CSE110A: Compilers

June 3, 2024

Topics:

- *Advanced Loop Optimizations*
- *Intro to instruction scheduling*

Announcements

- HW 5 is due on Friday
  - Hope you’ve made good progress on it.
  - Several office hours remaining from TAs, tutors, and me.

- We are doing our best on grading. HW 3 grades should be released imminently.

- 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).

More 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];
}
```

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];
}
```

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

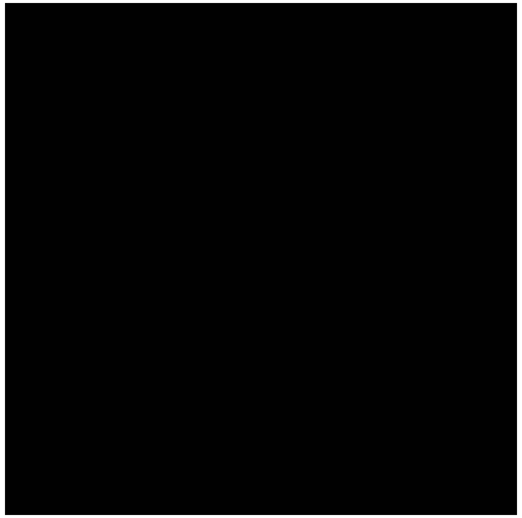
(1 pass over all pixels)

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

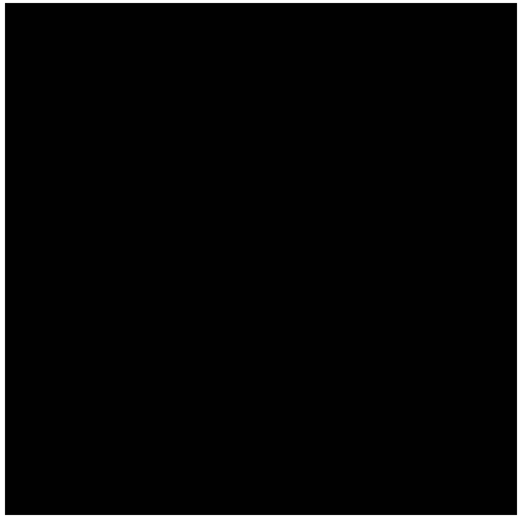
from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/



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

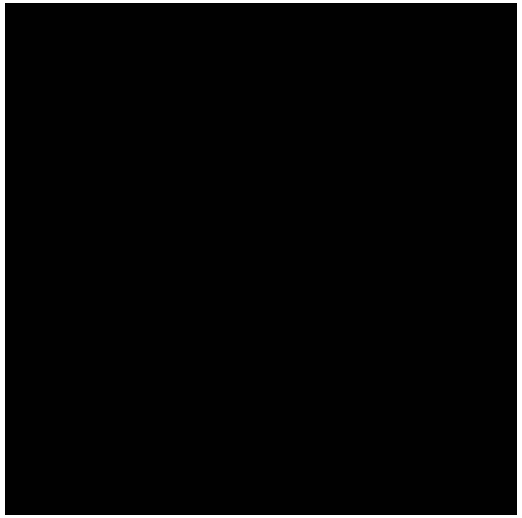
you can compute the pixels in any order you want, you just have to compute all of them!

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?

