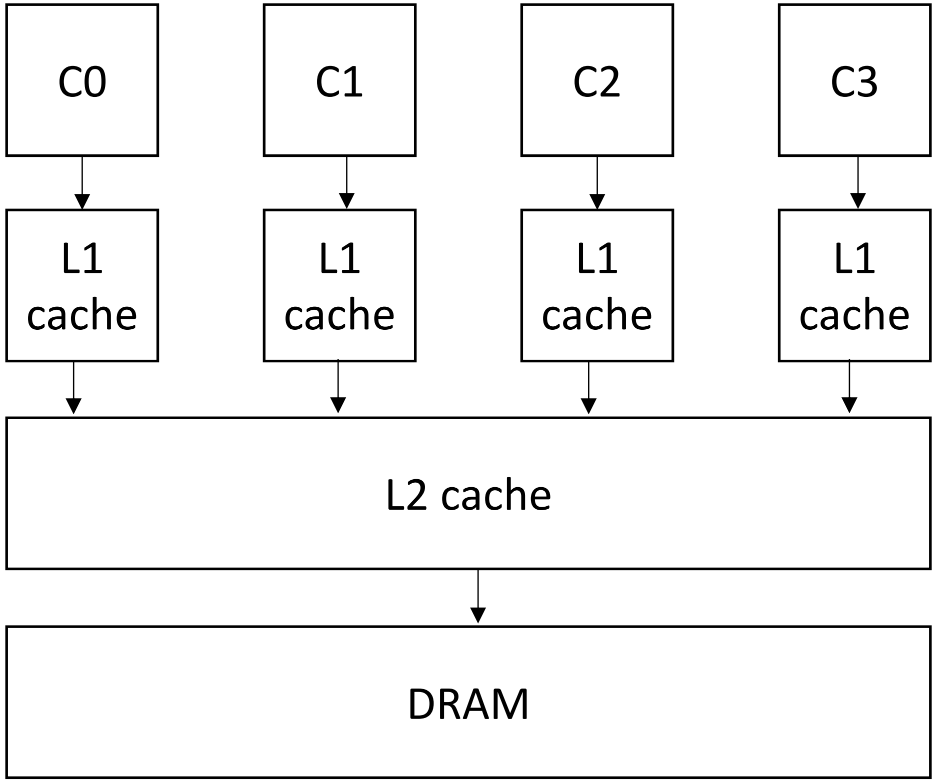
CSE211: Compiler Design

Nov. 17, 2020

- **Topic**: SMP parallelism
  - Nesting orders
  - Reordering nestings
  - Irregular parallelism

- **Discussion questions**:
  - What is memory locality and why does it matter?
  - What do you do if parallel threads do not have the same amount of work?



Announcements

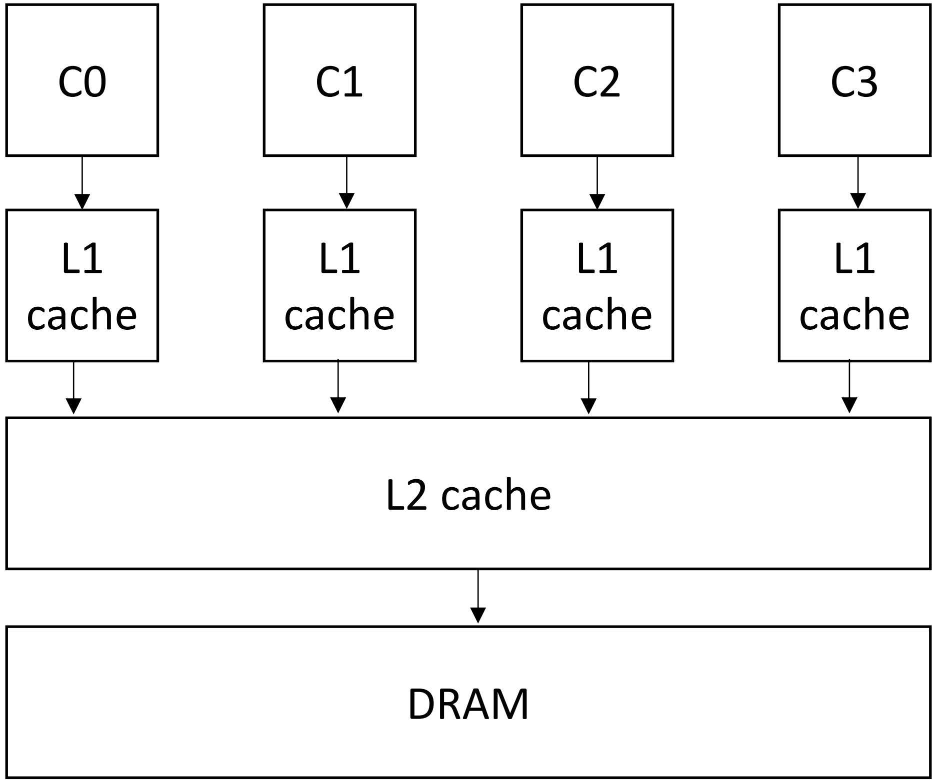
- Midterm is out. Clarification questions are posted as discussions on Canvas. Forgot to publish...
  - Due on Thursday.
- HW3 is released. Due Dec. 4
- HW1 is graded: good job everyone!
- Paper/projects proposals due Nov. 24

CSE211: Compiler Design

Nov. 17, 2020

- **Topic**: SMP parallelism
  - Nesting orders
  - Reordering nestings
  - Irregular parallelism

- **Discussion questions**:
  - What is memory locality and why does it matter?
  - What do you do if parallel threads do not have the same amount of work?



From last week:

- We want to find loops that are safe to parallelize
- Condition: outer loop iterations must be independent: they can execute in any order and provide the same result
  - using a constraint solver to detect write-write conflicts and read-write conflicts
- *new:* push/pop Z3 commands: this can save the state of the solver.

Example

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]**2;
}
```

two integers: i_x != i_y
i_x >= 0
i_x < 128
i_y >= 0
i_y < 128

read-write conflict check
i_x % 64 == i_y + 64

write-write conflict check
i_x % 64 == i_y % 64

General formula:

```
for (int i0 = init0; i0 < bound0(); i0++) {
    for (int i1 = init1(i0); i1 < bound1(i0); i1++) {
        ...
        for (int iN = initN(i0, i1, ...); iN < boundN(i0, i1 ...); iN++) {
            write(a, write_index(i0, i1 .. iN))
            read(a, read_index(i0, i1 .. iN));
        }
    }
}
```

1. Create two variables for each loop variable: $i0_x$, $i0_y$, $i1_x$, $i1_y$ ...

Set outer loop: $i0_x$!= $i0_y$

2. Constrain them to be inside their bounds:

for w in from (0,N): $iw_{x,y}$ >= initw(...), $iw_{x,y}$ < boundN(...)

3. Enumerate all pairs of potential write-write conflicts:

`check: write_index(i0x, i1x .. iNx) == write_index(i0y, i1y, ... iNy)`

4. Do the same for write-read conflicts

DOALL Loops

- These loops are called DOALL loops
- Once found, they can be passed to additional passes to fine-tune the parallelism (locality, number of threads, scheduling etc.)
- This lecture: nesting order and scheduling

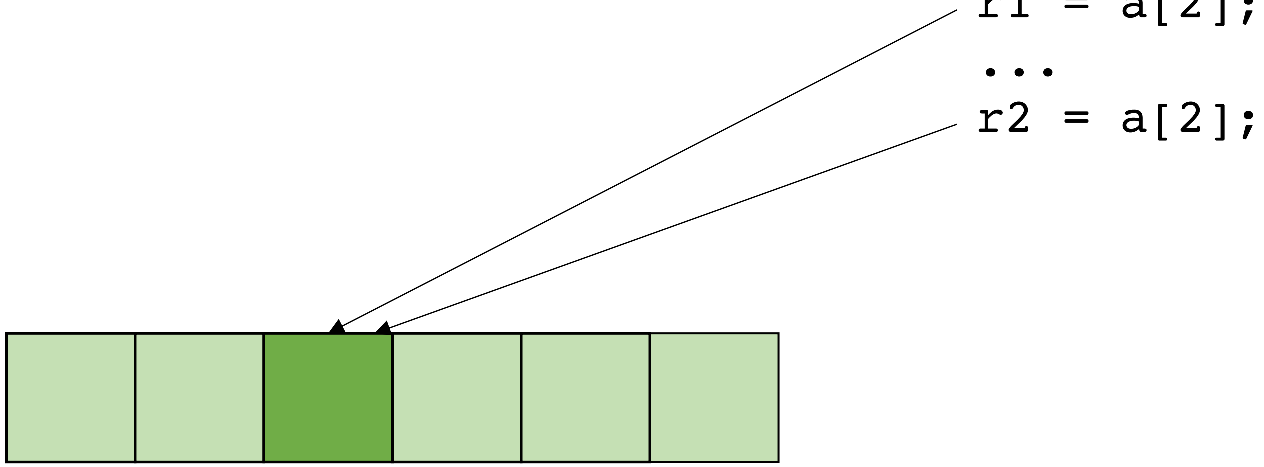
Transforming Loops

- Locality is key for good parallel performance:

Transforming Loops

- Locality is key for good parallel performance:

- Two types of locality:
  - Temporal locality
  - Spatial locality



temporal locality

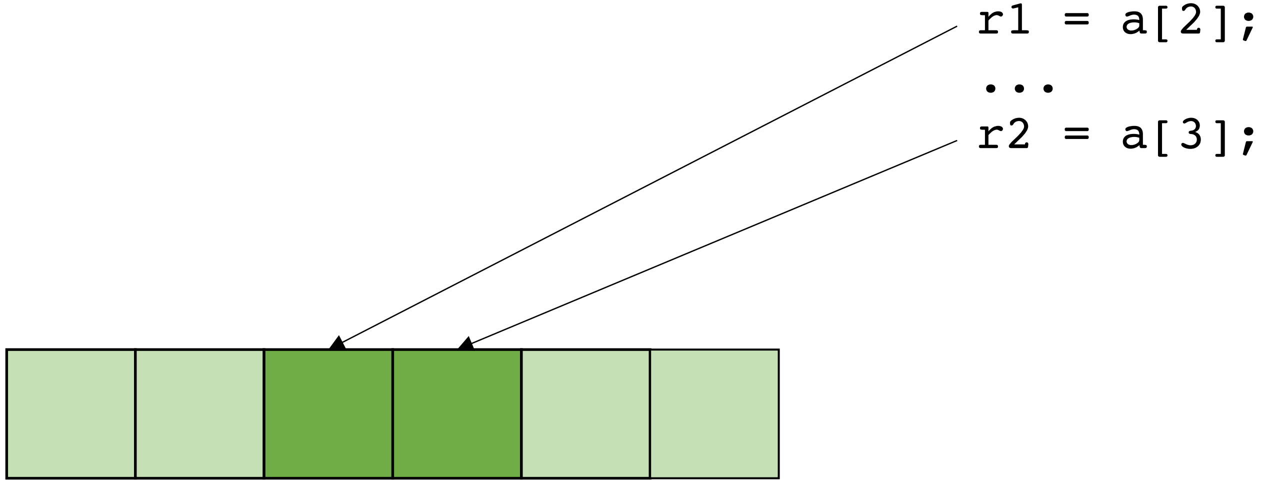
```
r1 = a[2];
...
r2 = a[2];
```

Transforming Loops

- Locality is key for good parallel performance:

- Two types of locality:
  - Temporal locality
  - Spatial locality

spatial locality

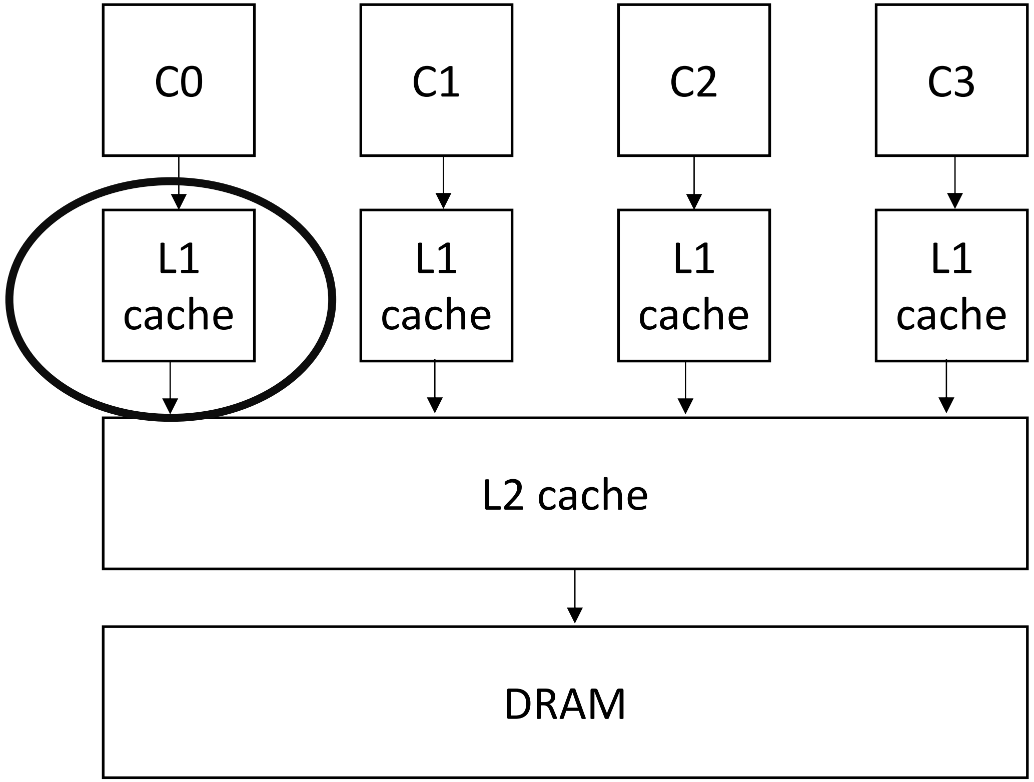


