CSE110A: Compilers

May 27, 2022

Topics:

- *Homework overview*
- *More loop transforms*
- *Control flow graphs*

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

What is a good optimization that should immediately follow local value numbering?

- ◯ Constant propagation
- ◯ Copy propagation
- ◯ Loop unrolling
- ◯ Common sub expressions elimination

Discussion

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

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

There lots of copies here

Whole example:

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

```
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;
```

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

how many times i gets updated?

memory access optimizations

loop jamming (we will see next slide)

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

You can only unroll the outer for loop in the following program

```
for(i = 0; i < 4; i++){
for(j = 0; j < 4; j++){
  SOME_INSTRUCTION;
}
}
```

Discussion

```
for (i = 0; i < 4; i++){
  for (j = 0; j < 4; j++){
    SOME_INSTRUCTION;
  }
}
```

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

Discussion

```
for (i = 0; i < 4; i++){
  for (j = 0; j < 4; j++){
    SOME_INSTRUCTION;
  }
}
```

Lets think about how unrolling this 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...

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.

Review

- Quiz was a good review and we've spent enough time on local value numbering (on the slides at least)

Homework overview

- discussion and demo
- code structure (command line args)
- loop unrolling:
  - test 4 is the right place to start
- local value numbering
  - basic blocks
  - parsing classier
  - speedup demos

