# CSE110A: Compilers

May 25, 2022

**Topics**:

- *Loop transformations*
- *Homework overview*
- *More loop transforms*

# Announcements

- New grades:
  - Midterm grades are out
  - Let us know within a week if there are issues.
  - You should be able to see comments for each subsection if you missed points
    - If not let us know
  - Double check the comments. If we messed up let us know
  - Average was ~76%

- HW 3 is due
  - It was due yesterday
  - get it in ASAP if you have not

- Homework 4 is released
  - my opinion:
    - conceptually it is not as hard as HW2 or HW3.
    - Practically it is difficult to deal with all the corner cases.
  - ***start early!***

# Quiz

# Quiz

During local value numbering, we should reorder the operands before inserting them as keys in the hashtable.

- ◯ True
- ◯ False

# Discussion

```
a = c + b;
d = b + c;
```

```
H = {
}
```

# Quiz

What do you need to fill in at the '___HERE__' after local value numbering on each basic block to make it the program valid.

label0:

a = b + c

d = b + c

___HERE__

label1:

e = a + f

- ◯ NOP
- ◯ a = a0
- ◯ d = a0
- ◯ d = d0

# Discussion

work through the example

```
label0:
a = b + c
d = b + c
___HERE__
label1:
e = a + f
```

# Quiz

```
int foo(int x) {
return x + x;
}
```

This function does not has side effects, so the following is valid for compiler:

```
c = foo(a);
e = foo(a);
```

optimized to

```
c = foo(a);
e = c;
```

# Quiz

how many times can you unroll this loop;

```
for (int i = 0; i < 16; ++i) {
c[i] = a[i ]+ b[i];
}
```

# Discussion

work through the example
how many comparisons and branches can you save?
What if i was a memory location?

```
for (int i = 0; i < 16; ++i) {
   c[i] = a[i] + b[i];
}
```

# Review

- Stitching optimized blocks back into the whole program

# How to stitch optimized code back into the program

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

# How to stitch optimized code back into the program

gets optimized to this:

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

# How to stitch optimized code back into the program

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

gets optimized to this:

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

Can it be stitched together like this?

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
label_0:
h2 = g0 + a1;
k3 = h2;
```

# How to stitch optimized code back into the program

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

gets optimized to this:

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

easiest way
showing only first basic block :

```
b0 = b;
c1 = c;
e3 = e;
f4 = f;

a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;

a = a2;
d = d5;
g = g6;
```

# How to stitch optimized code back into the program

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

gets optimized to this:

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

For a language like ClassIeR:
record new vrs:
{b0, c1, e3, f4, a2, d5, g6}

easiest way
showing only first basic block :

```
b0 = b;
c1 = c;
e3 = e;
f4 = f;

a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;

a = a2;
d = d5;
g = g6;
```

# How to stitch optimized code bac program

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

**Whole example:**

```
b0 = b;
c1 = c;
e3 = e;
f4 = f;
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
a = a2;
d = d5;
g = g6;

label_0:
g0 = g;
a1 = a;
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

new variables:
{b0, c1, e3, f4, a2, d5, g6}
{g0, a1, h2, k3}

# How to stitch optimized code bac program

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

Special cases: labels and branches:
how to stitch with them?

**Whole example:**

```
b0 = b;
c1 = c;
e3 = e;
f4 = f;
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
a = a2;
d = d5;
g = g6;

label_0:
g0 = g;
a1 = a;
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

new variables:
{b0, c1, e3, f4, a2, d5, g6}
{g0, a1, h2, k3}

# Picking back up where we left off: Loop Unrolling

# For loops terminology

- Loop body:
  - A series of statements that are executed each loop iteration
- Loop condition:
  - the condition that decides whether the loop body is executed
- Iteration variable:
  - A variable that is updated exactly once during the loop
  - The loop condition depends on the iteration variable
  - The loop condition is only updated through the iteration variable

# Examples

*iteration variable*
*loop body*
*loop condition*

```
for (int i = 0; i < 1024; i++) {
    counter += 1;
  }
```

```
for (; i < 1024; i+=counter) {
    counter += 1;
  }
```

```
  while (1) {
    i++;
    counter += 1;
    if (i < 1024) {
      break;
    }
  }
```

*In general, is it possible to determine if an iteration variable exists or not?*

# Loop unrolling

# Loop unrolling

- Executing multiple instances of the loop body without checking the loop condition.

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

unrolled by a **factor** of 2

```
for (int i = 0; i < 128; i++) {
   // body
}
```

```
for (int i = 0; i < 128; i++) {
   // body
   i++
   // body
}
```

*could we unroll more?*

# Loop unrolling conditions

- Under what conditions can we unroll?

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

# Loop unrolling conditions

- Under what conditions can we unroll?

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a variable (cond) or literal
   * check that cond is not assigned in body

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a variable or literal (cond)
   * check that cond is not assigned in body

*Do these guarantee we will find an iteration variable?*
*What happens if we don't find one?*

*how does C-simple help us here?*

# Loop unrolling conditions

- Several ways to unroll
  - More constraints: Simpler to unroll in code generation
  - Less constraints: Harder to unroll in code generation

***Base constraints (required for any unrolling):***

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a variable or literal (cond)
   * check that cond is not assigned in body

# Loop unrolling conditions

- Simple unroll
  - Most constraints
  - Easiest code generation

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body ... F times
- perform codegen

# Loop unrolling conditions

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 128; i++) {
  // body
}
```

how to do these steps?

