CSE110A: Compilers

June 7, 2023

Topics:

- *More loop transforms*

Announcements

- Homework 3 grades are out
  - If you think the grade is off, come see us (or post a private piazza post)
  - If you only threw the wrong exception, then fix your code (ONLY CODE DEALING WITH THE TYPE OF EXCEPTION YOU ARE THROWING) by Friday and post on Piazza

- HW 5 is out
  - Due on Sunday
  - But try to get it done by Friday so that you have time to study for the final.
  - Lots of good questions on Piazza.
  - Get started if you haven’t!

Announcements

- Final is on Monday (June 12) at 8 AM in this classroom.
  - Comprehensive
  - You can have 3 pages of notes (front and back)
  - Like the midterm but 4 questions instead of 3
  - Guest lecture material will not be on the final.

Quiz

Quiz

Here are two ways of unrolling a for loop; what are some of the advantages or disadvantages of each method?

```
for(...){
a[i] = b[i] + c[i];
a[i + 1] = b[i + 1] + c[1 + 1];
a[i + 2] = b[i + 2] + c[1 + 2];
a[i + 3] = b[i + 3] + c[1 + 3];
i += 4;
}
```

```
for(...){
a[i] = b[i] + c[i];
i ++;
a[i] = b[i] + c[i];
i ++;
a[i] = b[i] + c[i];
i ++;
a[i] = b[i] + c[i];
i ++;
}
```

Quiz

Only loops without control flow in the loop body can be unrolled

- ◯ True
- ◯ False

Discussion

```
for (i = 0; i < 4; i++){
  for (j = 0; j < 4; j++){
    a[i] += b[j];
  }
}
```

Lets think about how unrolling the outer loop would look...

Discussion

```
for (i = 0; i < 4; i++){
  for (j = 0; j < 4; j++){
    a[i] += b[j];
  }
  i++;
  for (j = 0; j < 4; j++){
    a[i] += b[j];
  }
}
```

Lets think about how unrolling this loop would look...

Can’t do much now

Discussion

Lets think about how unrolling this loop would look...

```
for (i = 0; i < 4; i+=2){
  for (j = 0; j < 4; j++){
    a[i] += b[j];
  }
  for (j = 0; j < 4; j++){
    a[i+1] += b[j];
  }
}
```

What about now?

Discussion

Lets think about how unrolling this loop would look...

```
for (i = 0; i < 4; i+=2){
  for (j = 0; j < 4; j++){
    a[i]   += b[j];
    a[i+1] += b[j];
  }
}
```

This is an optimization called unroll and jam: unroll the outer loop and fuse the inner loop.

Quiz

Compilers allow you to annotate loops with `pragma` operations to tell the compiler to unroll loops, and by how much. For example, in Clang, you can annotate a loop with `#pragma clang loop unroll_count(2)` to unroll a loop by a factor of 2.

Describe a case where you as a program may want to tell the compiler how many times to unroll a loop (or tell the compiler not to unroll a loop at all)

More loop transforms

- Loop nesting order
- Loop tiling
- General area is called polyhedral compilation

https://en.wikipedia.org/wiki/Polytope_model

New constraints:

- Typically requires that loop iterations are independent
  - You can do the loop iterations in any order and get the same result

are these independent?

```
for (int i = 0; i < 2; i++) {
    counter += 1;
  }
```

vs

```
for (int i = 0; i < 1024; i++) {
    counter = i;
  }
```

adds two arrays

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

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

are they the same if you traverse them backwards?

adds two arrays

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

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] += a[i+1]
}
```

are they the same if you traverse them backwards?

adds two arrays

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

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] += a[i+1]
}
```

No!

adds two arrays

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

what about a random order?

```
for (pick i randomly) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (pick i randomly) {
  a[i] += a[i+1]
}
```

adds two arrays

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

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

what about a random order?

```
for (pick i randomly) {
  a[i] = b[i] + c[i];
}
```

```
for (pick i randomly) {
  a[i] += a[i+1]
}
```

No!

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

These are **DOALL** loops:

- Loop iterations are independent
- You can do them in ANY order and get the same results
- If a compiler can find a DOALL loop then there are lots of optimizations to apply!

Safety Criteria: independent iterations

- How do we check this?
  - If the property doesn't hold then there exists 2 iterations, such that if they are re-ordered, it causes different outcomes for the loop.
  - **Write-Write conflicts**: two distinct iterations write different values to the same location
  - **Read-Write conflicts**: two distinct iterations where one iteration reads from the location written to by another iteration.

Safety Criteria: independent iterations

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Safety Criteria: independent iterations

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

index calculation based on the loop variable

Safety Criteria: independent iterations

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

index calculation based on the loop variable
Computation to store in the memory location

Safety Criteria: independent iterations

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Write-write conflicts:

for two distinct iteration variables:
i_x != i_y
Check:
index(i_x) != index(i_y)

Safety Criteria: independent iterations

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Write-write conflicts:

for two distinct iteration variables:
i_x != i_y
Check:
index(i_x) != index(i_y)

Why?
Because if
index(i_x) == index(i_y)
then:
a[index(i_x)] will equal
either loop(i_x) or loop(i_y)
depending on the order

Safety Criteria: independent iterations

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
   a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:
i_x != i_y
Check:
write_index(i_x) != read_index(i_y)

Safety Criteria: independent iterations

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
   a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:
i_x != i_y
Check:
write_index(i_x) != read_index(i_y)

Why?

if i_x iteration happens first, then iteration i_y reads an updated value.

if i_y happens first, then it reads the original value

Examples:

```
for (i = 0; i < 128; i++) {
    a[i]= a[i]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]*2;
}
```

DOALL loops

- Very difficult for a compiler to prove
  - Although a decent amount of academic work, very little is done in actual compilers
- However, some domains naturally have DOALL loops?
  - Examples?
- People make ”Domain Specific Languages” that target only certain applications. Then you can provide more constrains and optimize more aggressively.

