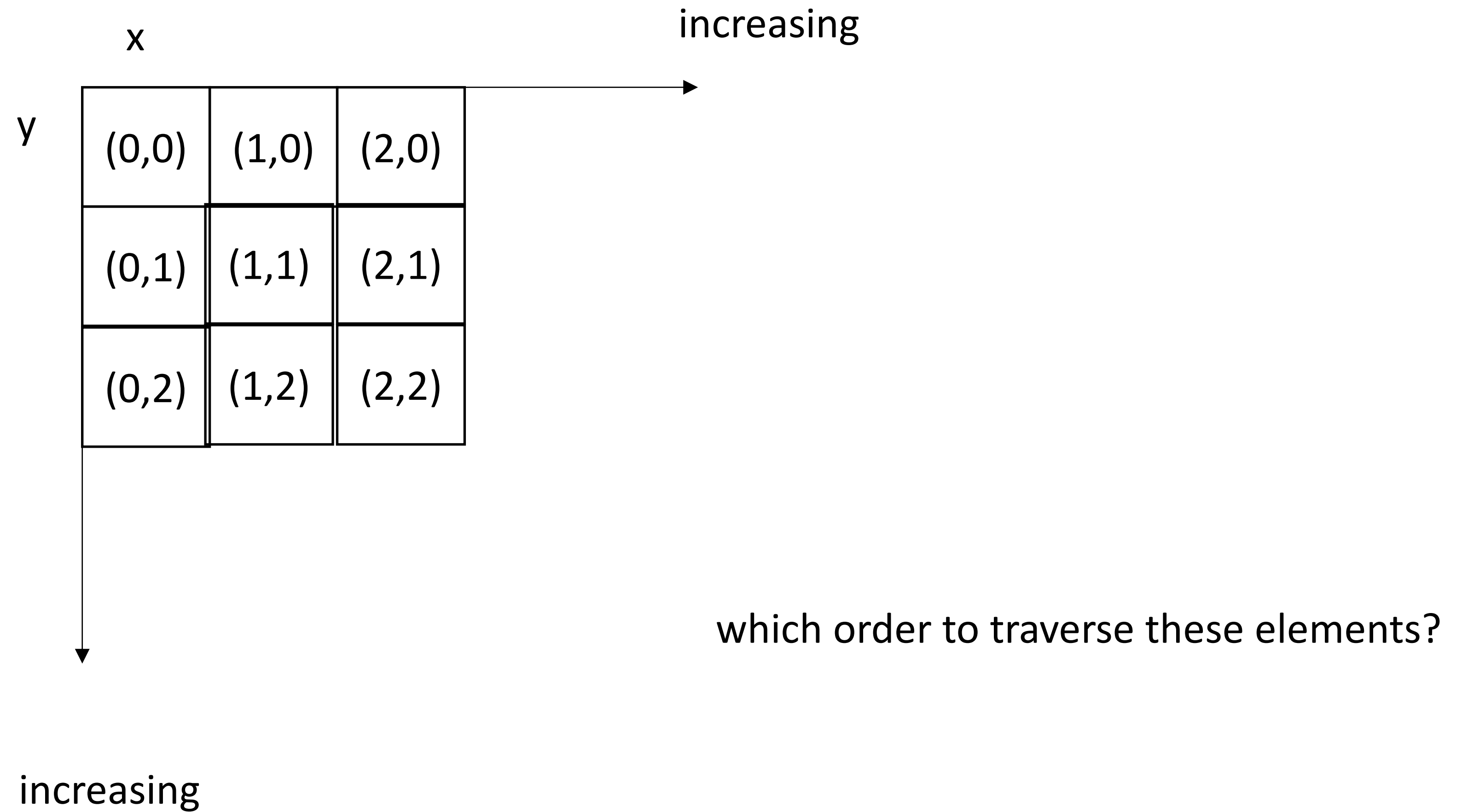
```
Halide::Buffer<uint8_t> input = load_image("parrot.png");

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});
```

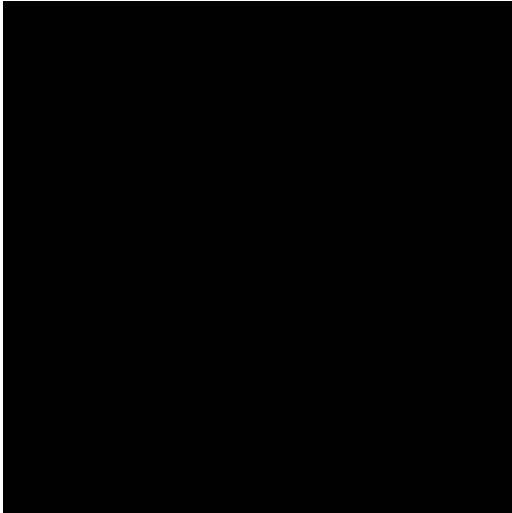


```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({4, 4});
```

```
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);
```