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body ... F times
- perform codegen

# Loop unrolling conditions

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 128; i++) {
   // body
}
```

result for a factor of 2

```
for (int i = 0; i < 128; i++) {
   // body
   i++
   // body
}
```

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body ... F times
- perform codegen

# Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 8; i+=3) {
  // body
}
```

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body ... F times
- perform codegen

# Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 8; i+=3) {
  // body
}
```

Actually this is fine as long as i is updated with a constant addition. but we need a more complicated formula to calculate LI:

```
ceil((end - start)/update)
```

But you may want to keep your life simpler by constraining it. We will keep it for now

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body ... F times
- perform codegen

# Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 8; i+=3) {
  // body
}
```

Do example

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body ... F times
- perform codegen

# Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 4; i++) {
  // body
}
```

What if we try to unroll this by a factor of 3?

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body ... F times
- perform codegen

# Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 4; i++) {
  // body
}
```

```
for (int i = 0; i < 4; i++) {
  // body
  i++
  // body
  i++
  // body
}
```

What if we try to unroll this by a factor of 3?

How many times do we execute body?

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body ... F times
- perform codegen

# Loop unrolling conditions

Let's examine this a bit closer?

```
for (int i = 0; i < 4; i++) {
   // body
}
```

What if we try to unroll this by a factor of 3?

```
for (int i = 0; i < 4; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

How many times do we execute body?

# Loop unrolling conditions

Let's examine this a bit closer?

```
for (int i = 0; i < 4; i++) {
   // body
}
```

```
for (int i = 0; i < 4; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

What if we try to unroll this by a factor of 3?

How many times do we execute body?

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = ?; i < ?; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

```
for (int i = ?; i < ?; i++) {
   // body
}
```

# Loop unrolling conditions

initially the loop starts the same as the original loop

```
for (int i = 0; i < 4; i++) {
  // body
}
```

find out how many unrolled loops we can execute:
?
This gives us the first bound

second loop is initialized with the first bound

second loop's bound is same as the original loop

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = ?; i < ?; i++) {
  // body
  i++
  // body
  i++
  // body
}
```

```
for (int i = ?; i < ?; i++) {
  // body
}
```

# Loop unrolling conditions

initially the loop starts the same as the original loop

```
for (int i = 0; i < 4; i++) {
  // body
}
```

find out how many unrolled loops we can execute:
(4 / 3) * 3 = 3
This gives us the first bound

second loop is initialized with the first bound

second loop's bound is same as the original loop

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = ?; i < ?; i++) {
  // body
  i++
  // body
  i++
  // body
}
```

```
for (int i = ?; i < ?; i++) {
  // body
}
```

# Loop unrolling conditions

What about in the general case? For unroll factor F?

```
for (int i = x; i < y; i++) {
   // body
}
```

find out how many unrolled loops we can execute:
?
This gives us the first bound

second loop is initialized with the first bound

second loop's bound is same as the original loop

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = ?; i < ?; i++) {
   // body
   i++
   ...
}
```

```
for (int i = ?; i < ?; i++) {
   // body
}
```

# Loop unrolling conditions

- general unroll

For unroll factor F

**General unroll constraints:**
- Loop update increments by 1
- Find the concrete number of loop iterations, LI

**General unroll code generation:**
- Create simple unrolled loop with new bound: (LI/F)*F
- Create cleanup (basic) loop with initialization: (LI/F)*F
- perform codegen

*None of these numbers have to be concrete!*

# Homework overview

- discussion and demo on command line

# 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;
  }
```

# Motivation:

**Image processing**

pretty straight
forward computation
for brightening

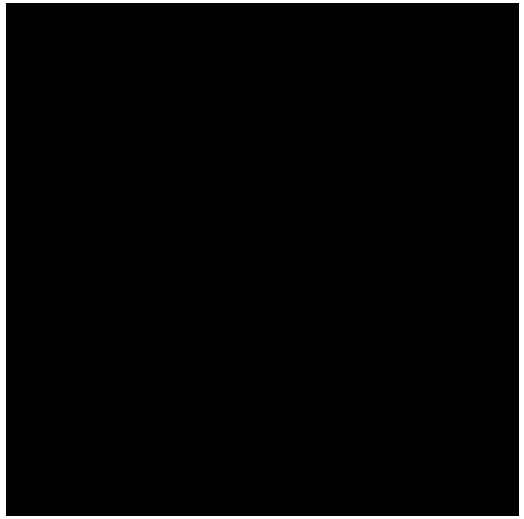
(1 pass over all pixels)