Motivation:

Image processing

Taken from Halide:
A DSL project out of MIT

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 an image DSL 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/

Motivation:

Image processing

Taken from Halide:
A DSL project out of MIT

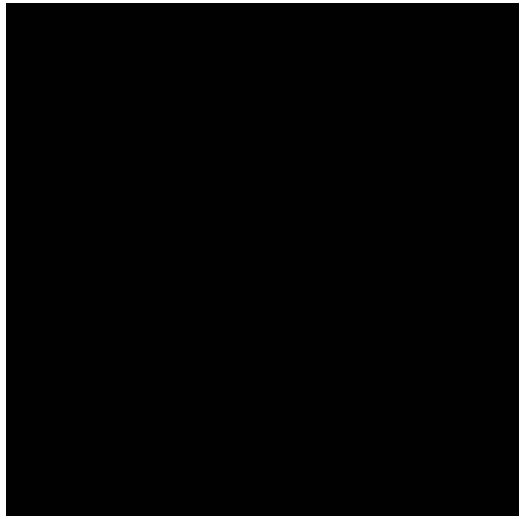
pretty straight forward computation for brightening

(1 pass over all pixels)

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

DSL provides two languages: one for the computation, and one for the optimizations and orders

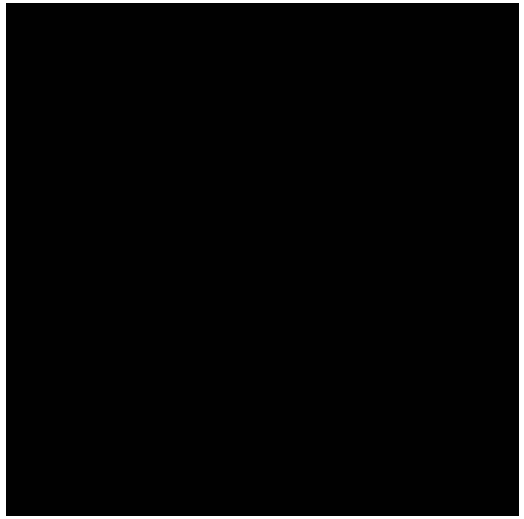
from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/



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

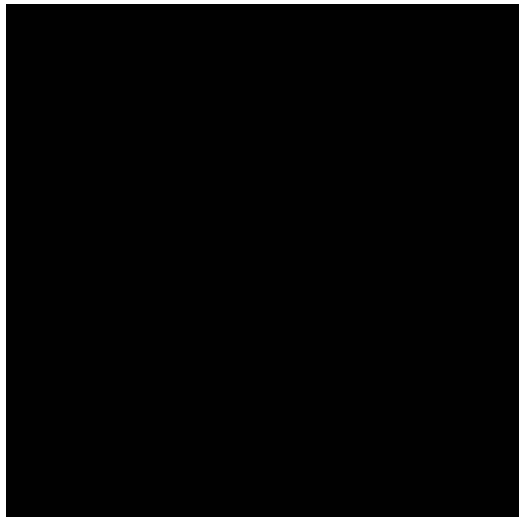
you can compute the pixels in any order you want, you just have to compute all of them!

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



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

you can compute the pixels in any order you want, you just have to compute all of them!



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

What is the difference here? What will the difference be?

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

Demo

- Why do we see the performance difference?

