CSE110A: Compilers

June 3, 2024

Topics:

- *Wrapping up LVN*
- *Advanced Loop Optimizations*

Announcements

- HW 5 is out
  - Due on Friday! Please get started on it!
- Most of the grades for HW 3 are in
  - Graders have said that they will have things ready by EOD
- Come see us in office hours for homework help (or anything else!)
- No more quizzes for the rest of the quarter

Announcements

- Final Exam
  - In Person
  - Monday June 10: Noon – 3 PM
  - 3 pages of notes (front and back)
  - Like the midterm
    - Designed to be 2x as long, but final has 3x time.
    - 4 questions instead of 3
  - Comprehensive, slightly more weight to last part of class

Topics to study for final

- **Module 1:** Token definitions, Regular expressions, Scanner API, Scanner implementations.
- **Module 2:** Grammars (BNF Form), parse trees, ambiguous grammars (and how to fix them). Precedence, associativity (of the operators in your homework), Top down parsers
- **Module 3:** ASTs - how to create them, node types and members, modifications. Simple type systems, linearizing ASTs into 3 address code.
- **Module 4:** basic blocks, local value numbering, for loop analysis (loop unrolling).

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];
i ++;
a[i] = b[i] + c[i];
i ++;
a[i] = b[i] + c[i];
i ++;
a[i] = b[i] + c[i];
i ++;
}
```

```
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;
}
```

Quiz

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

- ◯ True
- ◯ False

Quiz

Many 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)

Advanced Local Value Numbering

Local value numbering: Memory

- Consider a 3 address code that allows memory accesses

```
a[i] = x[j] + y[k];
b[i] = x[j] + y[k];
```

Not allowed!

```
a[i] = x[j] + y[k];
b[i] = a[i];
```

Example, initially:

```
i = j
a = x
y[k] = 1
x[j] = 1
```

What does b[i] equal at the end of each computation?

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair

```
a[i] = x[j] + y[k];
b = x[j] + y[k];
```

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b,6) = (x[j],?) + (y[k],?);
```

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b,6) = (x[j],4) + (y[k],5);
```

Does this help at all?

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b,6) = (x[j],4) + (y[k],5);
(c,7) = (x[j],4) + (y[k],5);
```

Does this help at all?

If there is no memory writes between an assignment to a variable then we can do a replacement

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b,6) = (x[j],4) + (y[k],5);
(c,7) = (b,6);
```

Does this help at all?

If there is no memory writes between an assignment to a variable then we can do a replacement

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

A compiler analysis might try to determine that addresses can't alias

can we trace `a,x,y` to

```
a = malloc(…);
x = malloc(…);
y = malloc(…);
```

// `a,x,y` are never overwritten

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],1) + (y[k],2);
```

in this case we do not have to update the number

A compiler analysis might try to determine that addresses can't alias

can we trace `a,x,y` to

```
a = malloc(…);
x = malloc(…);
y = malloc(…);
```

// `a,x,y` are never overwritten

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

in this case we do not have to update the number

```
restrict a
```

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Warning: the compiler does not enforce this!

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (a[i],3);
```

Local value numbering: functions

Local value numbering: functions

How to number?

```
a = foo(x);
x = b;
c = foo(x);
```

Local value numbering: functions

How to number?

```
a = foo(x);
x = b;
c = foo(x);
```

the same way

```
a1 = foo(x0);
x3 = b2;
c4 = foo(x3);
```

Local value numbering: functions

How to number?

```
a = foo(x);
x = b;
c = foo(x);
```

the same way

```
a1 = foo(x0);
x3 = b2;
c4 = foo(x3);
```

Can we replace?

Local value numbering: functions

How to number?

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

the same way

```
a1 = foo(x0);
c2 = foo(x0);
```

How about now?

Local value numbering: functions

How to number?

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

the same way

```
a1 = foo(x0);
c2 = foo(x0);
```

How about now?

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

What if foo had this implementation?

Local value numbering: functions

How to number?

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

the same way

```
a1 = foo(x0);
c2 = foo(x0);
```

How about now?

side effects!

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

What if foo had this implementation?

Local value numbering: functions

```
a = foo(x);
c = foo(x);
print(count);
```

```
a1 = foo(x0);
c2 = a1;
print(count);
```

are these two programs the same?

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

Local value numbering: functions

- In C/++, functions are assumed to have side effects

- A function that does not have side effects is called “pure”
  - You can annotate a function as pure
  - `__attribute__((pure))`
  - warning: compiler does not check this and you can introduce subtle bugs

- Functional languages tend to have a pure-by-default design. Allows more compiler optimizations, but less control to the programmer.

Advanced loop optimizations

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.

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;
}
```

DOALL loops

- Very difficult for a C++ style 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 constraints and optimize more aggressively.

DSL example

Image processing:

pretty straight
forward computation
for brightening

(1 pass over all pixels)

Taken from Halide:
A DSL project out of MIT

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/

Image processing:

pretty straight forward computation for brightening

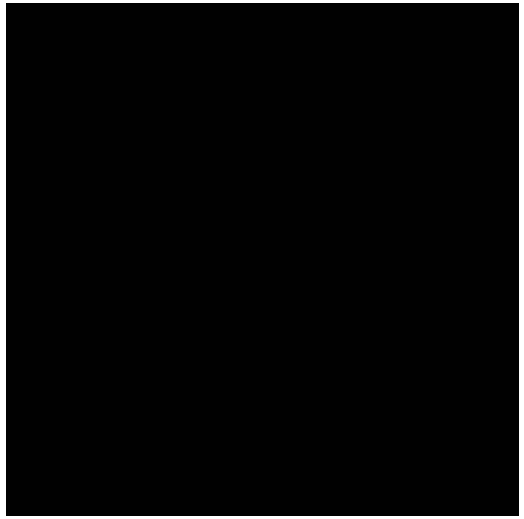
(1 pass over all pixels)

Taken from Halide:
A DSL project out of MIT

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

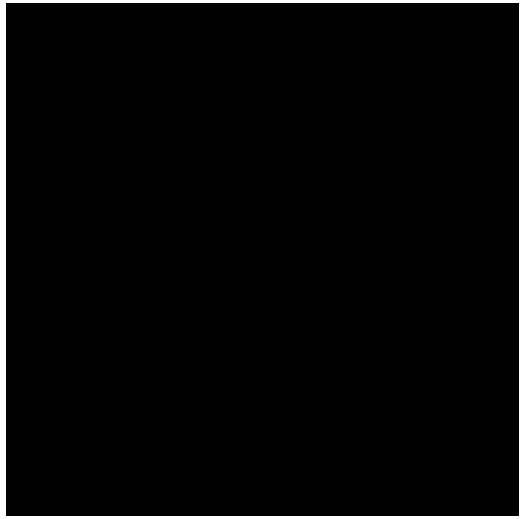
from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/