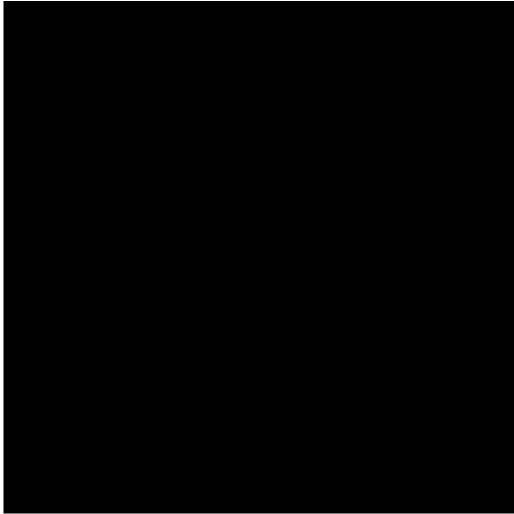
```
for (int x = 0; x < 4; x++) {
    for (int y = 0; y < 4; y++) {
        output[y,x] = x + y;
    }
}
```

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

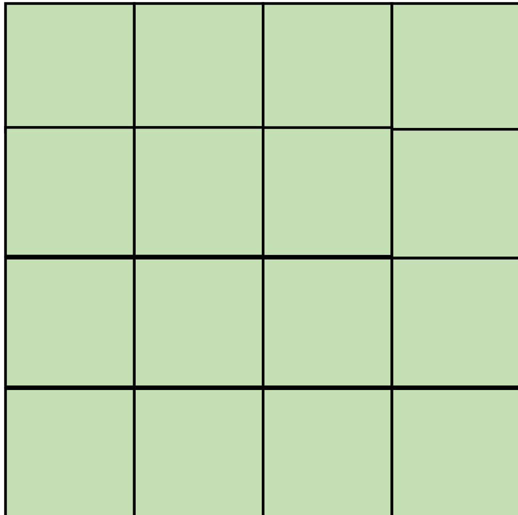
Tiling

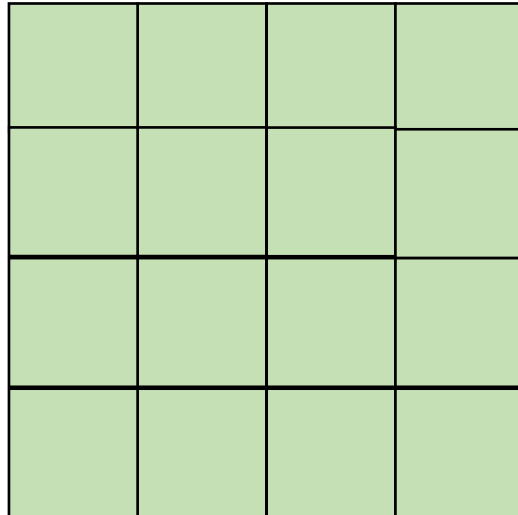
Adding loop nestings

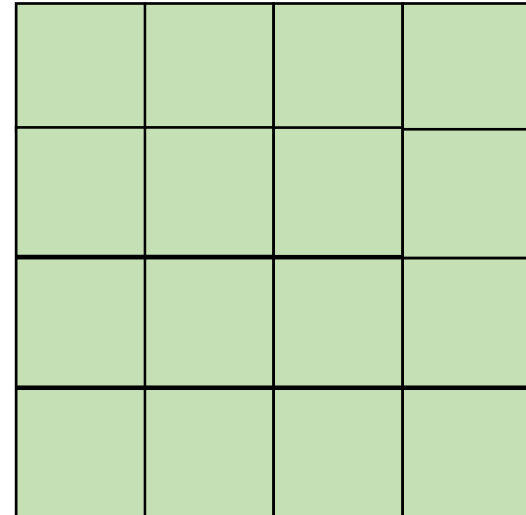
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