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;
}
```

Motivation:

Image processing

Taken from Halide:
A project out of MIT

pretty straight
forward computation
for brightening

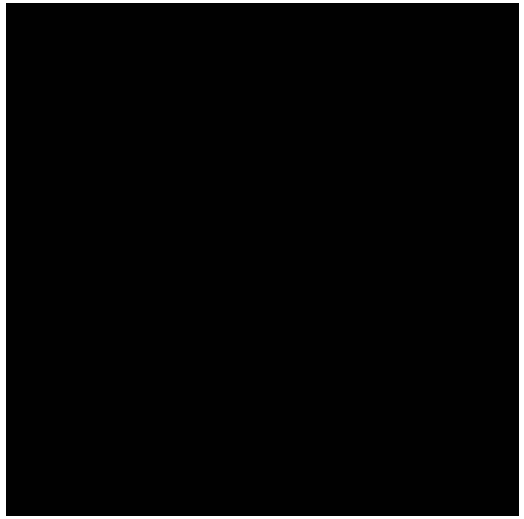
(1 pass over all pixels)

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

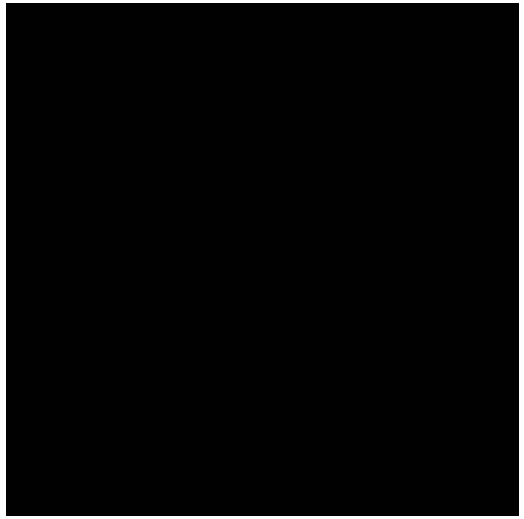
from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/



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

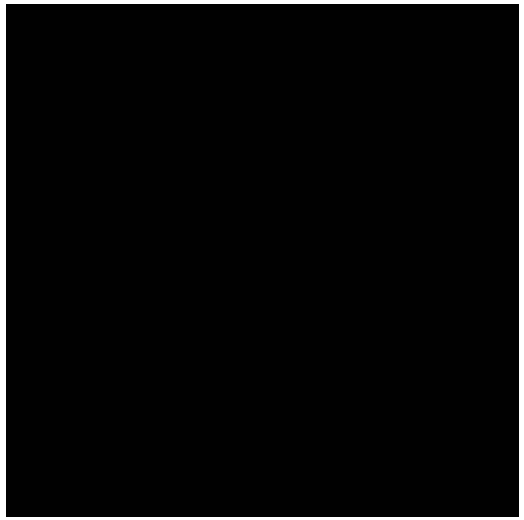
you can compute the pixels in any order you want, you just have to compute all of them!