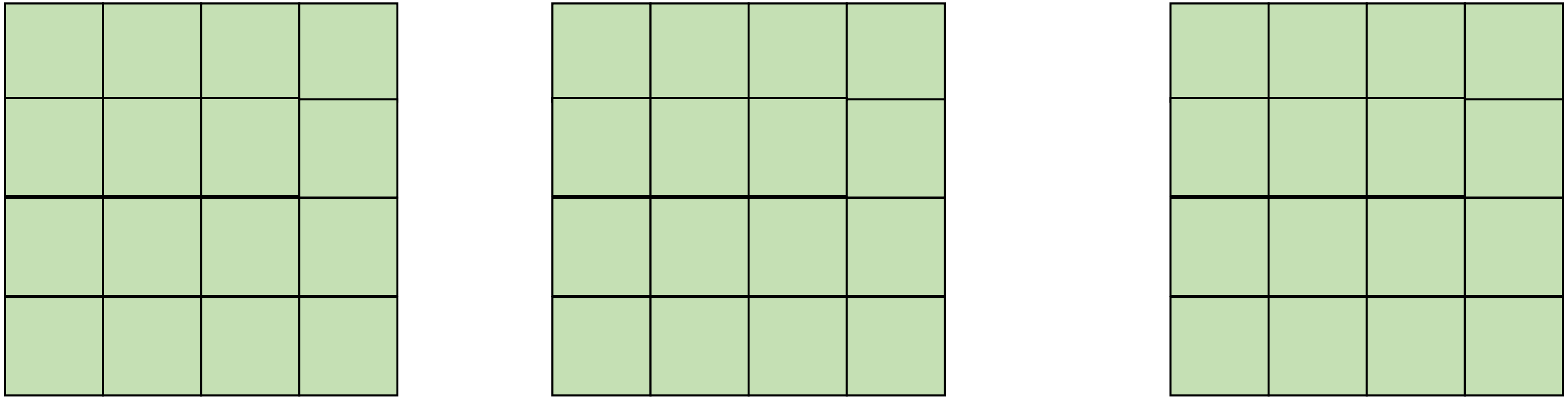
Adding 2D arrays together

Demo

- Memory accesses

$$A = B + C$$

A B C

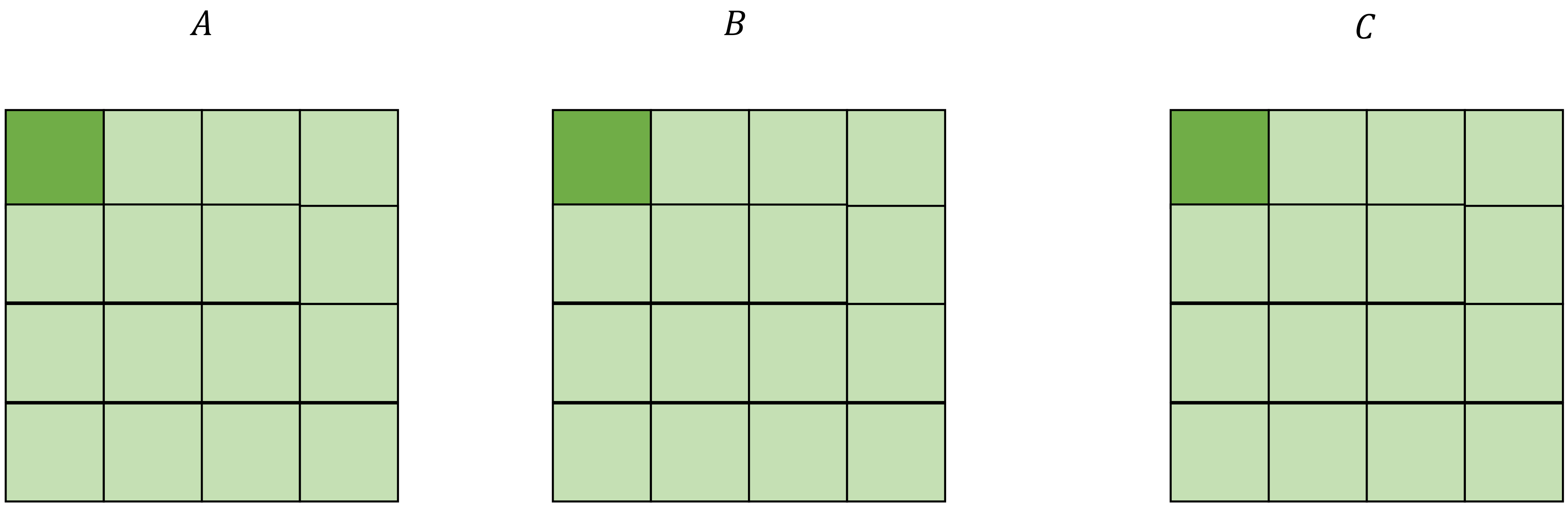


Adding 2D arrays together

Demo

- Memory accesses

$$A = B + C$$



Cache miss for all of them

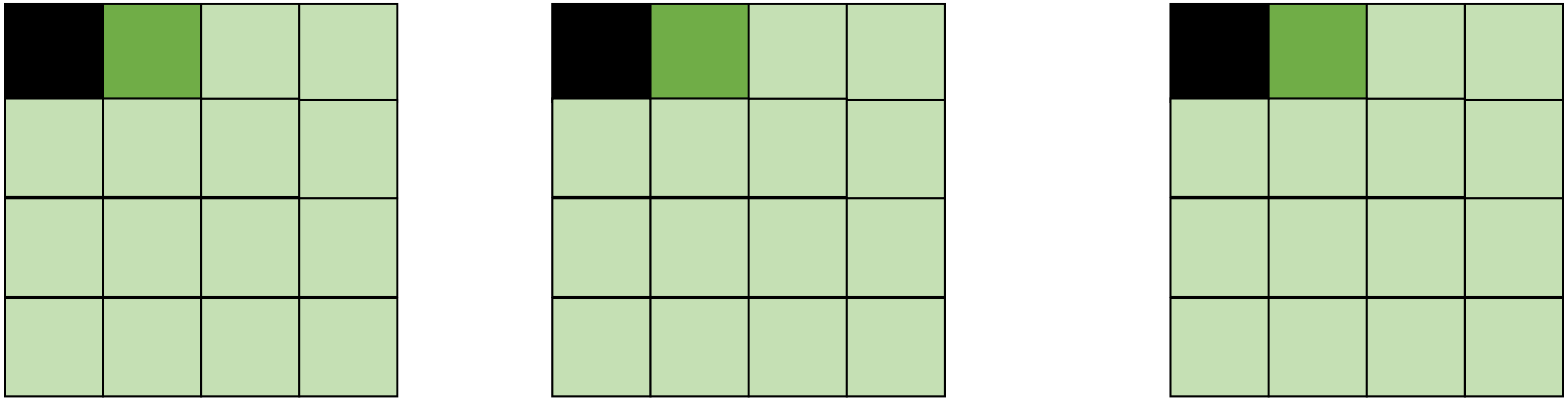
Adding 2D arrays together

Demo

- Memory accesses

$$A = B + C$$

A B C



Cache HIT for all of them

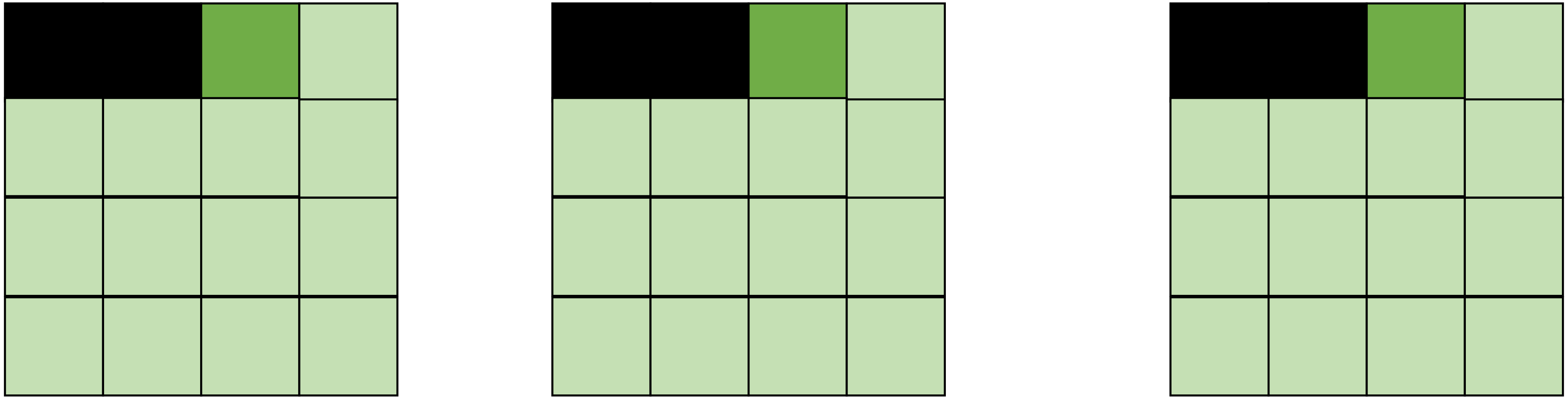
Adding 2D arrays together

Demo

- Memory accesses

$$A = B + C$$

A B C



Cache HIT for all of them

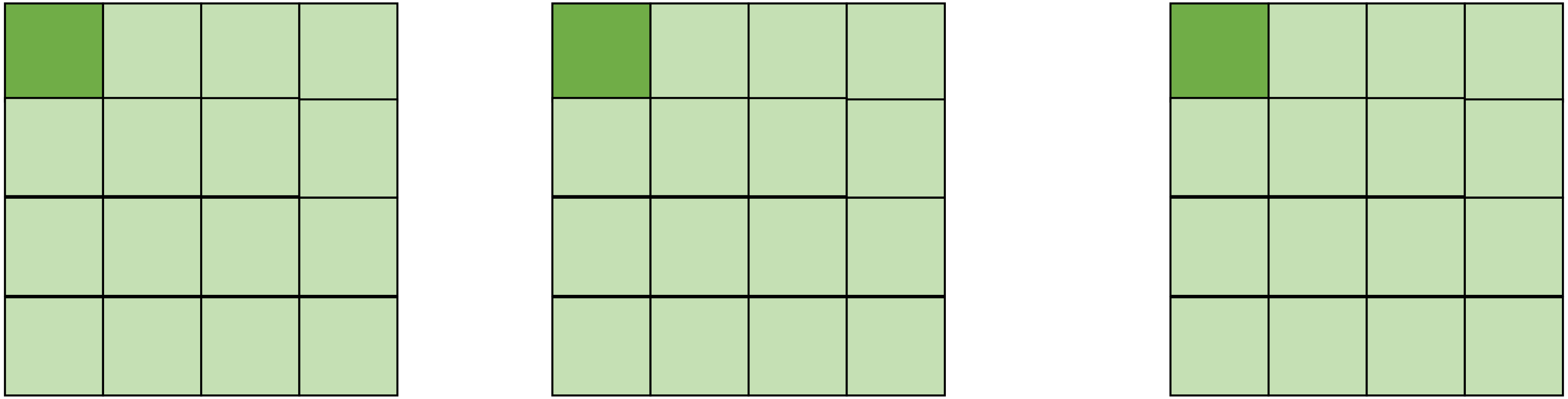
Adding 2D arrays together

Demo

- Memory accesses

$$A = B + C$$

A B C



Rewind! Cache miss for all of them

