CSE211: Compiler Design

Nov. 17, 2021

- **Topic**: Array processing DSL
- **Discussion questions**:
  - What is a DSL?
  - What are the benefits and drawbacks of a DSL?
  - What DSLs have you used?

Announcements

- Homework 2 and midterm are graded
  - Let me know if there are issues or if you have questions (Office hours on Thursday)
  - Last day to raise concerns is Friday
- Homework 3 is due TODAY
  - I will post homework 4 later today
- Starting on new module: DSLs
- Guest lecture for Nov. 22
  - Aviral Goel will talk about laziness in R

Announcements

- **Paper assignment**:
  - If you do not ”register” for a paper by Wednesday I will count it as late
  - 4 missing! Please sign up!

- **Project**:
  - Add your name and project title to sheet.
  - Last day to sign up. Any new projects will be presented on the 29.
  - You have until the 29th to switch to the final
  - Option for blog post, please mark your interest

Today we are moving on to DSLs

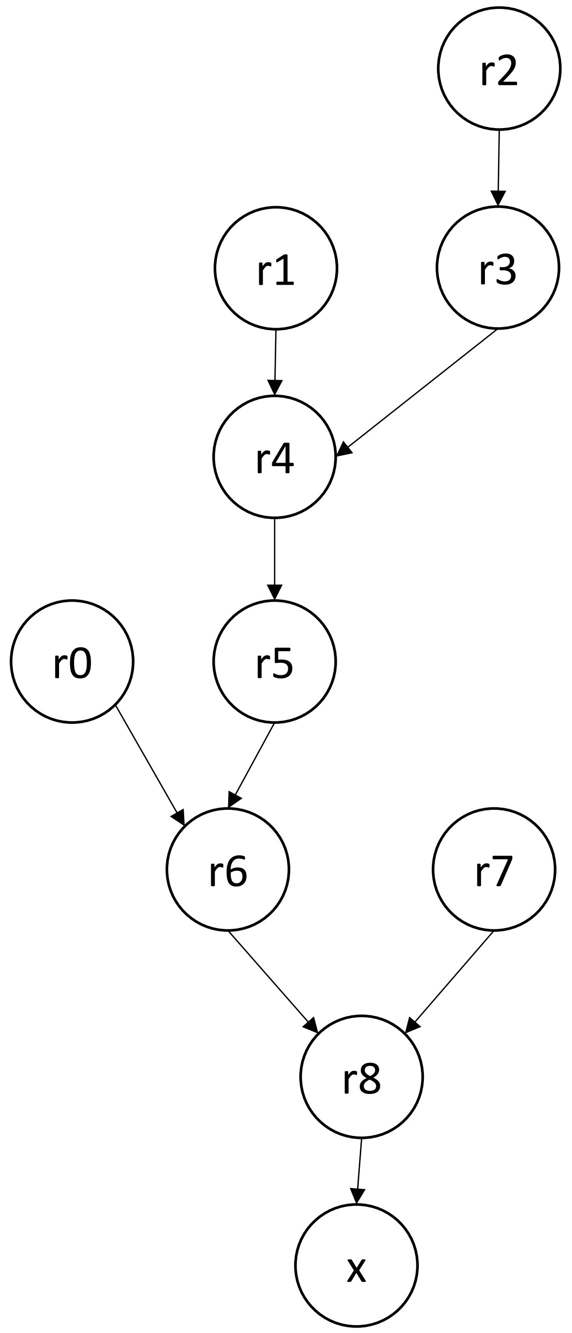
- But first let’s review

Review parallelism

- First thing we discussed:
  - Instruction level parallelism

Priority Topological Ordering of DDGs for Superscalar

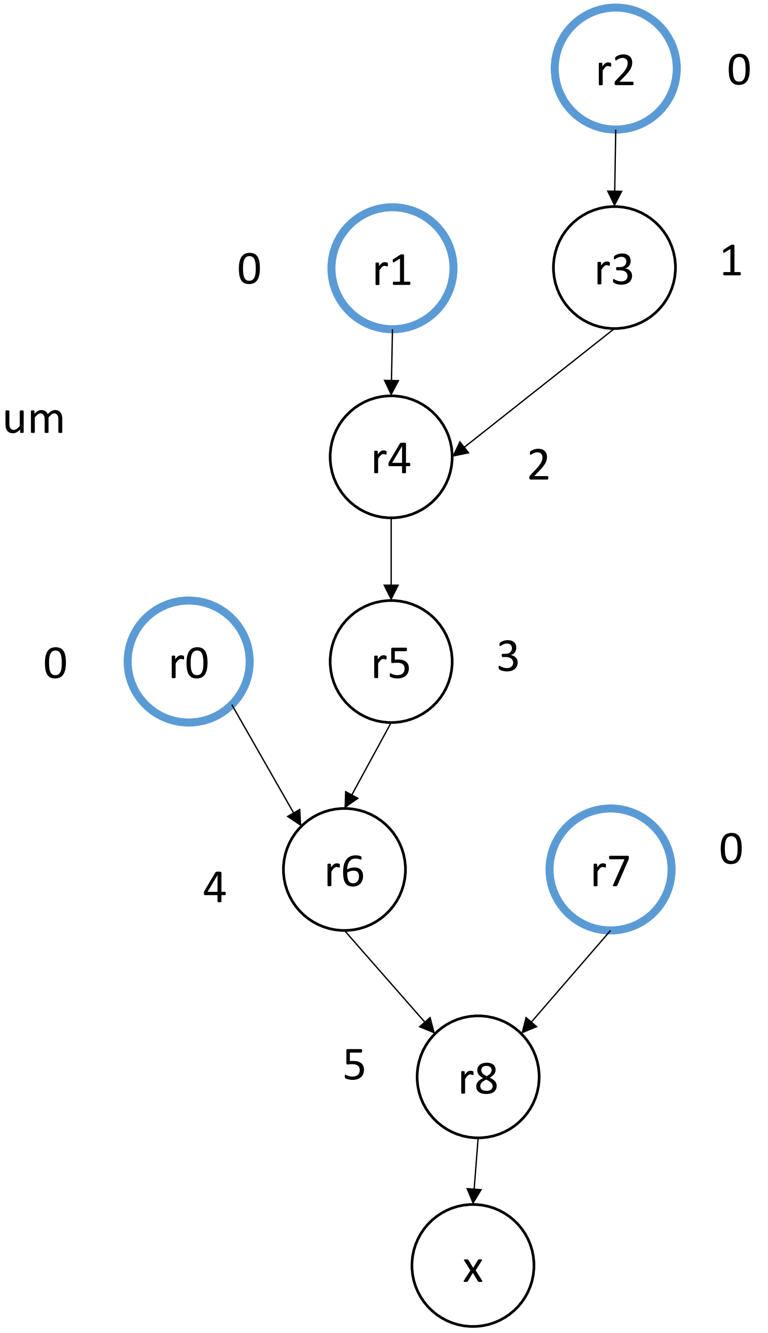
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```


$$\begin{tikzcd}
 & & r2 \arrow[d] \\
 & r1 \arrow[d] & r3 \arrow[dl] \\
 & r4 \arrow[d] & \\
r0 \arrow[dr] & r5 \arrow[d] & \\
 & r6 \arrow[dr] & & r7 \arrow[dl] \\
 & & r8 \arrow[d] & \\
 & & x &
\end{tikzcd}$$