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*

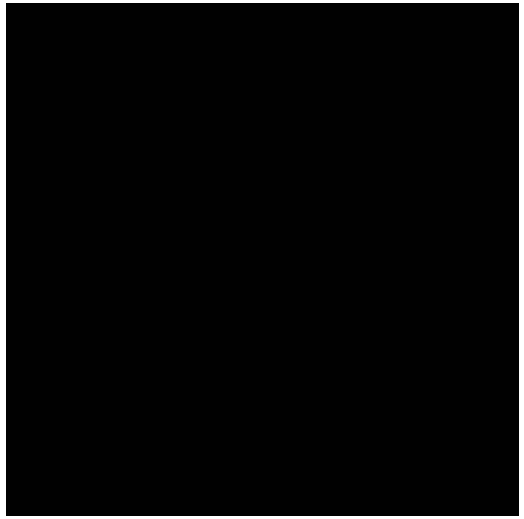
from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/



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

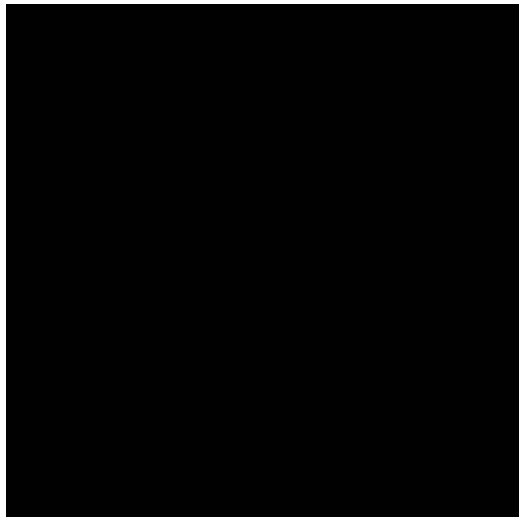
you can compute the pixels in any order you want, you just have to compute all of them!

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

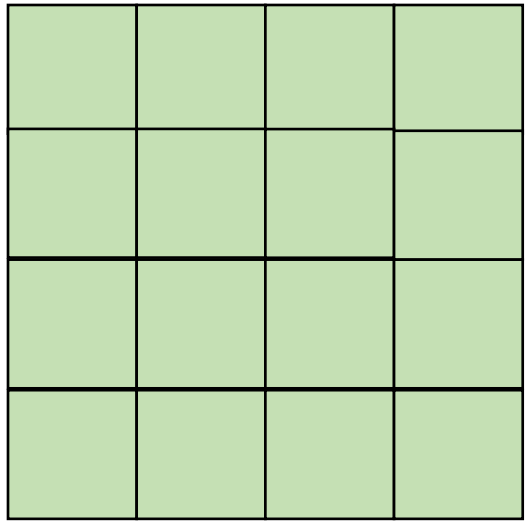
- Why do we see the performance difference?