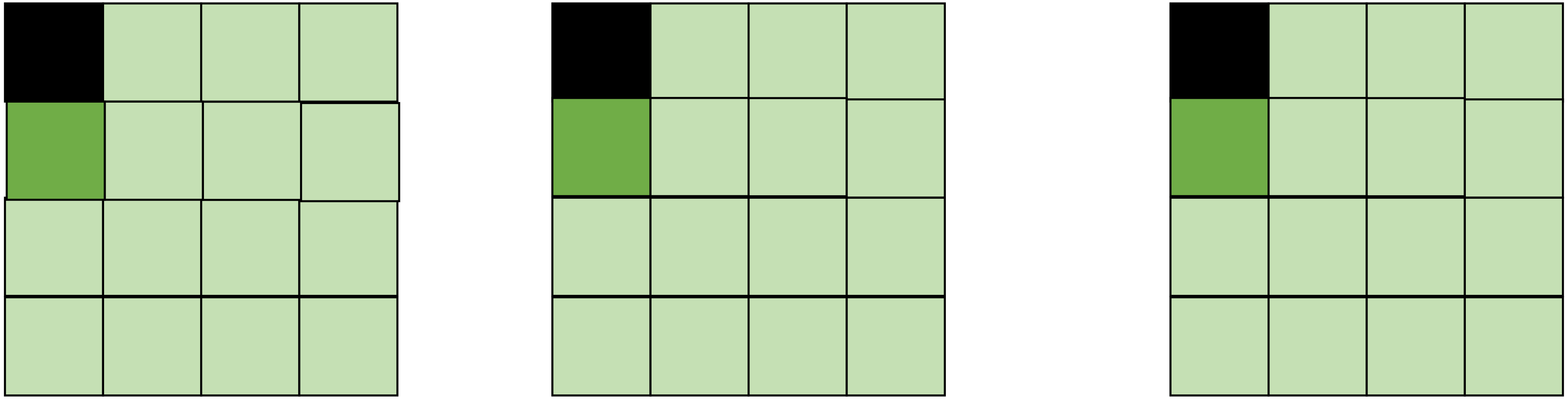
Adding 2D arrays together

Demo

- Memory accesses

$$A = B + C$$

A B C



Rewind! Cache miss for all of them

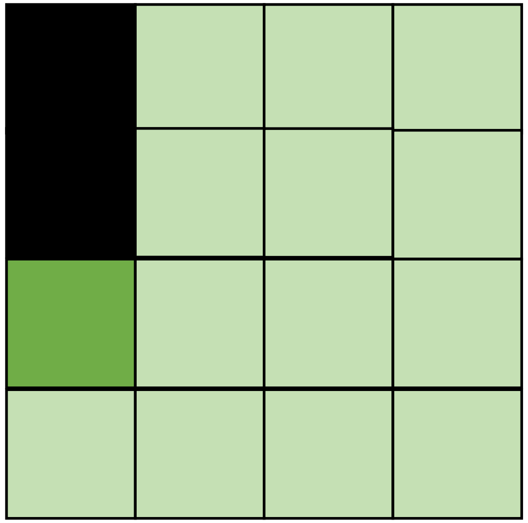
Adding 2D arrays together

Demo

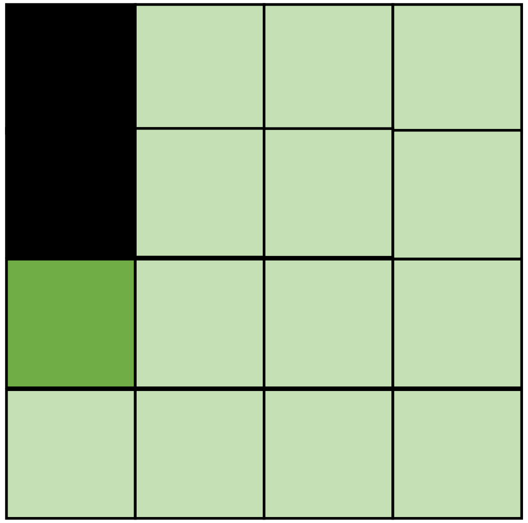
- Memory accesses

$$A = B + C$$

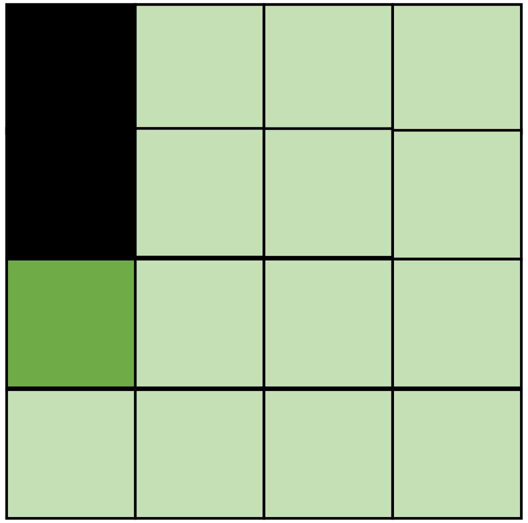
A



B



C



Rewind! Cache miss for all of them

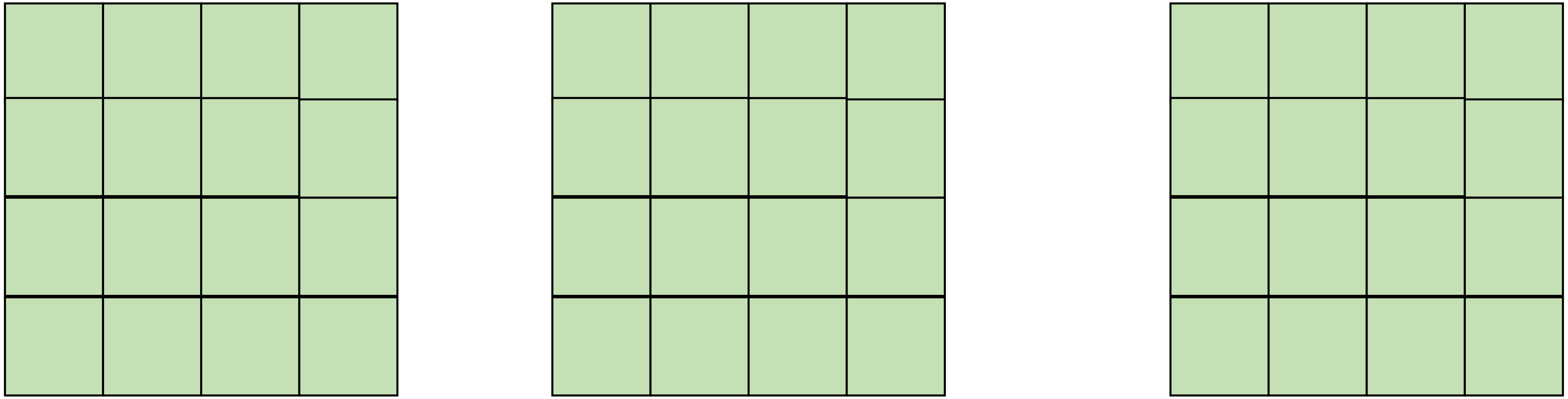
But sometimes there isn’t a good ordering

transposed arrays

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

$$A = B + C^T$$

A B C

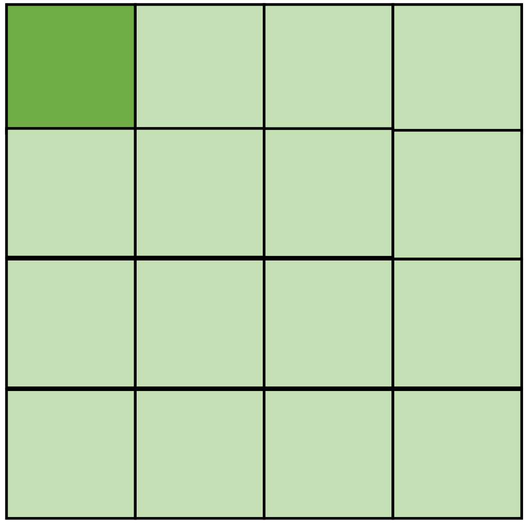


transposed arrays

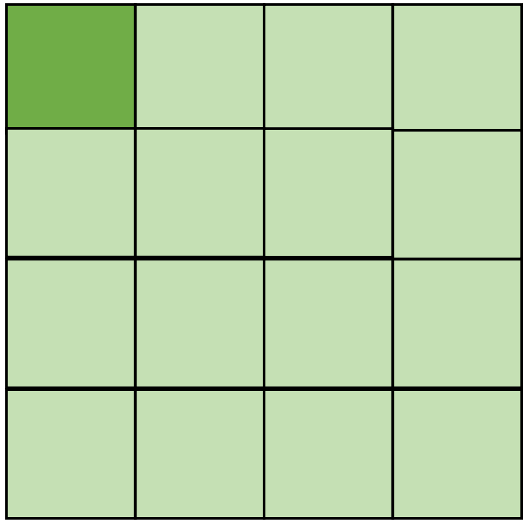
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