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

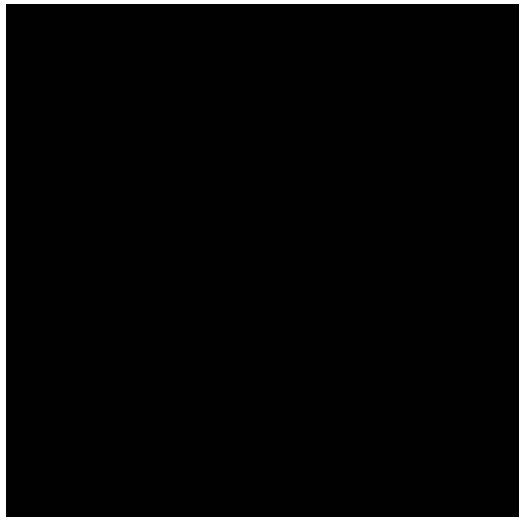
you can compute the pixels in any order you want, you just have to compute all of them!

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?

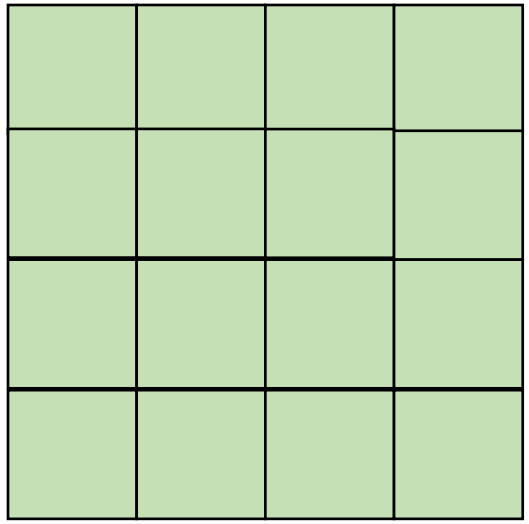
Adding 2D arrays together

Demo

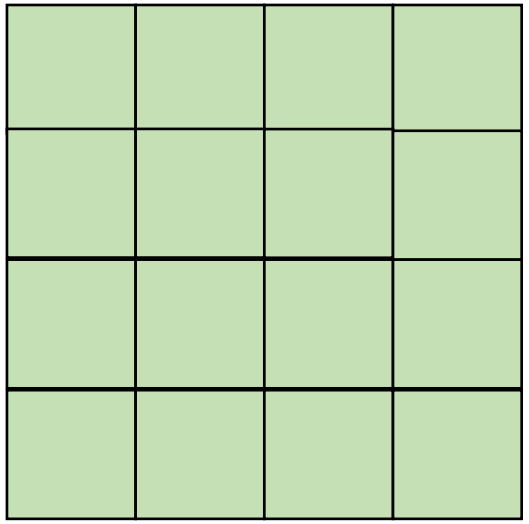
- Memory accesses

$$A = B + C$$

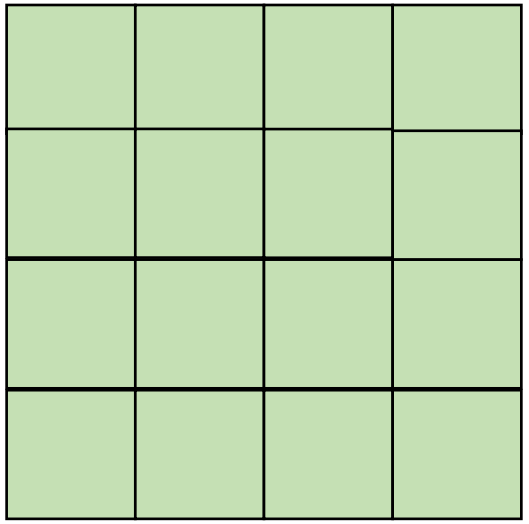
A



B



C



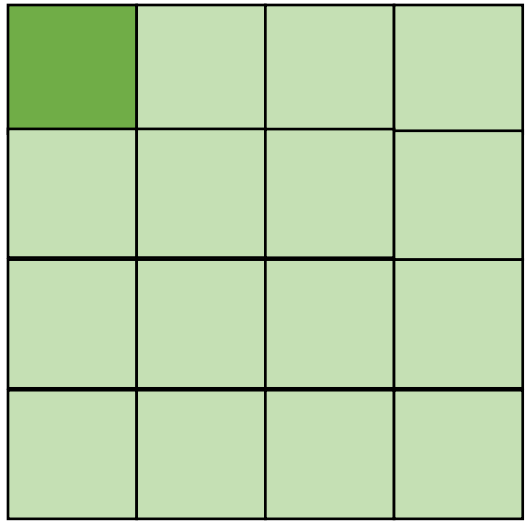
Adding 2D arrays together