A B C



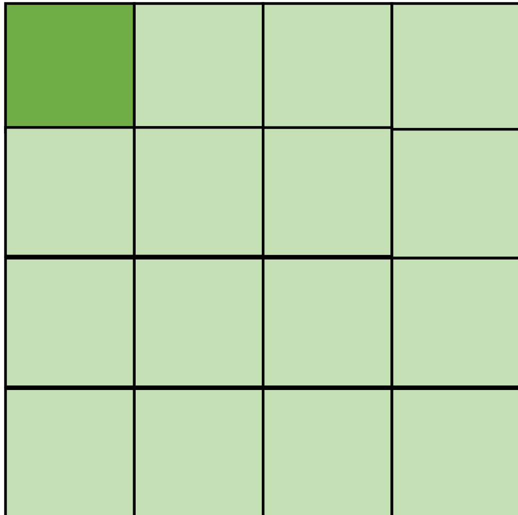


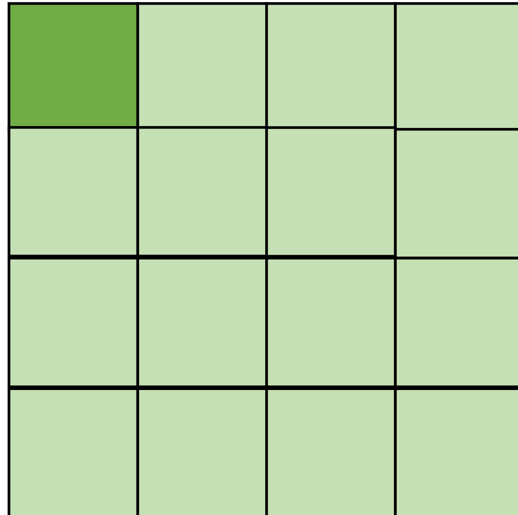


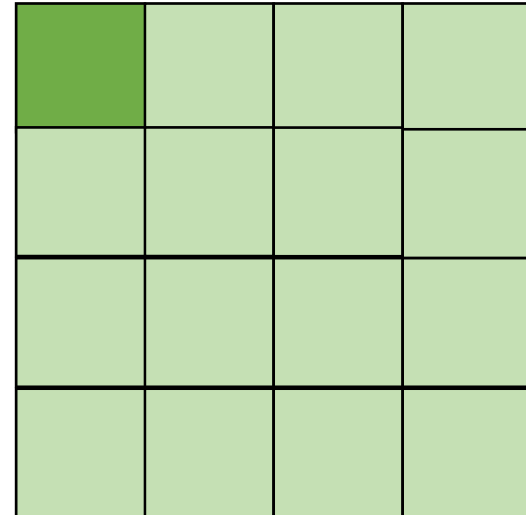
Adding loop nestings

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

$$A = B + C^T$$







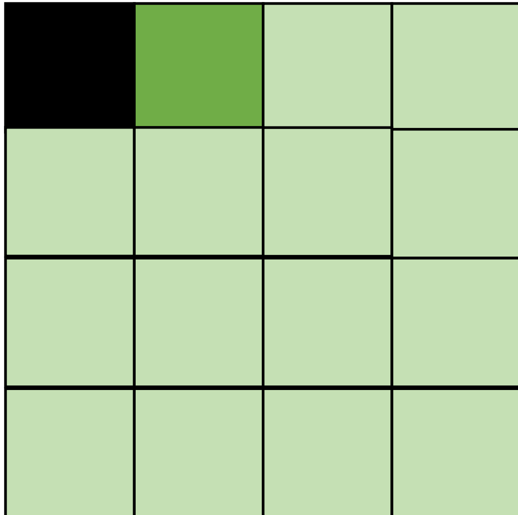
cold miss for all of them

Adding loop nestings

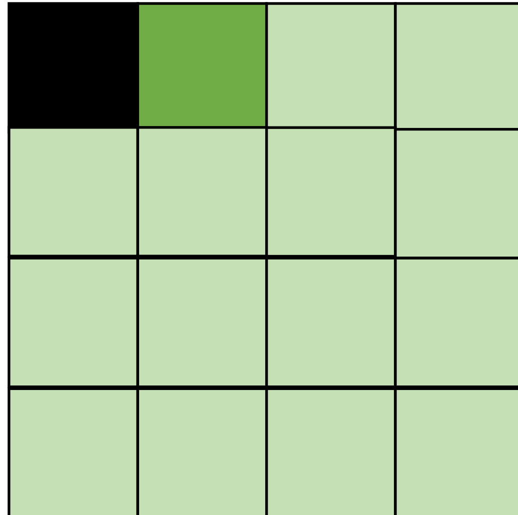
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