Priority Topological Ordering of DDGs for Superscalar

```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```

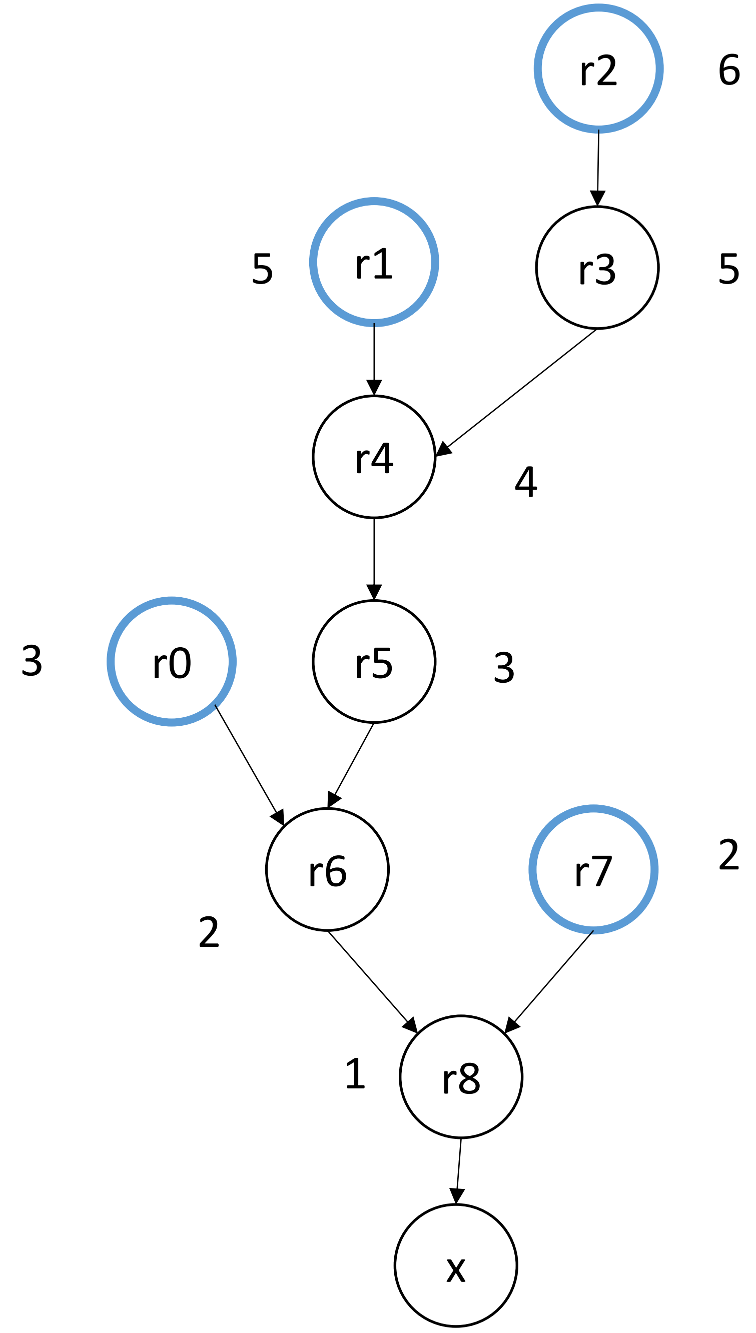
Label nodes with the maximum distance to a source



Priority Topological Ordering of DDGs for Pipelining

```
r2 = 4 * a;
r0 = neg(b);
r1 = b * b;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```

label each node with
a distance from the root.
Schedule each node according
to the level



How to deal with both super scalar and pipelining?

- Its complicated...
- Performance models and simulations (discussed in the dragon book)
- Intuitive model: place dependent instructions as far away from each other as possible

Across basic blocks

- Loop unrolling, creates a bigger basic block loop body
- Anticipatable expressions: move expressions to a location in the CFG where there are no control dependencies

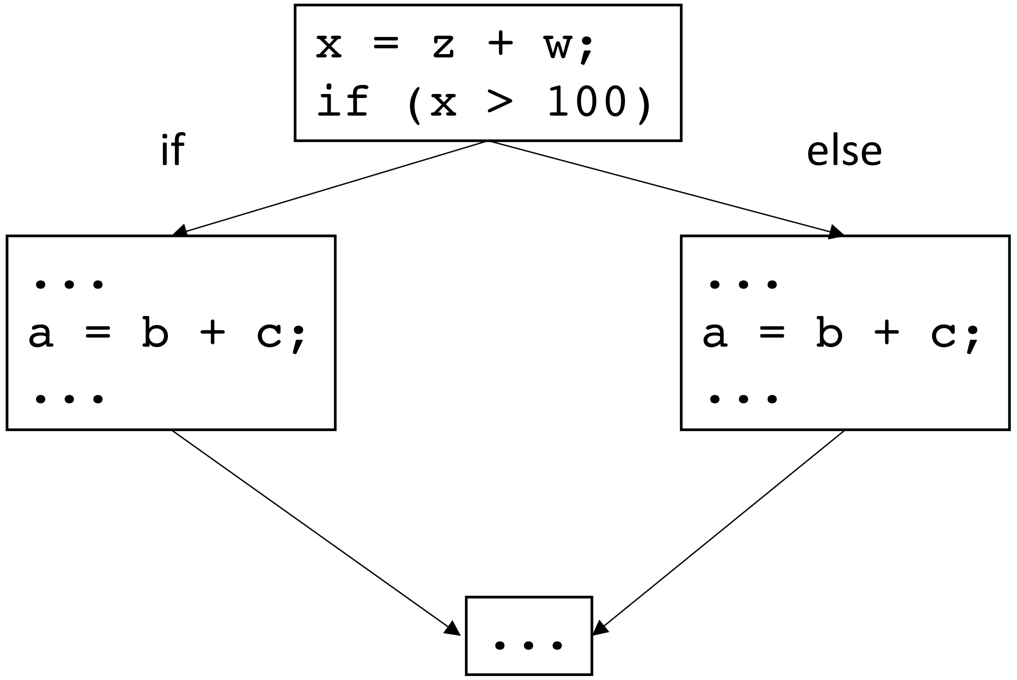
Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

An expression e is “anticipable” at a basic block b_x if for all paths that leave b_x, e is evaluated

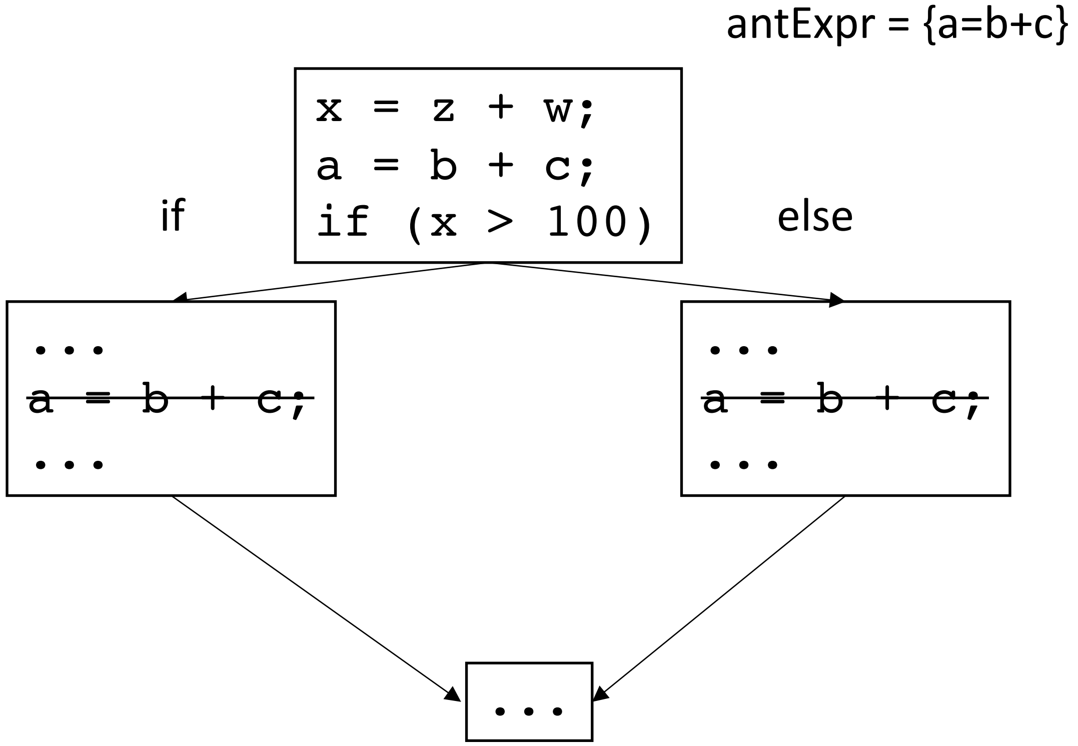
Anticipable Expressions

```
x = z + w;
if (x > 100) {
    ...
    a = b + c;
    ...
}
else {
    ...
    a = b + c;
    ...
}
```



Anticipable Expressions

```
x = z + w;
if (x > 100) {
    ...
    a = b + c;
    ...
}
else {
    ...
    a = b + c;
    ...
}
```



FOR loops are good candidates to do in parallel

- Safety:
  - What conditions?