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?

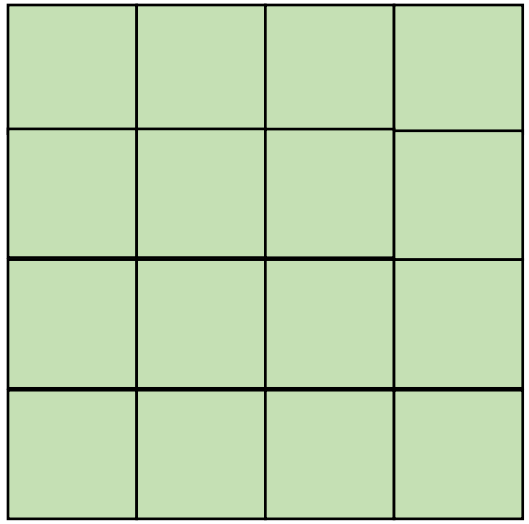
Adding 2D arrays together

Demo

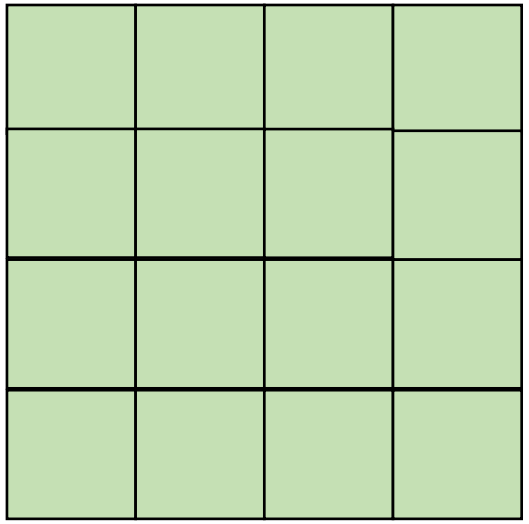
- Memory accesses

$$A = B + C$$

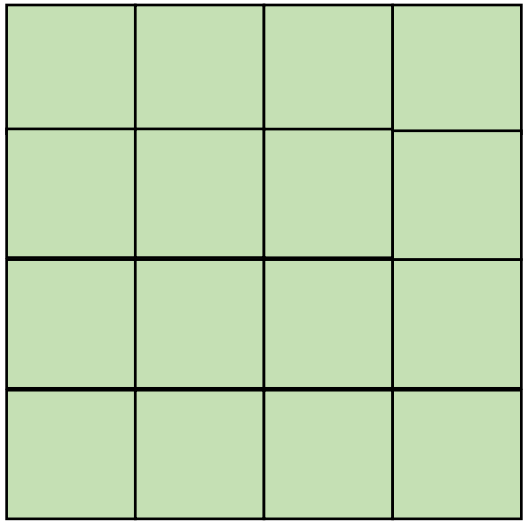
A



B



C



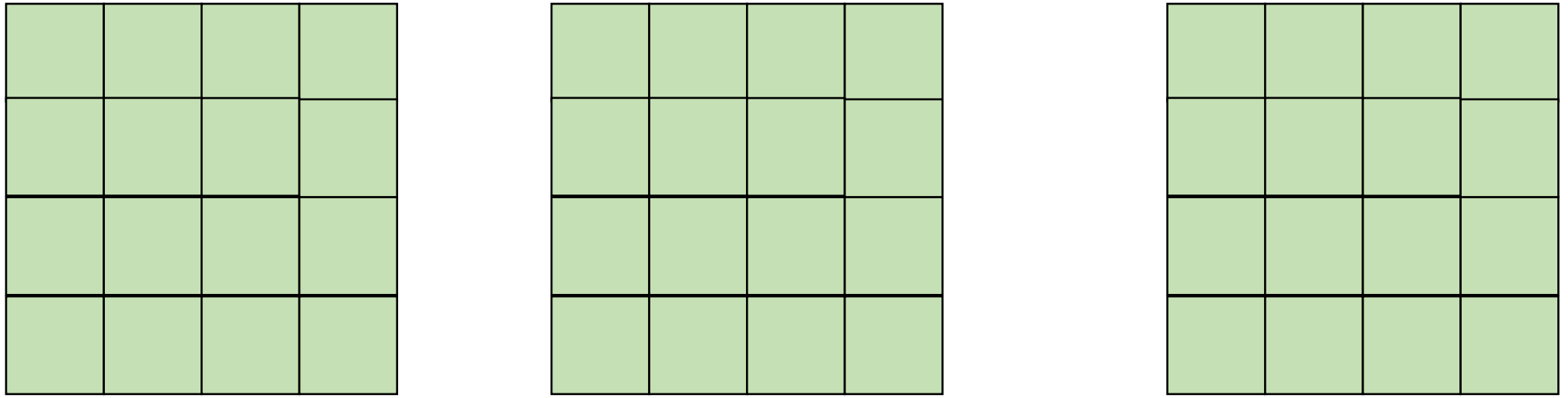
But sometimes there isn’t a good ordering

transposed arrays

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

$$A = B + C^T$$

A B C

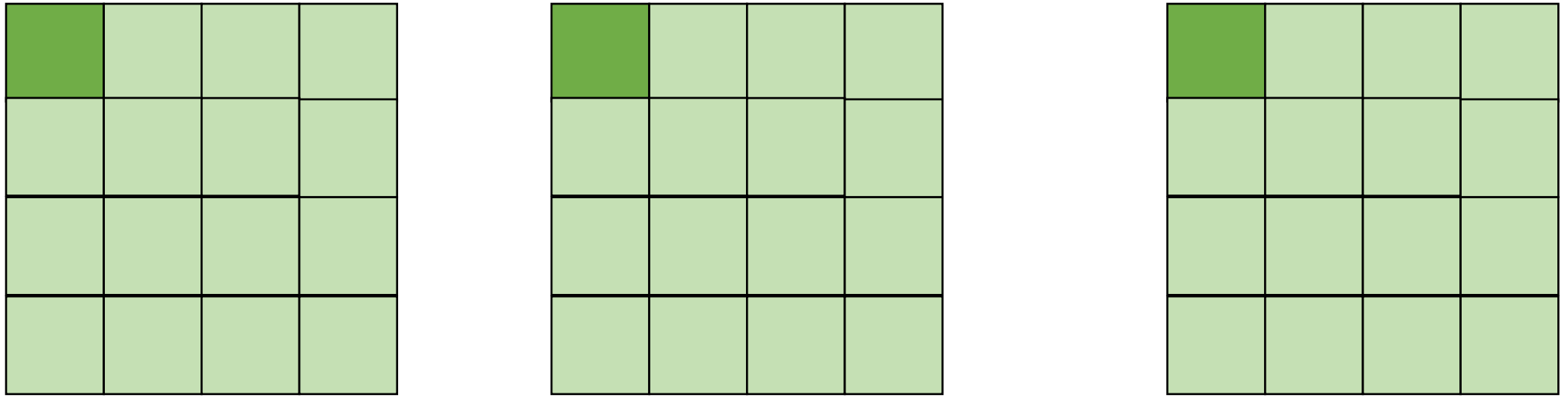


transposed arrays

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