# Adding 2D arrays together

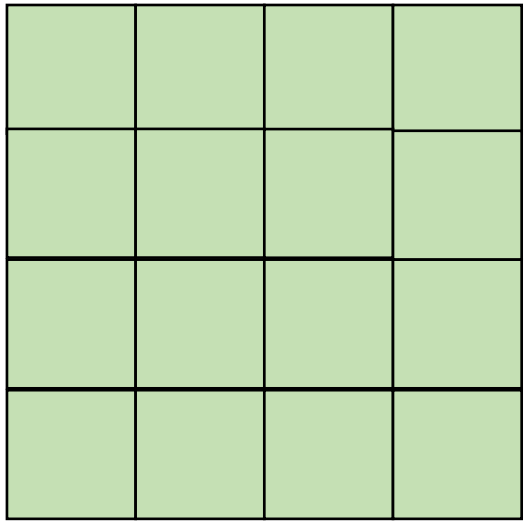
- Memory accesses

$$A = B + C$$

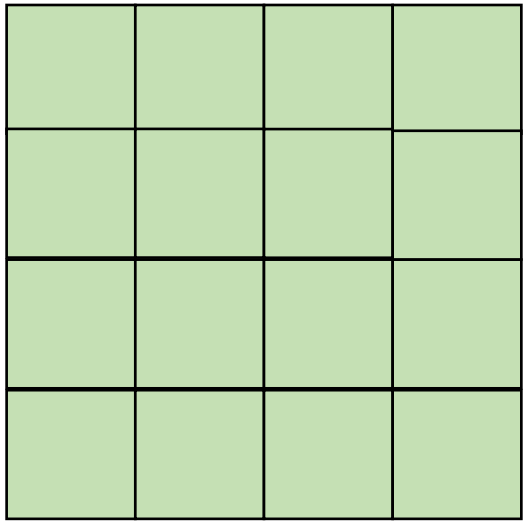
$A$



$B$



$C$



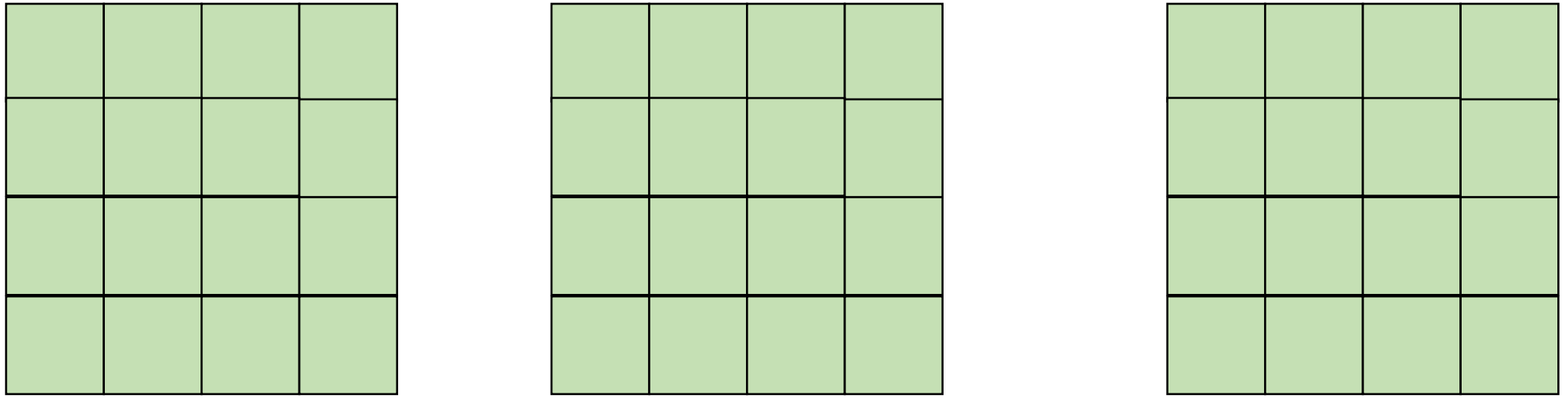
# But sometimes there isn’t a good ordering