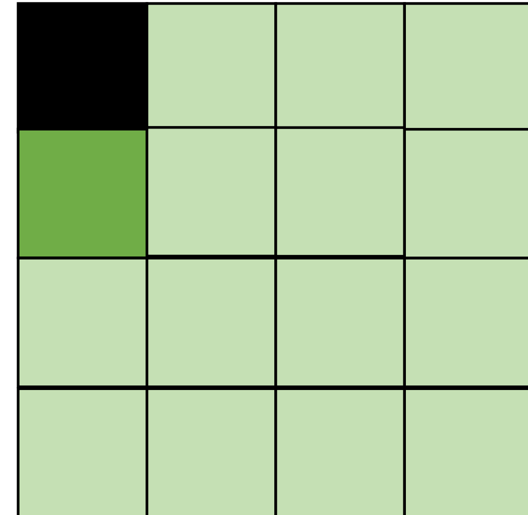
A



B



C



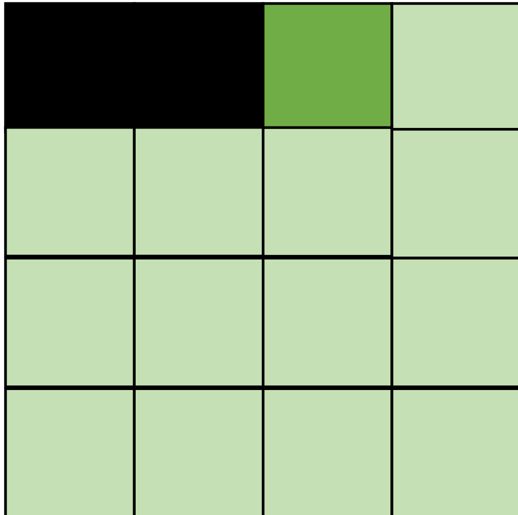
Hit on A and B. Miss on C

Adding loop nestings

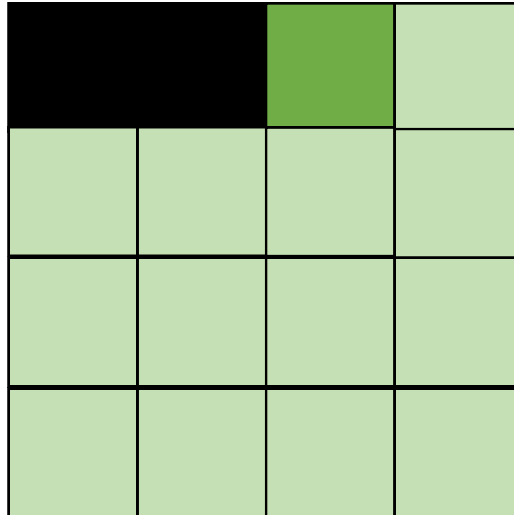
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