```
r1 = a[2];
...
r2 = a[3];
```

how far apart can memory locations be?

Transforming Loops

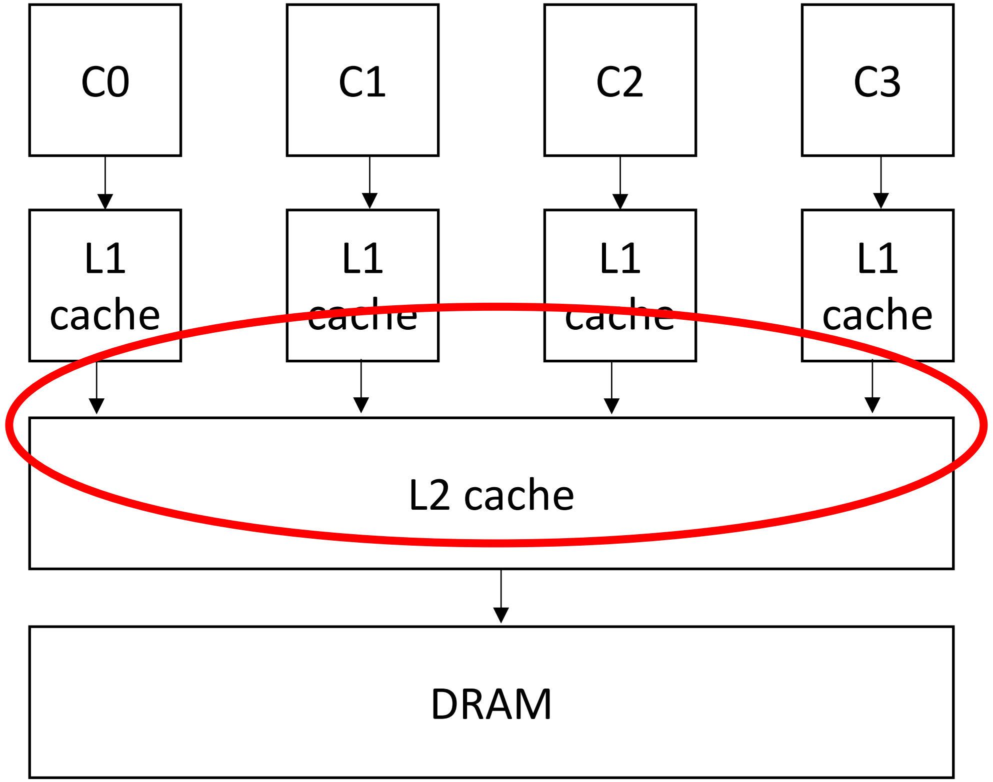
- Locality is key for good parallel performance:



good data locality: cores will spend most of their time accessing private caches

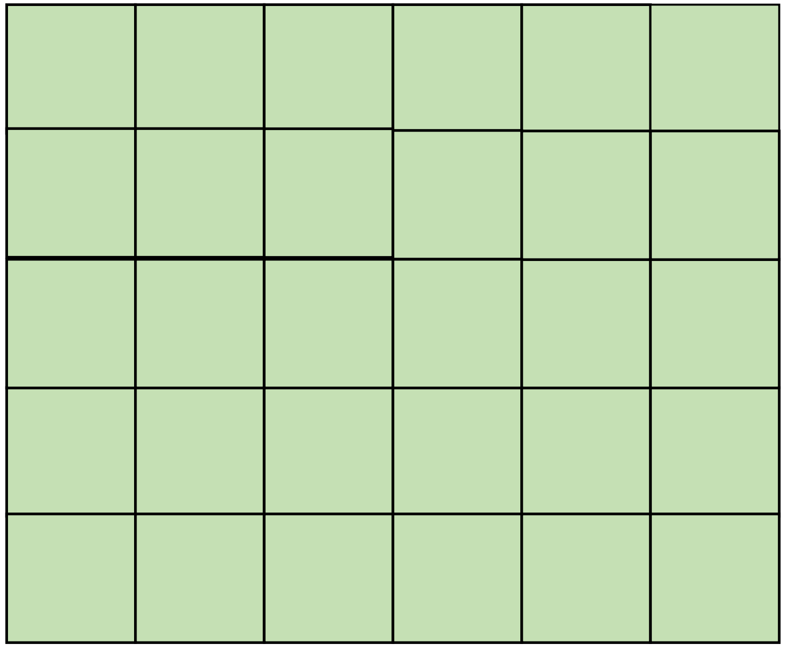
Transforming Loops

- Locality is key for good parallel performance:



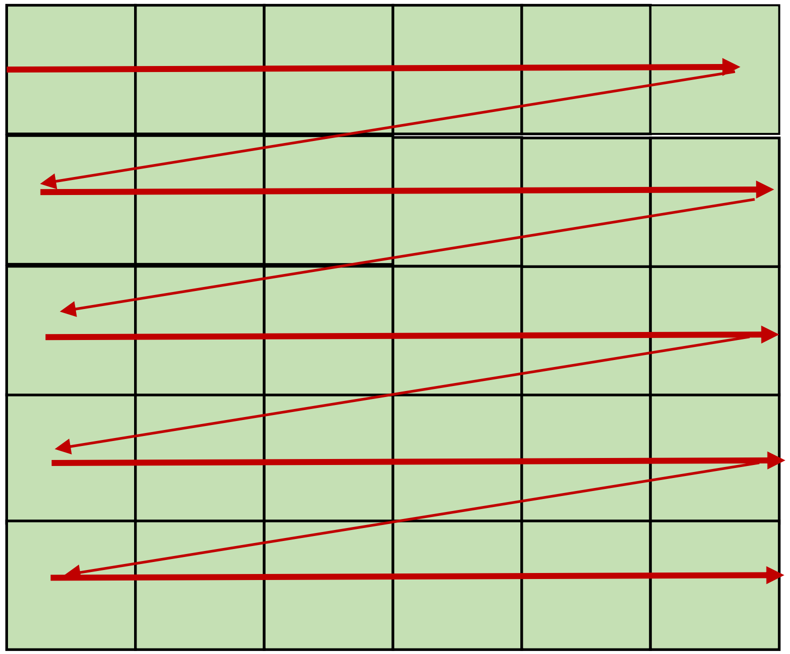
Bad data locality: cores will pressure and thrash shared memory resources

How multi dimensional arrays are stored:



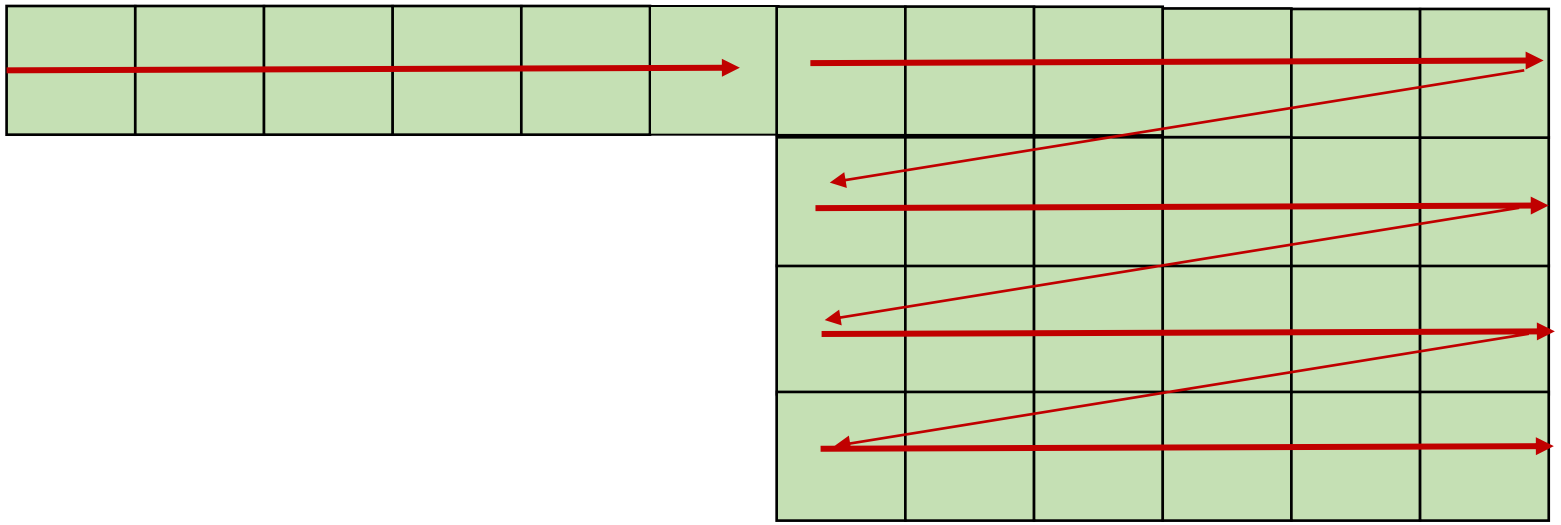
How multi dimensional arrays are stored:

Row major



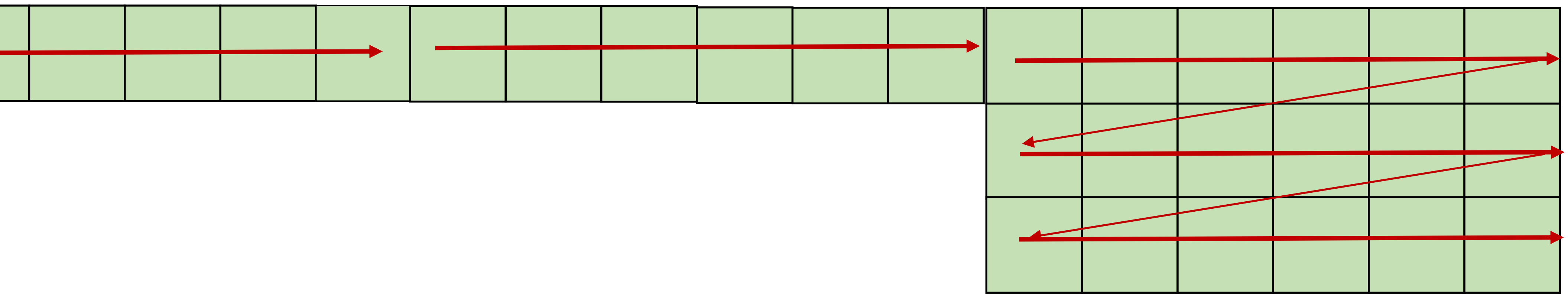
How multi dimensional arrays are stored:

Row major



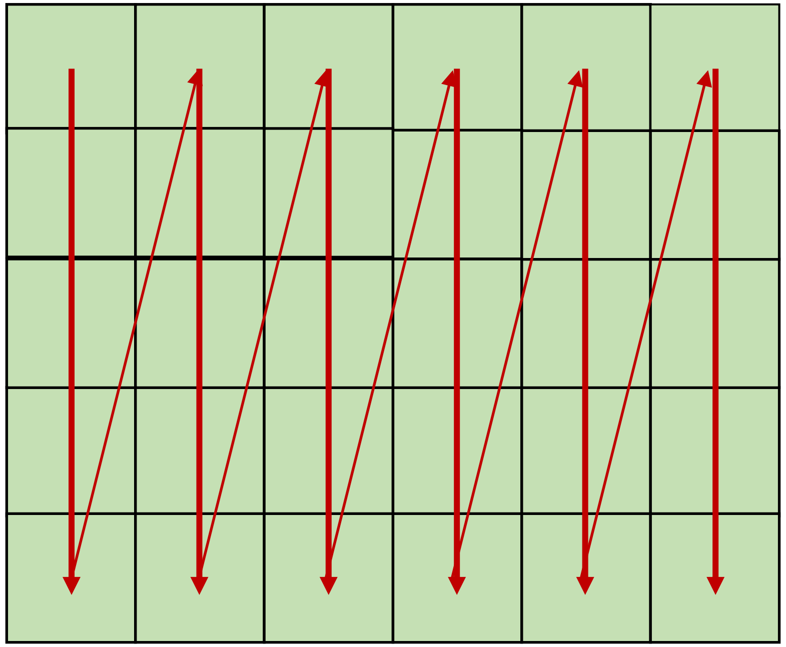
How multi dimensional arrays are stored:

Row major

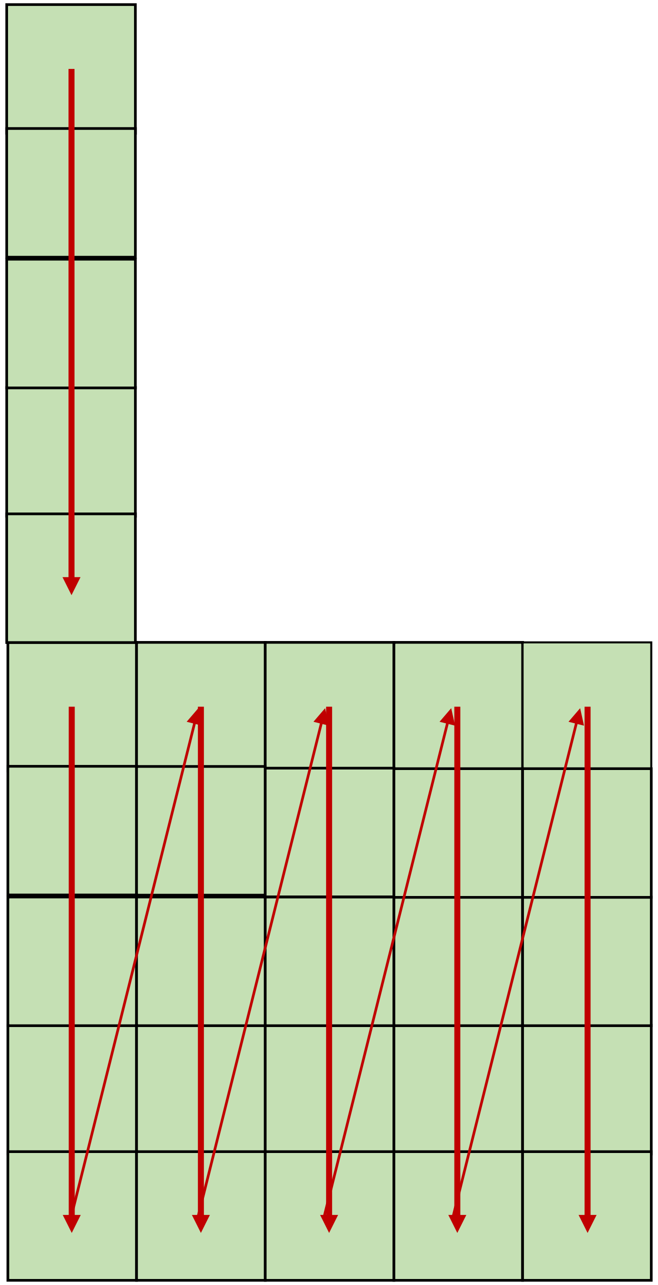


How multi dimensional arrays are stored:

Column major?
Fortran
Matlab
R

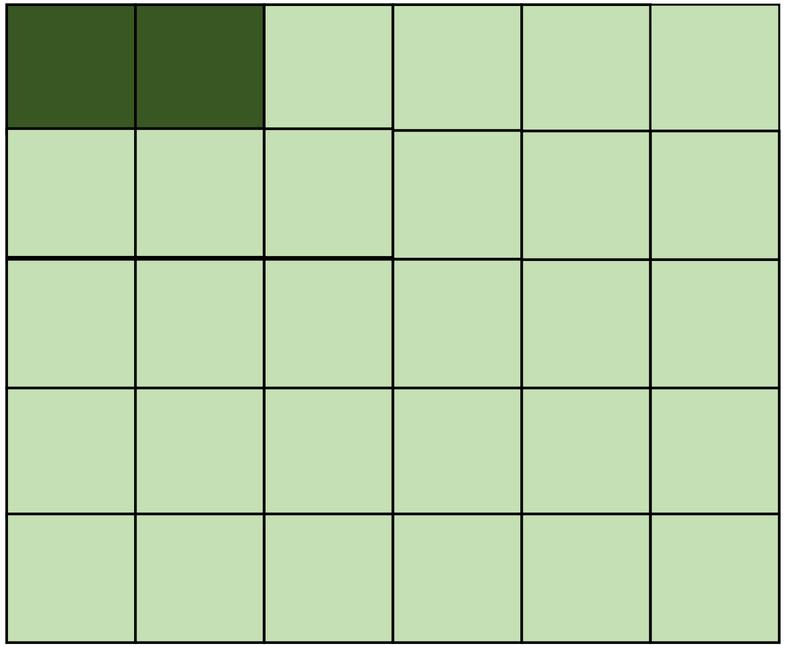


How multi dimensional arrays are stored:



Column major?
Fortran
Matlab
R

How multi dimensional arrays are stored:

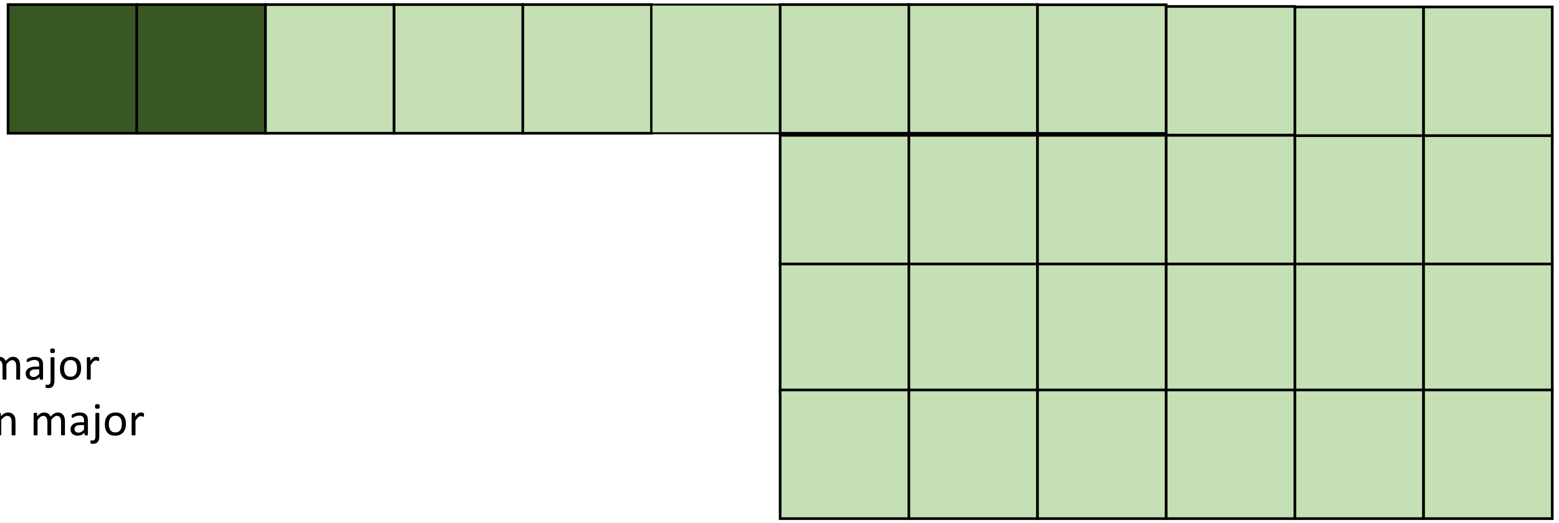