Demo

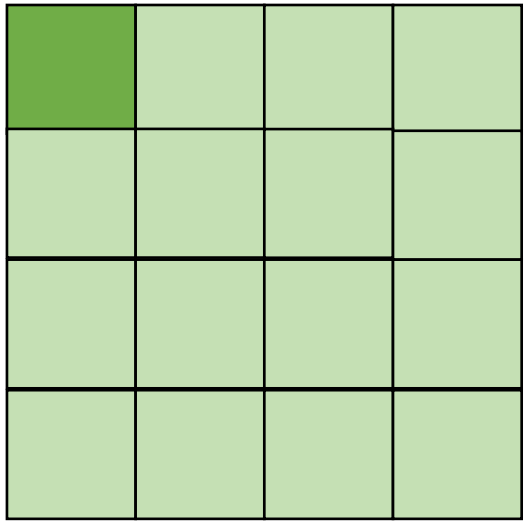
- Memory accesses

$$A = B + C$$

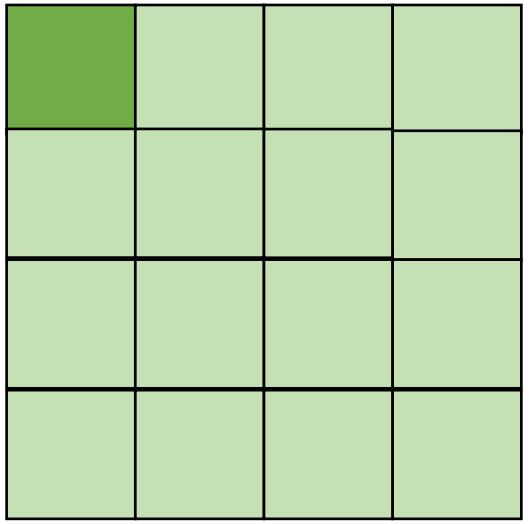
A



B



C



Cache miss for all of them

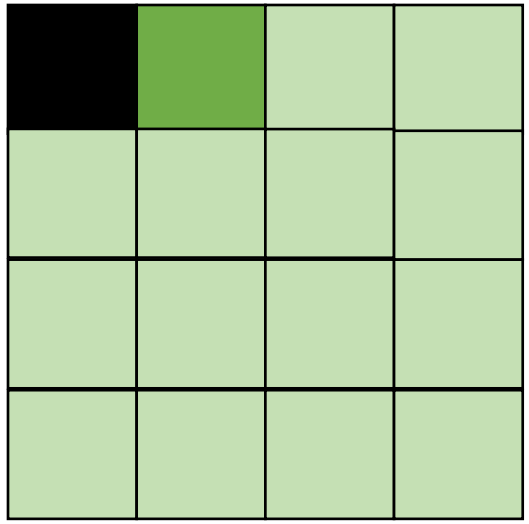
Adding 2D arrays together

Demo

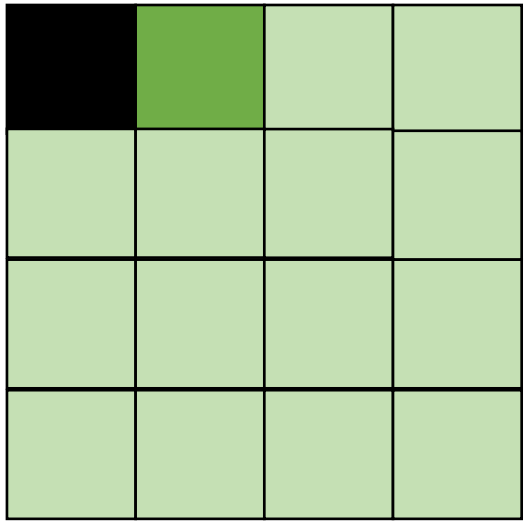
- Memory accesses

$$A = B + C$$

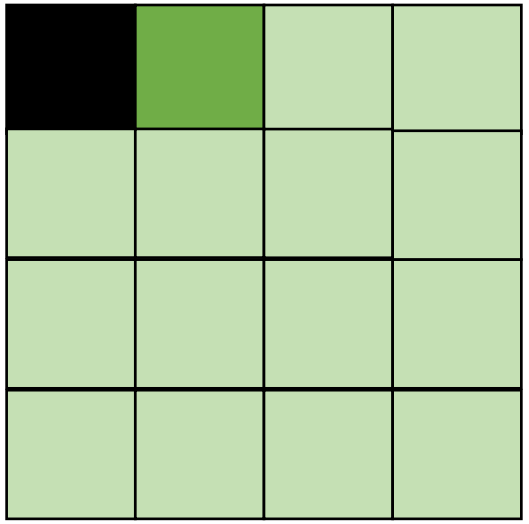
A



B



C



Cache HIT for all of them

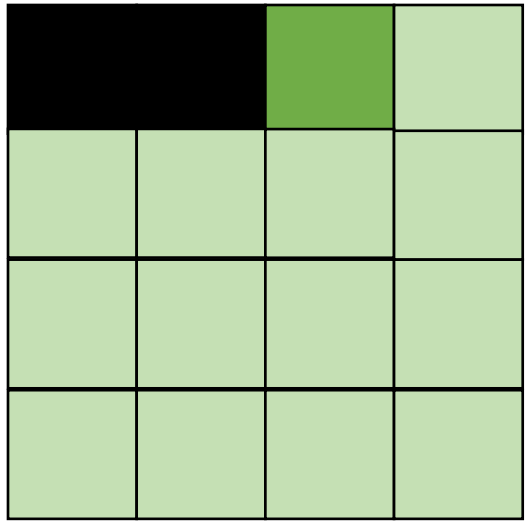
Adding 2D arrays together

Demo

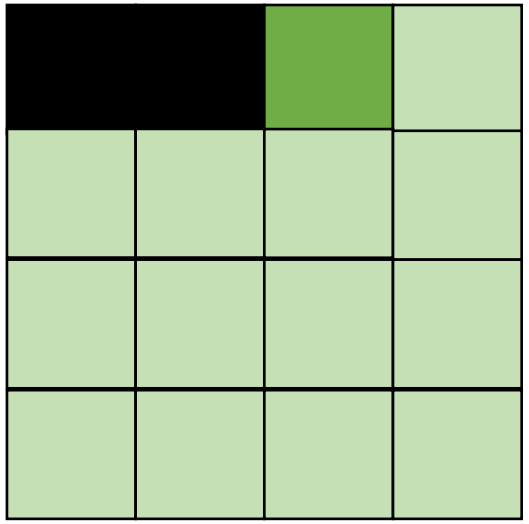