$$A = B + C^T$$

A B C



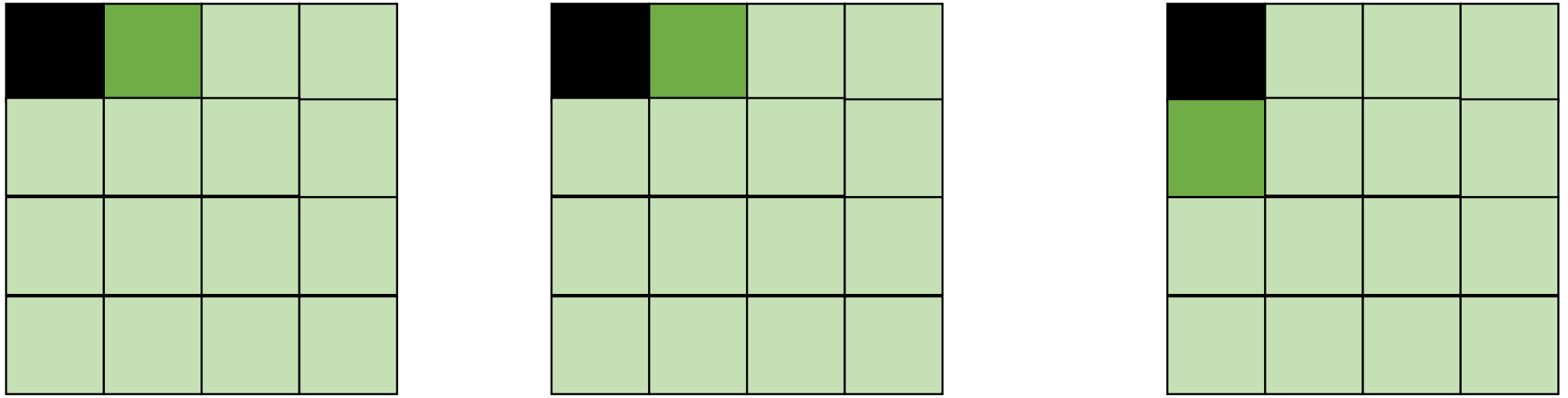
cold miss for all of them

transposed arrays

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

$$A = B + C^T$$

A B C



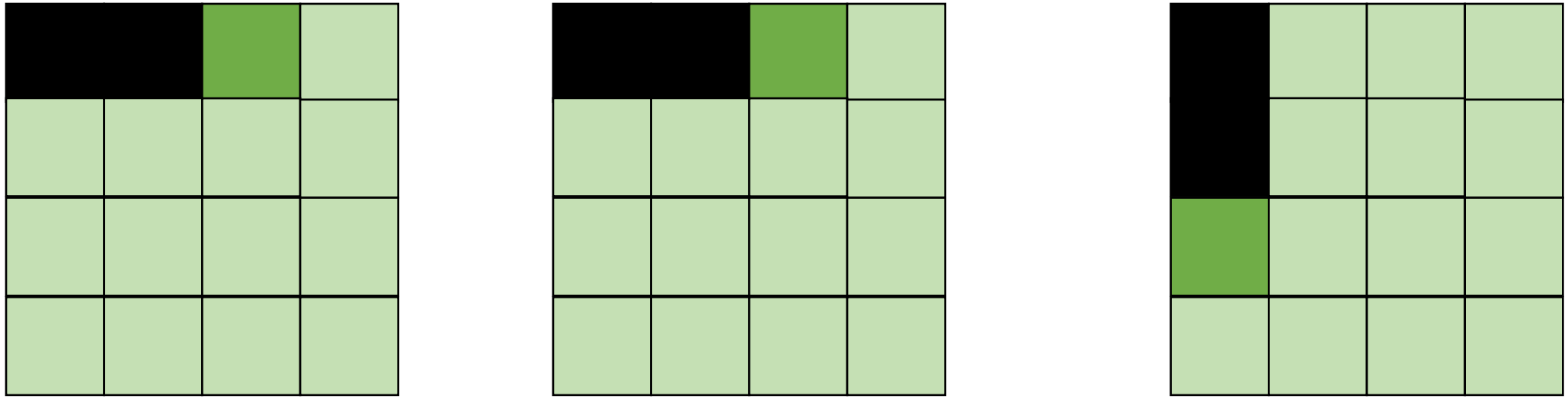
Hit on A and B. Miss on C

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

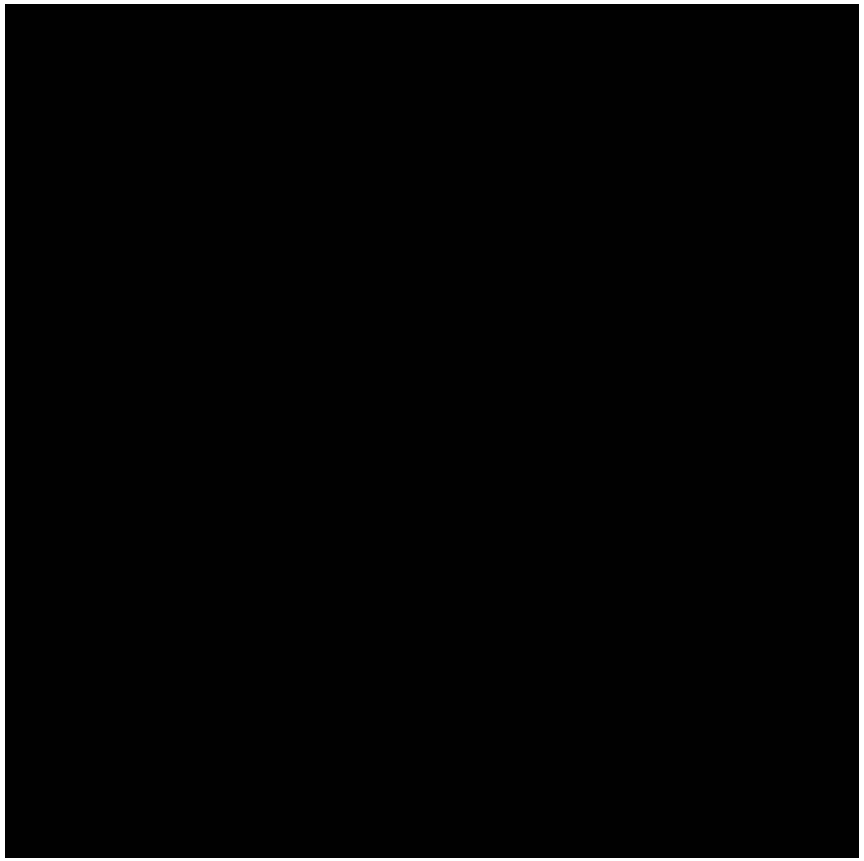
- Lets try chaining loop splitting and reorder
  - Demo

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

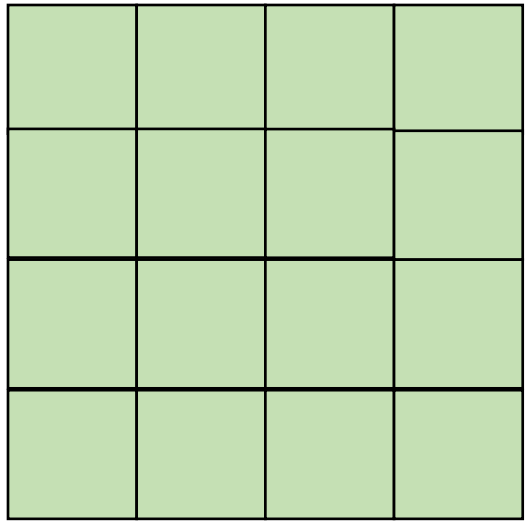
blocking

blocking