```
x1 = a[0,0];
x2 = a[0, 1];
```

good pattern for row major
bad pattern for column major

How multi dimensional arrays are stored:

unrolled row major: still has locality



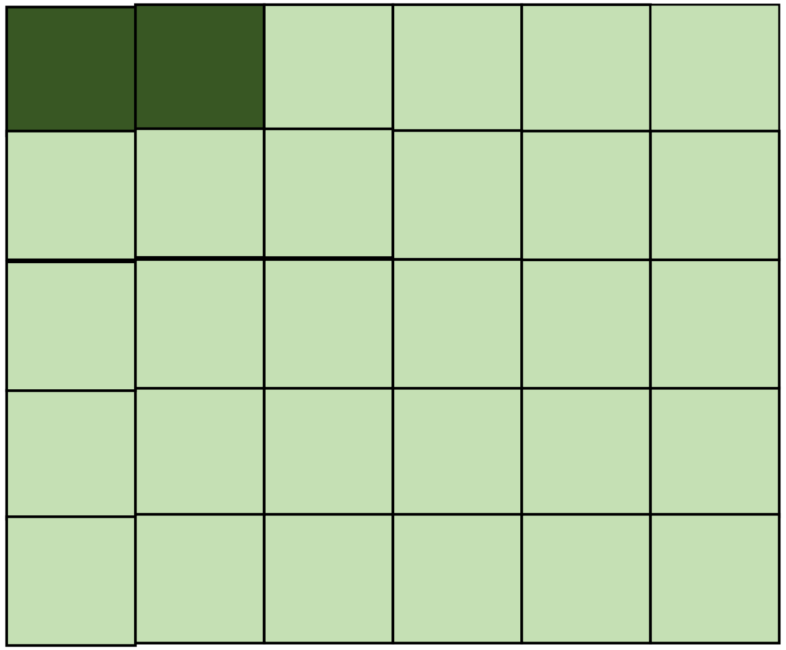
```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major

How multi dimensional arrays are stored:

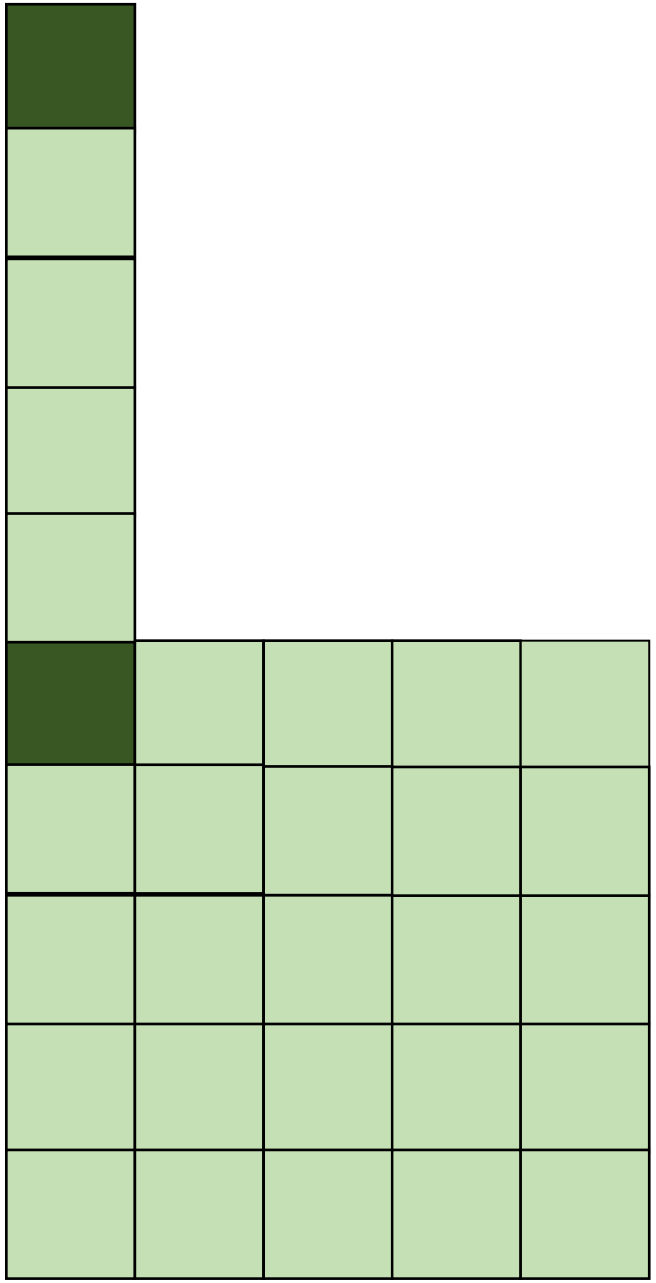
```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major



How multi dimensional arrays are stored:

unrolled
column
major:
Bad locality



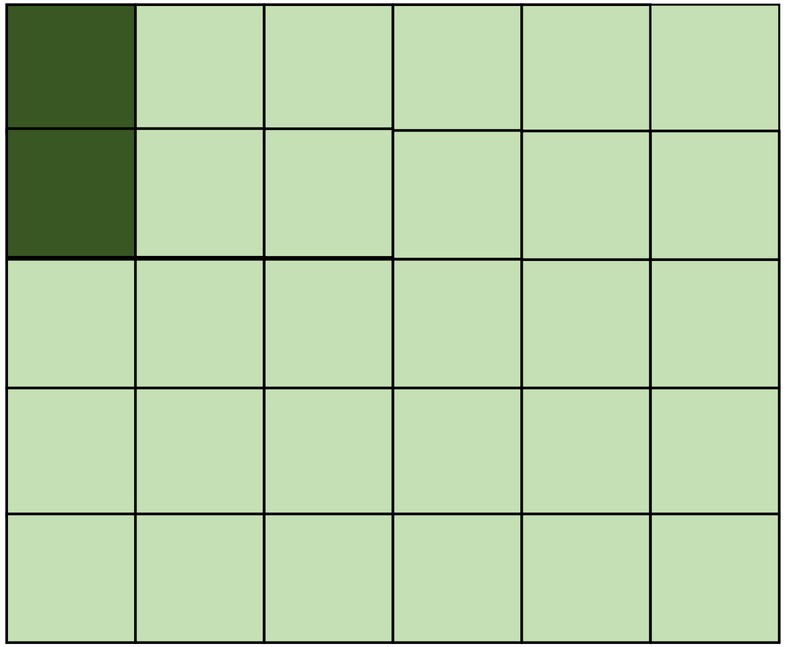
```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major