# transposed arrays

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

$$A = B + C^T$$

$A$ $B$ $C$

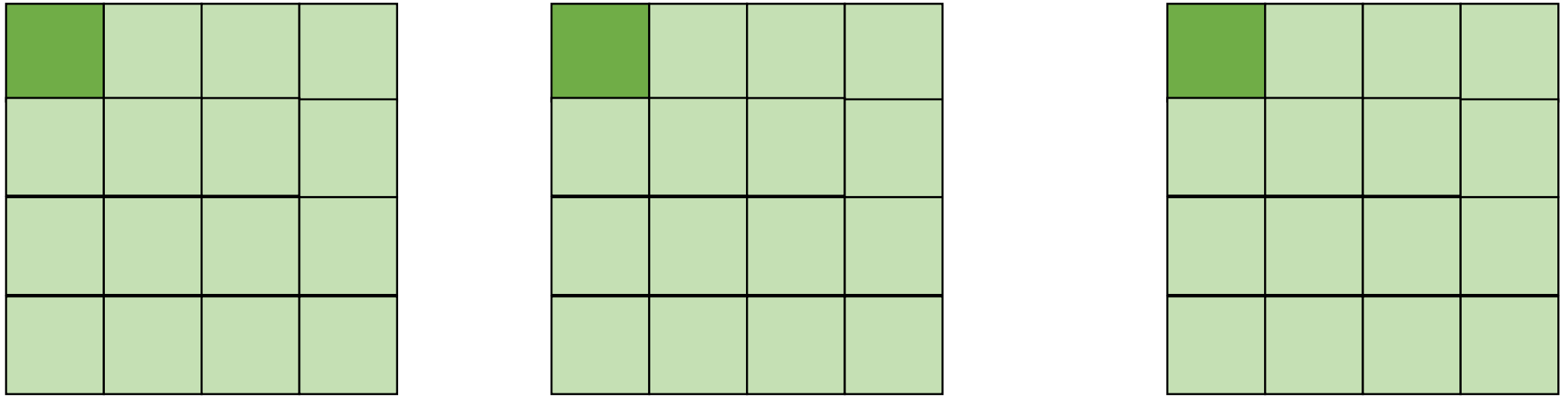


# transposed arrays

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

$$A = B + C^T$$

$A$ $B$ $C$



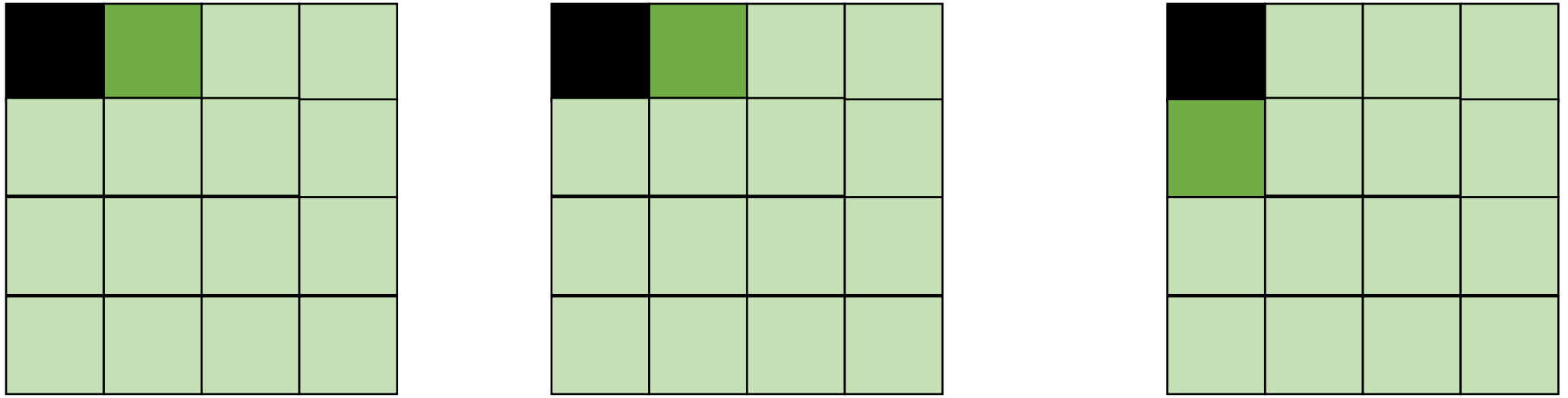
*cold miss for all of them*

# transposed arrays

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

$$A = B + C^T$$

$A$ $B$ $C$