- Efficiently:
  - What were some issues here?
  - Why is it important for parallelism?

Review

- Creating constraints

What if the loop index did not start at 0?

What if the the loop index does not increment by 1?

```
for (i = 5; i < 128; i+=2) {
   a[i]= a[i]*2;
}
```

two integers: i_x != i_y

i_x >= 5

i_x < 128

i_y >= 5

i_x % 2 == 1

i_y % 2 == 1

i_y < 128

i_x == i_y

i_x == i_y

write-write conflict

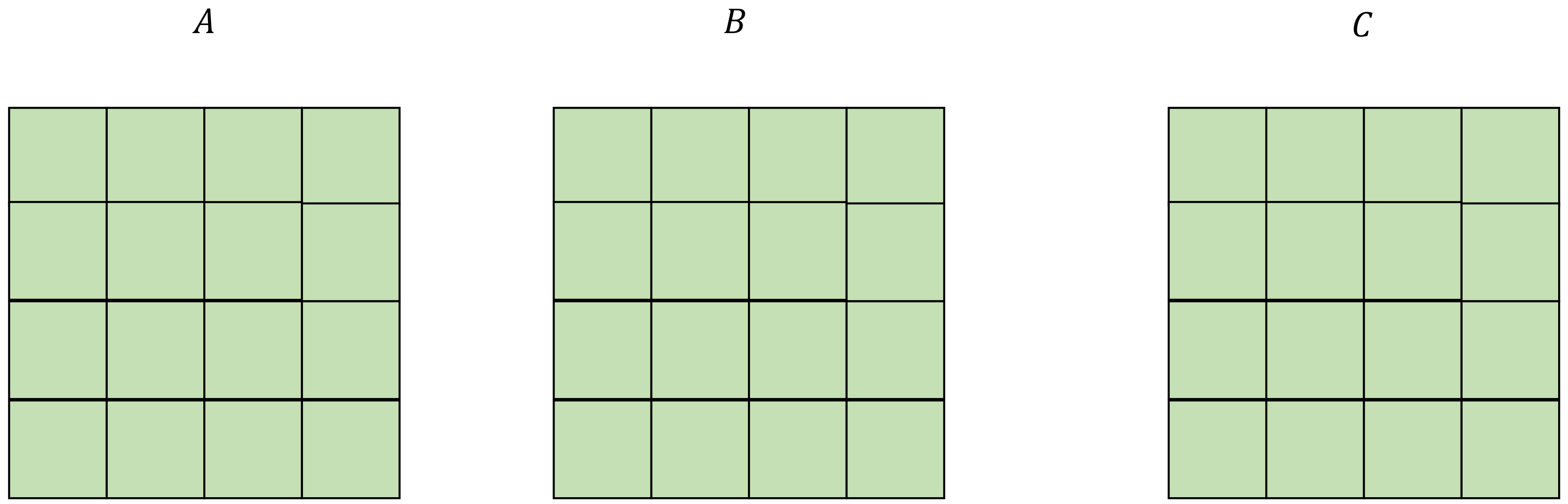
read-write conflict

Ask if these constraints are satisfiable (if so, it is not safe to parallelize)