How multi dimensional arrays are stored:

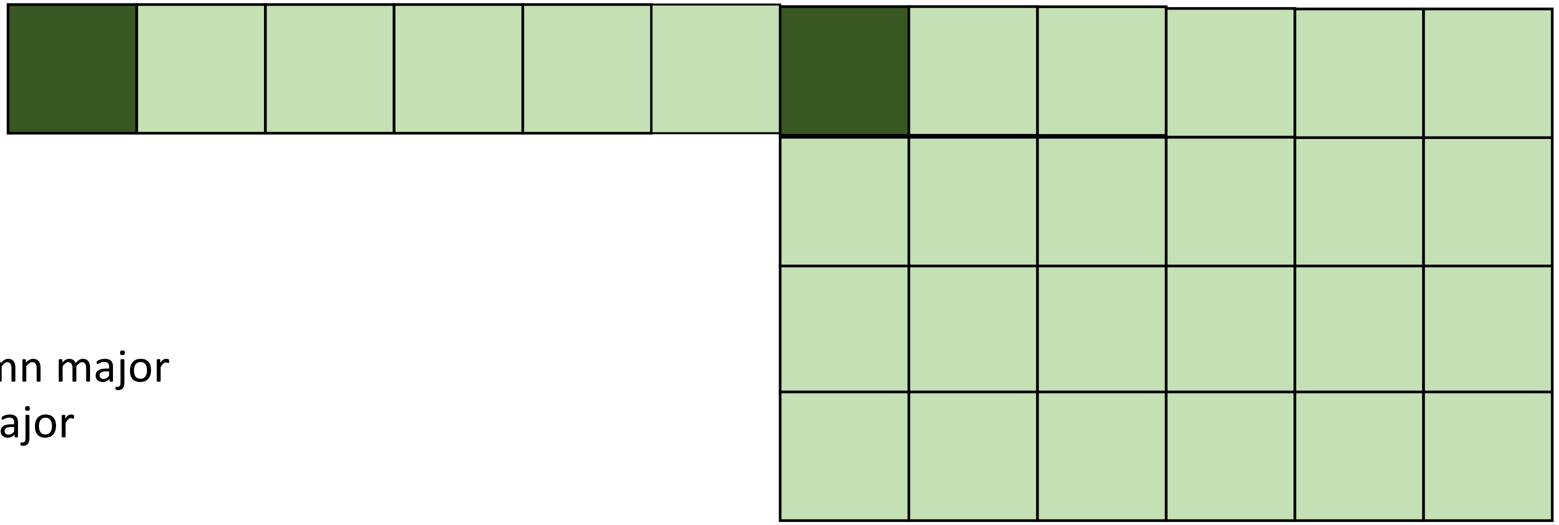
```
x1 = a[0,0];
x2 = a[1, 0];
```

good pattern for column major
bad pattern for row major



How multi dimensional arrays are stored:

row major unrolled: bad spatial locality

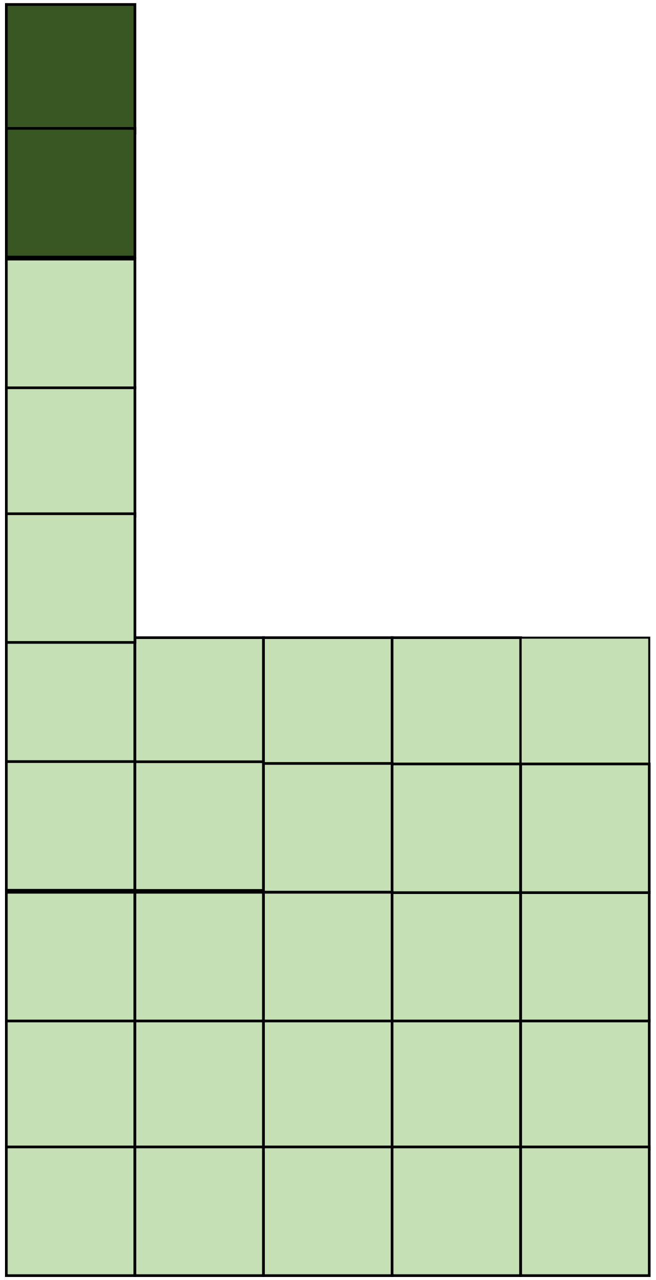


```
x1 = a[x,y];
x2 = a[x+1, y];
```

good pattern for column major
bad pattern for row major

How multi dimensional arrays are stored:

unrolled
column
major:
good locality



```
x1 = a[x,y];
x2 = a[x+1, y];
```

good pattern for column major
bad pattern for row major

How much does this matter?

```
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];
   }
}
```

```
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];
   }
}
```

which will be faster?
by how much?

Demo

How to reorder loop nestings?

- For a DOALL loop, if loop bounds are independent, they can simply be re-ordered.
- If they are dependent...

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

bad nesting order for row-major!

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

bad nesting order for row-major!

but iteration variables are dependent

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

bad nesting order for row-major!

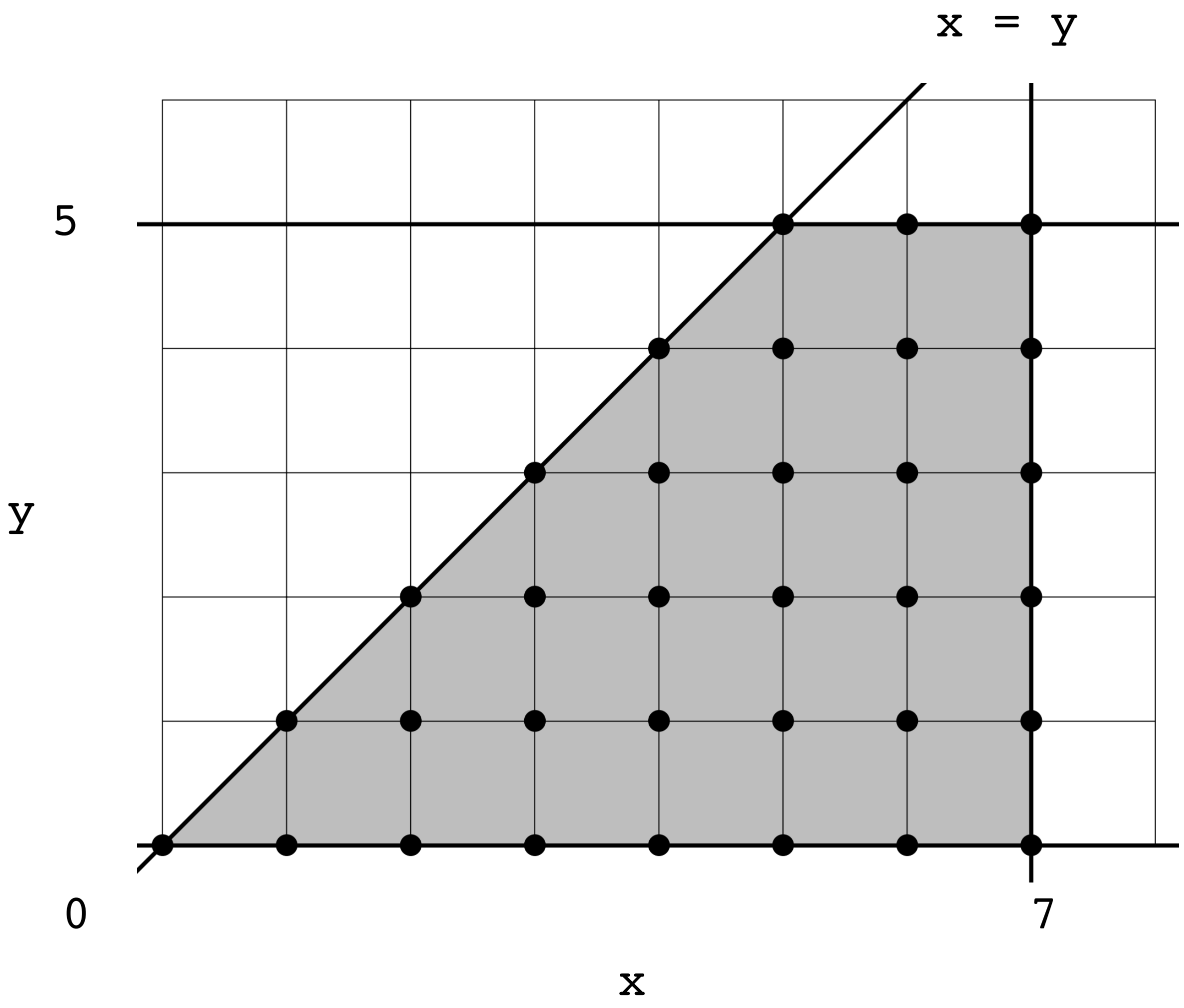
but iteration variables are dependent

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

Example:

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

System with N variables can be viewed as an N dimensional polyhedron



Fourier-Motzkin elimination:

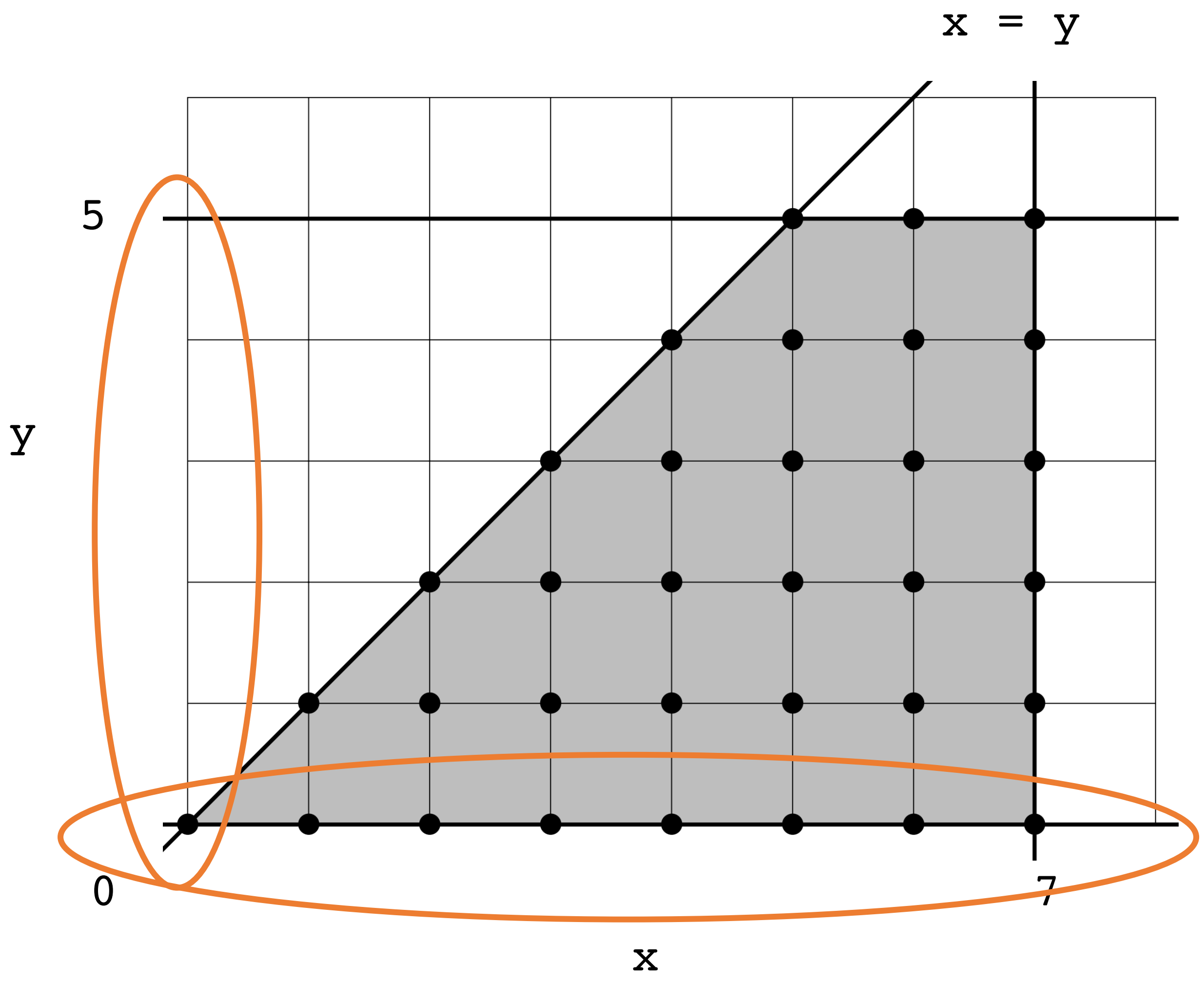
- Given a system of inequalities with N variables, reduce it to a system with N - 1 variables.

- A system of inequalities describes an N-dimensional polyhedron. Produce a system of equations that projects the polyhedron onto an N-1 dimensional space

Example:

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

System with N variables can be viewed as an N dimensional polyhedron



Fourier-Motzkin elimination:

- To eliminate variable x_i:
  For every pair of lower bound L_i and upper bound U_i on x_i, create:

  $$L_i \leq x_i \leq U_i$$

  Then simply remove x_i :

  $$L_i \leq U_i$$

Example: remove y from the constraints

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

All pairs of upper/lower bounds on y:

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

Example: remove y from the constraints

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

All pairs of upper/lower bounds on y:

```
0 <= y <= 5
0 <= y <= x
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

Example: remove y from the constraints

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

All pairs of upper/lower bounds on y:

```
0 <= y <= 5
0 <= y <= x
```

Then eliminate y:

```
0 <= 5
0 <= x
```

Example: remove y from the constraints

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

All pairs of upper/lower bounds on y:

```
0 <= y <= 5
0 <= y <= x
```

Then eliminate y:

```
0 <= 5
0 <= x
```

Example: remove y from the constraints

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

All pairs of upper/lower bounds on y:

`0 <= y <= 5`
`0 <= y <= x`

Then eliminate y:

`0 <= x`

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

Example: remove y from the constraints

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

All pairs of upper/lower bounds on y:

```
0 <= y <= 5
0 <= y <= x
```

Then eliminate y:

```
0 <= x
```

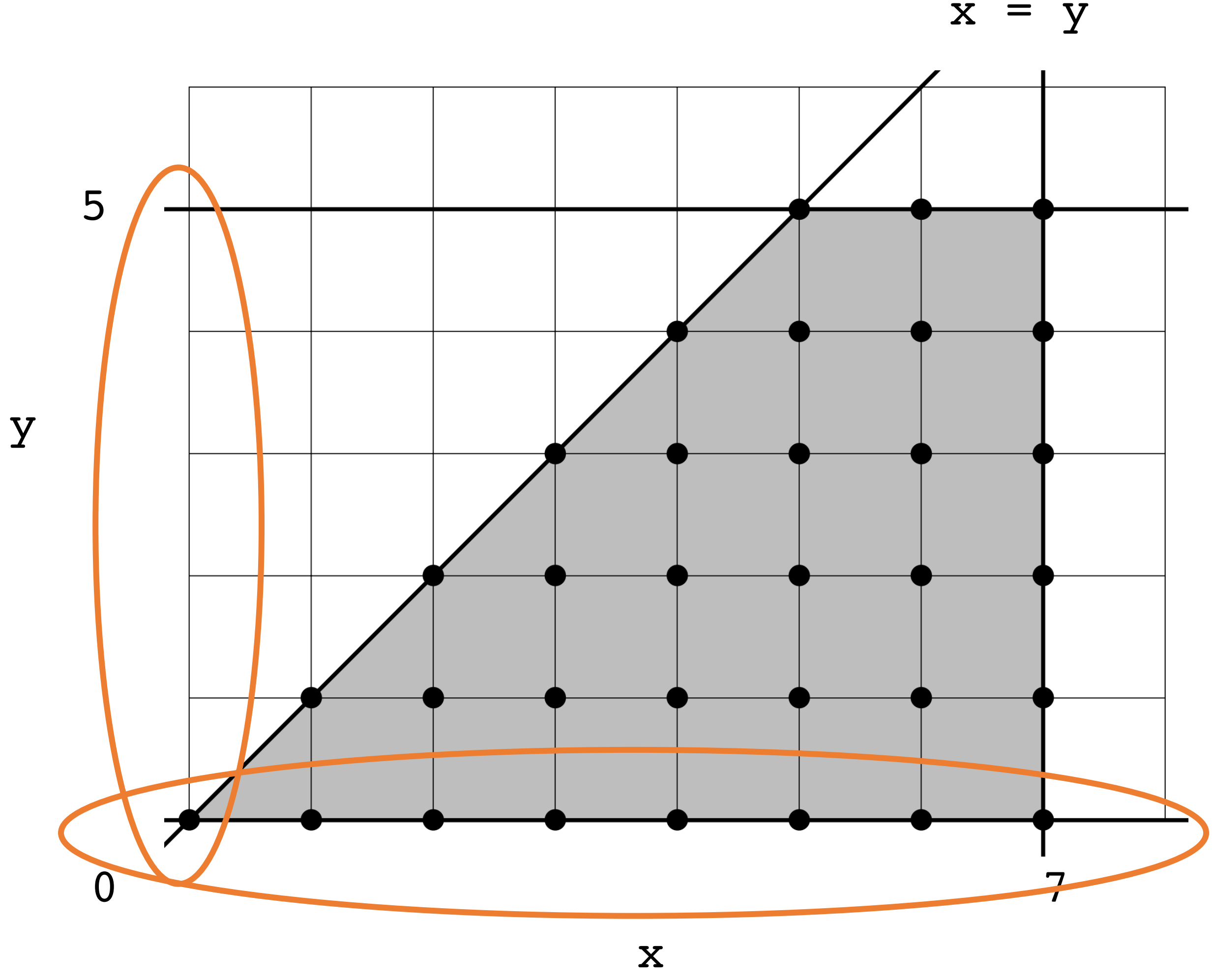
loop constraints without y:

```
x >= 0
x <= 7
```

Example:

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

System with N variables can be viewed as an N dimensional polyhedron



Reording Loop bounds:

- Given a new order: $[x_0, x_1, x_2, \ldots x_n]$
- For each variable x_i : perform Fourier-Motzkin elimination to eliminate any variables that come after x_i in the new order.
- Instantiate loop conditions for x_i, potentially using `max/min` operators

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

new order: [x,y]

for x: eliminate y using FM elimination:

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

new order: [x,y]

for x: eliminate y using FM elimination:

x loop constraints without y:

```
x >= 0
x <= 7
```

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

new order: [x,y]

for x: eliminate y using FM elimination:

x loop constraints without y:

```
x >= 0
x <= 7
```

```
y loop constraints:
y >= 0
y <= 5
y <= x
```

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

new order: [x,y]

for x: eliminate y using FM elimination:

x loop constraints without y:

```
x >= 0
x <= 7
```

```
y loop constraints:
y >= 0
y <= 5
y <= x
```

Example:

```
for (y = 0; y <= 5; y++) {
  for (x = y; x <= 7; x++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

```
loop constraints
y >= 0
y <= 5
x >= y
x <= 7
```

new order: [x,y]

for x: eliminate y using FM elimination:

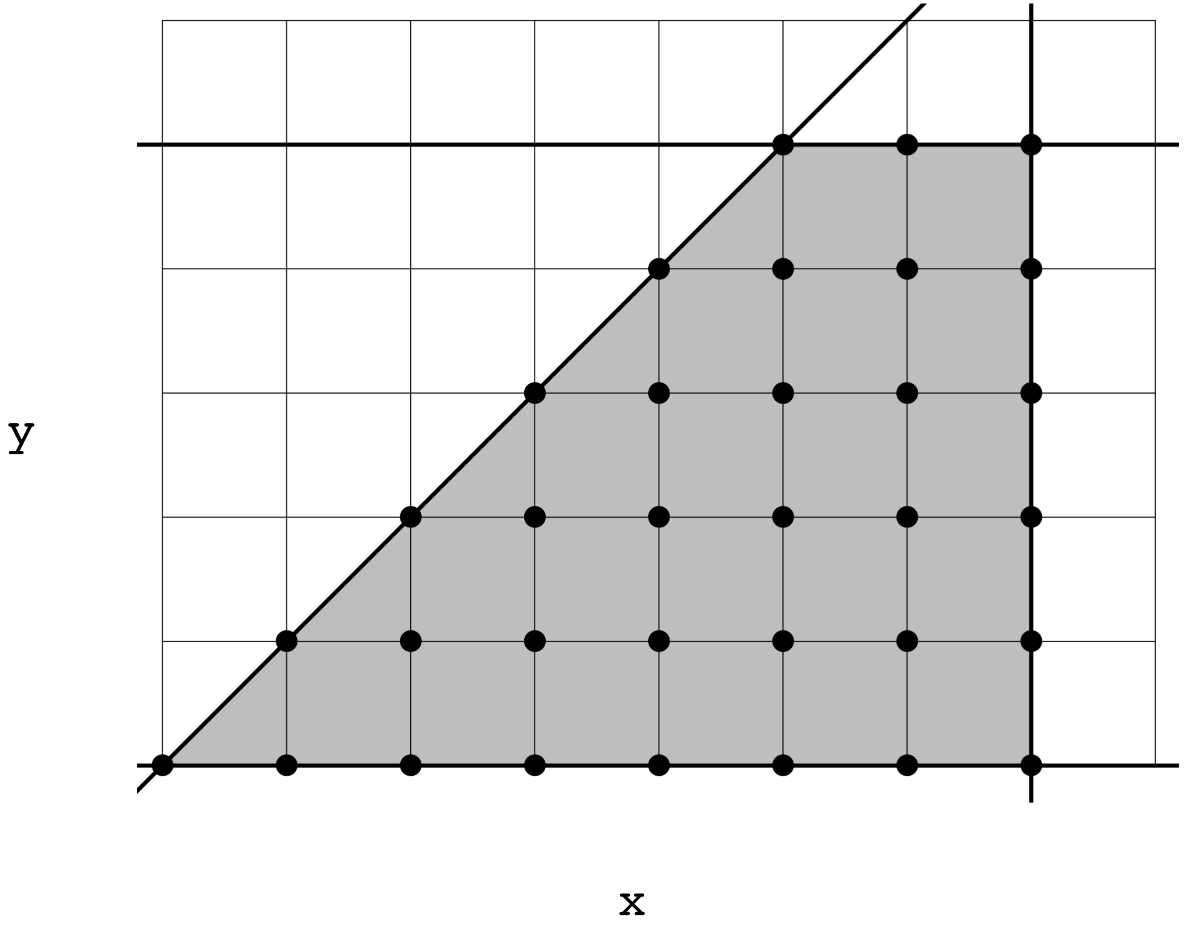
x loop constraints without y:

```
x >= 0
x <= 7
```

```
y loop constraints:
y >= 0
y <= min(x,5)
```

Example:

```
for (x = 0; x <= 7; x++) {
  for (y = 0; y <= min(x,5); y++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```



x loop constraints without y:

```
x >= 0
x <= 7
```

```
y loop constraints:
y >= 0
y <= min(x,5)
```

Reordering loop bounds

- only works if loop increments by 1; assumes a closed polyhedron

- best performance when array indexes are simple:
    - e.g.: `a[x,y]`
    - harder with, e.g.: `a[x*5+127, y+x*37]`
    - There exists schemes to automatically detect locality. Reach chapter 10 of the Dragon book

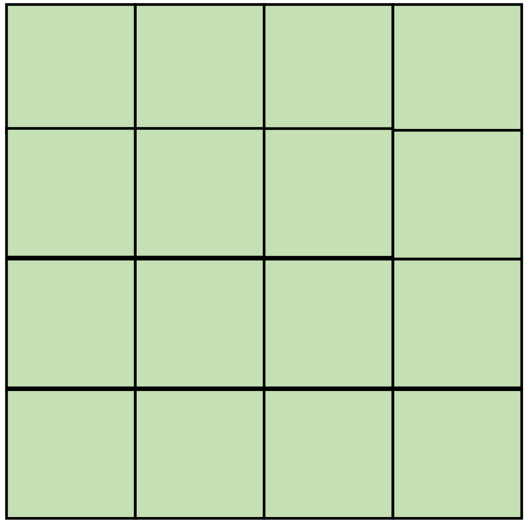
- compiler implementation allows exploration and auto-tuning

Adding loop nestings

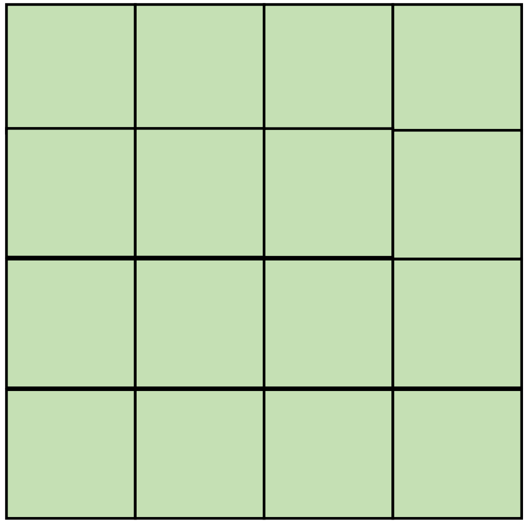
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

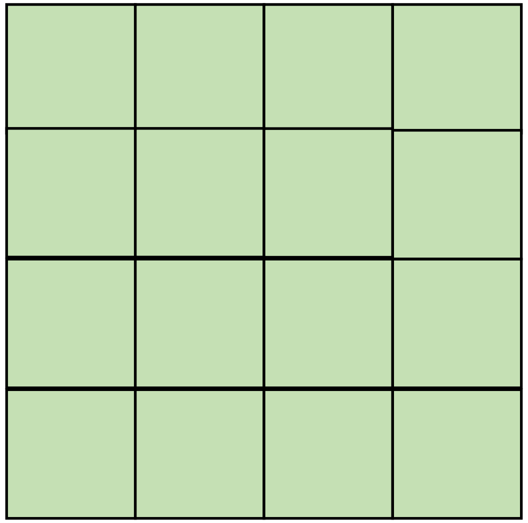
A



B



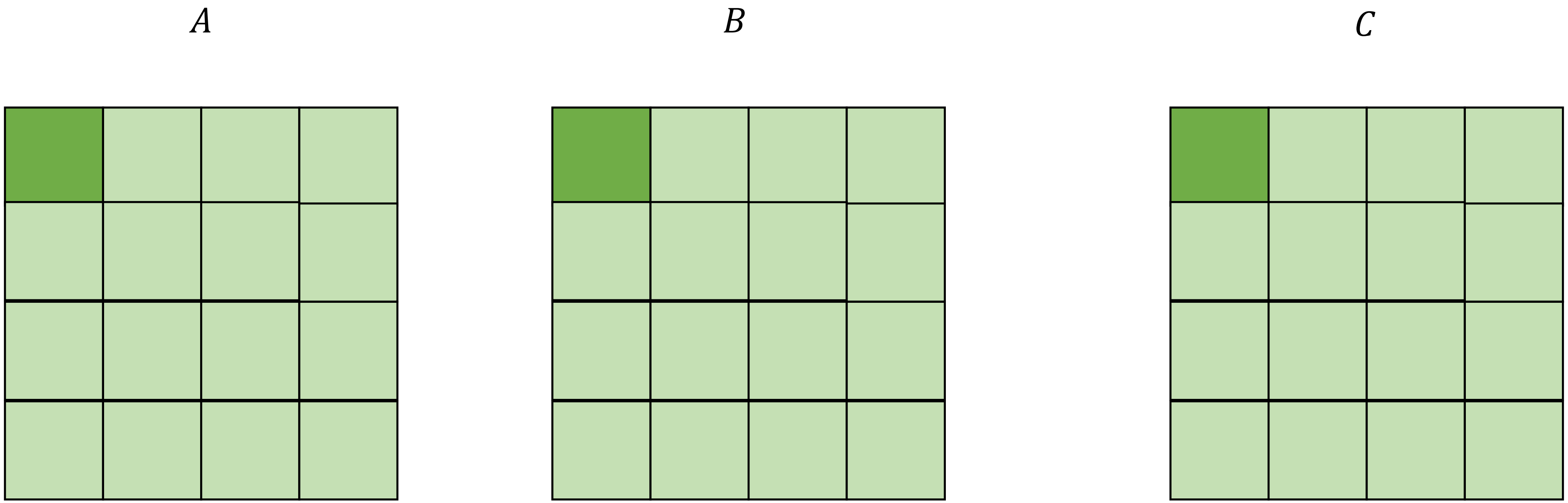
C



Adding loop nestings

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

$$A = B + C^T$$

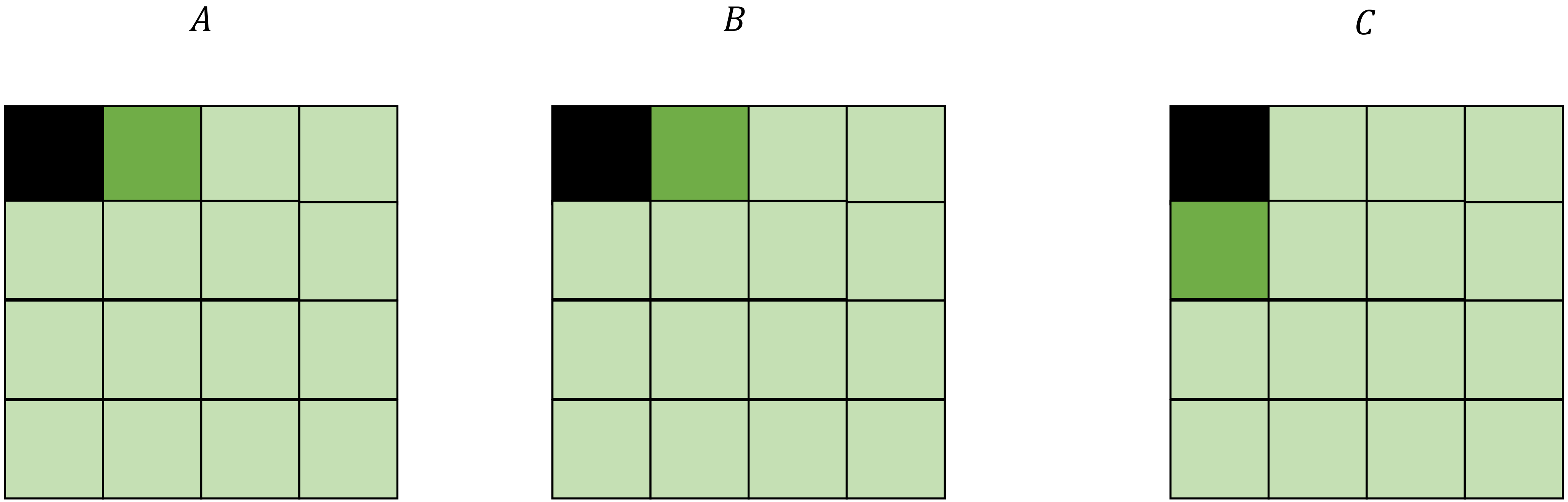


cold miss for all of them

Adding loop nestings

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

$$A = B + C^T$$



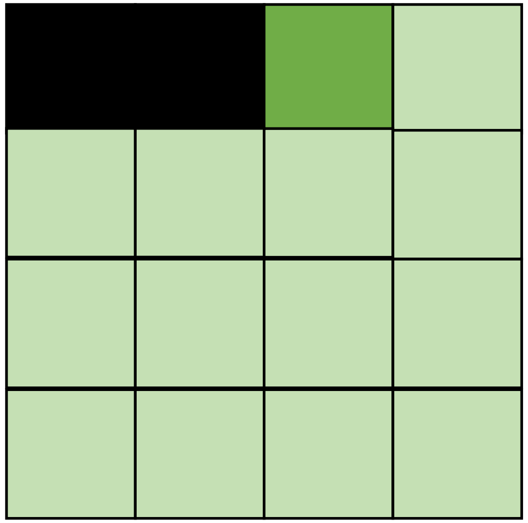
Hit on A and B. Miss on C

Adding loop nestings