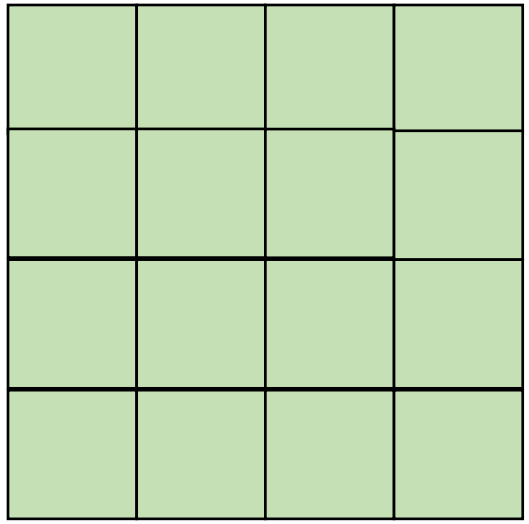
Adding 2D arrays together

Demo

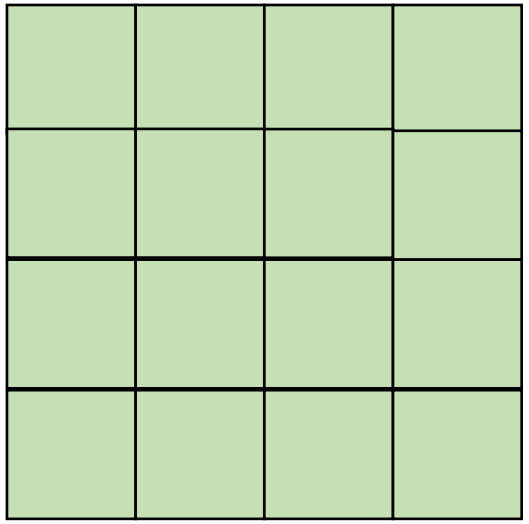
- Memory accesses

$$A = B + C$$

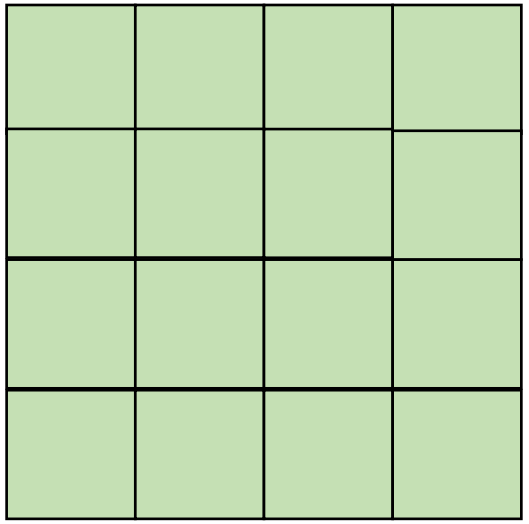
A



B



C



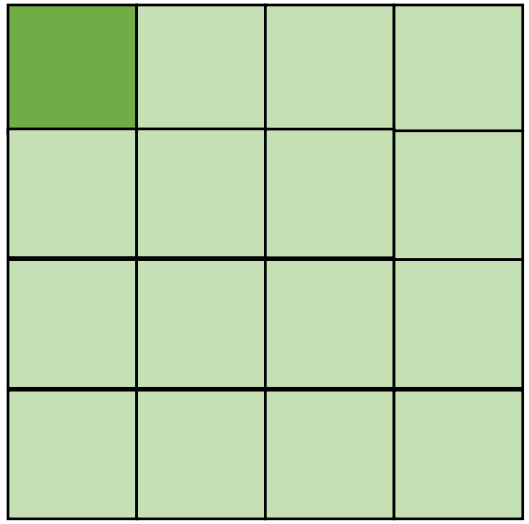
Adding 2D arrays together

Demo

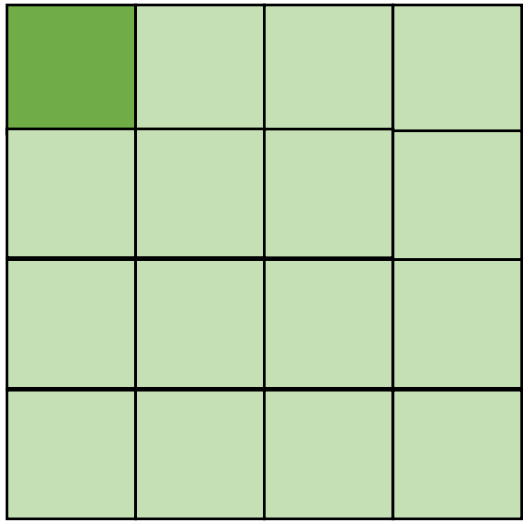
- Memory accesses

$$A = B + C$$

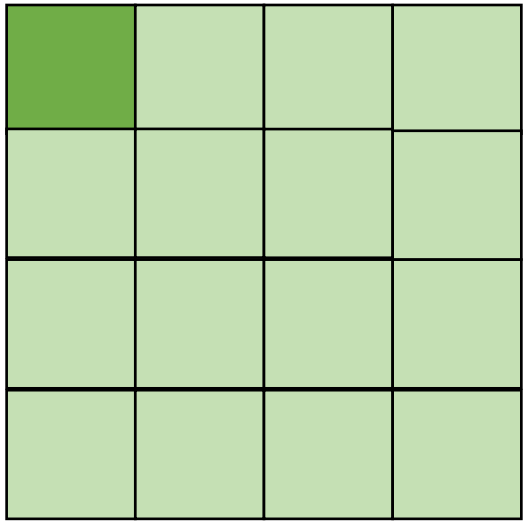
A



B



C



Cache miss for all of them

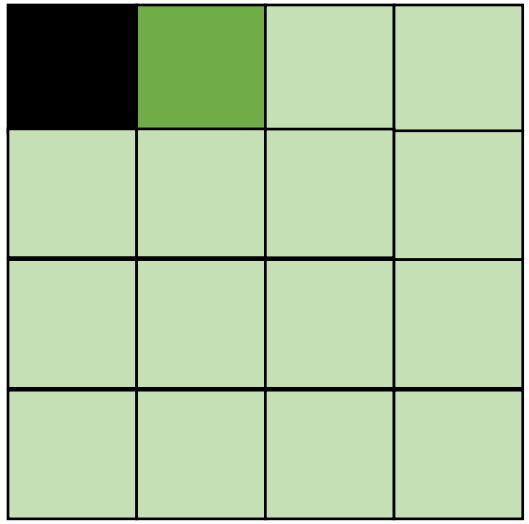
Adding 2D arrays together

Demo

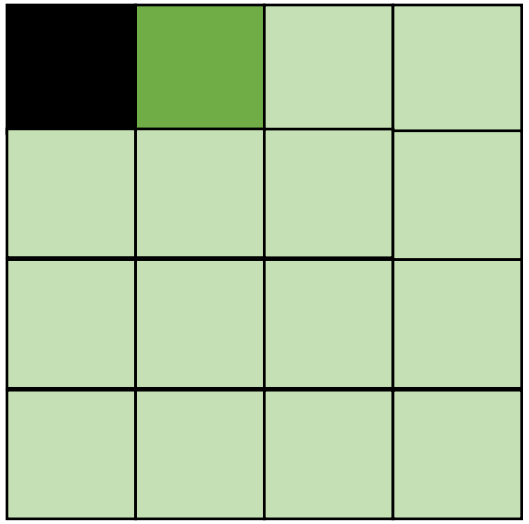
- Memory accesses

$$A = B + C$$

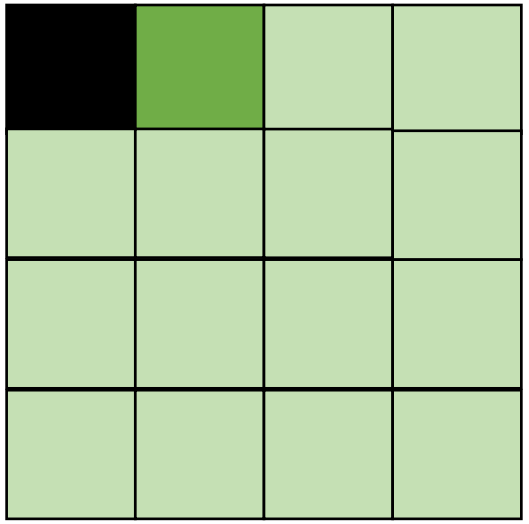
A