- Memory accesses

$$A = B + C$$

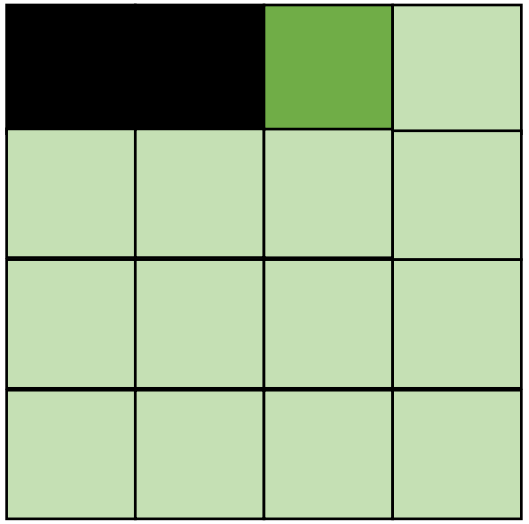
A



B



C



Cache HIT for all of them

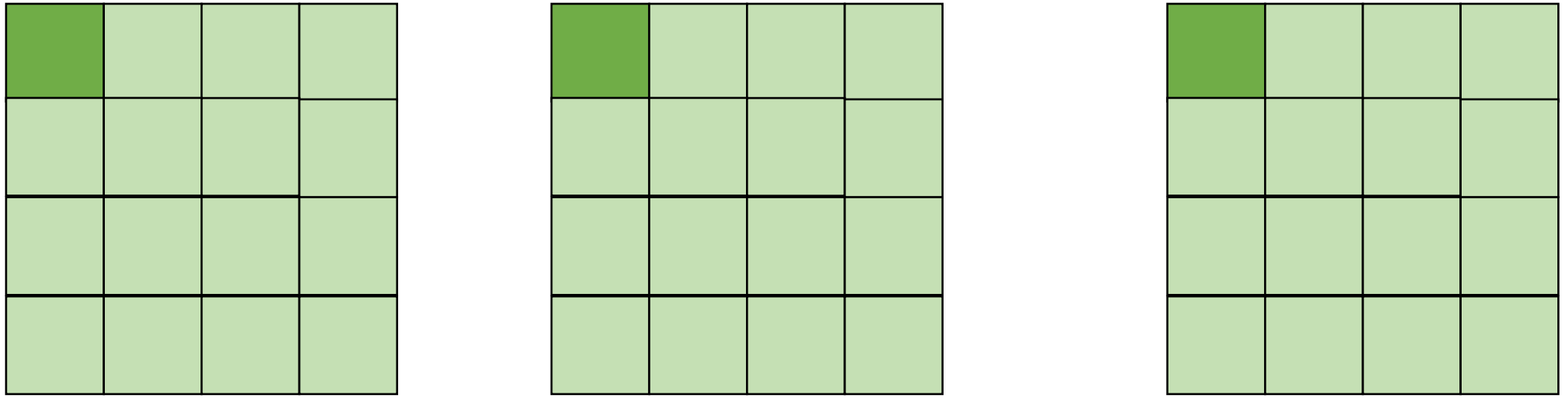
Adding 2D arrays together

Demo

- Memory accesses

$$A = B + C$$

A B C



Rewind! Cache miss for all of them

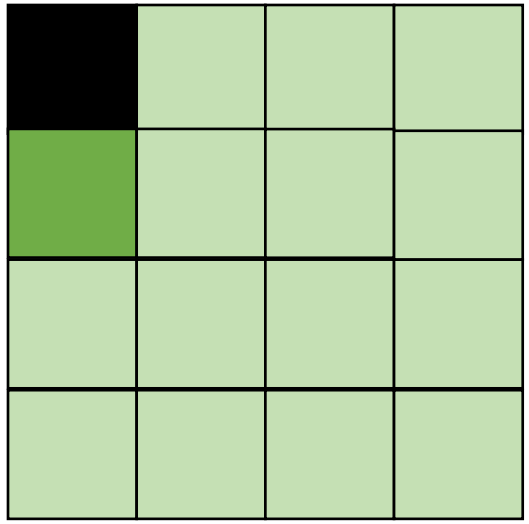
Adding 2D arrays together

Demo

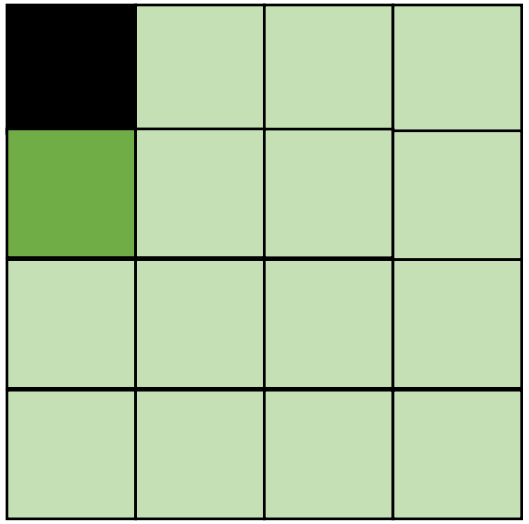
- Memory accesses

$$A = B + C$$

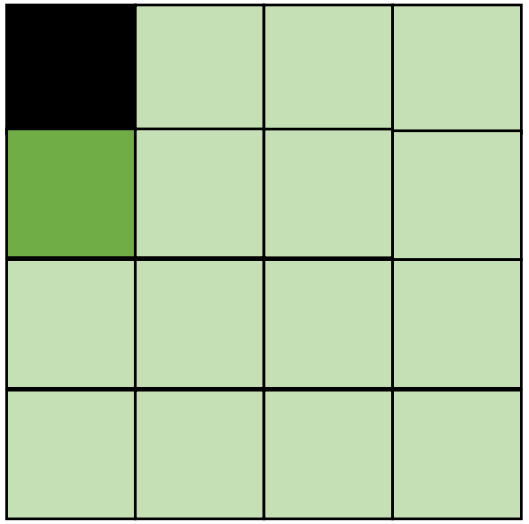
A



B



C



Rewind!