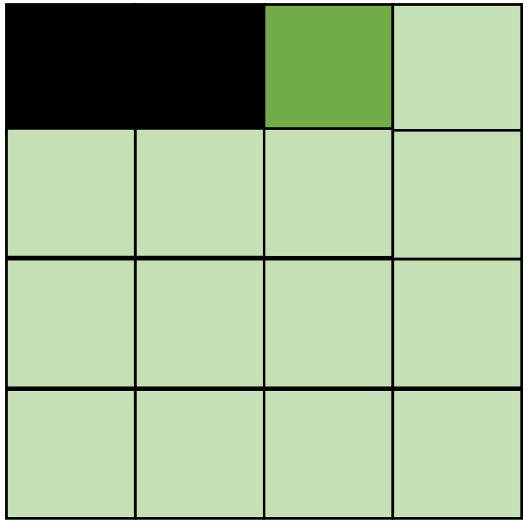
- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

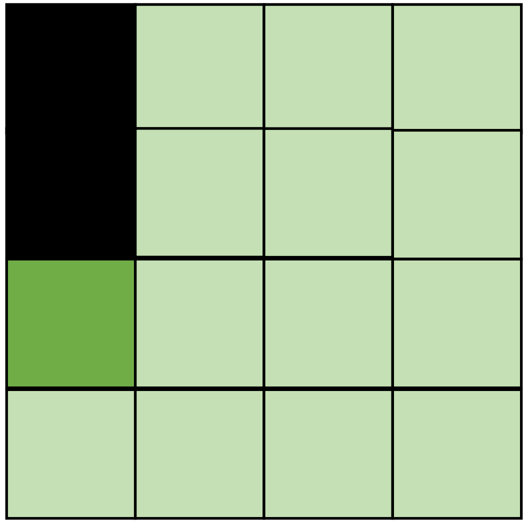
A



B



C

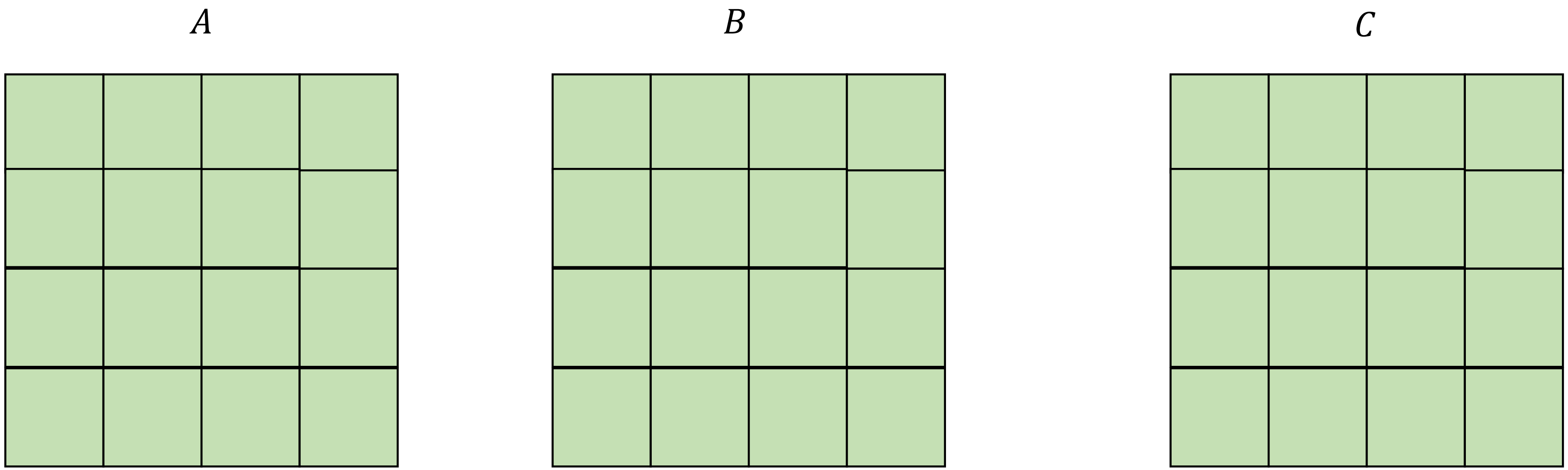


Hit on A and B. Miss on C

Adding loop nestings

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

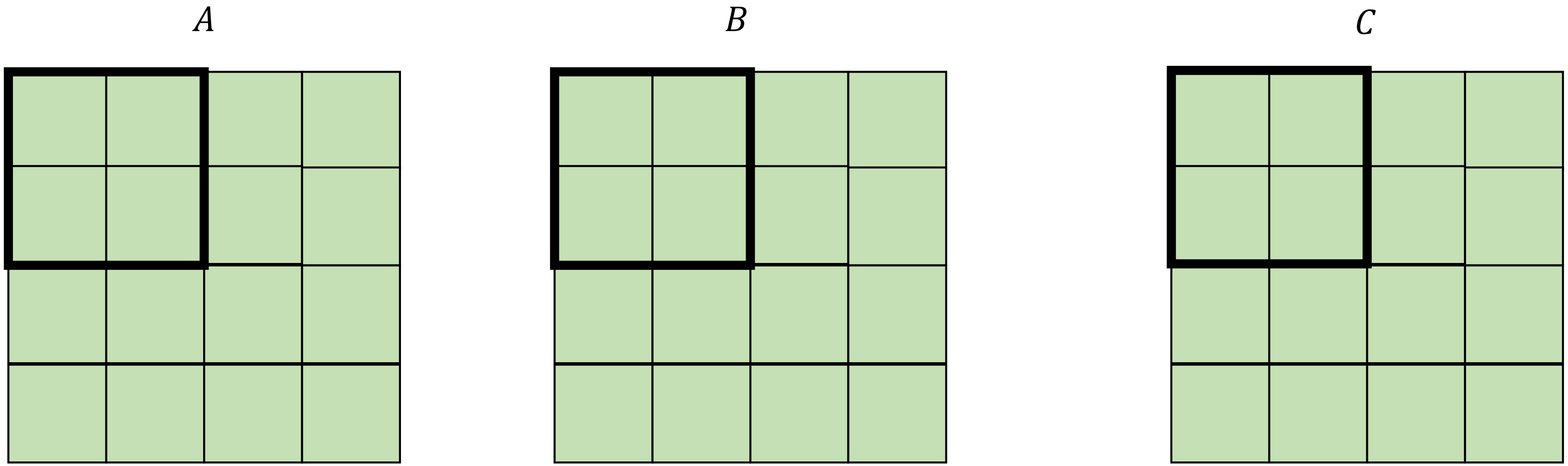
$$A = B + C^T$$



Adding loop nestings

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

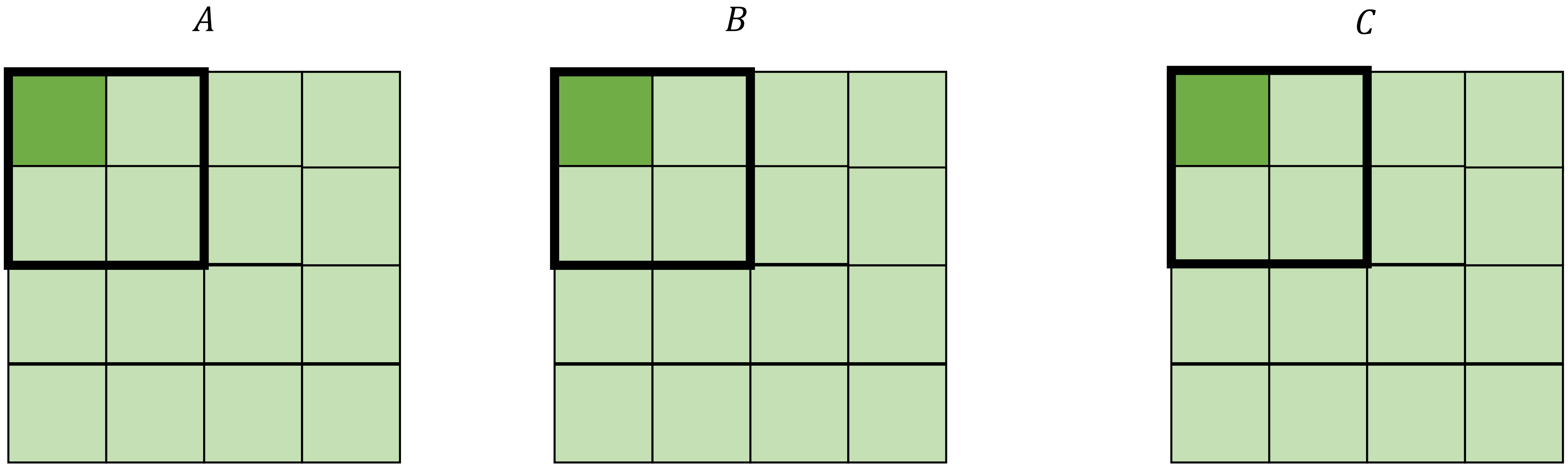
$$A = B + C^T$$



Adding loop nestings

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

$$A = B + C^T$$

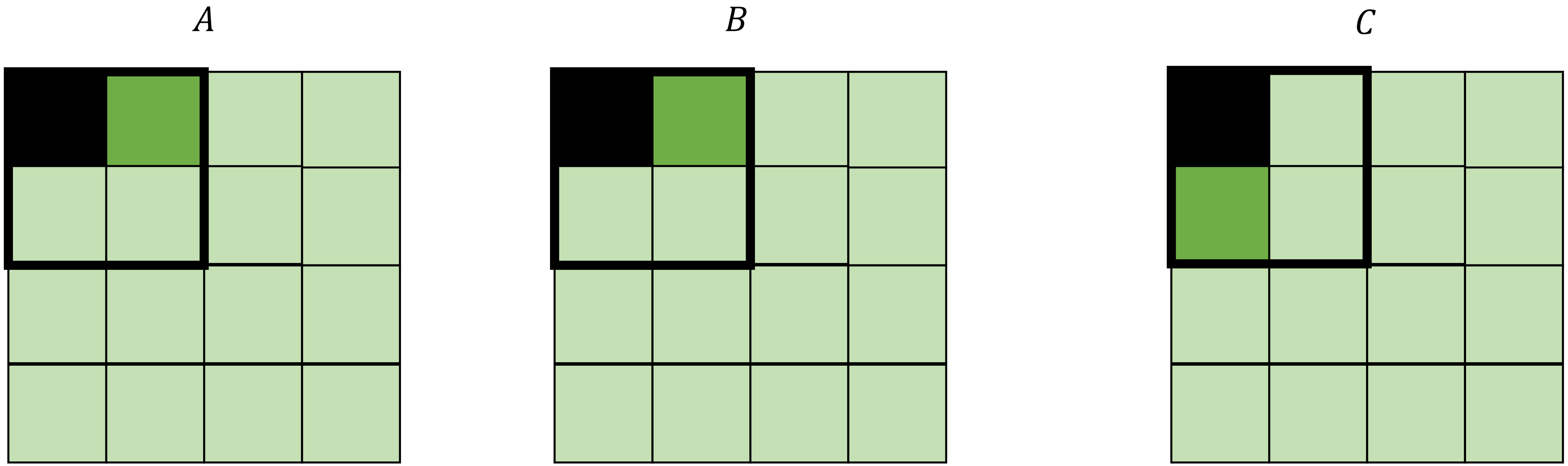


cold miss for all of them

Adding loop nestings

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

$$A = B + C^T$$

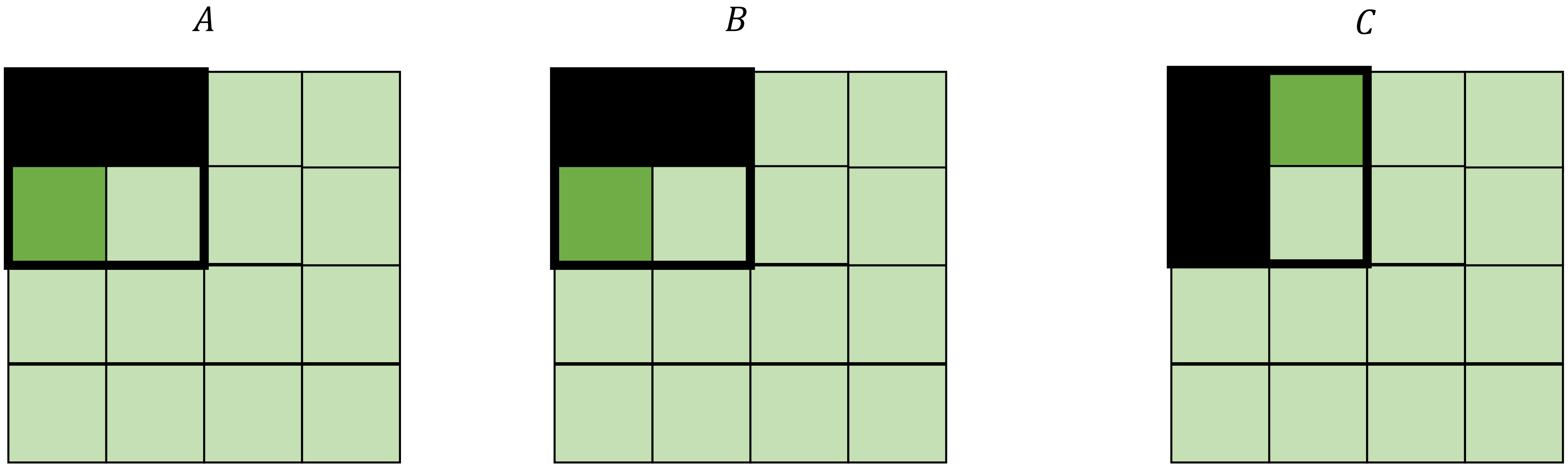


Miss on C

Adding loop nestings

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

$$A = B + C^T$$

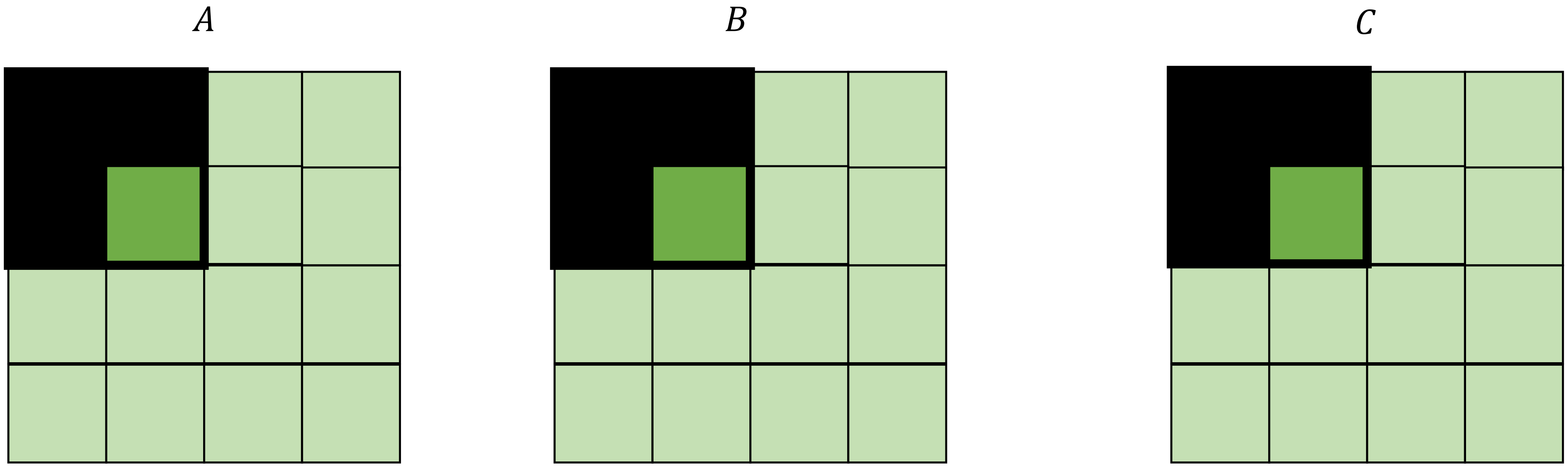


Miss on A,B, hit on C

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];
      }
    }
  }
}
```