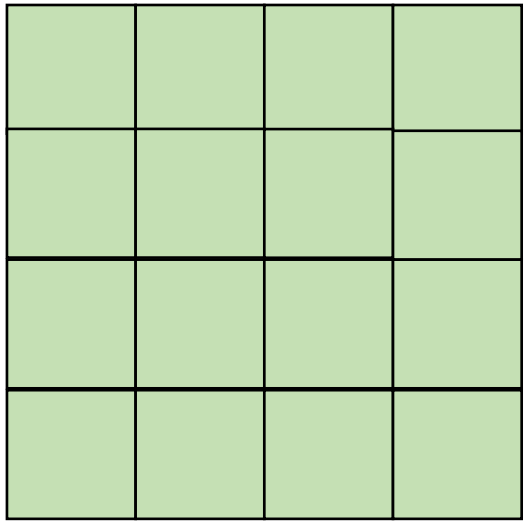
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

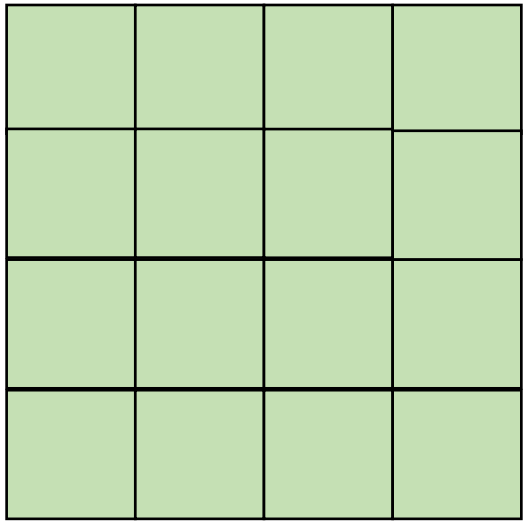
A



B



C

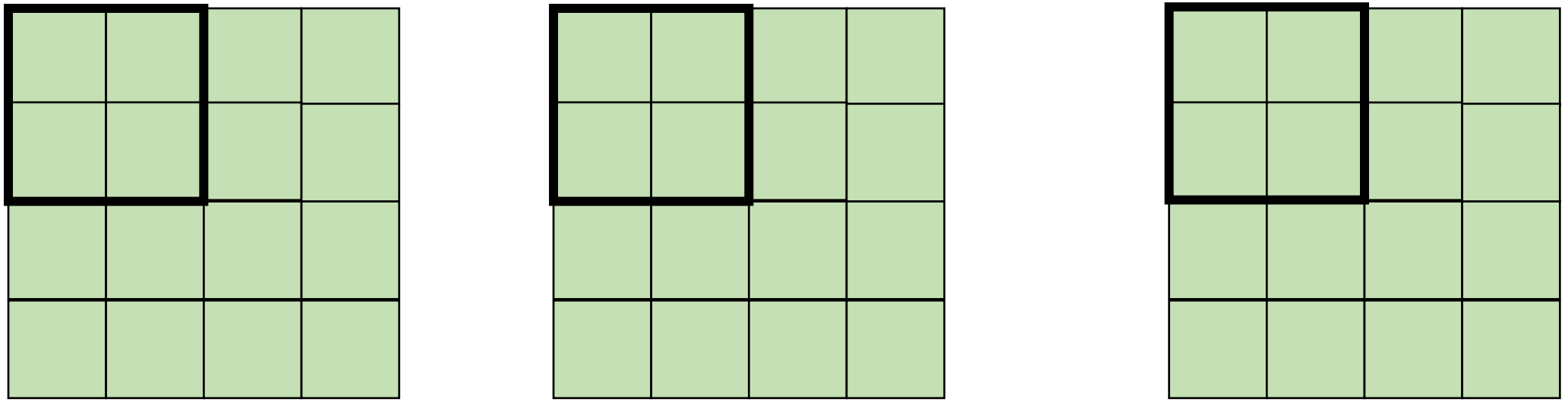


blocking

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

$$A = B + C^T$$

A B C

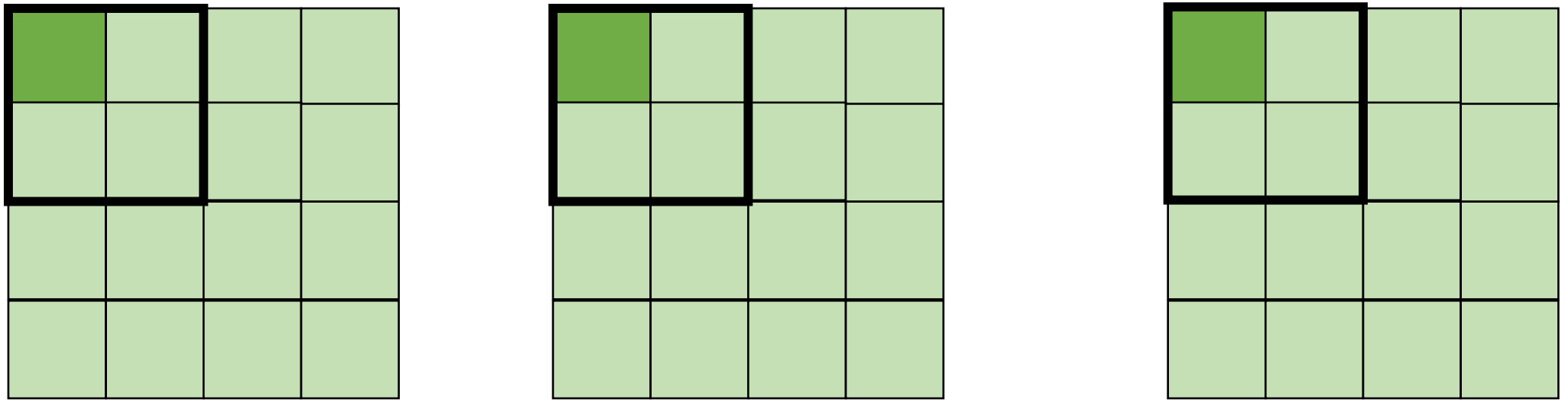


blocking

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

$$A = B + C^T$$

A B C



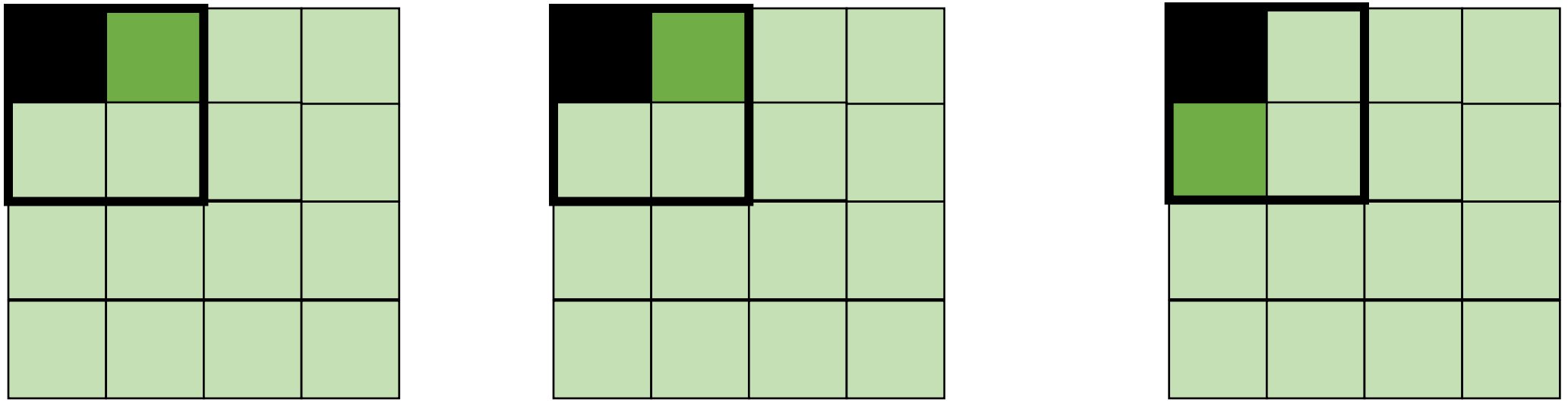
cold miss for all of them

blocking

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

$$A = B + C^T$$

A B C