Cache miss for all of them

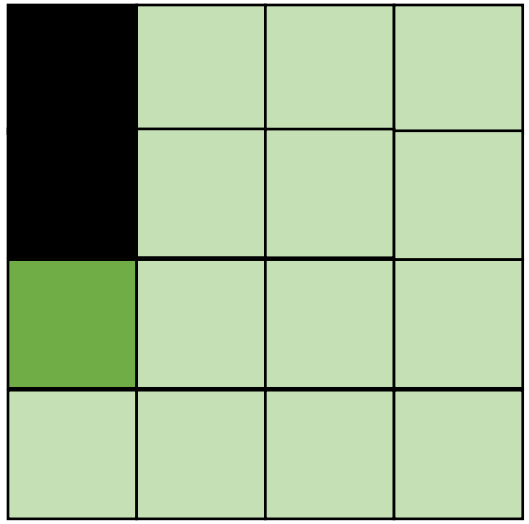
Adding 2D arrays together

Demo

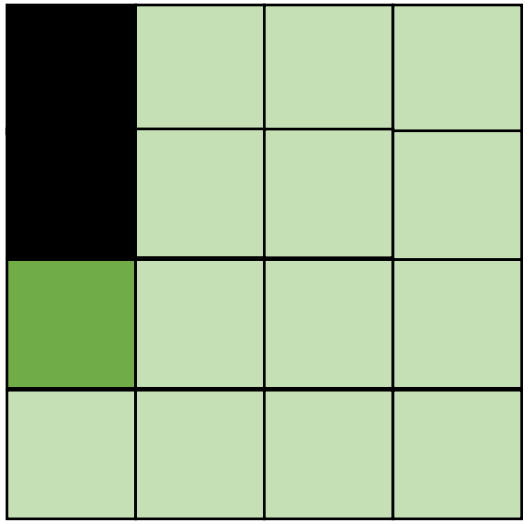
- Memory accesses

$$A = B + C$$

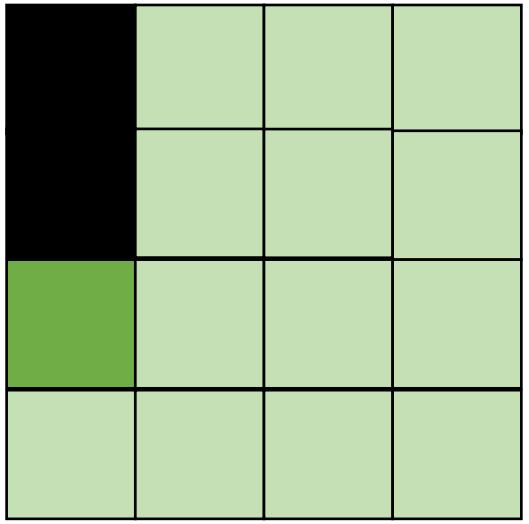
A



B



C



Rewind!

Cache miss for all of them

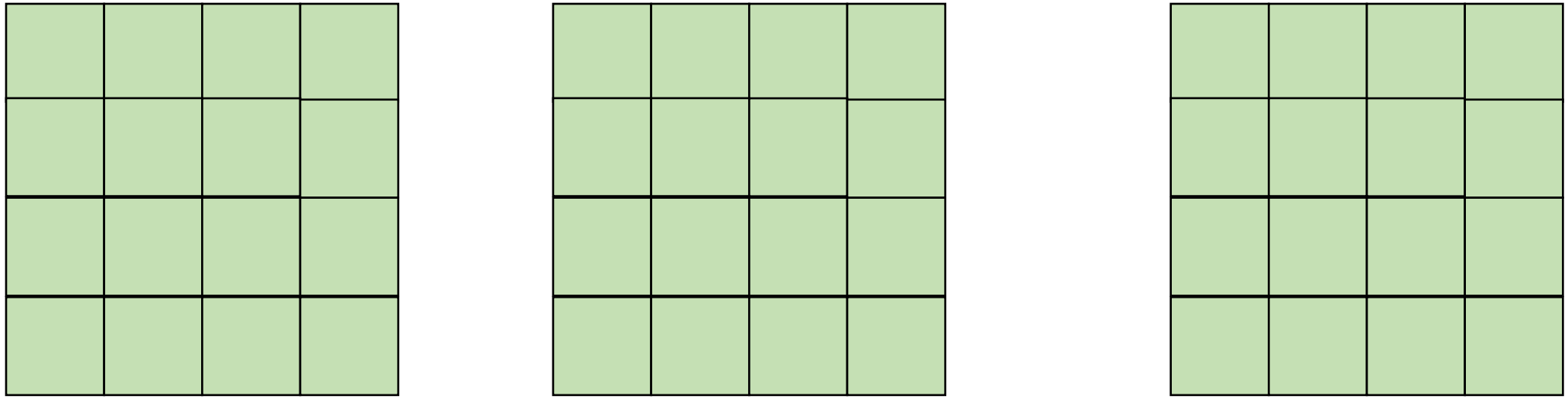
But sometimes there isn’t a good ordering