Adding loop nestings

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

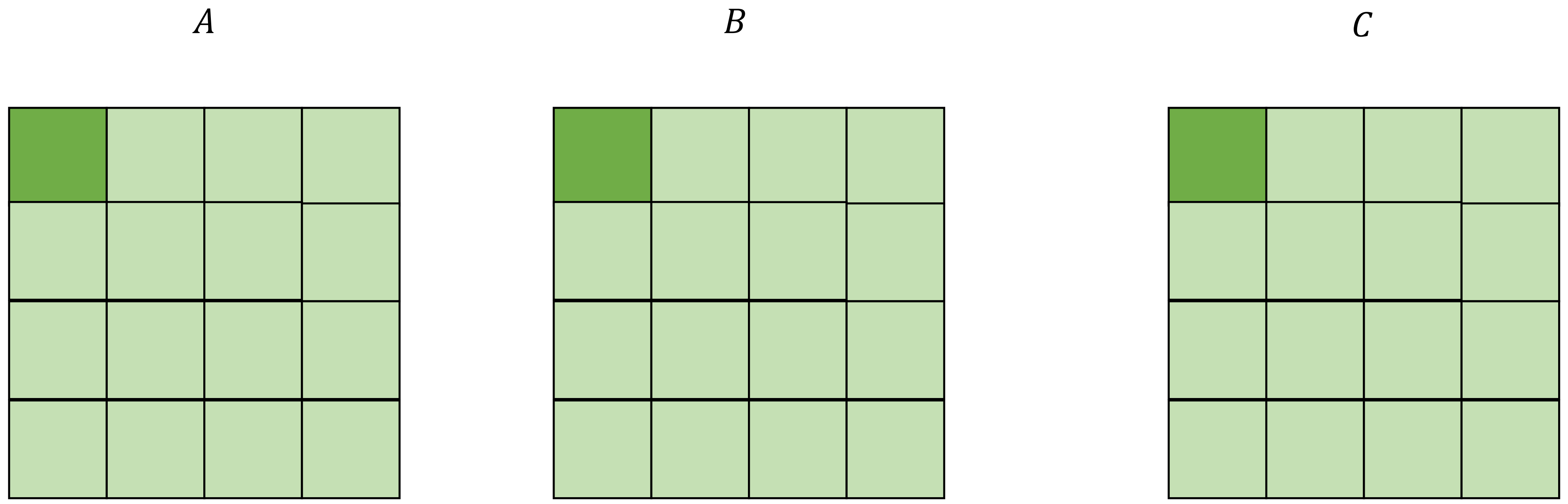
$$A = B + C^T$$



Adding loop nestings

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

$$A = B + C^T$$

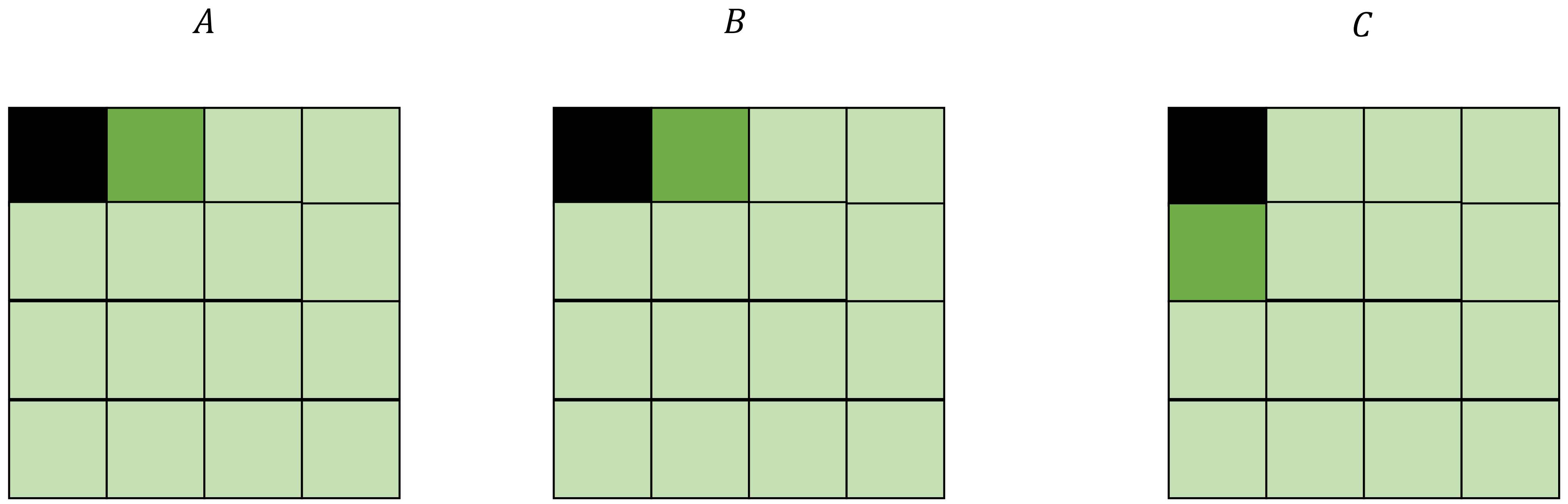


cold miss for all of them

Adding loop nestings

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

$$A = B + C^T$$

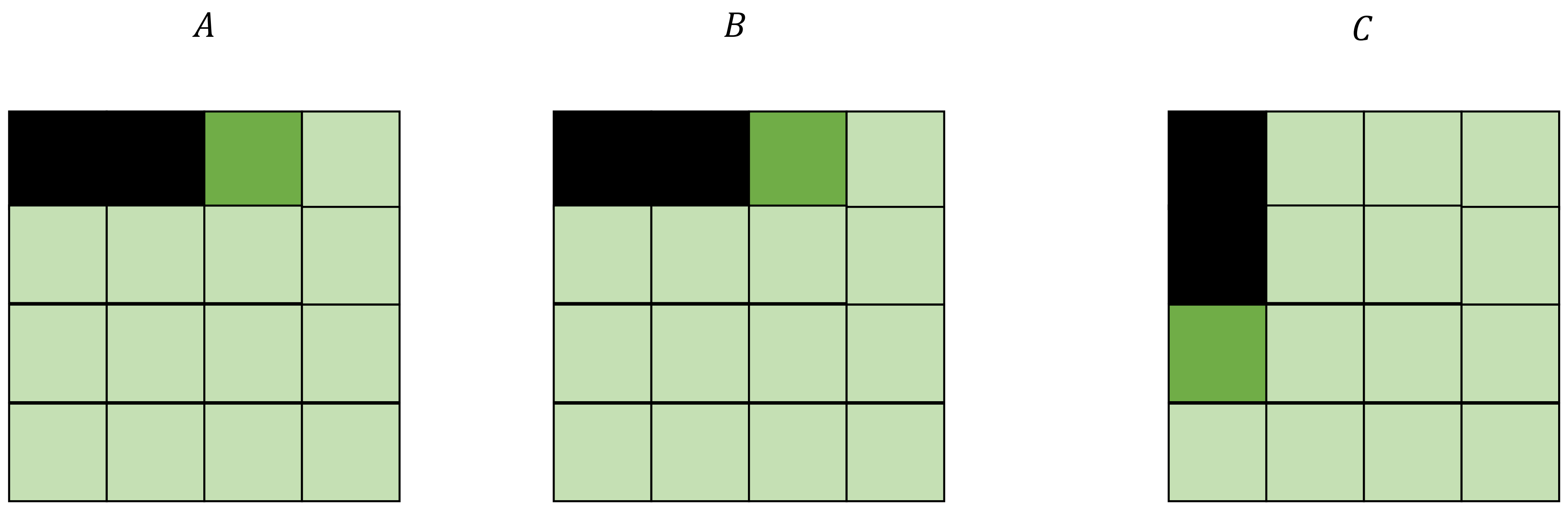


Hit on A and B. Miss on C

Adding loop nestings

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

$$A = B + C^T$$

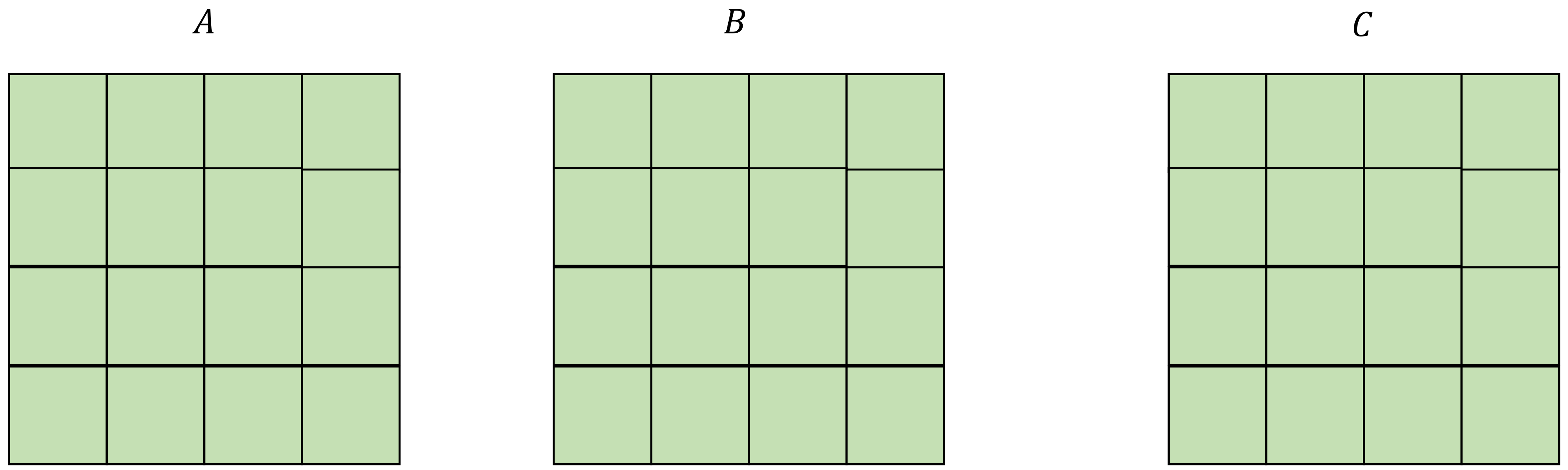


Hit on A and B. Miss on C

Adding loop nestings

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

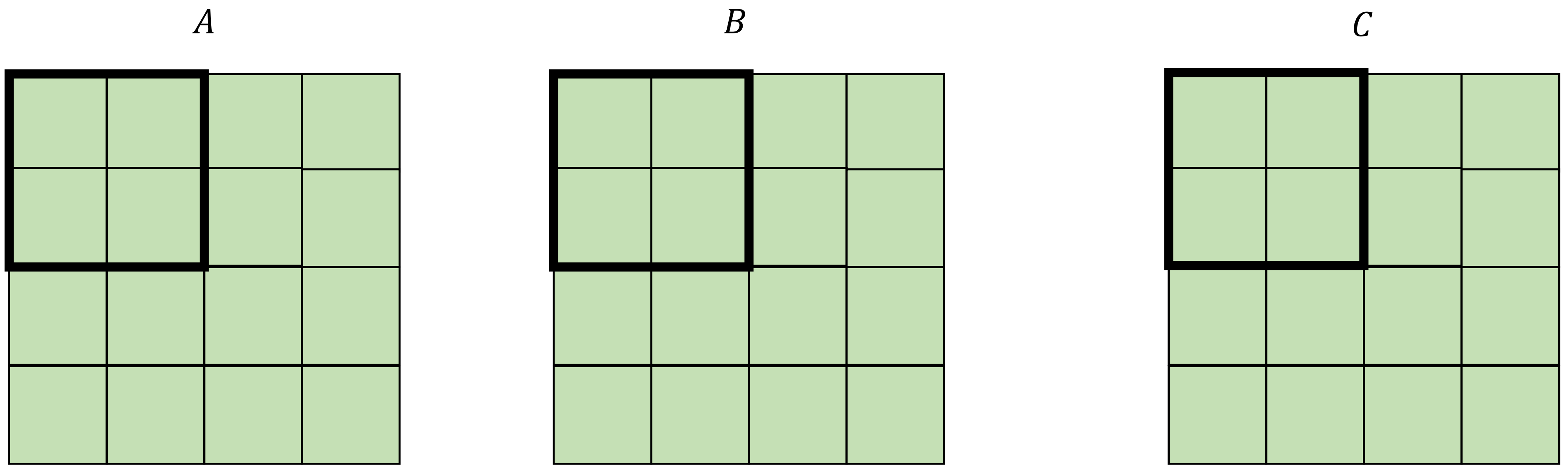