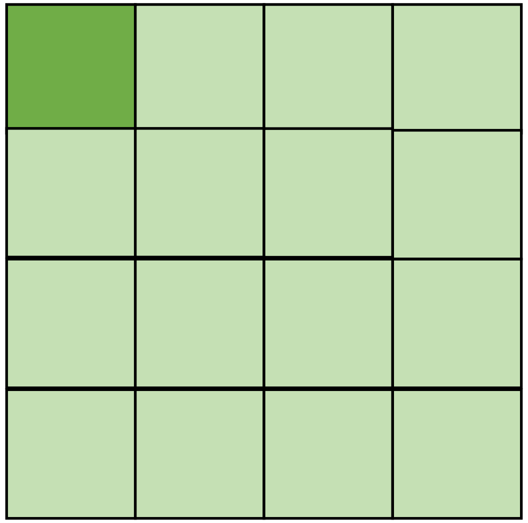
A



B



C

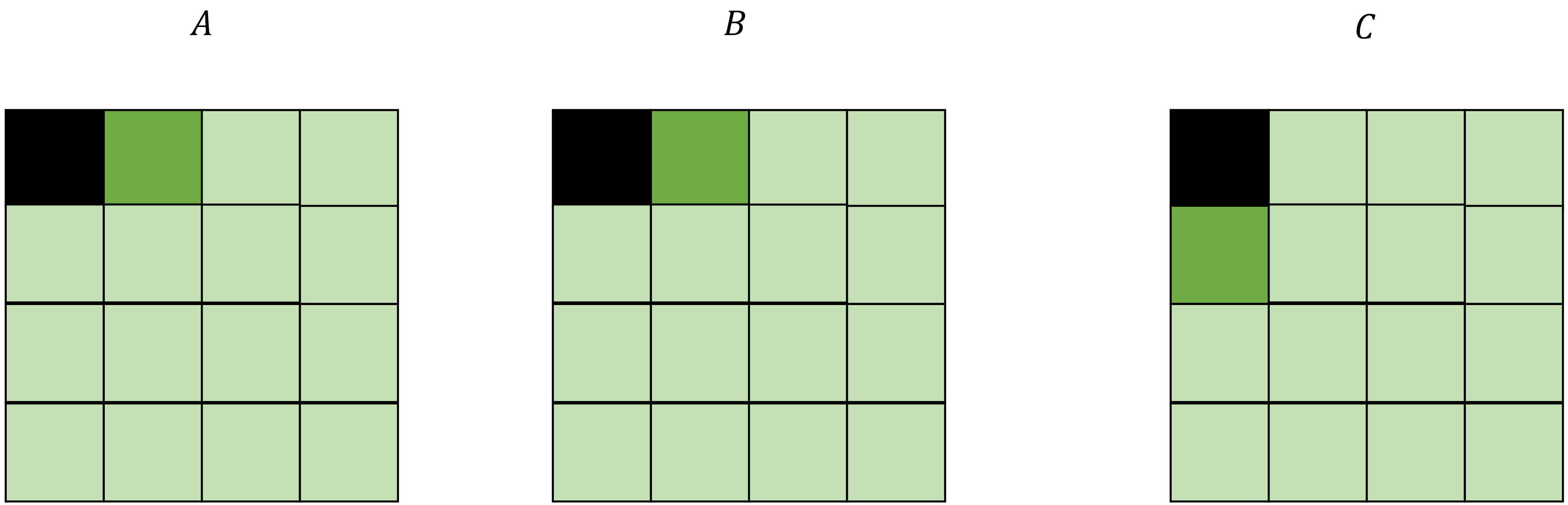


cold miss for all of them

transposed arrays

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

$$A = B + C^T$$

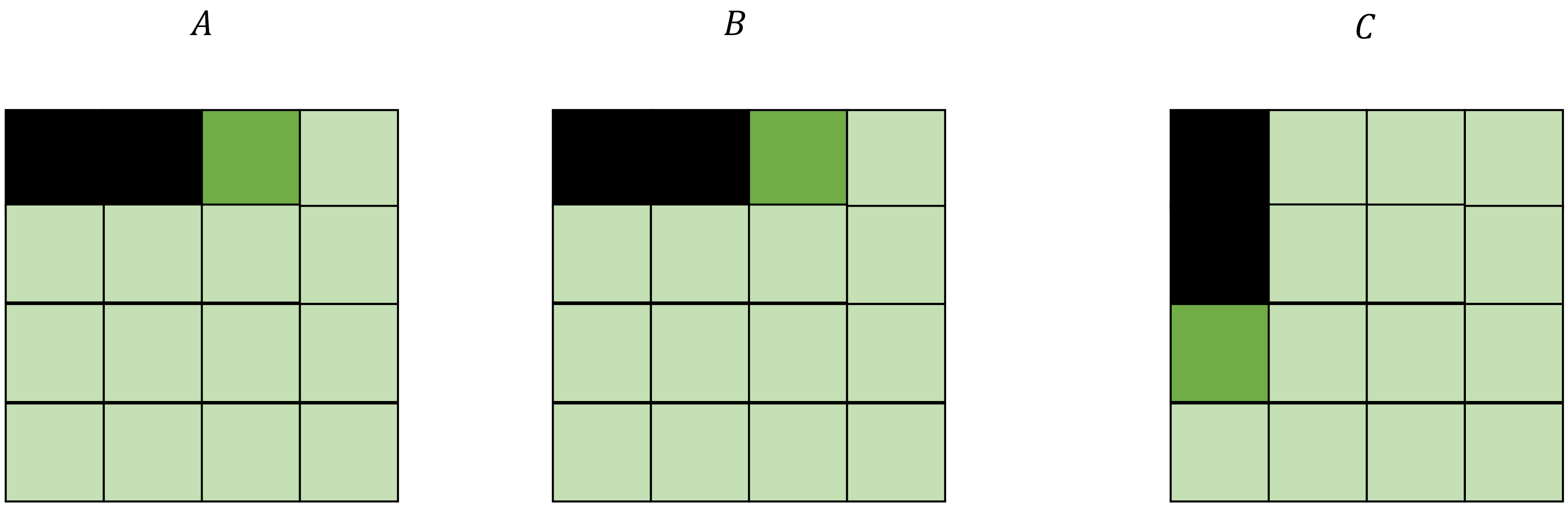


Hit on A and B. Miss on C

transposed arrays

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

$$A = B + C^T$$



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

First unroll,
Then put back into a loop

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;
    }
}
```

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

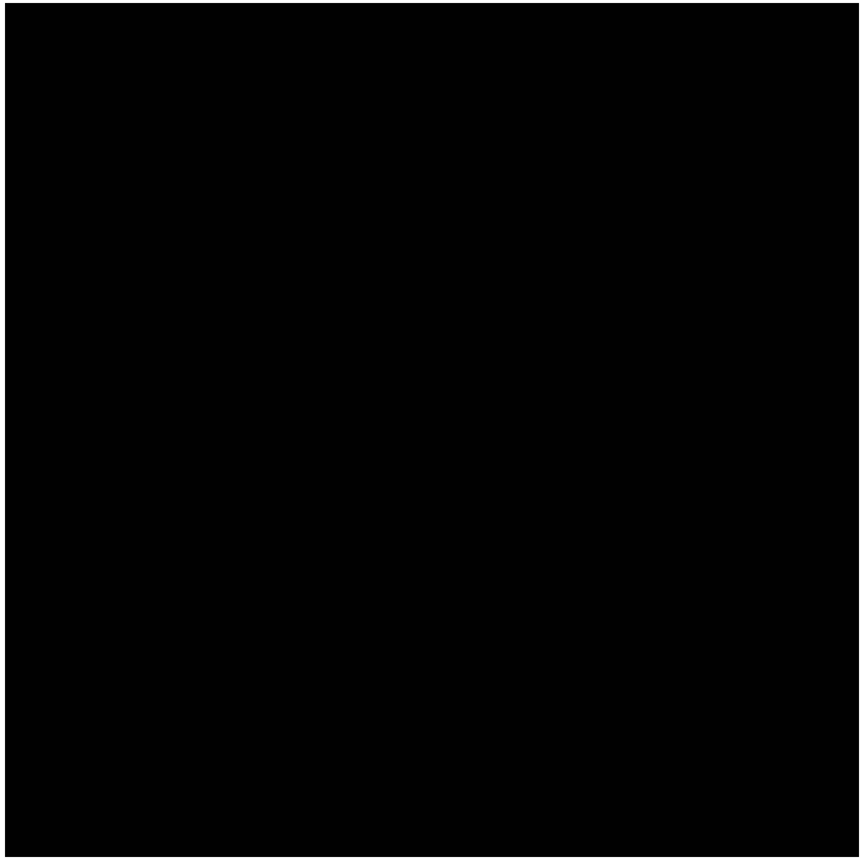
We can chain optimizations

- 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