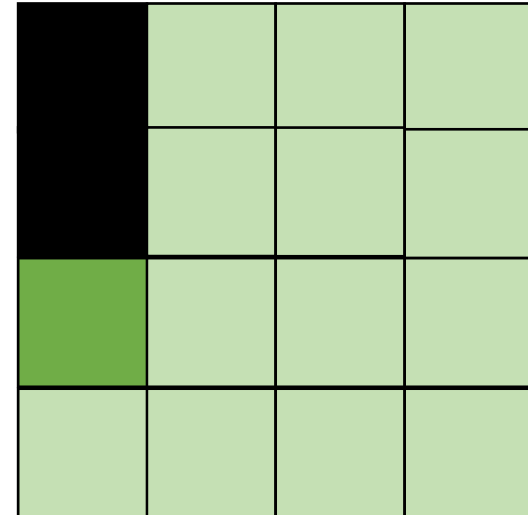
A



B



C



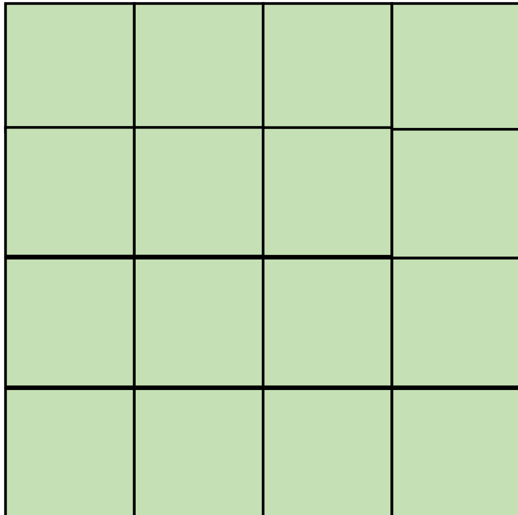
Hit on A and B. Miss on C

Adding loop nestings

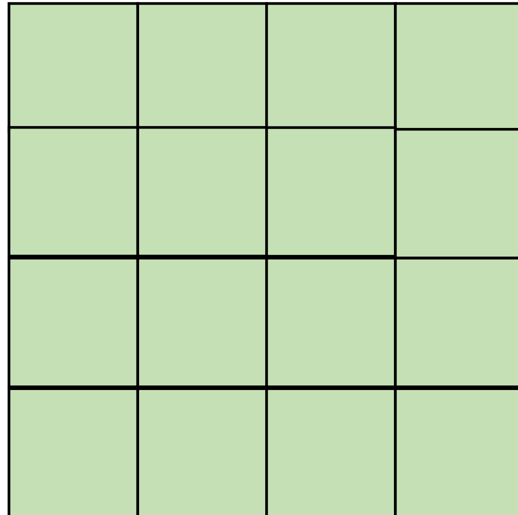
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