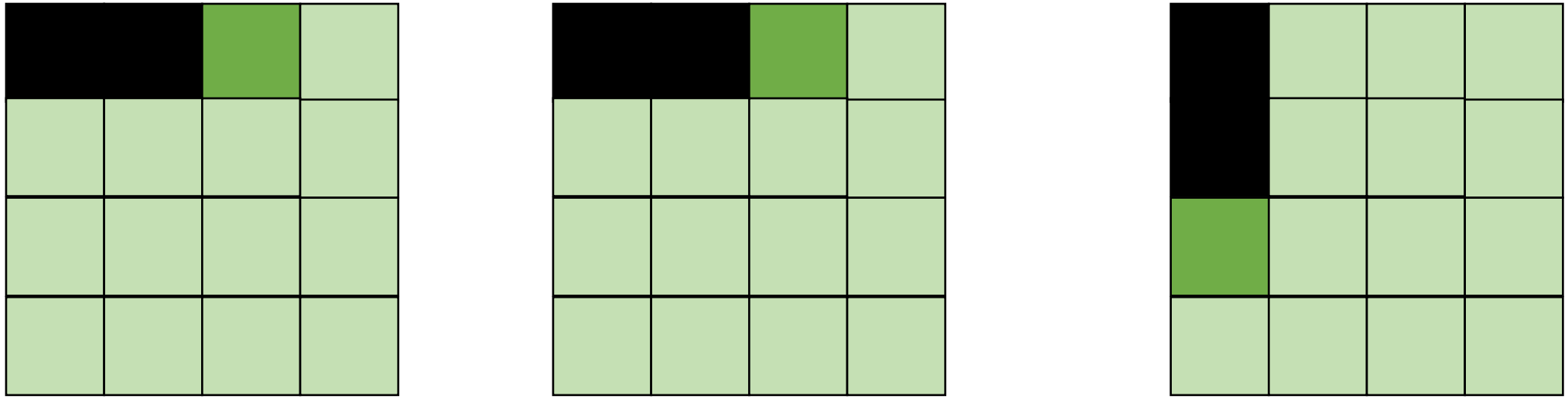
*Hit on A and B. Miss on C*

# 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?

```
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

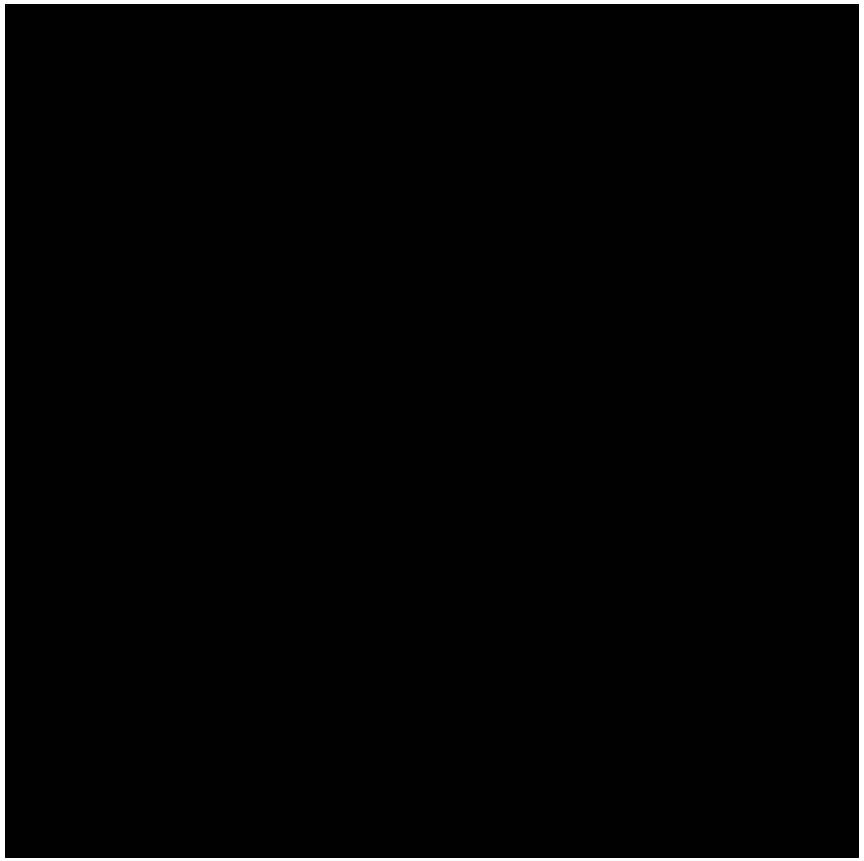
- Lets try chaining loop splitting and reorder
  - Demo

# 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

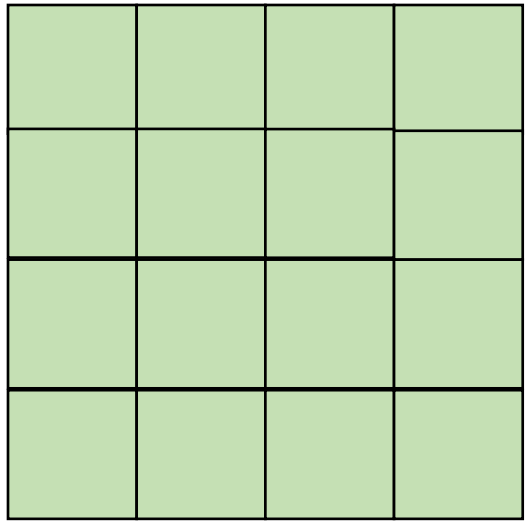
# blocking

# blocking

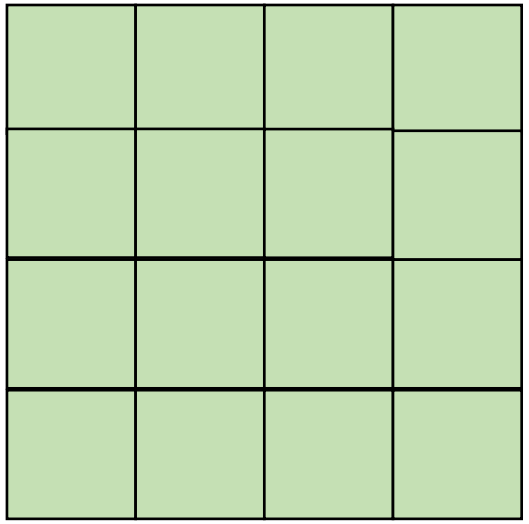