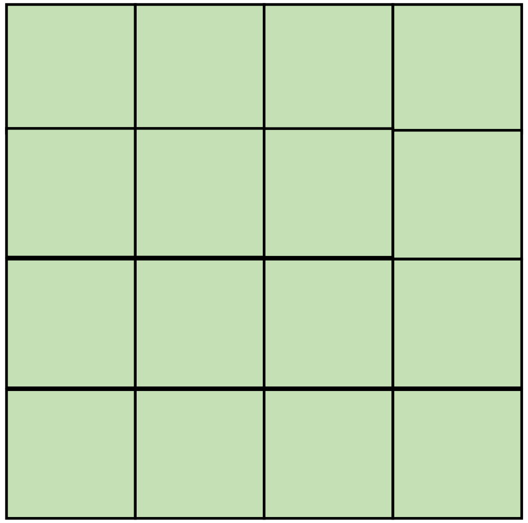
blocking

blocking

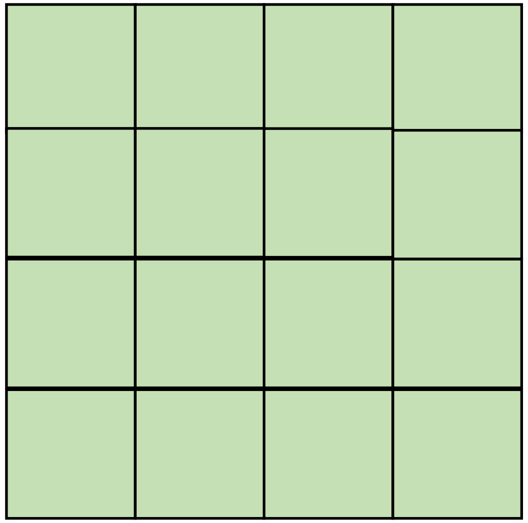
- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

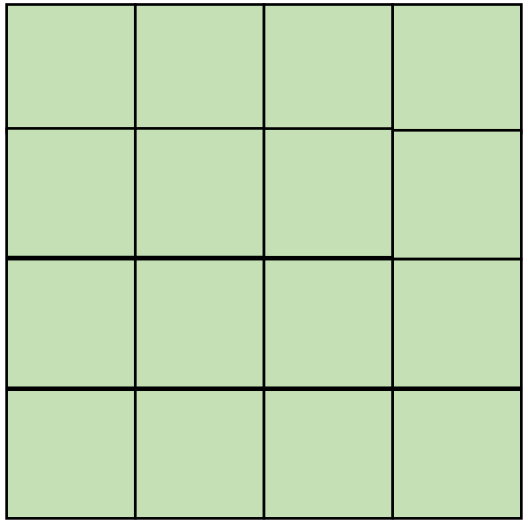
A



B



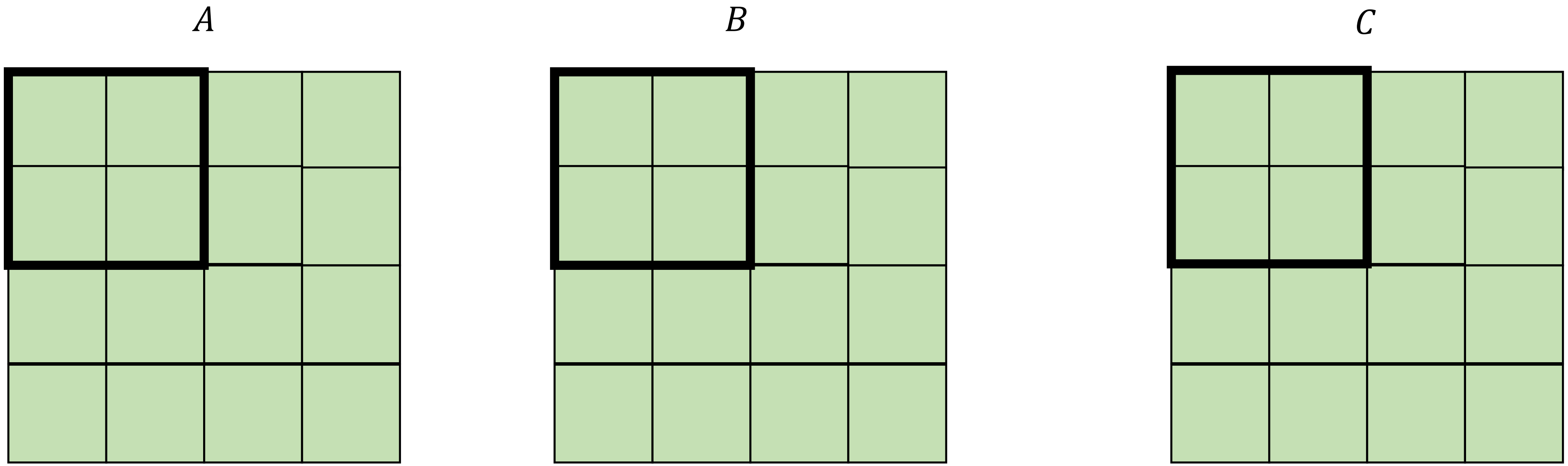
C



blocking

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

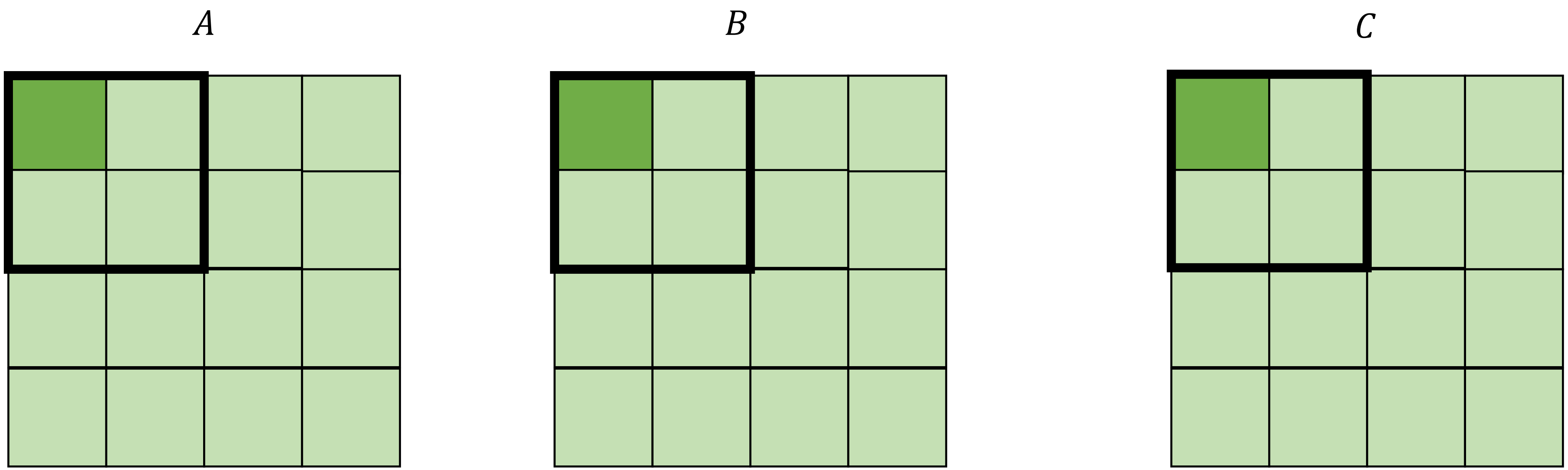
$$A = B + C^T$$



blocking

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

$$A = B + C^T$$

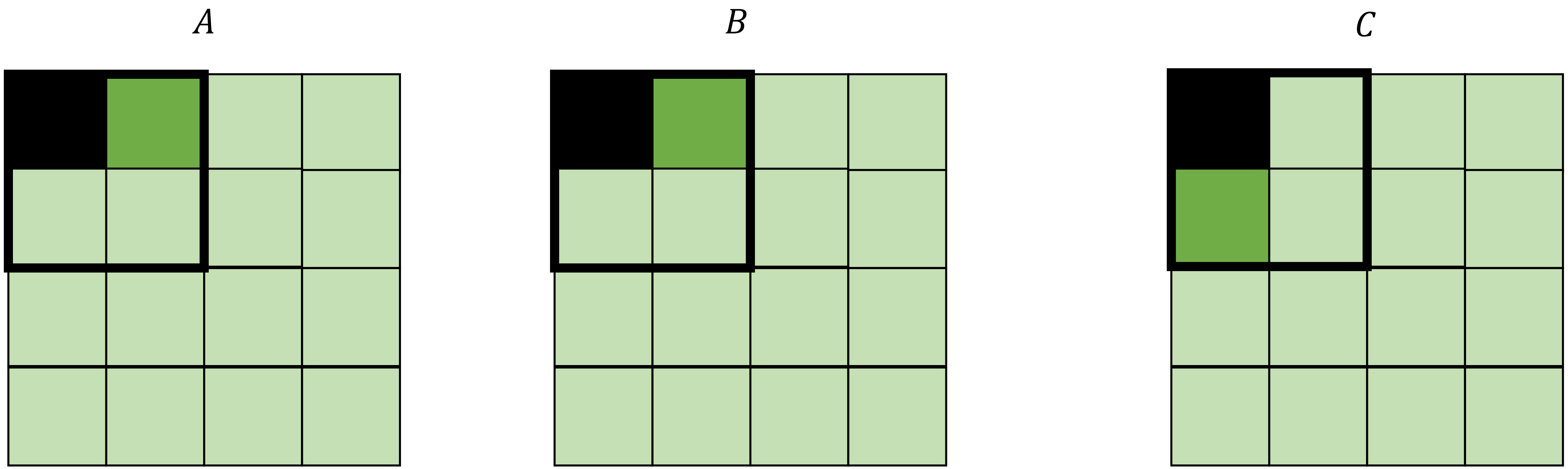


cold miss for all of them

blocking

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

$$A = B + C^T$$

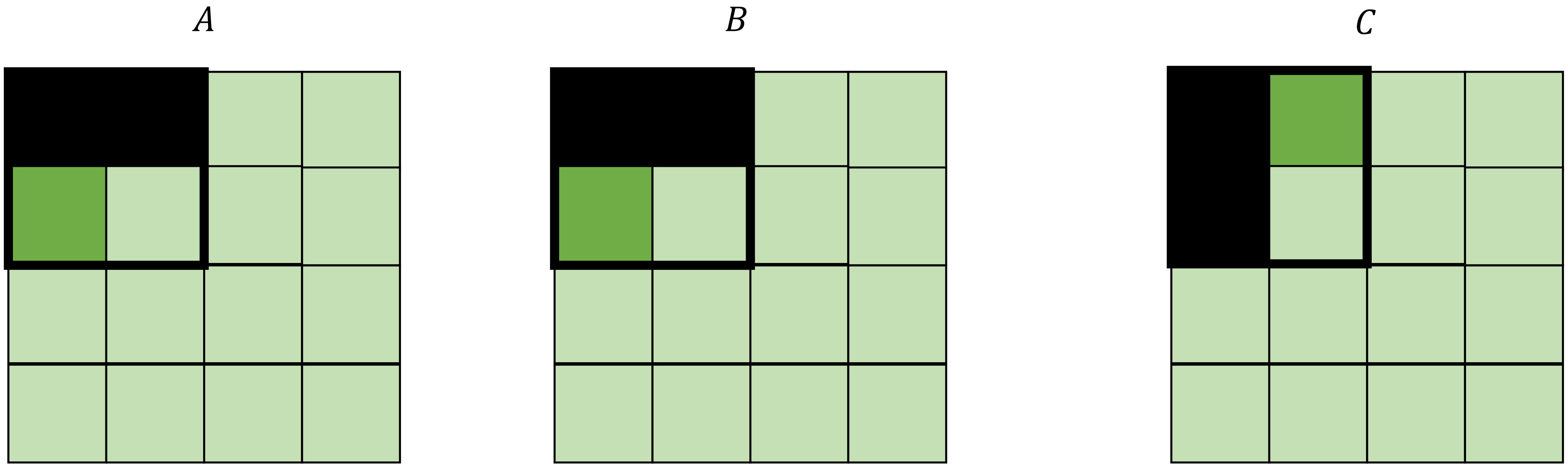


Miss on C

blocking