transposed arrays

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

$$A = B + C^T$$

A B C

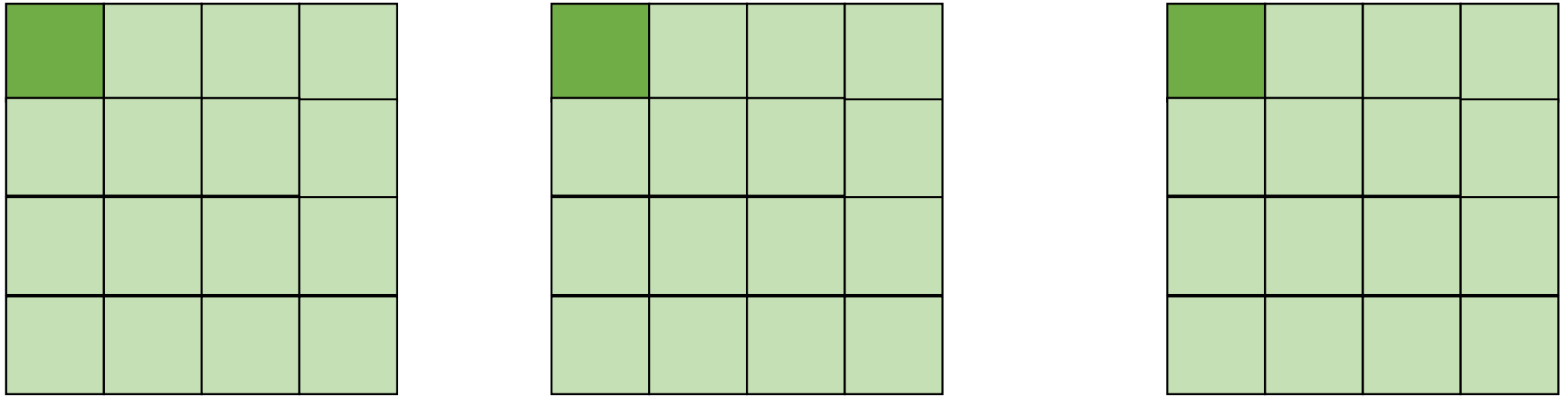


transposed arrays

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

$$A = B + C^T$$

A B C



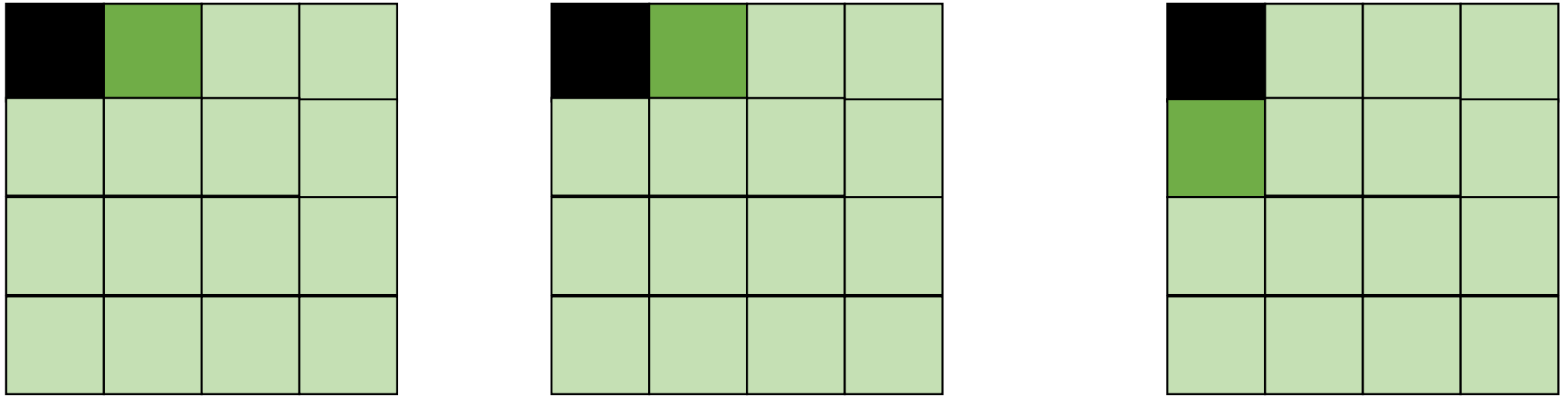
cold miss for all of them

transposed arrays

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

$$A = B + C^T$$

A B C



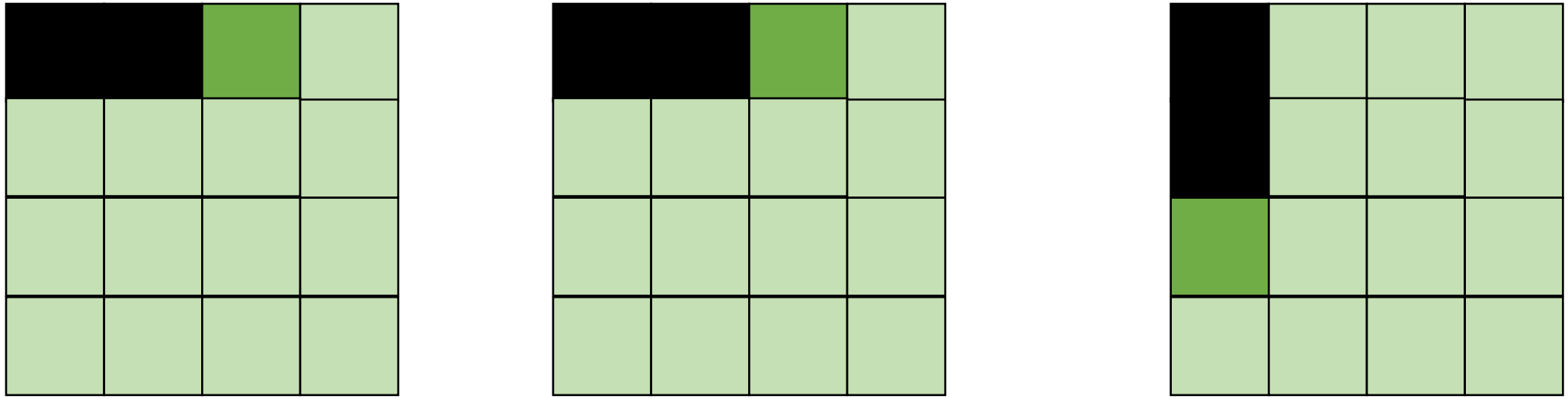
Hit on A and B. Miss on C

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?

```