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

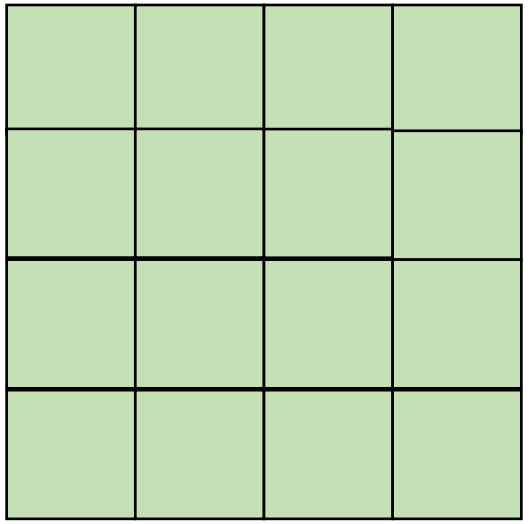
$A$



$B$



$C$

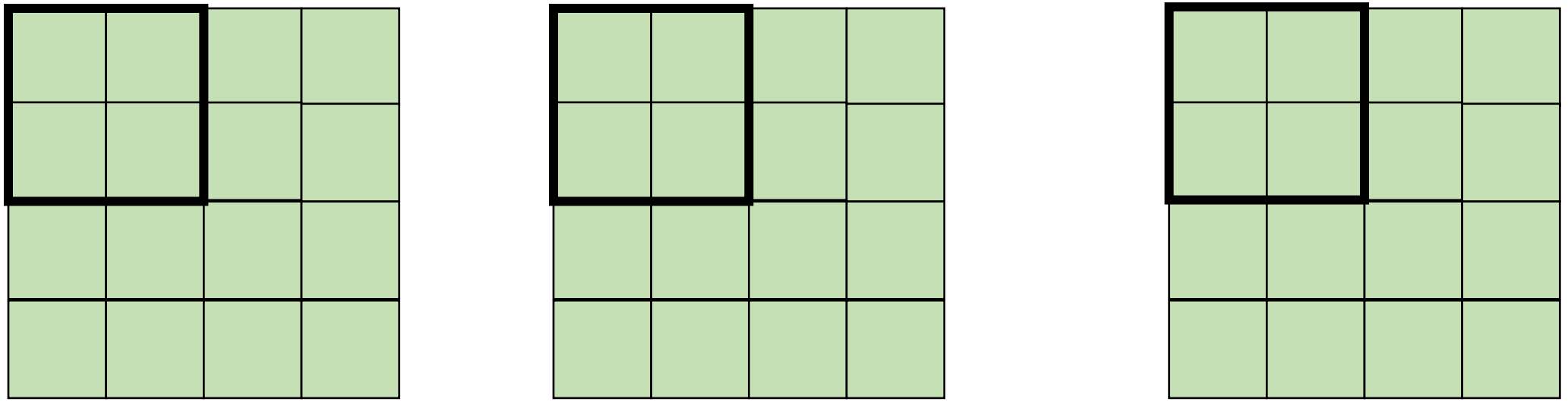


# blocking

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

$$A = B + C^T$$

$A$ $B$ $C$

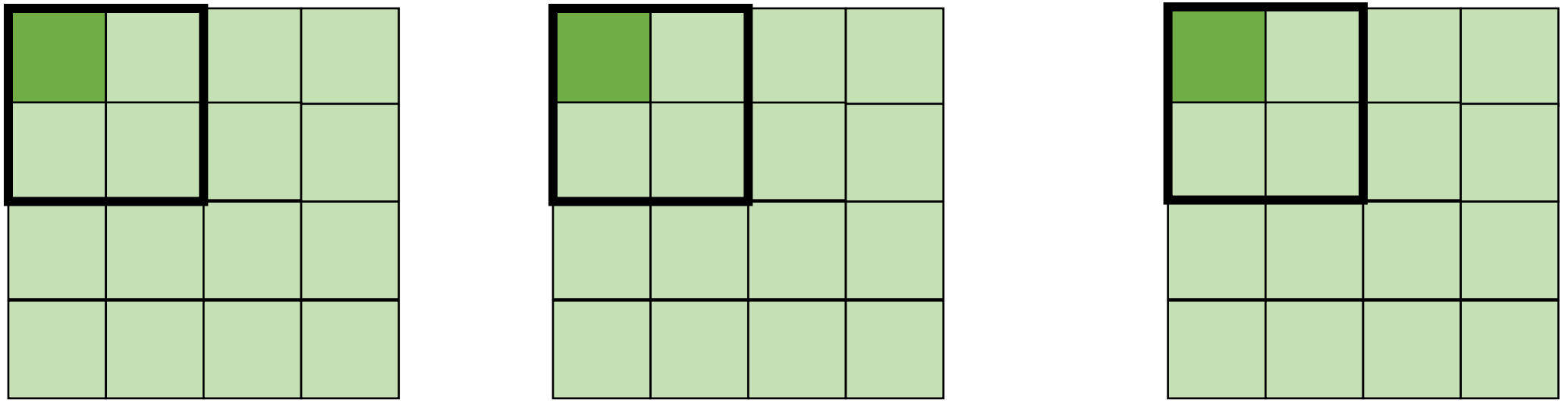


# blocking

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

$$A = B + C^T$$

$A$ $B$ $C$



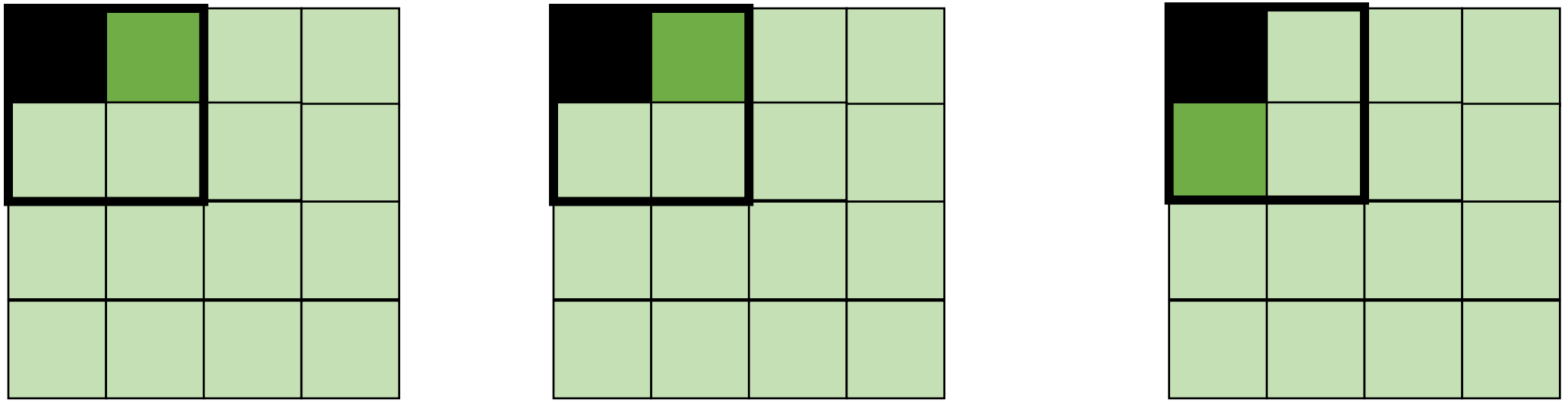