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

$$A = B + C^T$$

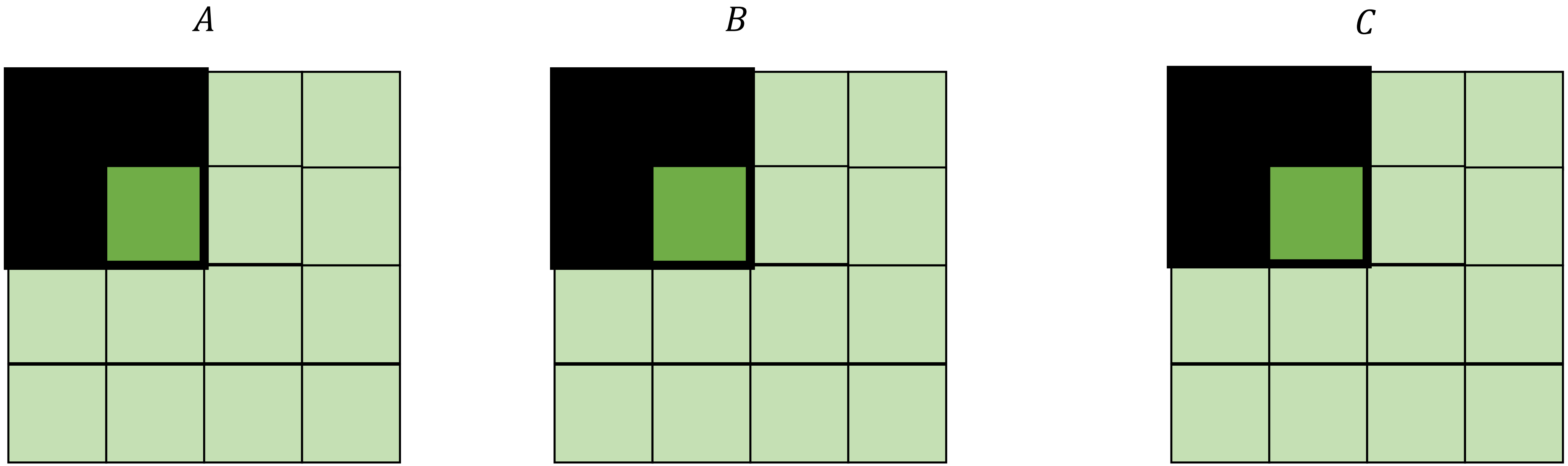


Miss on A,B, hit on C

blocking

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

$$A = B + C^T$$



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/

New material – Instruction Level Parallelism

Instruction-level Parallelism (ILP)

- Parallelism from a single stream of instructions.
  - Output of program must match exactly a sequential execution!
- Widely applicable:
  - most mainstream programming languages are sequential
  - most deployed hardware has components to execute ILP
- Done by a combination of programmer, compiler, and hardware

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