B



C



Cache HIT for all of them

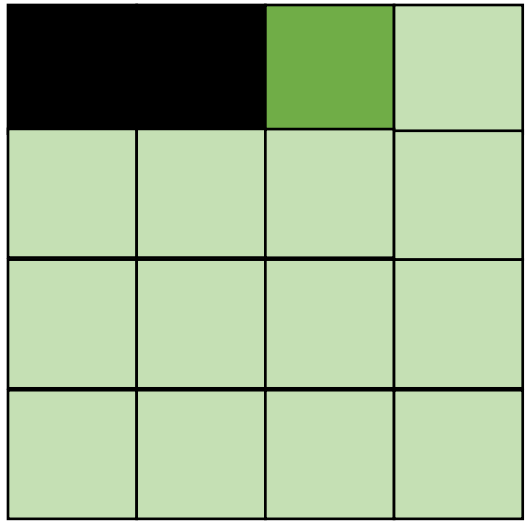
Adding 2D arrays together

Demo

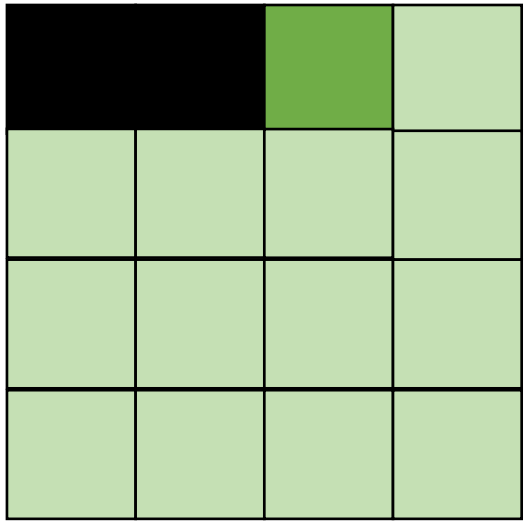
- Memory accesses

$$A = B + C$$

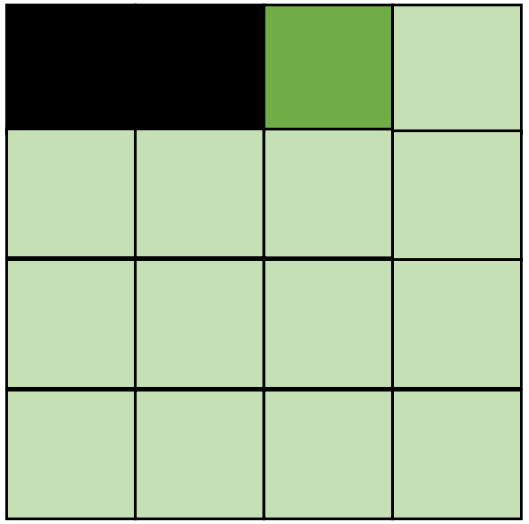
A



B



C



Cache HIT for all of them

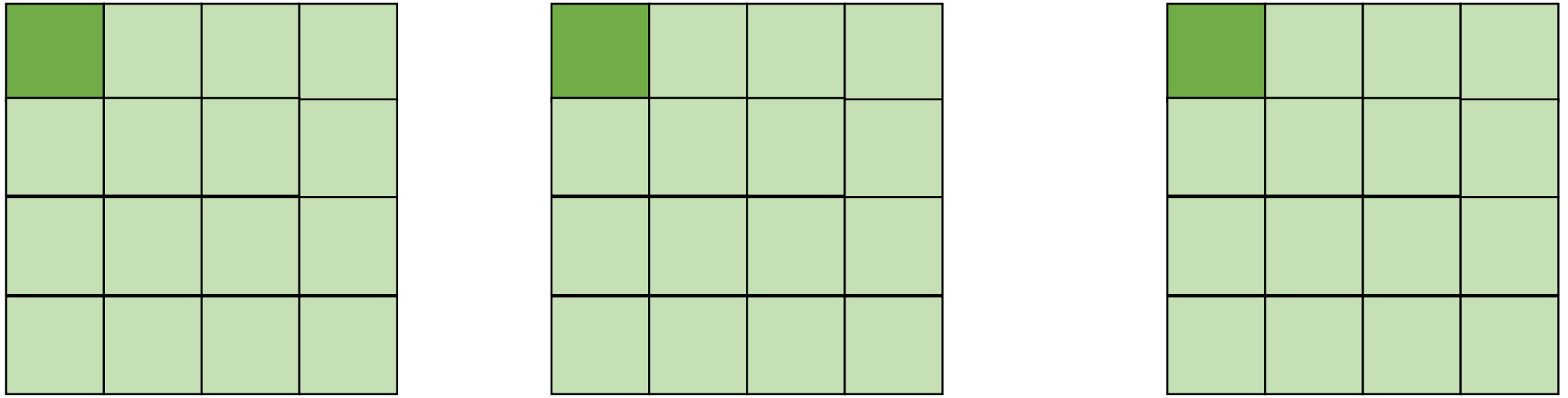
Adding 2D arrays together

Demo

- Memory accesses

$$A = B + C$$

A B C



Rewind! Cache miss for all of them

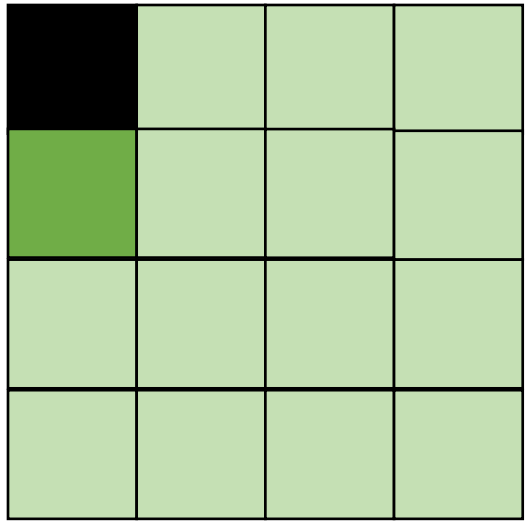
Adding 2D arrays together

Demo

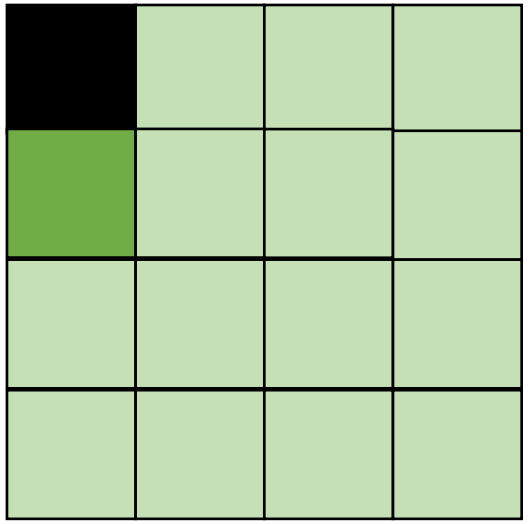
- Memory accesses

$$A = B + C$$

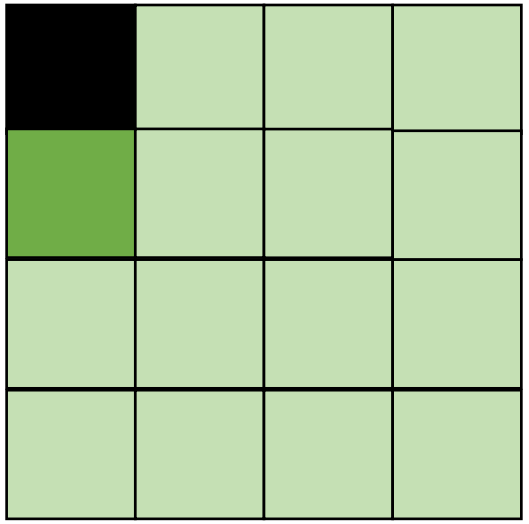
A



B



C



Rewind!