$$A = B + C^T$$



Adding loop nestings

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

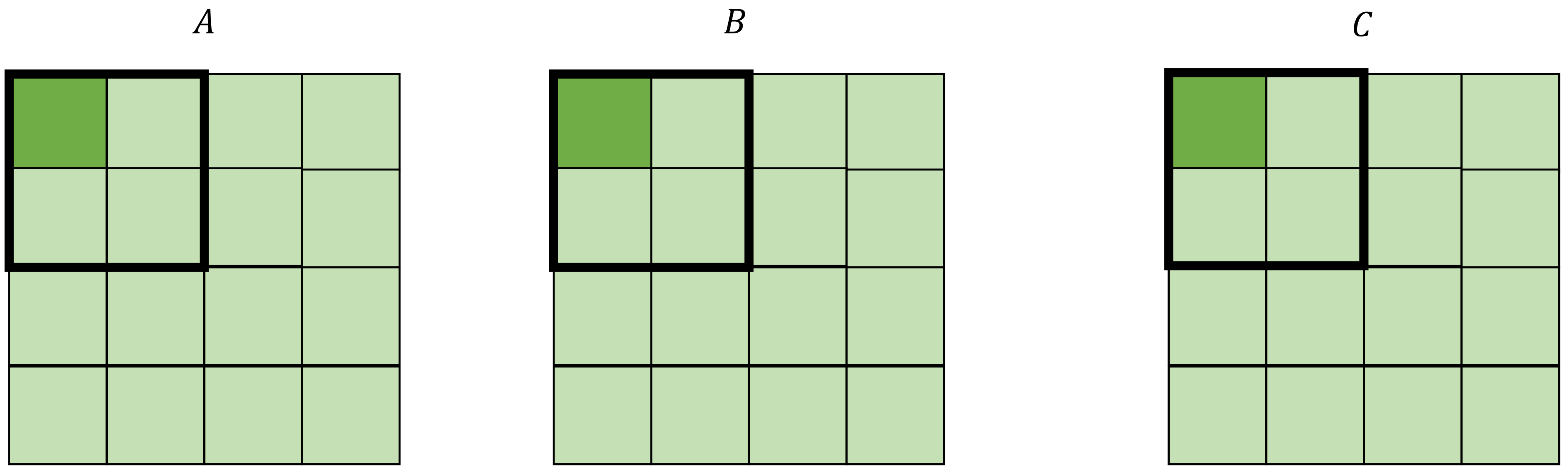
$$A = B + C^T$$



Adding loop nestings

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

$$A = B + C^T$$

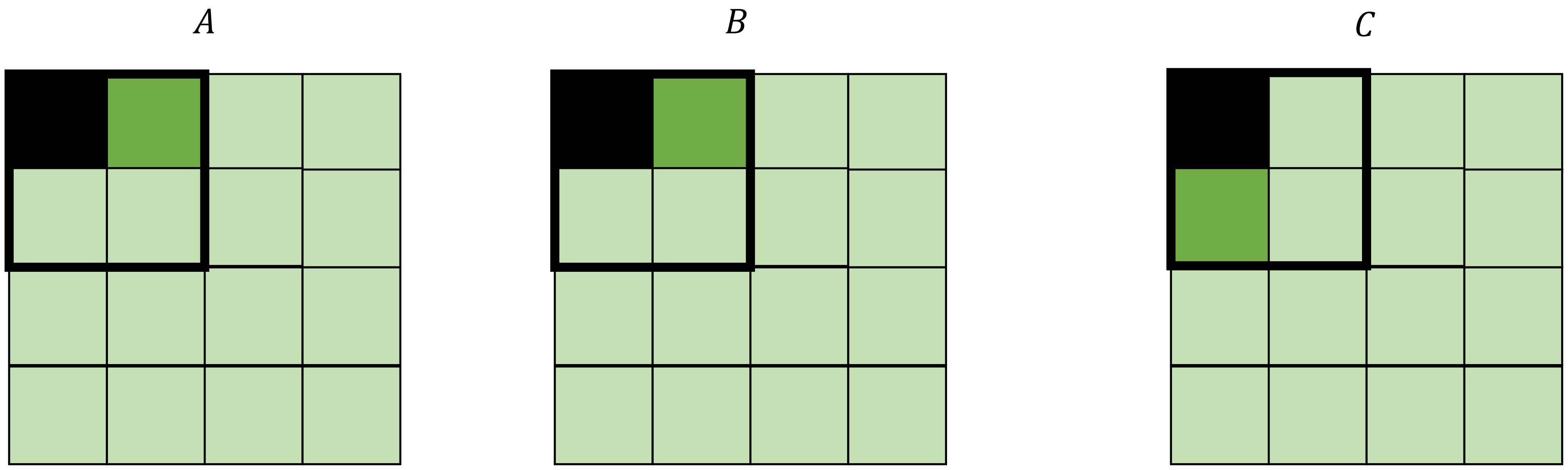


cold miss for all of them

Adding loop nestings

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

$$A = B + C^T$$

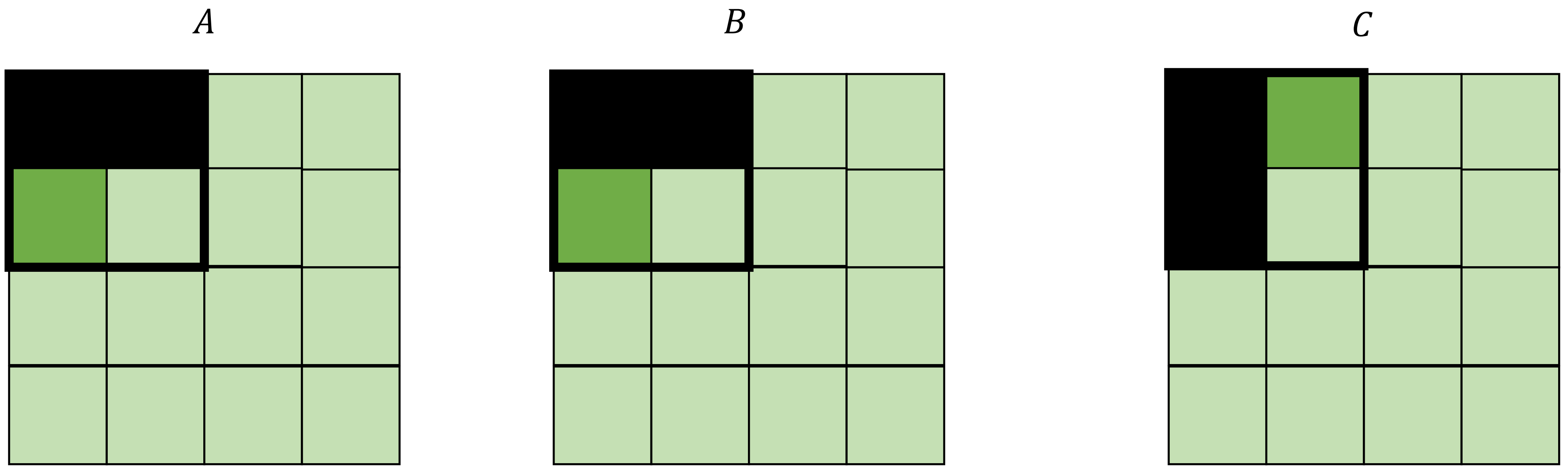


Miss on C

Adding loop nestings

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

$$A = B + C^T$$

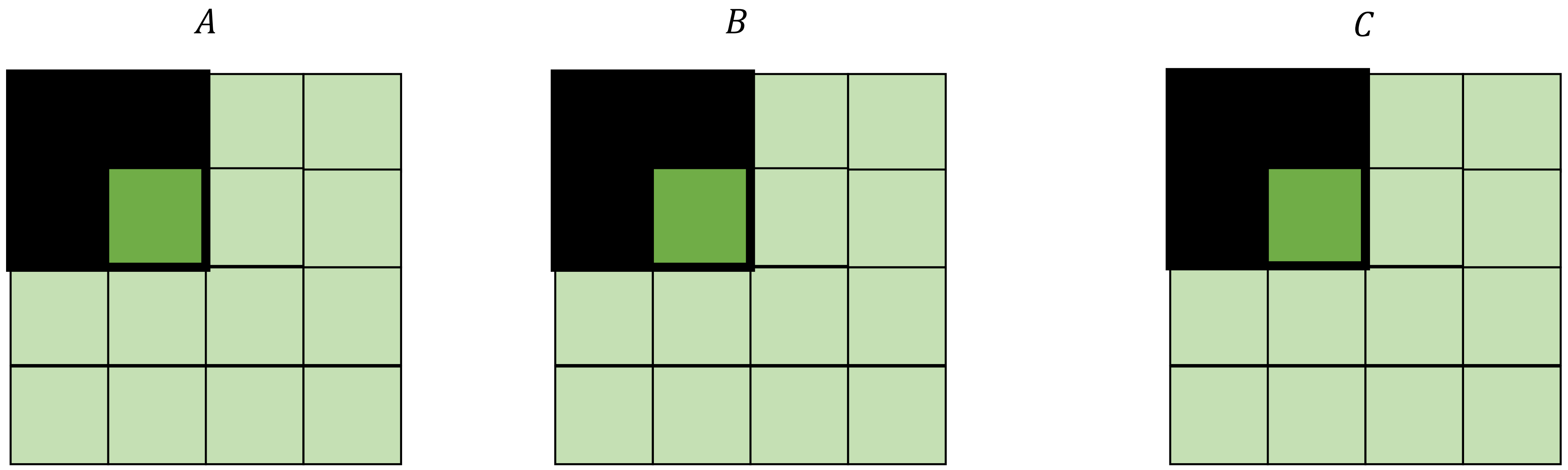


Miss on A,B, hit on C

Adding loop nestings

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

$$A = B + C^T$$



Hit on all!