```
x = z + w;
a = b + c;
```

Two instructions are independent if the operand registers are disjoint from the result registers

(assume all letter variables are registers)

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

```
x = z + w;
a = b + c;
```

Two instructions are independent if the operand registers are disjoint from the result registers

(assume all letter variables are registers)

instructions that are not independent cannot be executed in parallel

```
x = z + w;
a = b + x;
```

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

```
x = z + w;
a = b + c;
```

Two instructions are independent if the operand registers are disjoint from the result registers

(assume all letter variables are registers)

instructions that are not independent cannot be executed in parallel

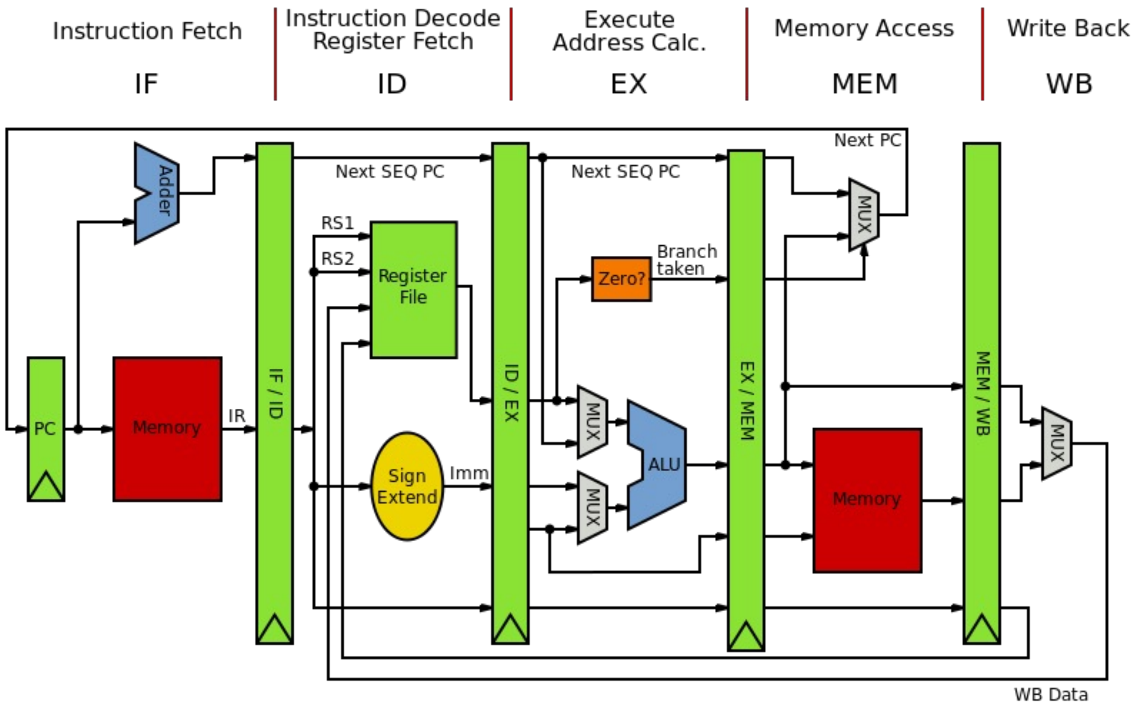
```
x = z + w;
a = b + x;
```

Many times, dependencies can be easily tracked in the compiler:

How can hardware execute ILP?

- Pipeline parallelism

- Abstract mental model:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

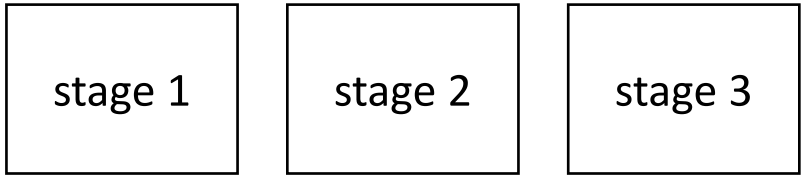


MIPS pipeline image from:
https://commons.wikimedia.org/wiki/Pipeline_(computer_hardware)

Pipeline

- Pipeline parallelism

- Abstract mental model
  for compiler:
    - N-stage pipeline
    - N instructions can be in-flight
    - Dependencies stall pipeline

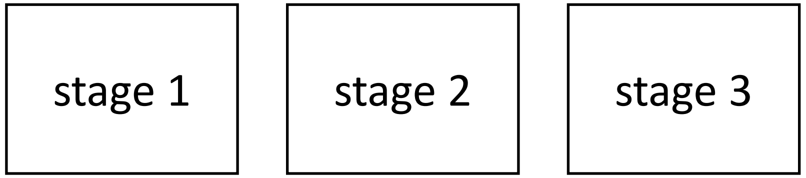



```
instr1;
instr2;
instr3;
```

Pipeline

- Pipeline parallelism

- Abstract mental model
  for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

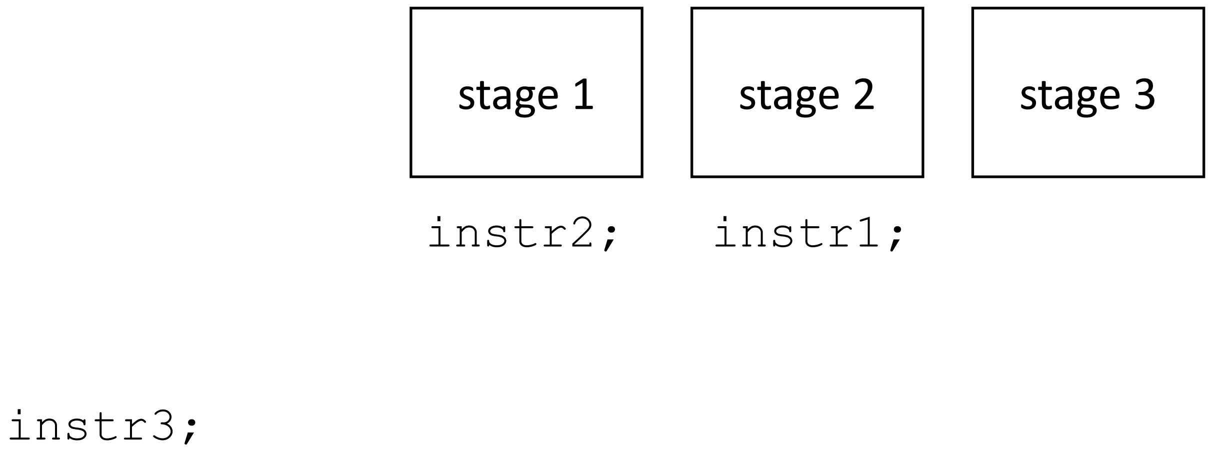



```
instr1;
```

```
instr2;
instr3;
```

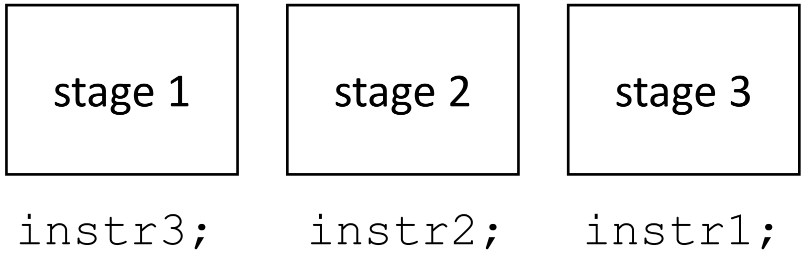
Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
    - N-stage pipeline
    - N instructions can be in-flight
    - Dependencies stall pipeline



Pipeline

- Pipeline parallelism

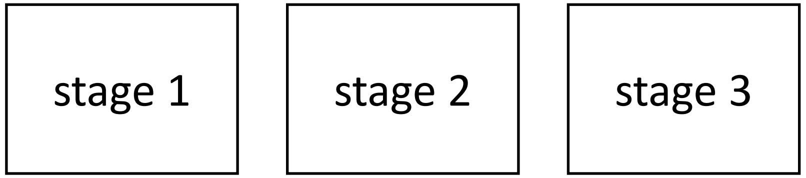


- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

Pipeline

- Pipeline parallelism

- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

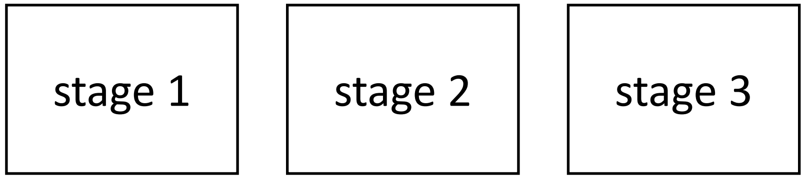


6 cycles for 3 independent instructions

Converges to 1 instruction per cycle

Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline




```
instr1;
instr2;
instr3;
```

What if the instructions depend on each other?

Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1 | stage 2 | stage 3

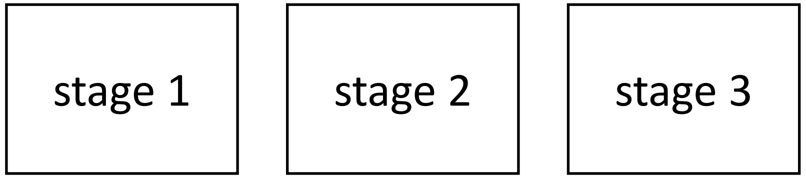
```
instr1;
```