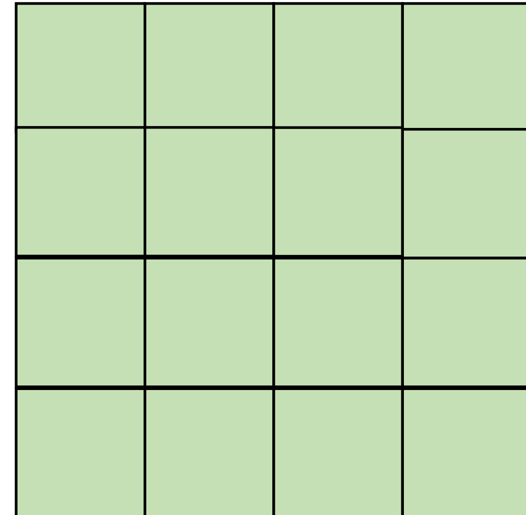
A



B



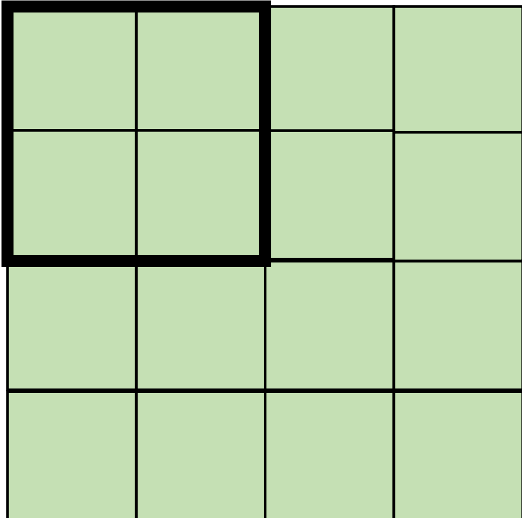
C

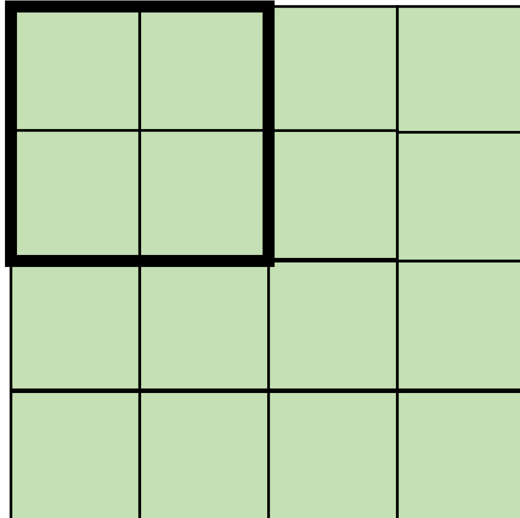


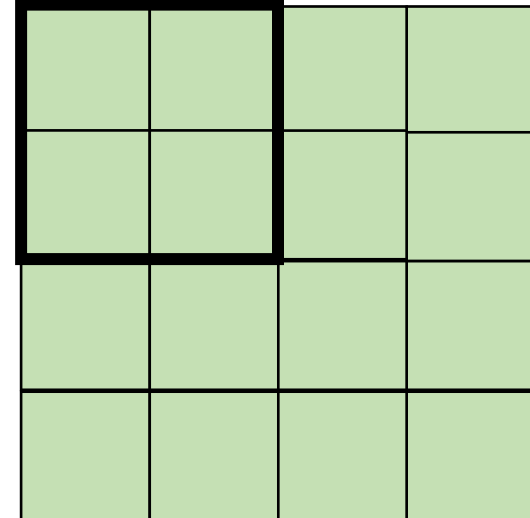
Adding loop nestings

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

$$A = B + C^T$$





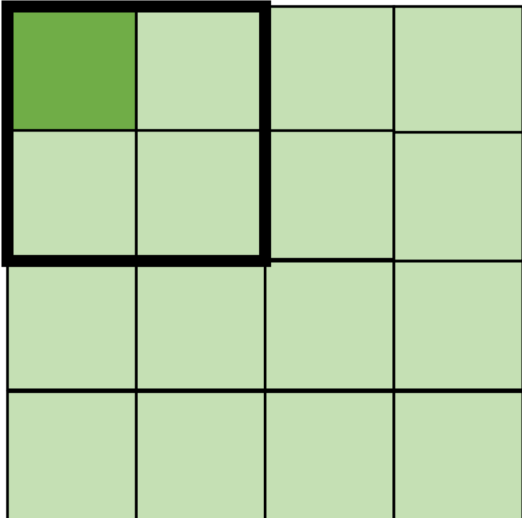


Adding loop nestings

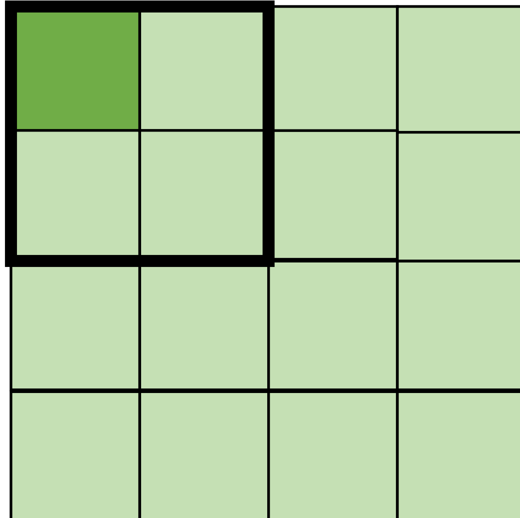
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