*cold miss for all of them*

# blocking

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

$$A = B + C^T$$

$A$ $B$ $C$



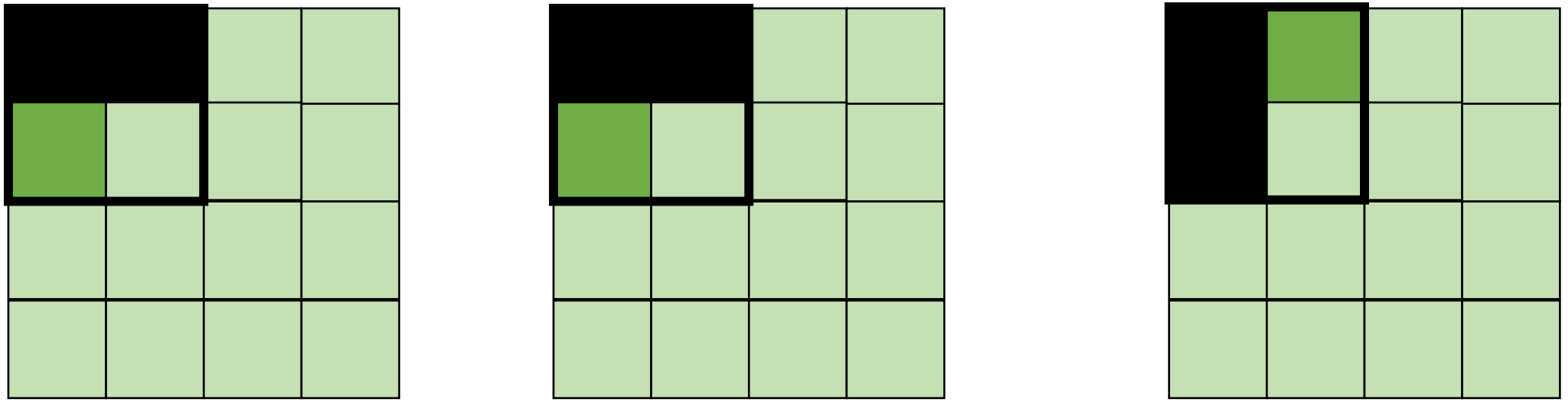
*Miss on C*

# blocking

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

$$A = B + C^T$$

$A$ $B$ $C$



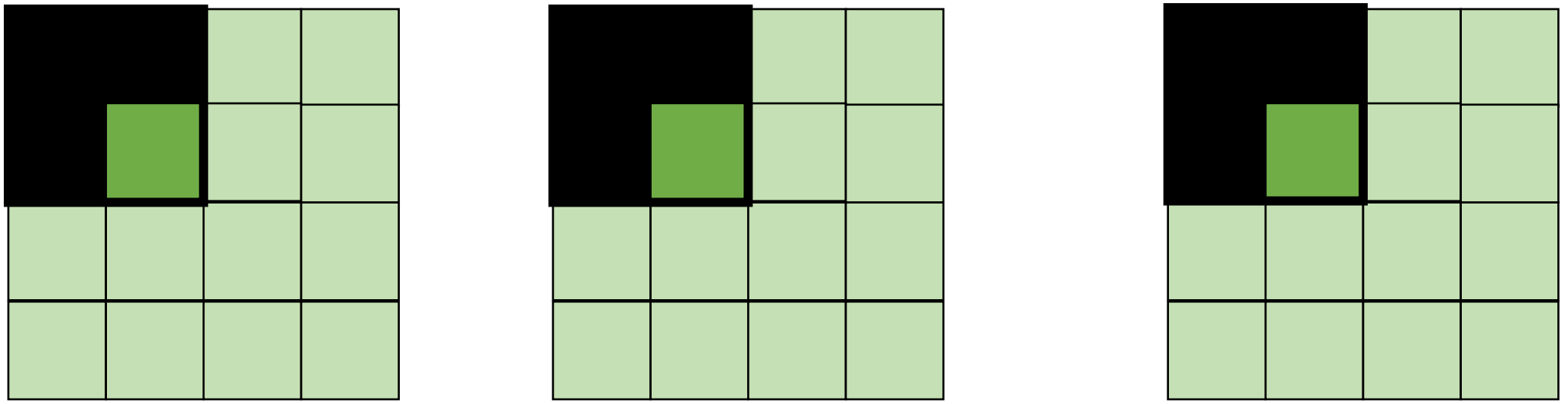
*Miss on A,B, hit on C*

# 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!*

# See everyone on Friday

- Control flow graphs