Cache miss for all of them

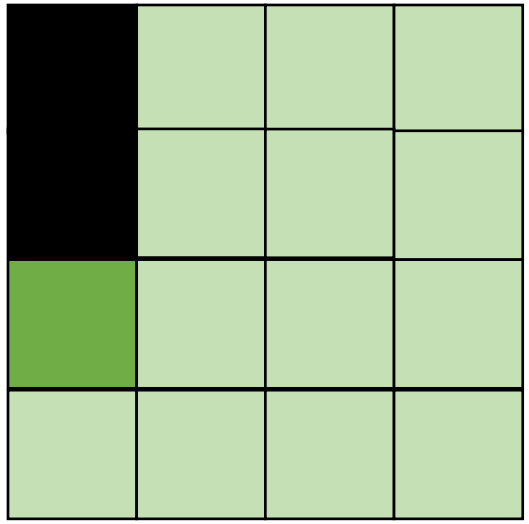
Adding 2D arrays together

Demo

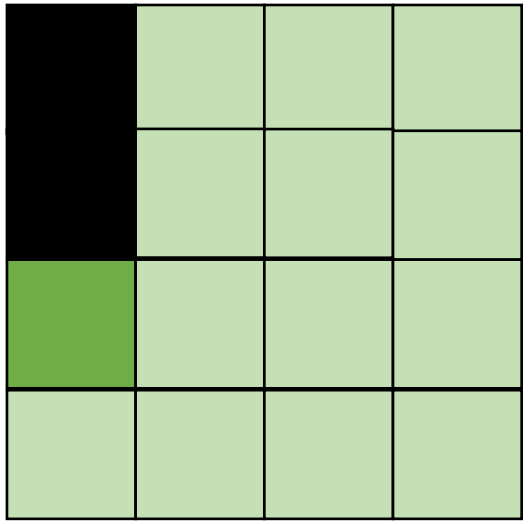
- Memory accesses

$$A = B + C$$

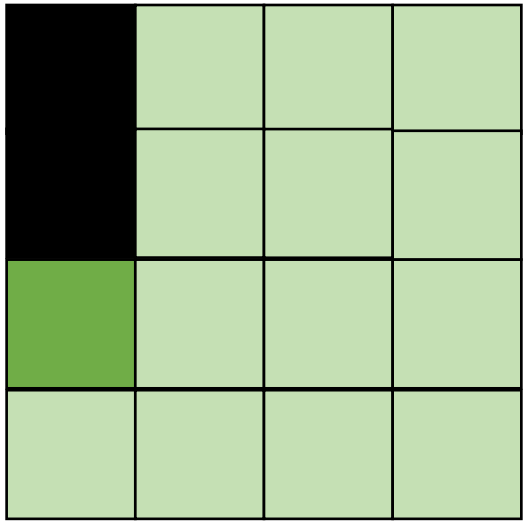
A



B



C



Rewind!

Cache miss for all of them

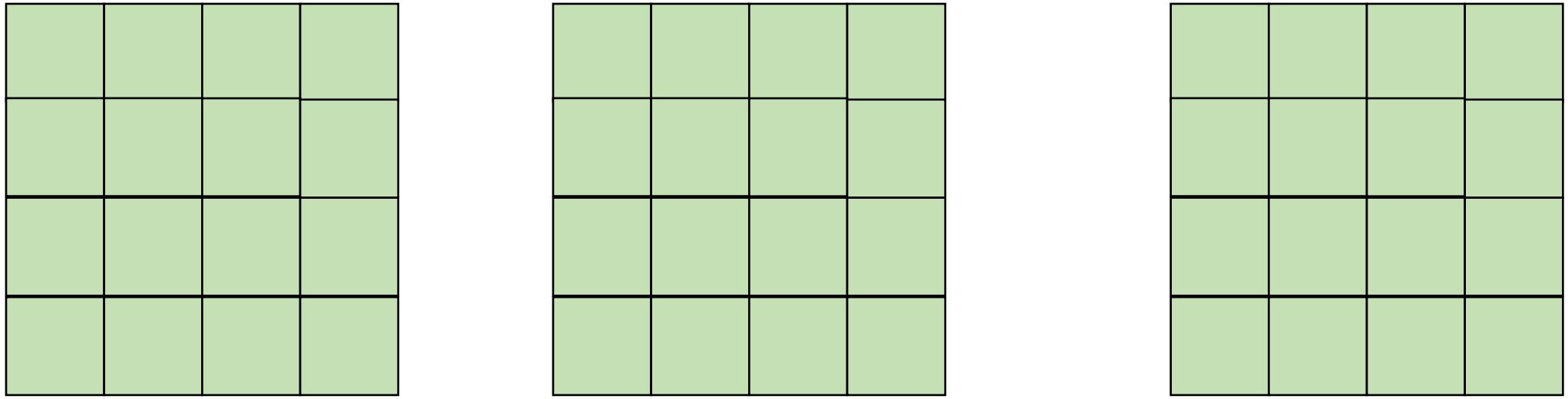
But sometimes there isn’t a good ordering