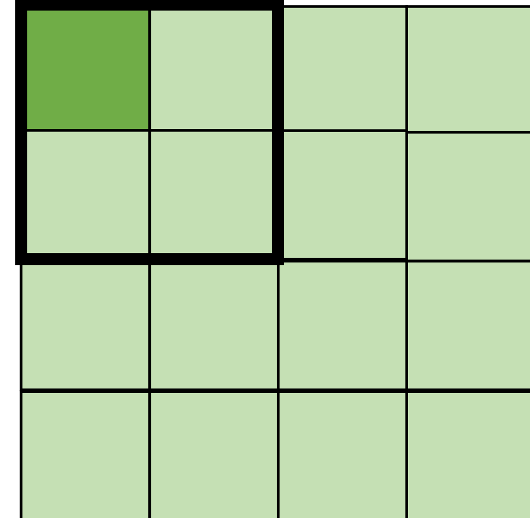
A



B



C



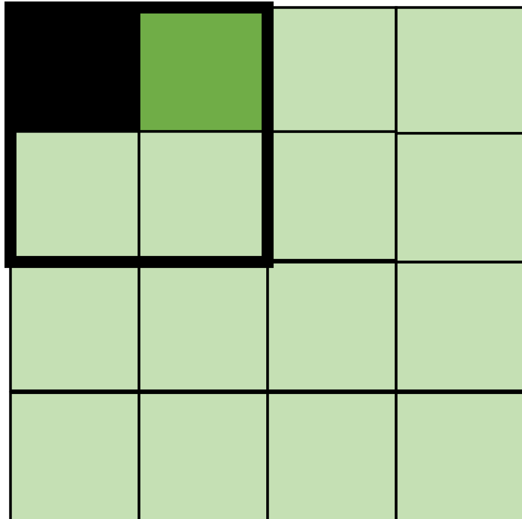
cold miss for all of them

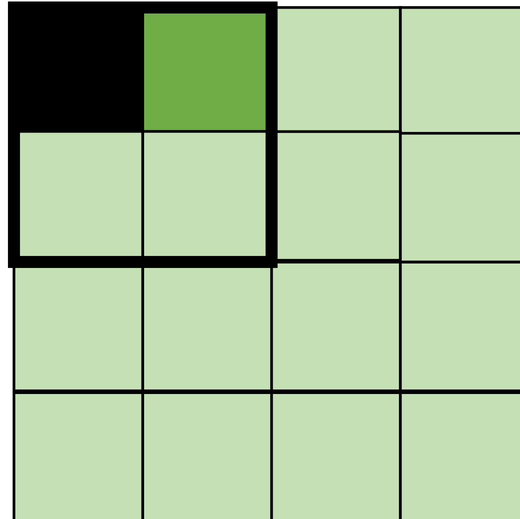
Adding loop nestings

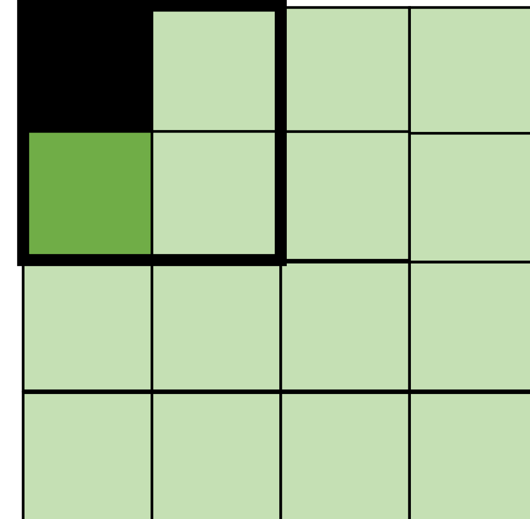
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

A B C







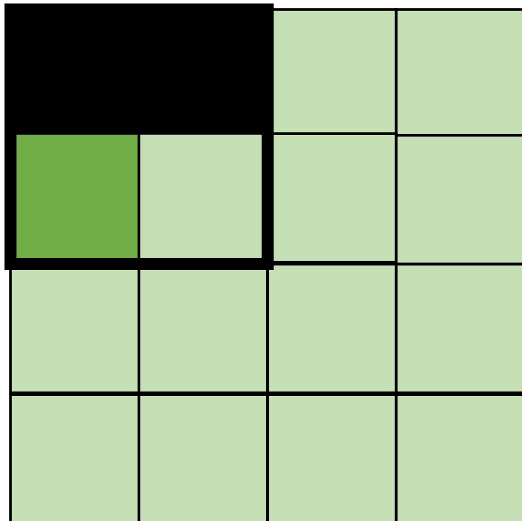
Miss on C

Adding loop nestings

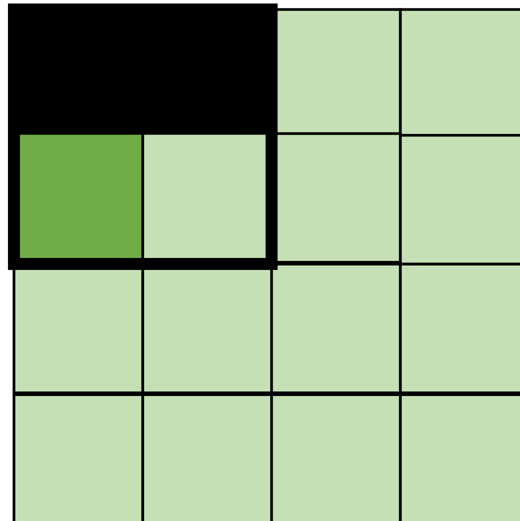
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