```
instr2;
instr3;
```

What if the instructions depend on each other?

Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



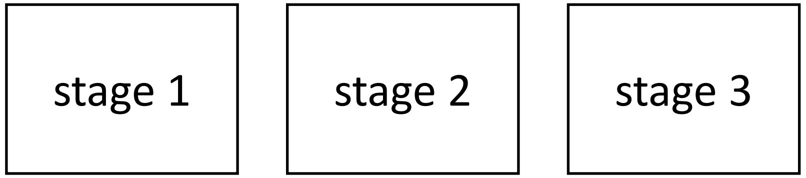

```
instr1;
```

```
instr2;
instr3;
```

What if the instructions depend on each other?

Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



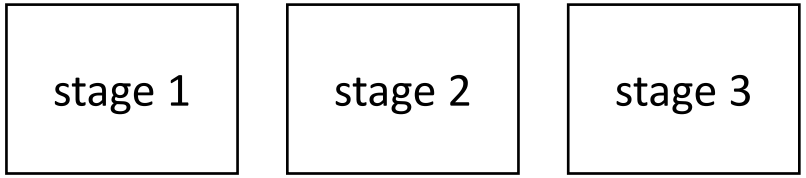

```
instr1;
```

```
instr2;
instr3;
```

What if the instructions depend on each other?

Pipeline

- Pipeline parallelism



- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

```
instr2;
instr3;
```

What if the instructions depend on each other?

Pipeline

- Pipeline parallelism

- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1

stage 2

stage 3

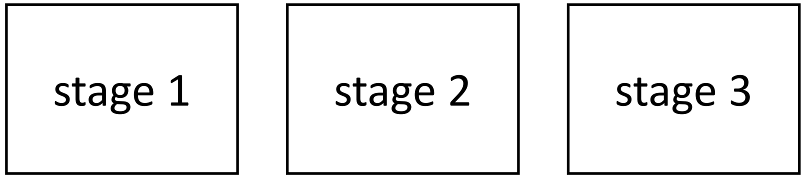
`instr2;`

`instr3;`

What if the instructions depend on each other?

Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



`instr2;`

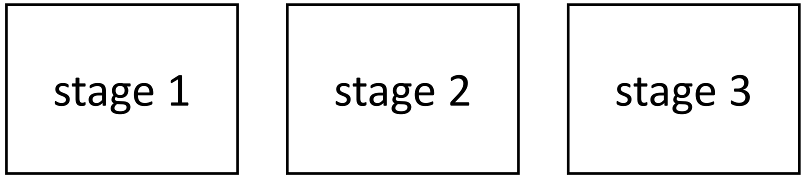
`instr3;`

and so on...

What if the instructions depend on each other?

Pipeline

- Pipeline parallelism



- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

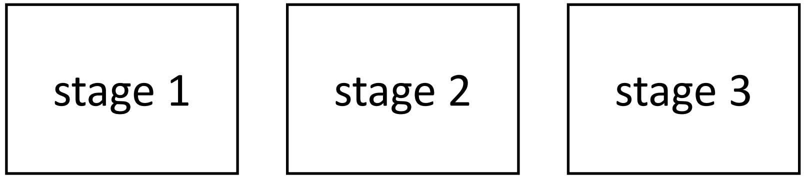
What if the instructions depend on each other?

9 cycles for 3 instructions

converges to 3 cycles per instruction

Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline




```
instr1;
instrX0;
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

- Pipeline parallelism

- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1 | stage 2 | stage 3

```
instr1;
```

```
instrX0;
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

- Pipeline parallelism

- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1	stage 2	stage 3
`instrX0;`	`instr1;`	

```
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

- Pipeline parallelism

- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

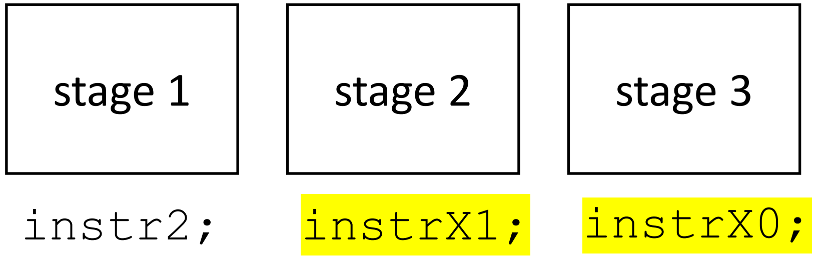
stage 1	stage 2	stage 3
`instrX1;`	`instrX0;`	`instr1;`

```
instr2;
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

- Pipeline parallelism
- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline




```
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

- Pipeline parallelism

- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1	stage 2	stage 3
`instr2;`	`instrX1;`	`instrX0;`

```
instrX2;
instrX3;
instr3;
```

and so on...

We converge to 1 cycle per instruction again!

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

How can hardware execute ILP?

- Executing multiple instructions at once:

- Superscalar architecture:
  - Several sequential operations are issued in parallel
  - hardware detects dependencies

issue-width is maximum number of instructions that can be issued in parallel

```
instr0;
instr1;
instr2;
```

if instr0 and instr1 are independent, they will be issued in parallel

It’s even more complicated

- Out-of-order execution delays dependent instructions
  - Reorder buffers (RoB) track dependencies
  - Load-Store Queues (LSQ) hold outstanding memory requests

What does this look like in the real world?

- Intel Haswell (2013):
  - Issue width of 4
  - 14-19 stage pipeline
  - OoO execution
- Intel Nehalem (2008)
  - 20-24 stage pipeline
  - Issue width of 2-4
  - OoO execution
- ARM
  - V7 has 3 stage pipeline; Cortex V8 has 13
  - Cortex V8 has issue width of 2
  - OoO execution
- RISC-V
  - Ariane and Rocket are In-Order
  - 3-6 stage pipelines
  - some super scaler implementations (BOOM)

What does this mean for us?

- We should have an abstract and parametrized performance model for instruction scheduling (the order of instructions)
- Try not to place dependent instructions in sequence
- Many times the compiler will help us here, but sometimes it cannot!

Three techniques to optimize for ILP

- Independent for loops (loop unrolling)
- Reduction for loops (loop unrolling)
- Priority topological ordering (if there is time)

What is loop unrolling?

can we unroll this loop?

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

Using Loop Unrolling to Exploit ILP

- for loops with independent chains of computation

```
for (int i = 0; i < SIZE; i++) {
    SEQ(i);
}
```

where:

```
SEQ(i) = instr1;
         instr2;
         ...
         a[i] = instrN;
```

and let instr(N) depends on instr(N-1)

loops only write to memory addressed by the loop variable

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

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

Saves one addition and one comparison per loop, but doesn't help with ILP

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

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

Let green highlights indicate instructions from iteration `i`.

Let blue highlights indicate instructions from iteration `i + 1`.

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

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

Let `SEQ(i,j)` be the jth instruction of `SEQ(i)`.

Let each instruction chain have N instructions

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
    SEQ(i,1);
    SEQ(i,2);
    ...
    SEQ(i,N); // end iteration for i
    SEQ(i+1,1);
    SEQ(i+1,2);
    ...
    SEQ(i+1, N); // end iteration for i + 1
}
```

Let `SEQ(i,j)` be the jth instruction of `SEQ(i)`.

Let each instruction chain have N instructions

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
      SEQ(i,1);
      SEQ(i+1,1);
      SEQ(i,2);
      SEQ(i+1,2);
      ...
      SEQ(i,N);
      SEQ(i+1, N);
}
```

They can be interleaved

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
      SEQ(i,1);
      SEQ(i+1,1);
      SEQ(i,2);
      SEQ(i+1,2);
      ...
      SEQ(i,N);
      SEQ(i+1, N);
}
```

They can be interleaved

two instructions can be pipelined, or executed on a superscalar processor

General case

- Dependency graphs
  - Nodes are IR instructions
  - Edges are dependencies
  - Different traversals can provide better ILP!
  - Not enough time to go into detail this quarter, but the topic is known as instruction scheduling