transposed arrays

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

$$A = B + C^T$$

A B C

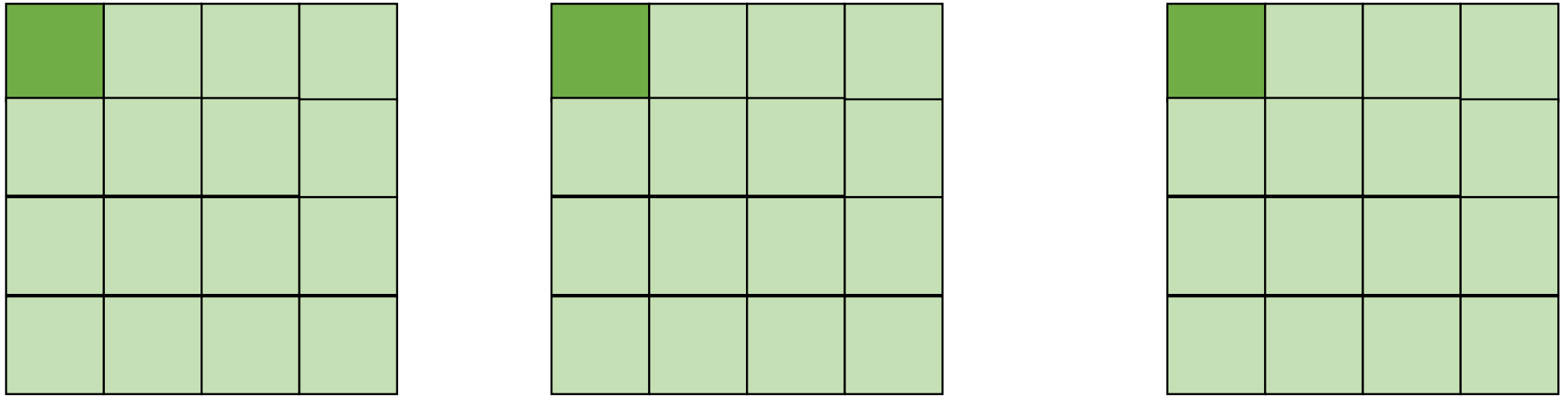


transposed arrays

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

$$A = B + C^T$$

A B C



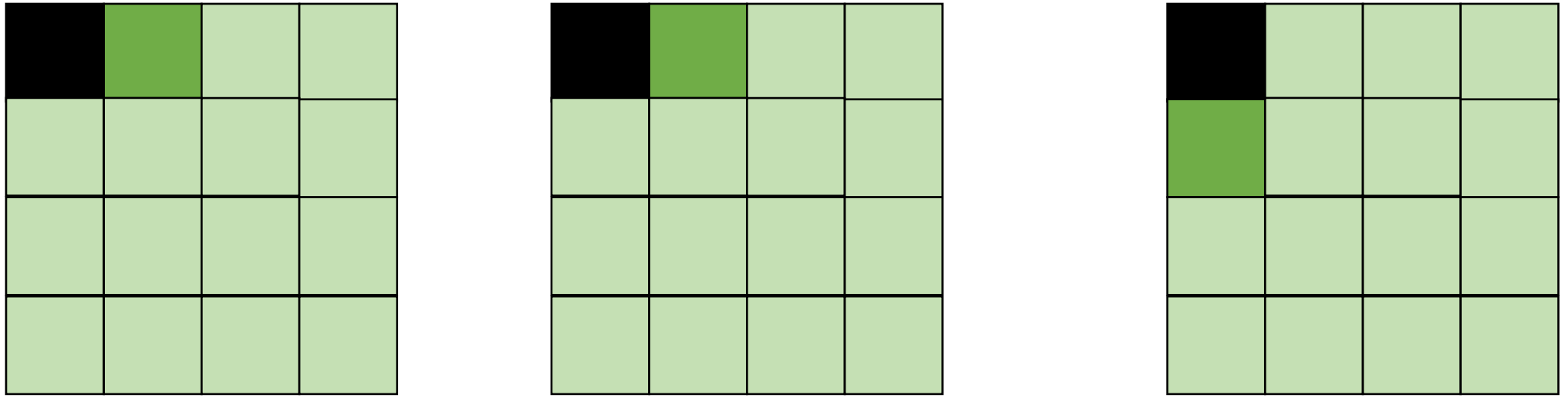
cold miss for all of them

transposed arrays

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

$$A = B + C^T$$

A B C



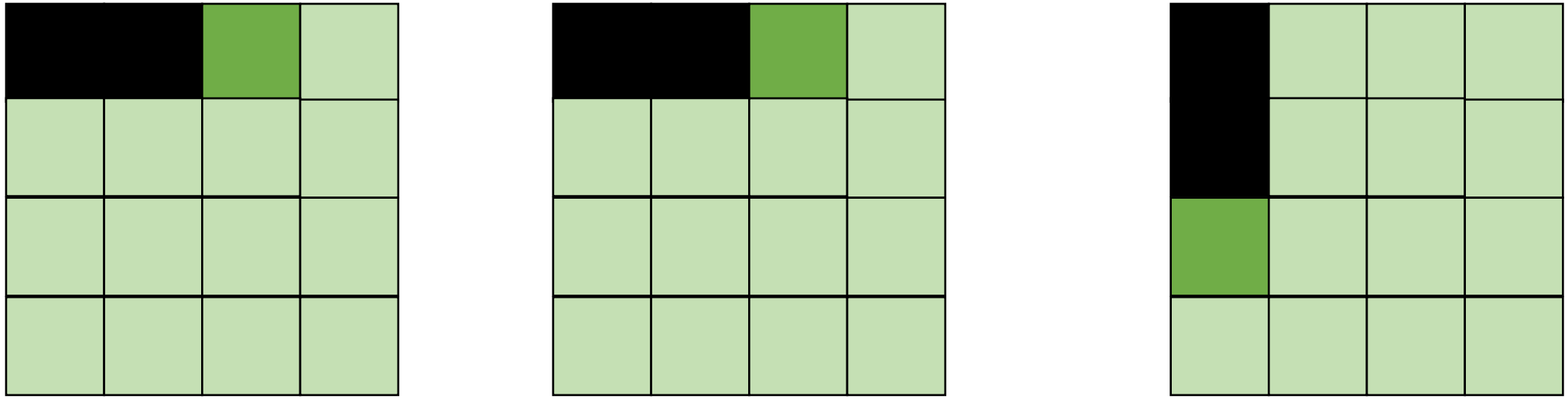
Hit on A and B. Miss on C

transposed arrays

- 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

What happens here?

- Demo

How can we fix it?

- Can we use the compiler?
- Does loop order matter?

Loop transformations

- We can change loop order
- What else?

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

Loop splitting:

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

What is the difference here?

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

Does loop splitting by itself work?

- Lets try it
  - demo

We can chain optimizations

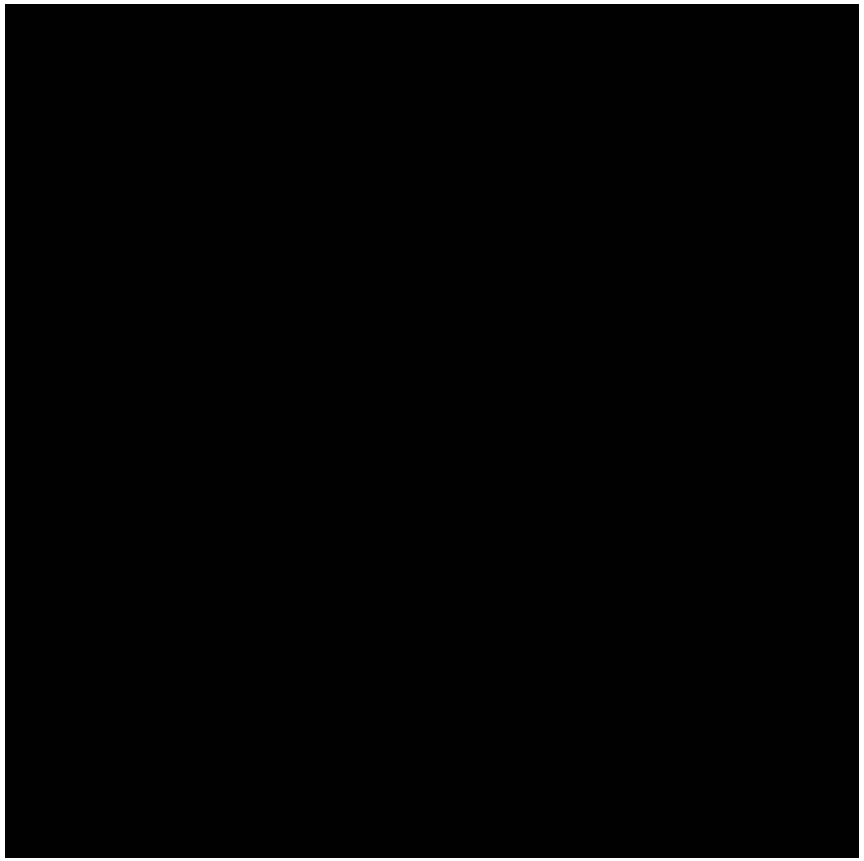
- Lets try chaining loop splitting and reorder
  - Demo

We can chain optimizations

- Lets try chaining loop splitting and reorder
  - Demo

- What happened?!

Our new schedule looks like this:



Why is this beneficial?