Adding loop nestings

- Add two outer loops for both x and y

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

Adding loop nestings

- Add two outer loops for both x and y

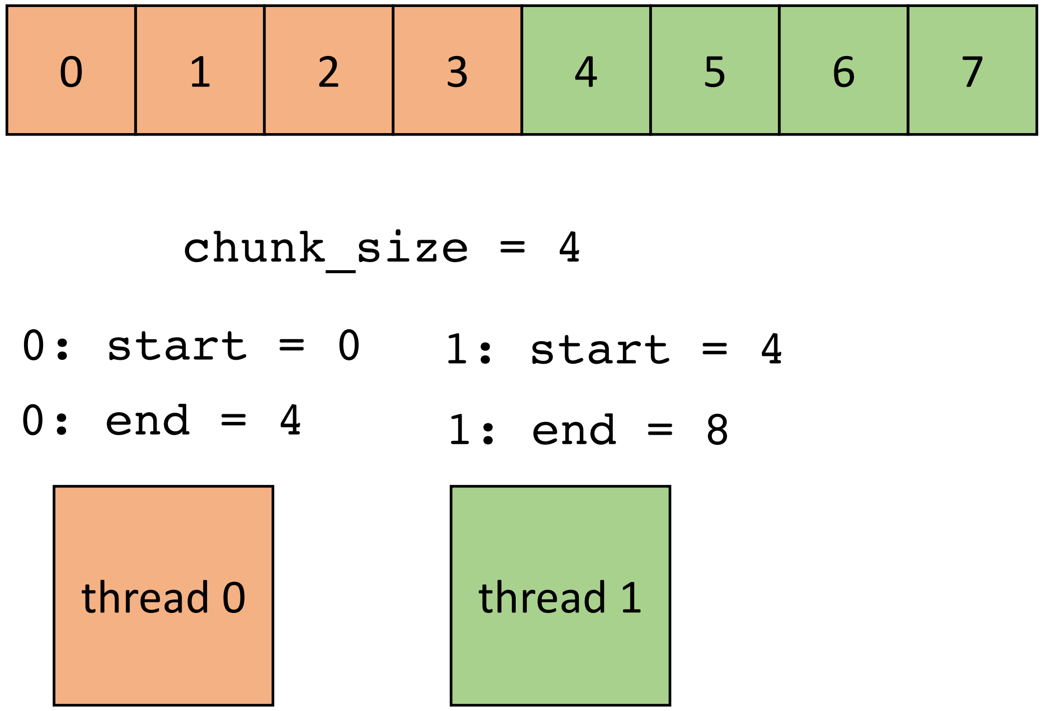
```
for (int xx = 0; xx < SIZE; xx += B) {
  for (int yy = 0; yy < SIZE; yy += B) {
    for (int x = xx; x < xx+B; x++) {
      for (int y = yy; y < yy+B; y++) {
        a[x*SIZE + y] = b[x*SIZE + y] + c[y*SIZE + x];
      }
    }
  }
}
```

How to parallelize

- different approaches

Regular Parallel Loops

- Example, 2 threads/cores, array of size 8



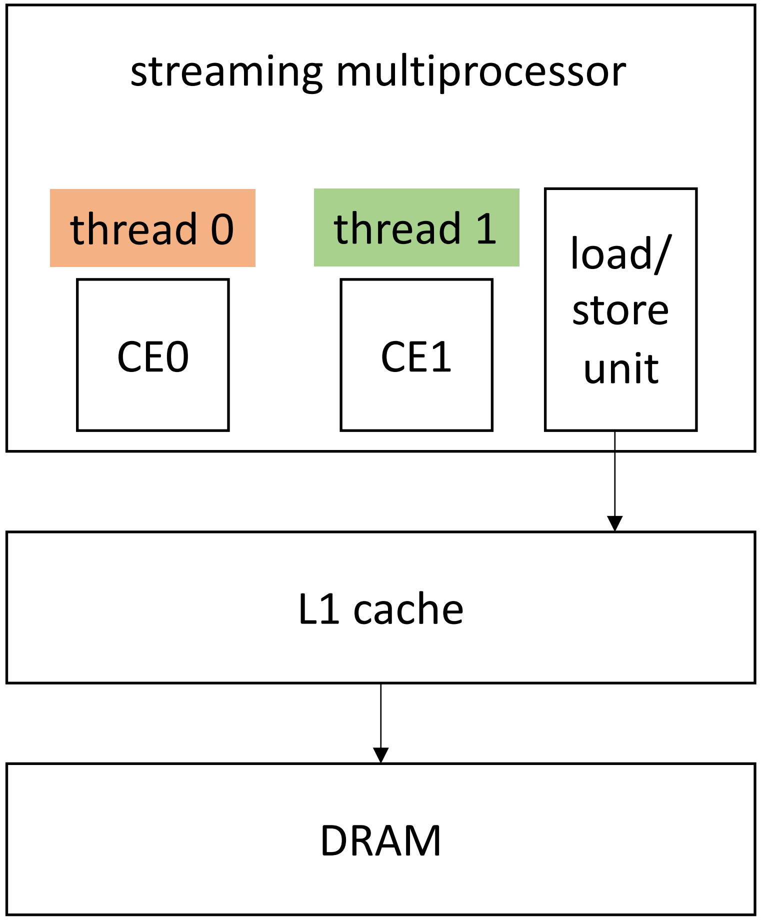
```
void parallel_loop(..., int tid) {

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size
  for (x = start; x < end; x++) {
    // work based on x
  }
}
```

What about streaming multiprocessors (GPUs)?

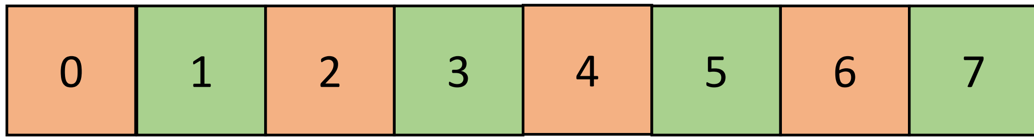
one streaming multiprocessor contains many small Compute Elements (CE)

CEs Can load adjacent memory locations simultaneously



ITER 0:

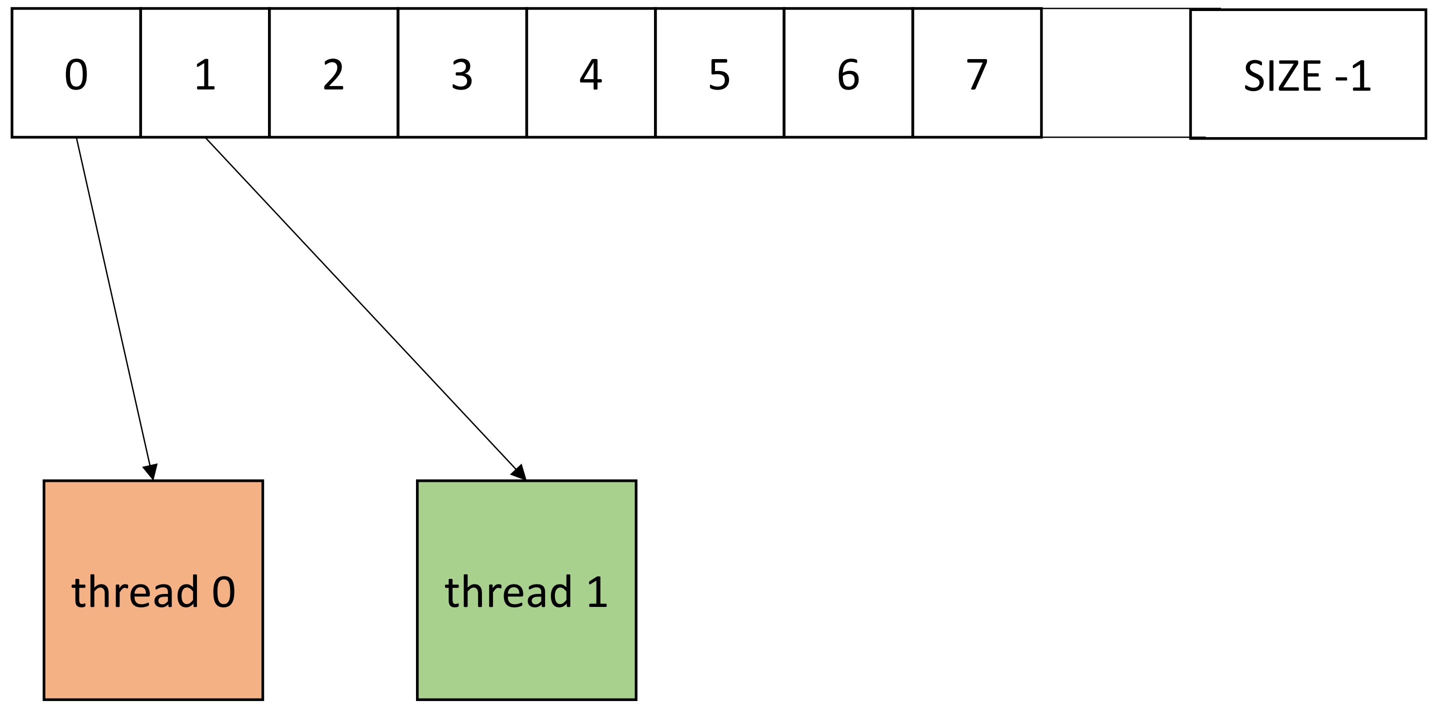
What about a striped pattern?



For irregular workloads

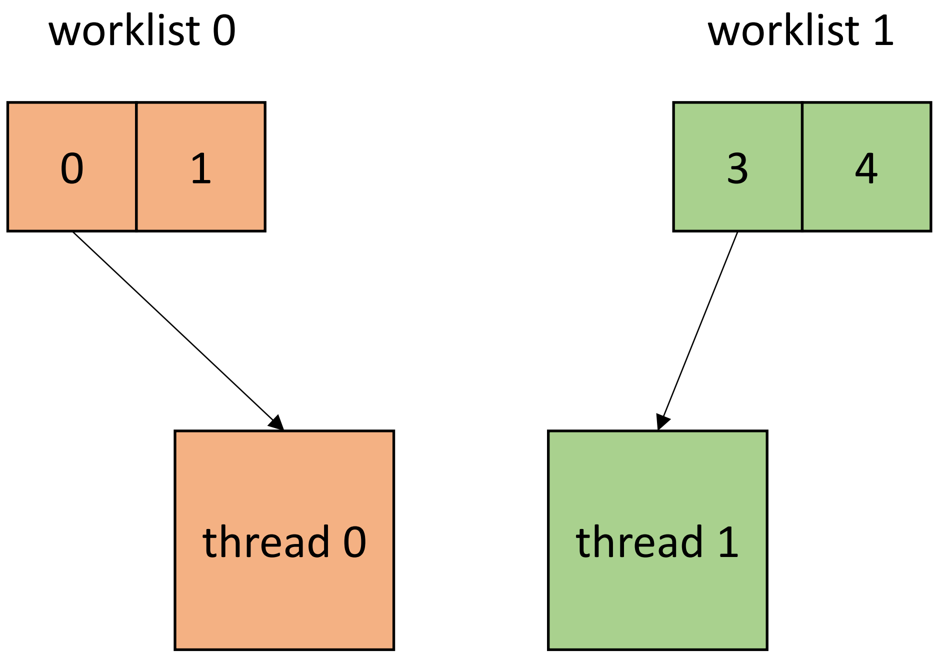
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



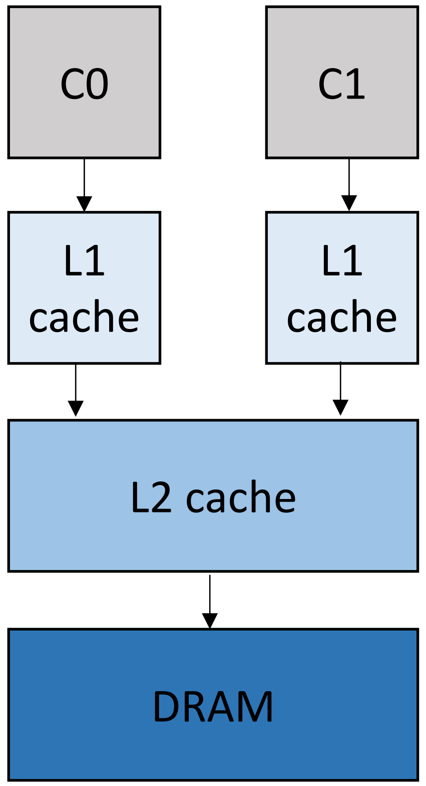
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread

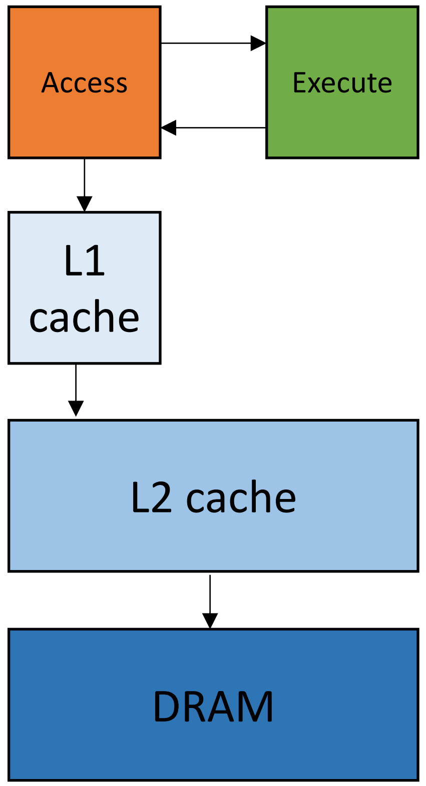


Finally, looking forward

Traditional SMP System



Decoupled Access/Execute System



Moving on to DSLs

CSE211: Compiler Design

Nov. 17, 2021

- **Topic**: Array processing DSL
- **Discussion questions**:
  - What is a DSL?
  - What are the benefits and drawbacks of a DSL?
  - What DSLs have you used?

What is a DSL

What is a DSL

- Objects in an object oriented language?
    - operator overloading (C++ vs. Java)

No clear answer!

- Libraries?
    - Numpy

- Does it need syntax?
    - Pytorch/Tensorflow

What is a DSL

- Not designed for general computation, instead designed for a domain
- How wide or narrow can this be?
  - Numpy vs TensorFlow
  - Pros and cons of this design?
- Domain specific optimizations
  - Optimizations do not have to work well in all cases

DSL designs

- Ease of expressiveness

```
sed ‘s/Utah/California’ address.txt
```

gnuplot

```
set title ”Parallel timing experiments"
set xlabel ”Threads”
set ylabel ”Speedup"
plot ”data.dat" with lines
```

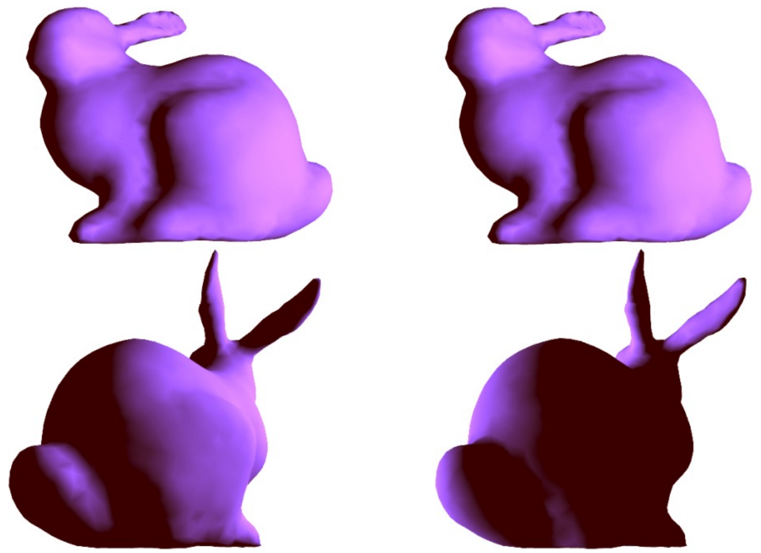
Other examples?