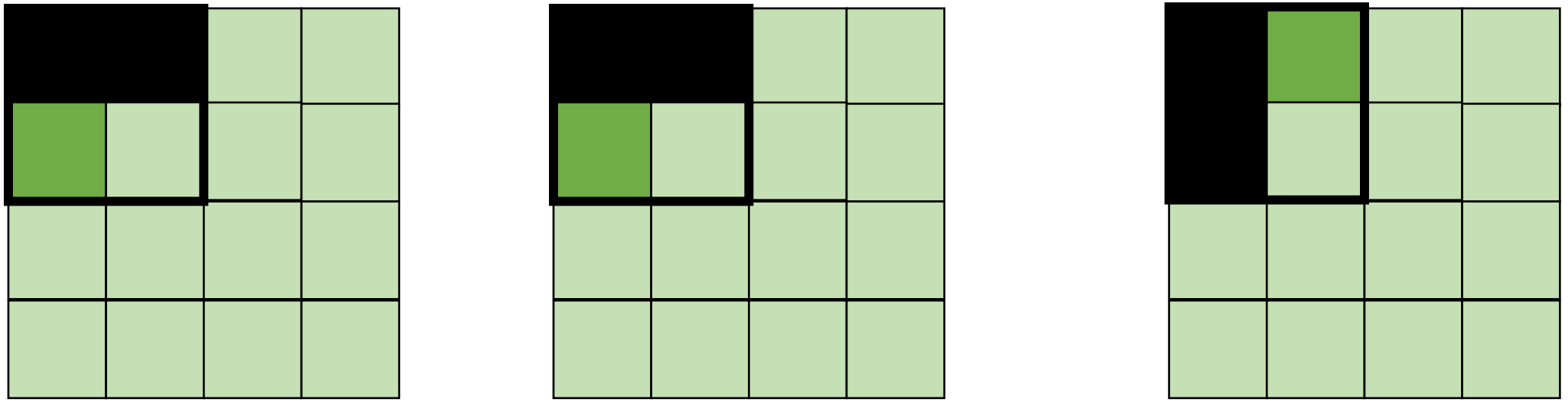
Miss on C

blocking

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

$$A = B + C^T$$

A B C



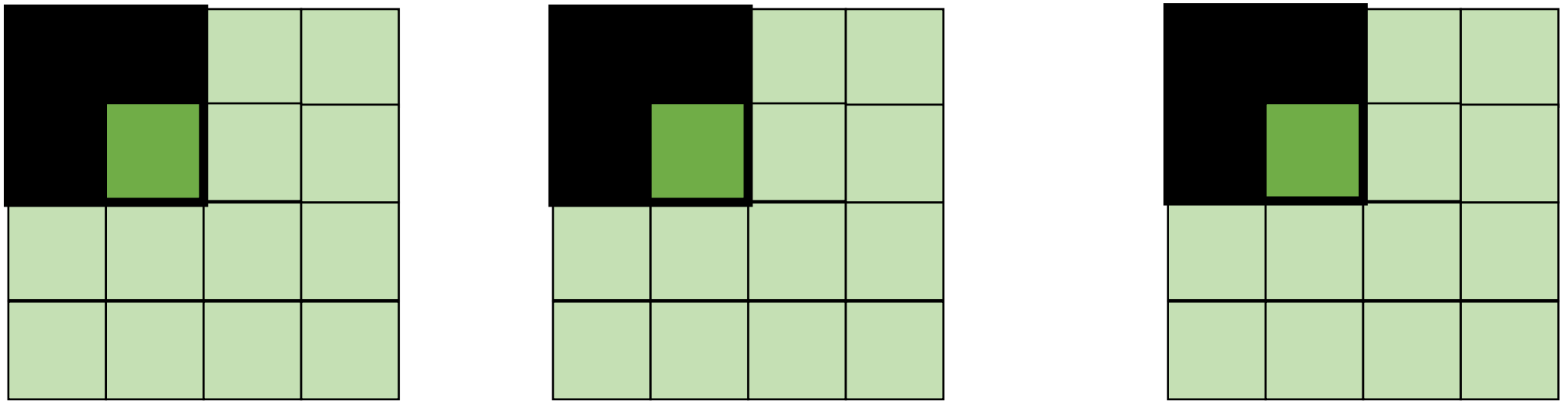
Miss on A,B, hit on C

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!

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/

Global Optimization (analysis)

- Loop transforms are a regional analysis
  - Compiler works hard to show that code fits a certain pattern
- Global analysis must account for arbitrary patterns
- Generality costs us! Lots of times these optimizations are not as effective or precise.