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

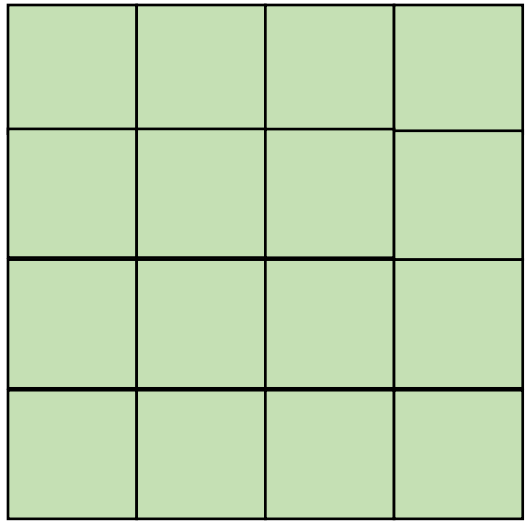
blocking

blocking

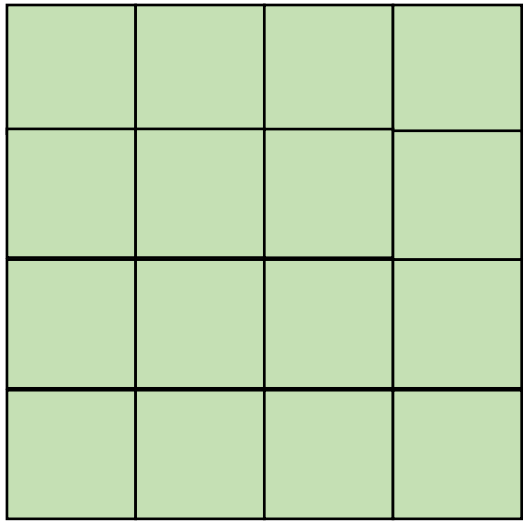
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