These require their own front end. What about Matplotlib?

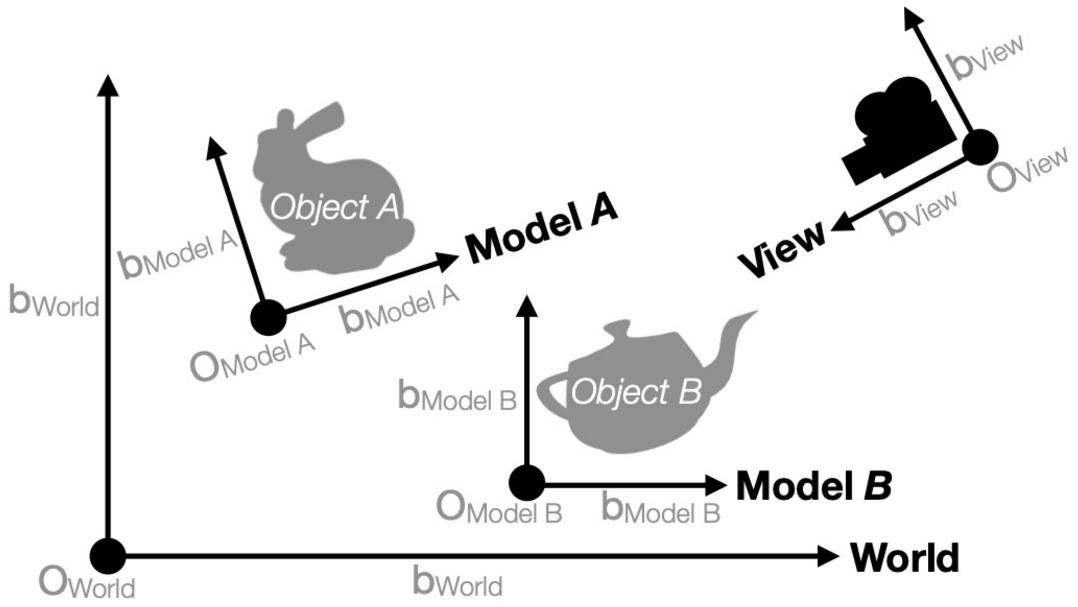
DSL designs

- Ease of expressiveness

make it harder to write bugs!



(a) Correct implementation. (b) With geometry bug.



Add reference tags to types: World or View

From: Geometry Types for Graphics Programming, OOPSLA 2020

DSL designs

- Ease of optimizations

Examples?

From homework 3:

What does this assume?
Optional in C++
Non-optional in Tensorflow

- reduction loops:
  - Entire computation is dependent
  - Typically short bodies (addition, multiplication, max, min)

1	2	3	4	5	6

addition: 21

max: 6

min: 1

Typically faster than implementations in general languages.

DSL designs

- Easier to reason about

Typically much fewer lines of code than implementations in general languages.

gnuplot example again

```
set title ”Parallel timing experiments"
set xlabel ”Threads”
set ylabel ”Speedup"
plot ”data.dat" with lines
```

tensorflow

```
tf.matmul(a, b)
```

What does an optimized matrix multiplication look like?

https://github.com/flame/blis/tree/master/kernels

DSL designs

- Easier to maintain

- *Optimizations and transforms are less general (more targeted).*
- *Less syntax (sometimes no syntax).*
- *Fewer corner cases.*

DSL design

- Recipe for a DSL talk:
  - Introduce your domain
  - Show scary looking optimized code
  - Show clean DLS code
  - Show performance improvement
  - Have a correctness argument

Homework 4

Homework 4

Choice of 3 DSLs

- **Halide**: classic DSL for image processing. Hugely influential in academic DSL design
- **GraphIt**: DSL for graph computations (BFS, PageRank, SSSP)
- **TVM**: DSL for machine learning

Homework 4

For the one that you pick:

- Read the original paper (I will specify the URL)
- Get the code running, in all cases there are tutorials
- Comment on design choices
- Comment on performance (run your own experiments!)

Homework 4

Submission: 5 page report (double spaced).

You can include graphs and code snippets, if you generate the data and code yourself!

Any figure or data that you use in the report that is not yours needs to be cited.

The rest of the lecture

- A discussion and overview of **Halide:**
  - Huge influence on modern DSL design
  - Great tooling
  - Great paper
- Originally: A DSL for image pipelining:

Brighten example

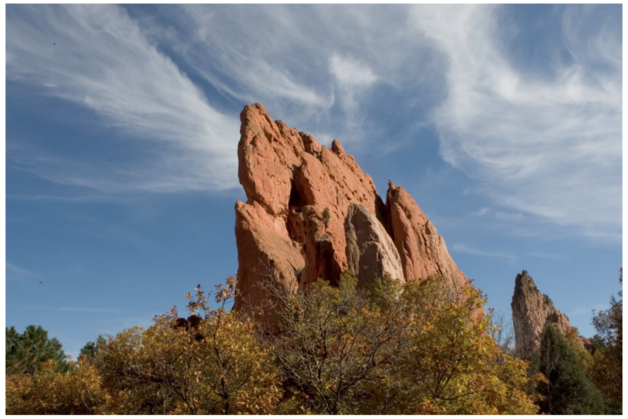
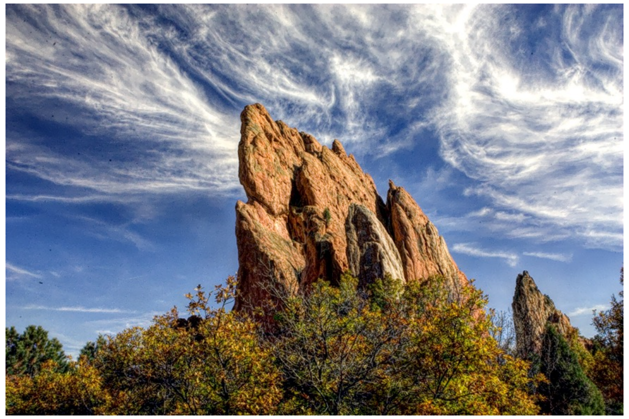
from: https://halide-lang.org/tutorials/tutorial_lesson_02_input_image.html

Motivation:

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 Halide 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/

Decoupling computation from optimization

- We love Halide not only because it can make pretty pictures very fast
- We love it because it changed the level of abstraction for thinking about computation and optimization
- (Halide has been applied in many other domains now, turns out everything is just linear algebra)

Example

- in C++

```
for (int x = 0; x < x_size; x++) {
  for (int y = 0; y < y_size; y++) {
        a[x,y] = b[x,y] + c[x,y];
   }
}
```

Which one would you write?

```
for (int y = 0; y < y_size; y++) {
  for (int x = 0; x < x_size; x++) {
        a[x,y] = b[x,y] + c[x,y];
   }
}
```

Optimizations are a black box

- What are the options?
  - -O0, -O1, -O2, -O3
  - Is that all of them?
  - What do they actually do?

https://stackoverflow.com/questions/15548023/clang-optimization-levels

Optimizations are a black box

- What are the options?
  - -O0, -O1, -O2, -O3
  - Is that all of them?
  - What do they actually do?

- ***Answer***: they do their best for a wide range of programs. The common case is that you should not have to think too hard about them.

- ***In practice***, to write high-performing code, you are juggling computation and optimization in your mind!

Next Class

- Continuing on Halide