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

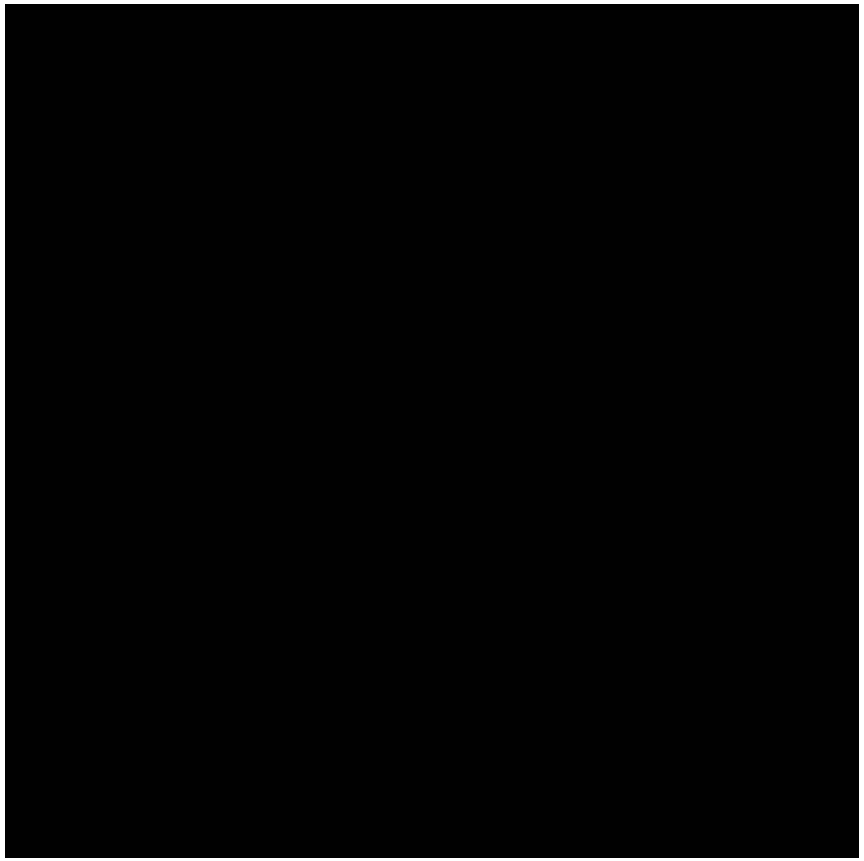
- Lets try chaining loop splitting and reorder
  - Demo

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

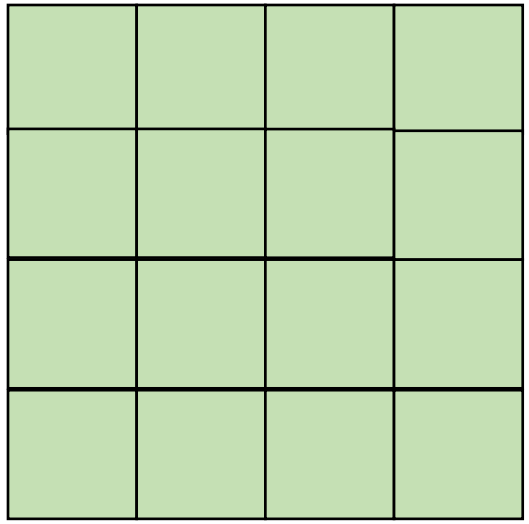
blocking

blocking

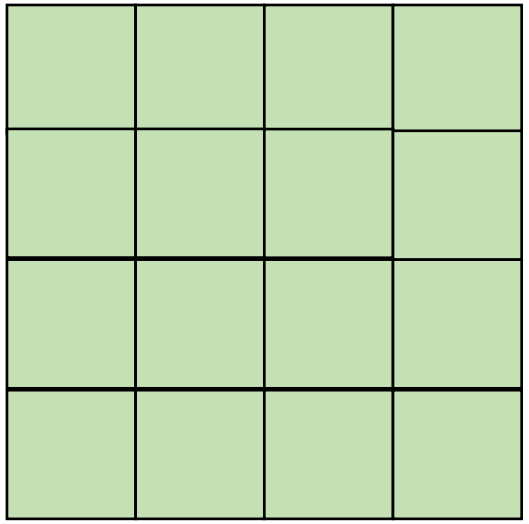
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

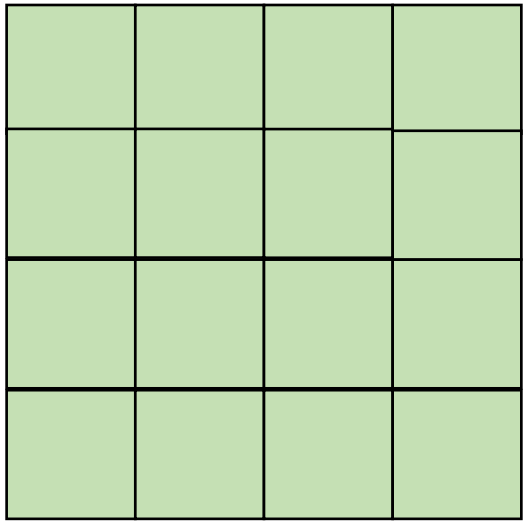
A



B



C

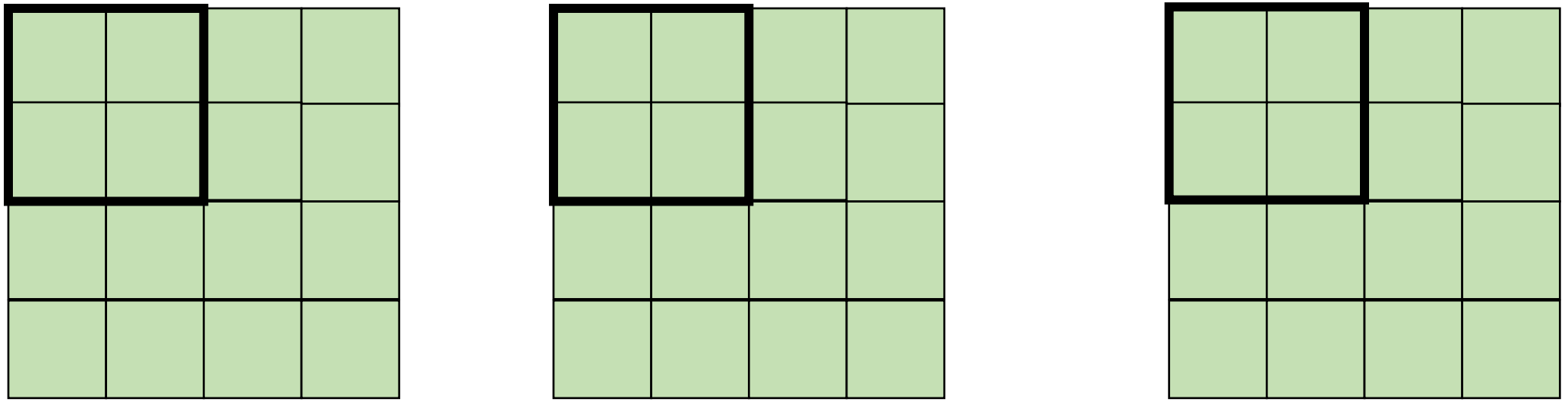


blocking

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

$$A = B + C^T$$

A B C