- But they can still help...

To finish up the class: Live variable analysis

- Not an analysis to make your code go faster
- An analysis to help warn programmers about potential bugs
- Optimizations that make code go faster are really fun but the reality i that programmers often spend ~70% of their time debugging and testing.
- Compilers can help!!

Control flow graphs

Control flow graphs

A graph where:

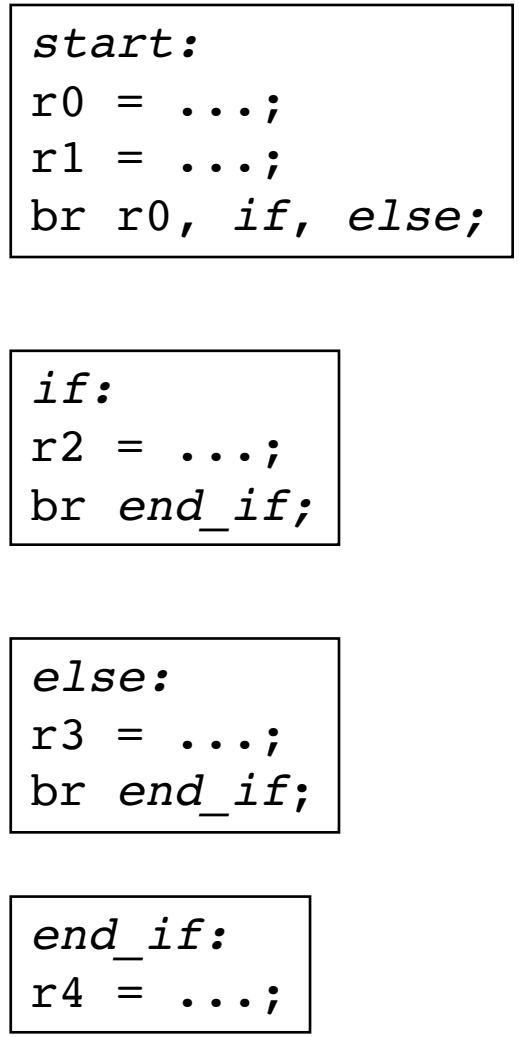
- nodes are basic blocks
- edges mean that it is possible for one block to branch to another