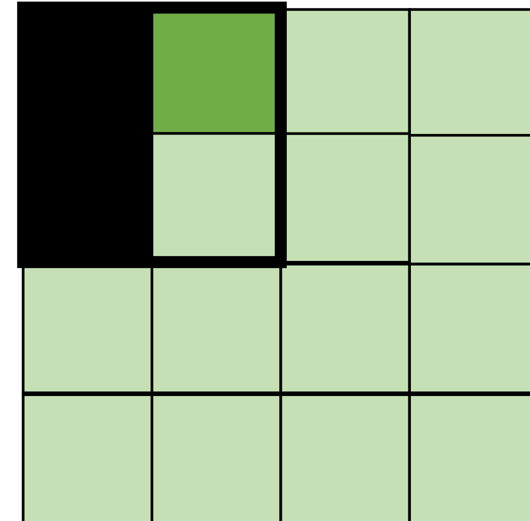
A



B



C



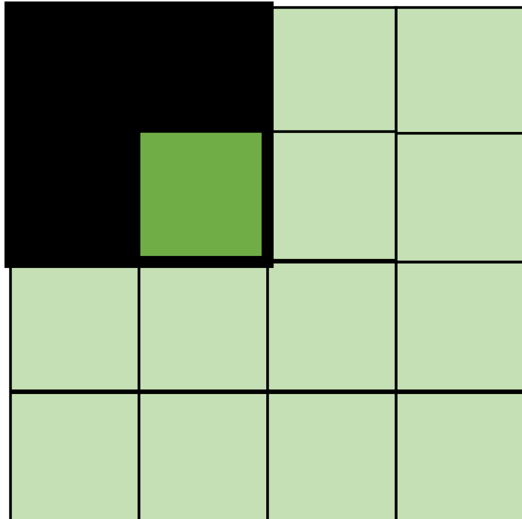
Miss on A,B, hit on C

Adding loop nestings

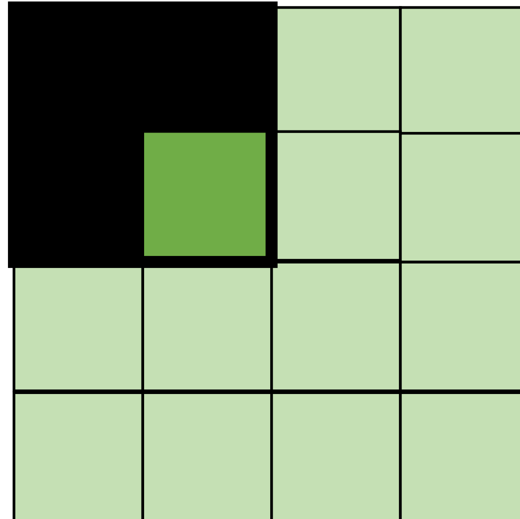
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

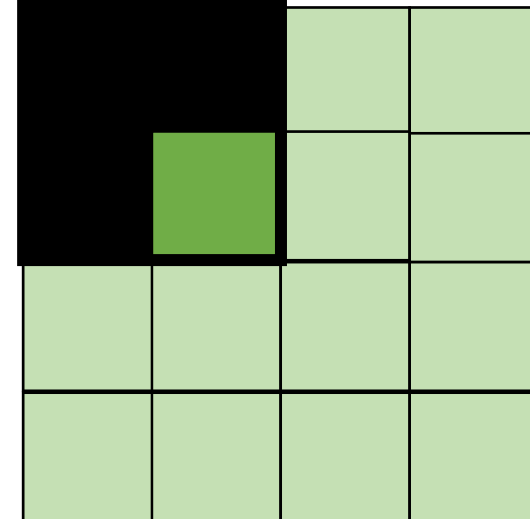
A



B



C



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});
```

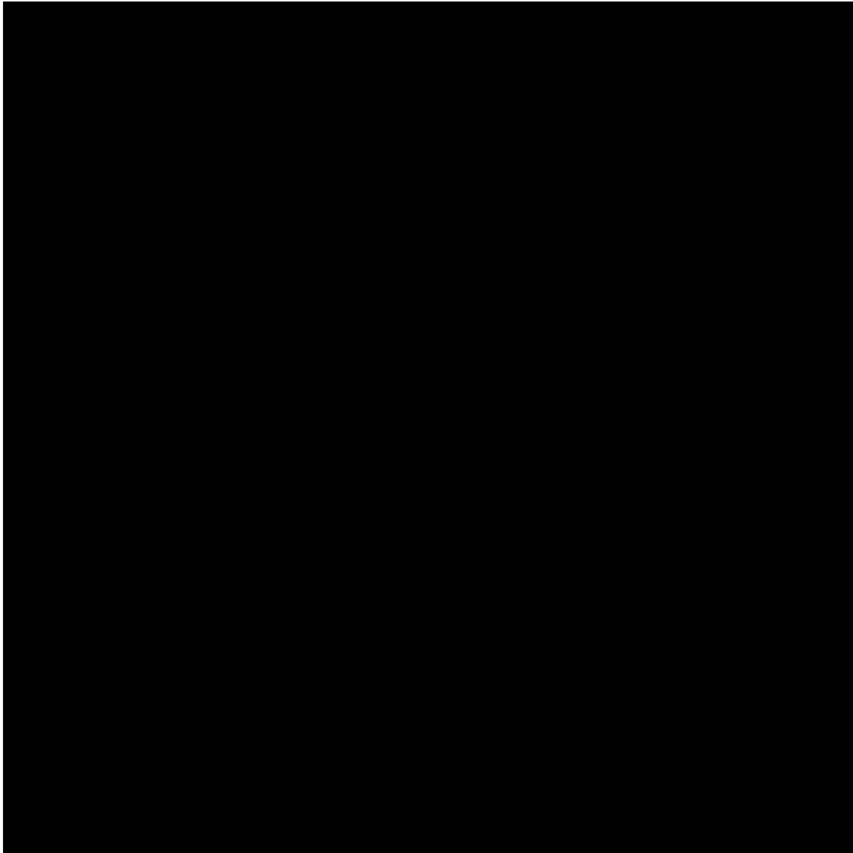
Schedule

```
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 < 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

Parallelism?

- Next lecture

Next class

- Continuing on DSL parallelism

- See you on Thursday
- Get a partner for homework 3!