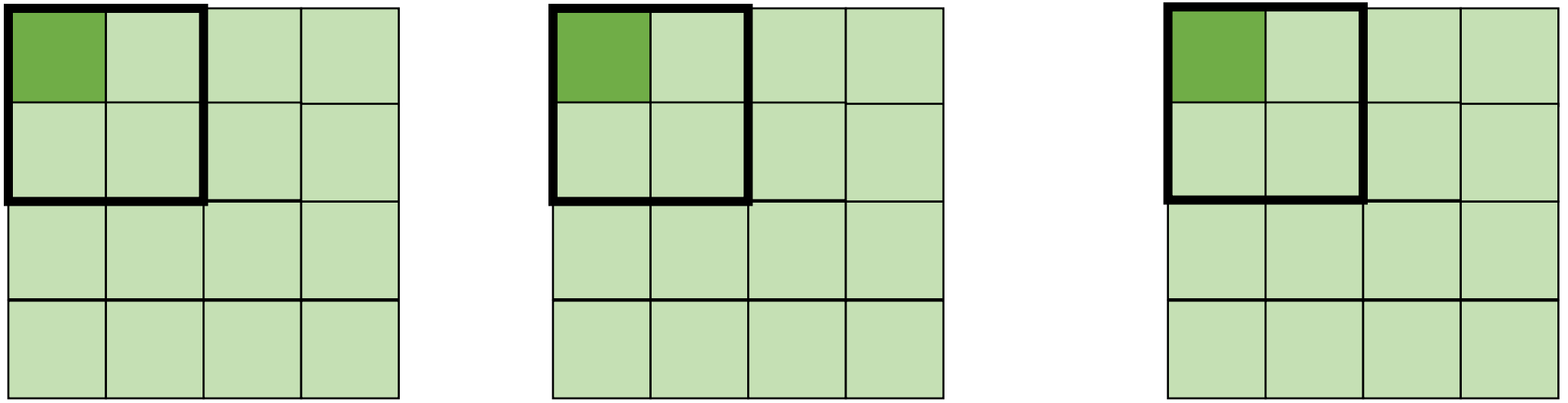


blocking

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

$$A = B + C^T$$

A B C



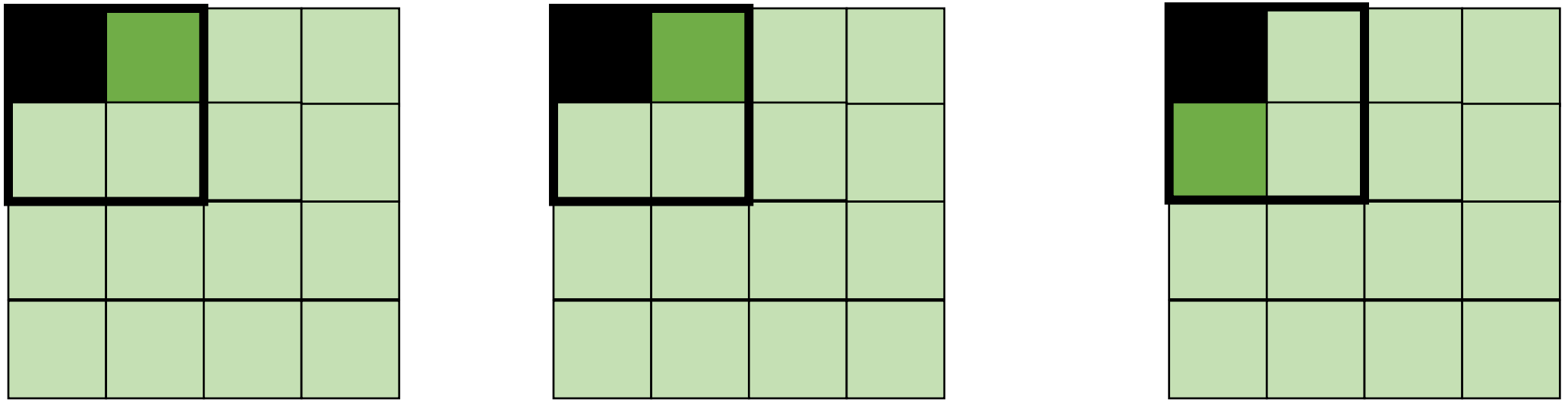
cold miss for all of them

blocking

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

$$A = B + C^T$$

A B C



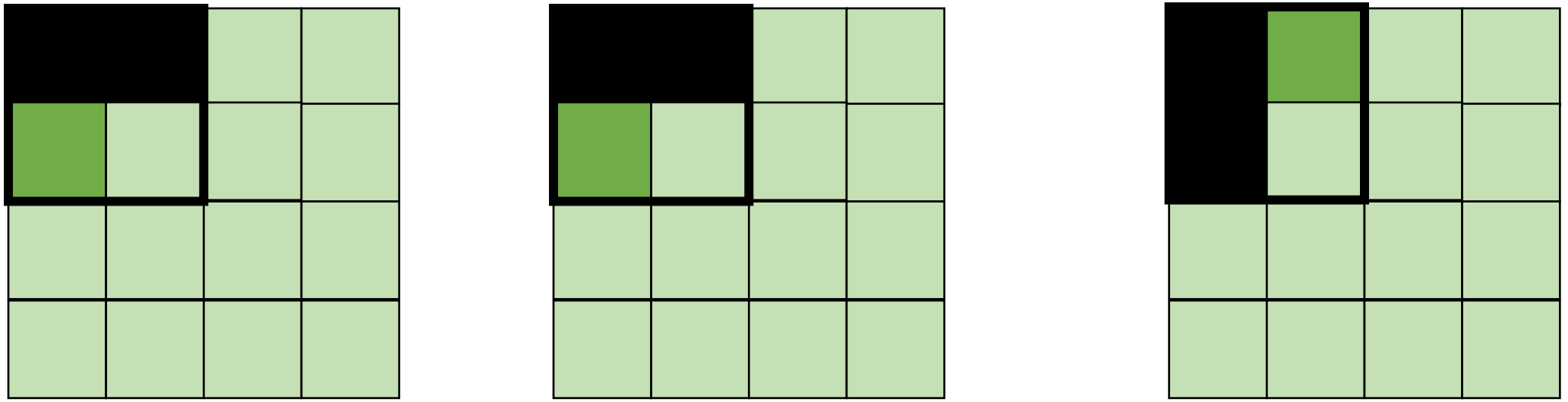
Miss on C

blocking

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

$$A = B + C^T$$

A B C



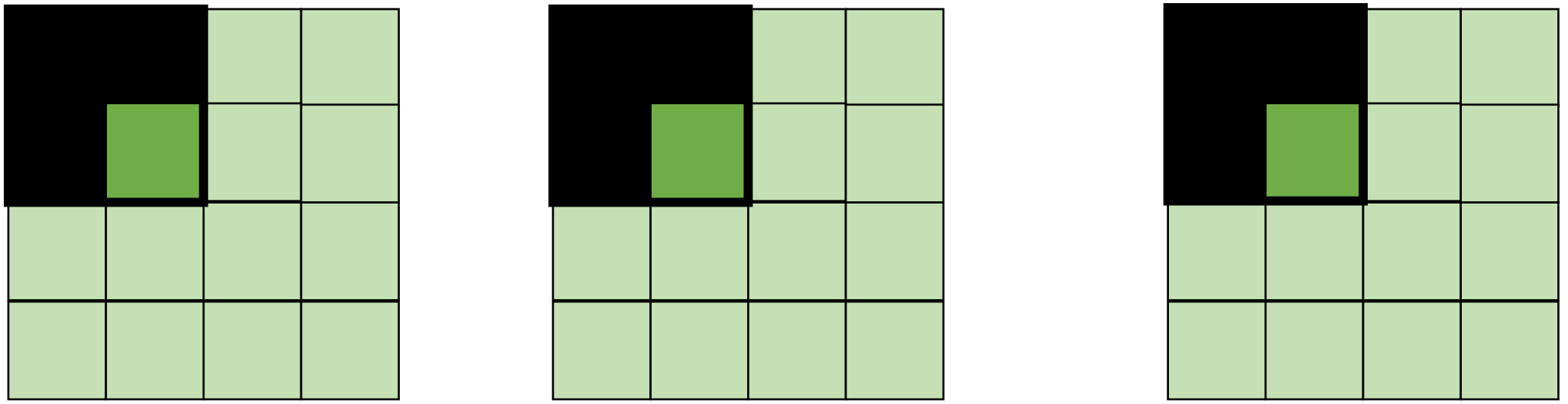
Miss on A,B, hit on C

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/