```
start:
r0 = ...;
r1 = ...;
br r0, if, else;

if:
r2 = ...;
br end_if;

else:
r3 = ...;

end_if:
r4 = ...;
```

Control flow graphs

A graph where:

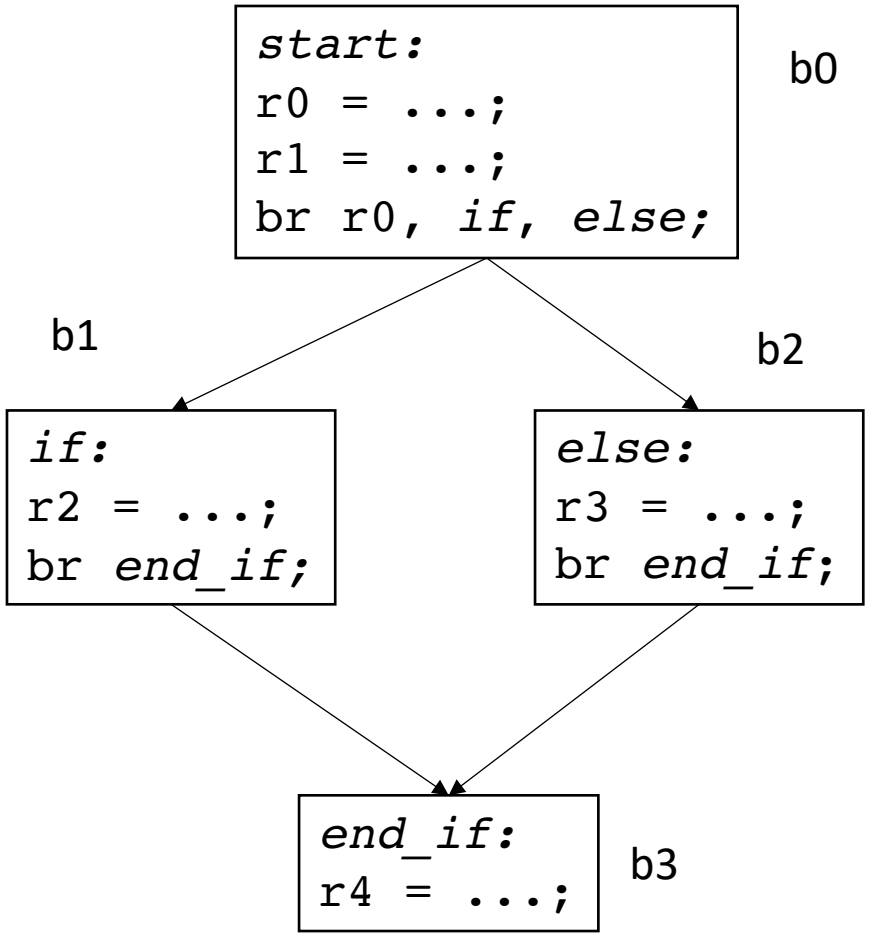
- nodes are basic blocks
- edges mean that it is possible for one block to branch to another



Control flow graphs

A graph where:

- nodes are basic blocks
- edges mean that it is possible for one block to branch to another



Interesting CFGs

Interesting CFGs

- Exceptions
- Break in a loop
- Switch statement (consider break, no break)
- first class branches (or functions)

CFG demo

- PyCFG

See everyone on Wednesday

- Control flow graphs