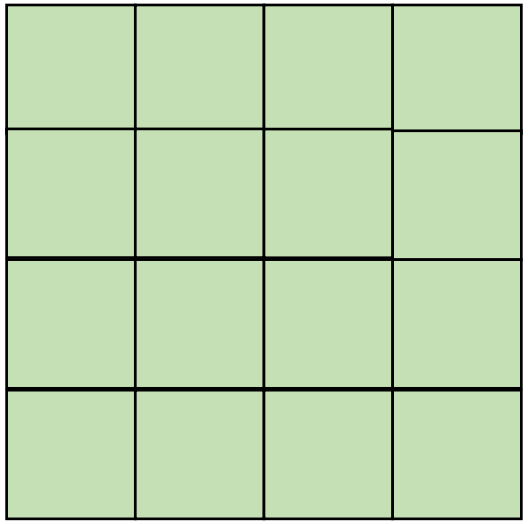
A



B



C

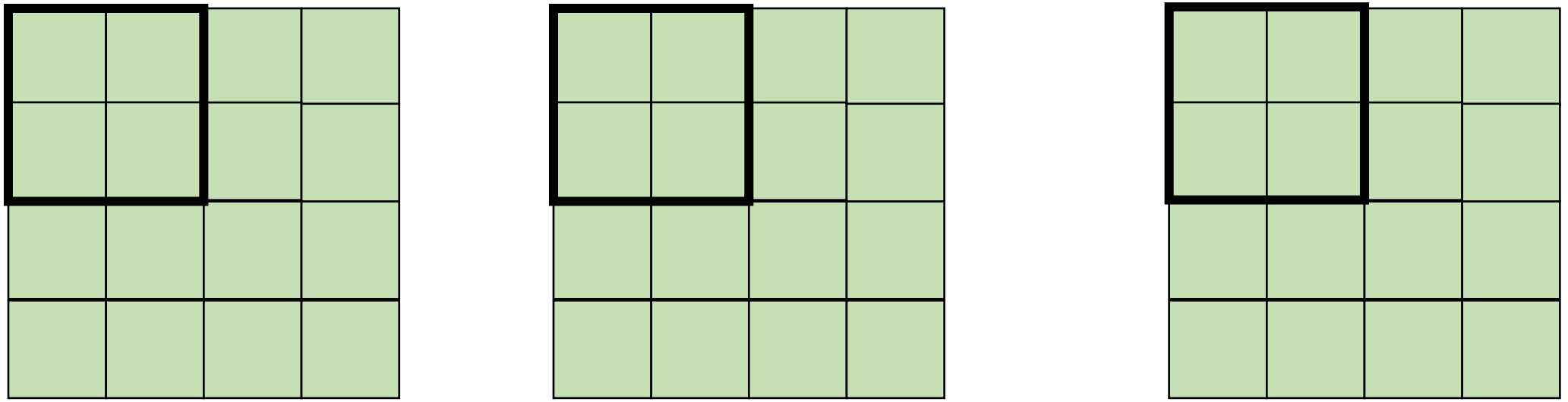


blocking

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

$$A = B + C^T$$

A B C

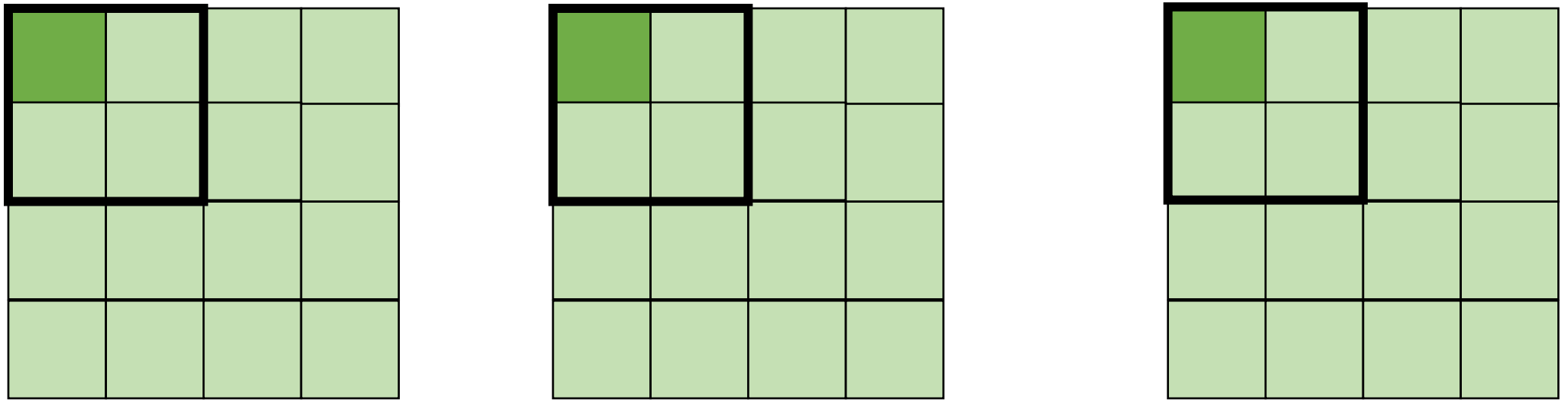


blocking

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

$$A = B + C^T$$

A B C



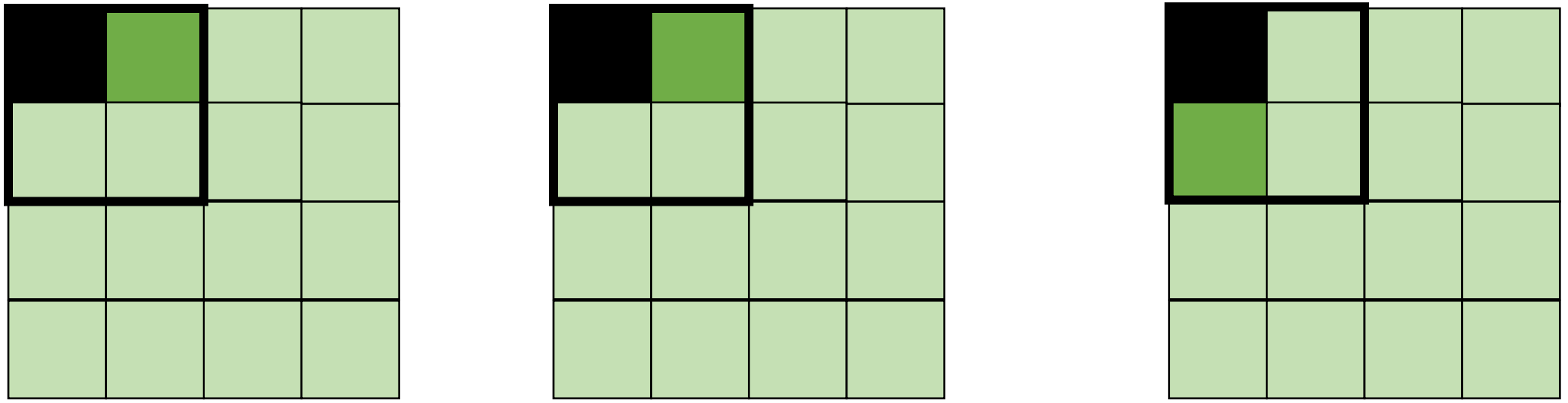
cold miss for all of them

blocking

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

$$A = B + C^T$$

A B C



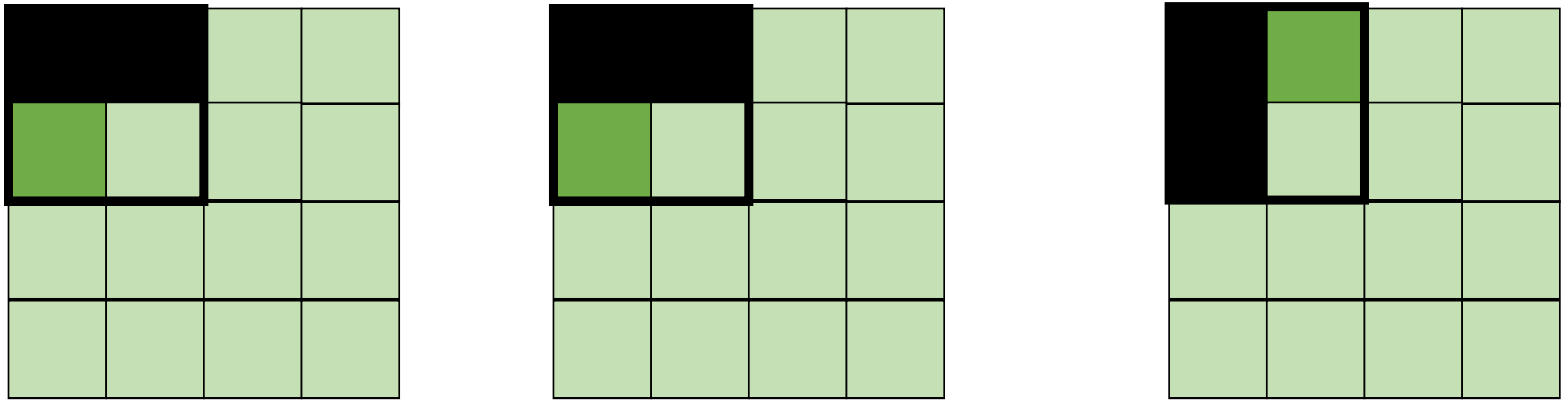
Miss on C

blocking

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

$$A = B + C^T$$

A B C



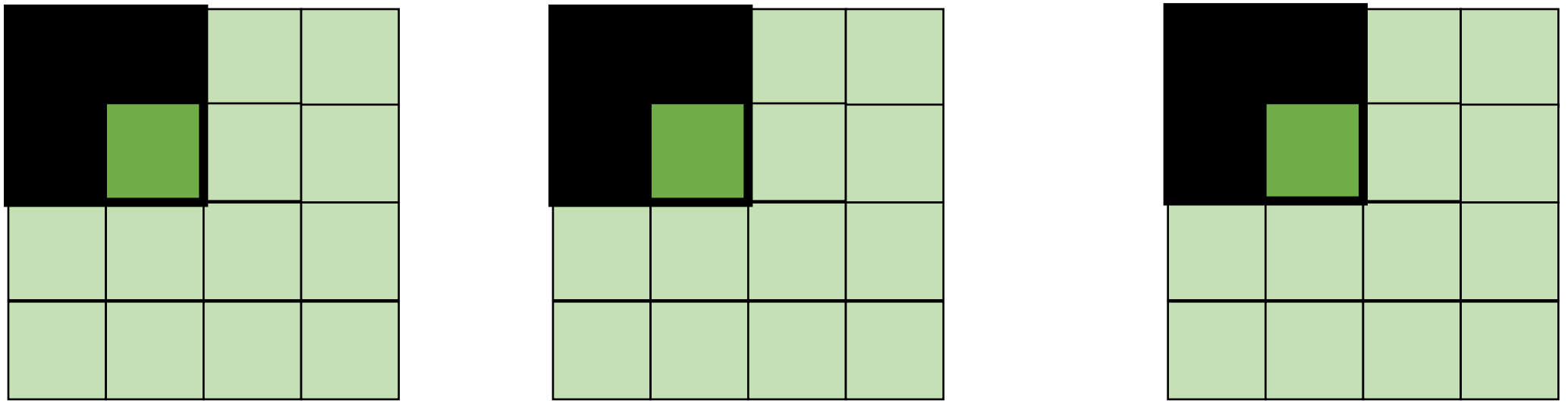
Miss on A,B, hit on C

blocking

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

$$A = B + C^T$$

A B C



Hit on all!

Other uses of loop split

- Say your processor can vectorize 4 elements at a time

```
for (int y = 0; y < 4; y++) {
    for (int x = 0; x < 8; x++) {
        output[y,x] = x + y;
    }
}
```

```
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 8; x_outer+=4) {
        for (int x = x_outer; x < x_outer+4; x++) {
            output[y,x] = x + y;
        }
    }
}
```

Other uses of loop split

```
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 8; x_outer+=4) {
        for (int x = x_outer; x < x_outer+4; x++) {
            output[y,x] = x + y;
        }
    }
}
```

specify vectorize

Loop transformation summary

- If the compiler can prove different properties about your loops, you can automatically make code go a lot faster

- It is hard in languages like C/C++. But in constrained languages (often called domain specific languages (DSLs) it is easier!
  - Hot topic right now for Machine learning, graphics, graph analytics, etc!

Main results in from an image DSL 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/

Homework

See everyone on Friday!

- Class review for final