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];
      }
    }
  }
}
```

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];
      }
    }
  }
}
```

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];
      }
    }
  }
}
```

Demo

Moving on...

Irregular parallelism in loops

- Independent iterations have different amount of work to compute
- Threads with longer tasks take longer to compute.
- Threads with shorter tasks are underutilized.

Irregular parallelism in loops

- Independent iterations have different amount of work to compute
- Threads with longer tasks take longer to compute.
- Threads with shorter tasks are under utilized.

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

example: regular (or embarrassingly) parallelism:
each x iteration performs the same amount of work

Irregular parallelism in loops

- Independent iterations have different amount of work to compute
- Threads with longer tasks take longer to compute.
- Threads with shorter tasks are under utilized.

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

irregular (or unbalanced) parallelism: each x iteration performs different amount of work.

Irregular parallelism in loops

- Calculate imbalance cost if x is chunked:
  - Thread 1 takes iterations 0 - SIZE/2
  - Thread 2 takes iterations SIZE/2 - SIZE

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

Irregular parallelism in loops

- Calculate imbalance cost if x is chunked:
  - Thread 1 takes iterations 0 - SIZE/2
  - Thread 2 takes iterations SIZE/2 - SIZE

Calculate how much total work:

$$\text{total\_work} = \sum_{n=0}^{SIZE} n$$

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

Irregular parallelism in loops

- Calculate imbalance cost if x is chunked:
  - Thread 1 takes iterations 0 - SIZE/2
  - Thread 2 takes iterations SIZE/2 - SIZE

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

Calculate how much total work:

$$\text{total\_work} = \sum_{n=0}^{SIZE} n$$

Calculate work done by second thread:

$$\text{t2\_work} = \sum_{n=0}^{SIZE/2} n$$

Irregular parallelism in loops

- Calculate imbalance cost if x is chunked:
  - Thread 1 takes iterations 0 - SIZE/2
  - Thread 2 takes iterations SIZE/2 - SIZE

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

Calculate how much total work:

$$\text{total\_work} = \sum_{n=0}^{SIZE} n$$

Calculate work done by second thread:

$$\text{t2\_work} = \sum_{n=0}^{SIZE/2} n$$

Calculate work work done by first thread:

t1_work = total_work − t2_work

Irregular parallelism in loops

Example: SIZE = 64

total_work = 2016
t2_work = 496
t1_work = 1520

t1 does ~3x more work than t2

Only provides ~1.3x speedup

Potential solution:
Have T1 do only ¼ of the iterations
Gives a better speedup of 1.77x

Not a feasible solution because often times load imbalance is not given by a static equation on loop bounds!

Calculate how much total work:

$$\text{total\_work} = \sum_{n=0}^{SIZE} n$$

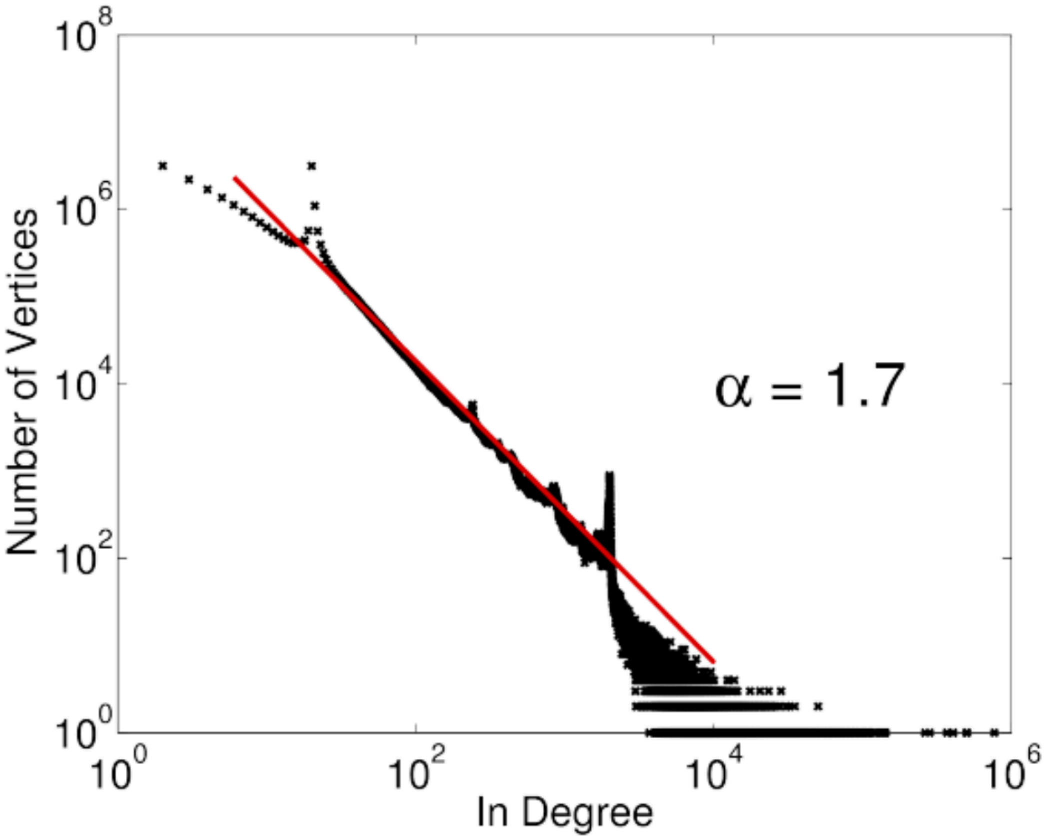
Calculate work done by second thread:

$$\text{t2\_work} = \sum_{n=0}^{SIZE/2} n$$

Calculate work work done by first thread:

t1_work = total_work − t2_work

Where does irregular parallelism show up?



(a) Twitter In-Degree

from “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”, OSDI 2012

Tyler Sorensen
@Tyler_UCSC

Computer Science Researcher
Assistant Professor at UC Santa Cruz in 2020

Santa Cruz, CA users.soe.ucsc.edu/~tsorensen/
Joined September 2019

303 Following **332** Followers

Lindsey Kuper
@lindsey Follows you

Mommy; blogger. CS professor at @ucsc (PL, distributed systems, verification).
“Permit yourself to open a book and start reading from anywhere.” pronoun.is/she

composition.al Joined October 2006

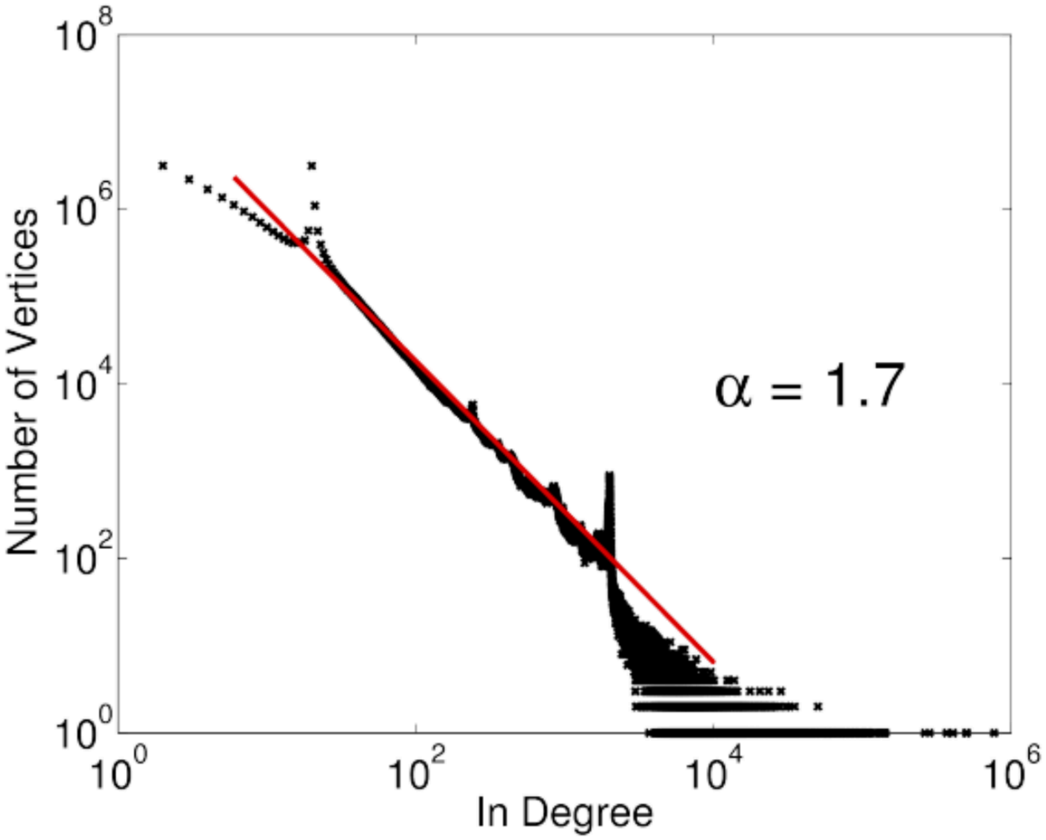
751 Following **6,840** Followers

Barack Obama ✓
@BarackObama

Dad, husband, President, citizen.

Washington, DC obama.org Born August 4, 1961
Joined March 2007

598.7K Following **126.2M** Followers



(a) Twitter In-Degree

- Vertex programming model iterates over each node in parallel.
- Each node pulls in values from neighbors
- Similar to flow analysis!

from “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”, OSDI 2012

Tyler Sorensen
@Tyler_UCSC

Computer Science Researcher
Assistant Professor at UC Santa Cruz in 2020

Santa Cruz, CA users.soe.ucsc.edu/~tsorensen/
Joined September 2019

303 Following **332** Followers

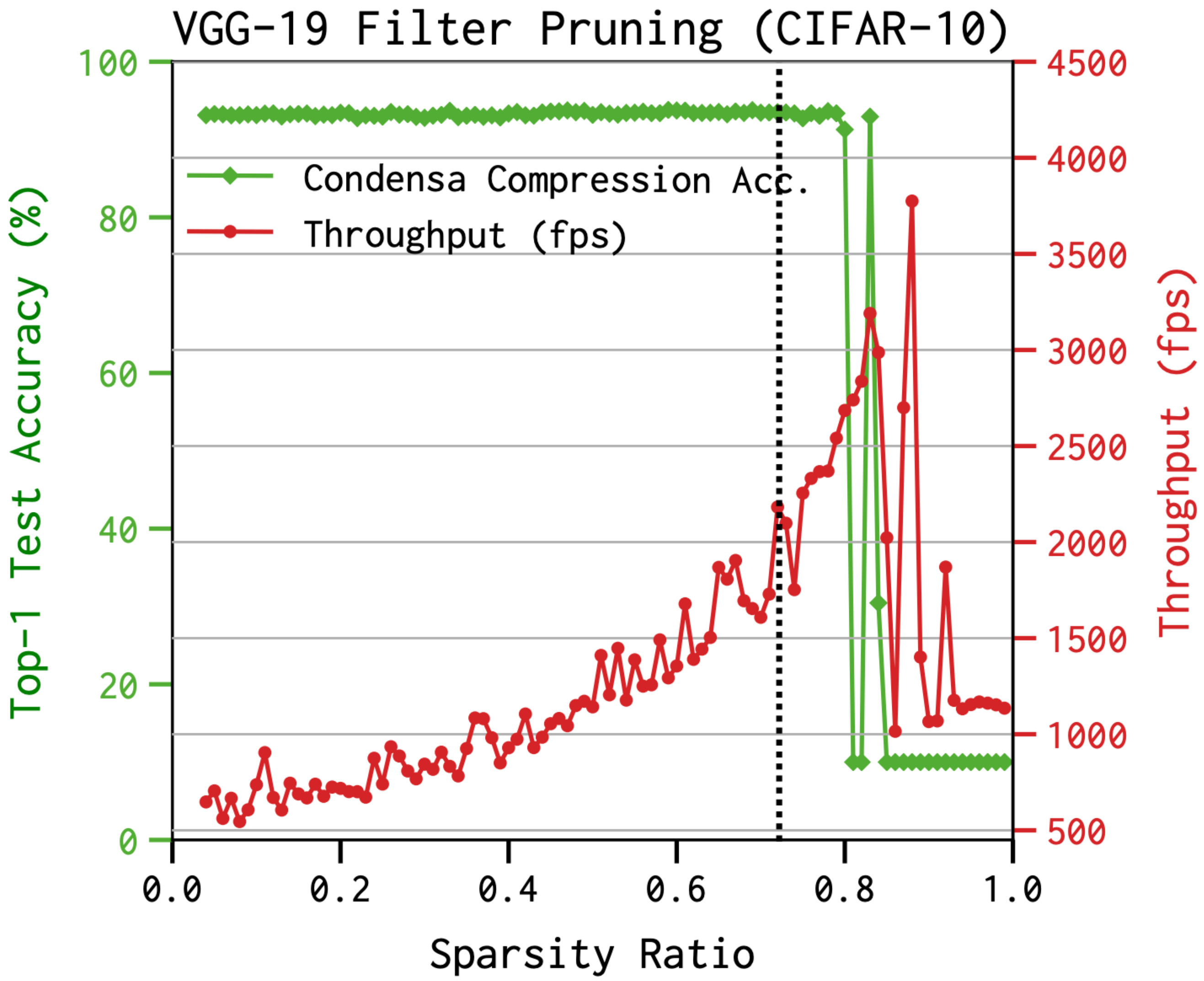
Barack Obama
@BarackObama

Dad, husband, President, citizen.

Washington, DC obama.org Born August 4, 1961
Joined March 2007

598.7K Following **126.2M** Followers

Sparse Neural Nets



from: "A PROGRAMMABLE APPROACH TO MODEL COMPRESSION". arxiv 2019.

How can we deal with load imbalance?

- Great research question! Changes per domain/architecture/input etc.

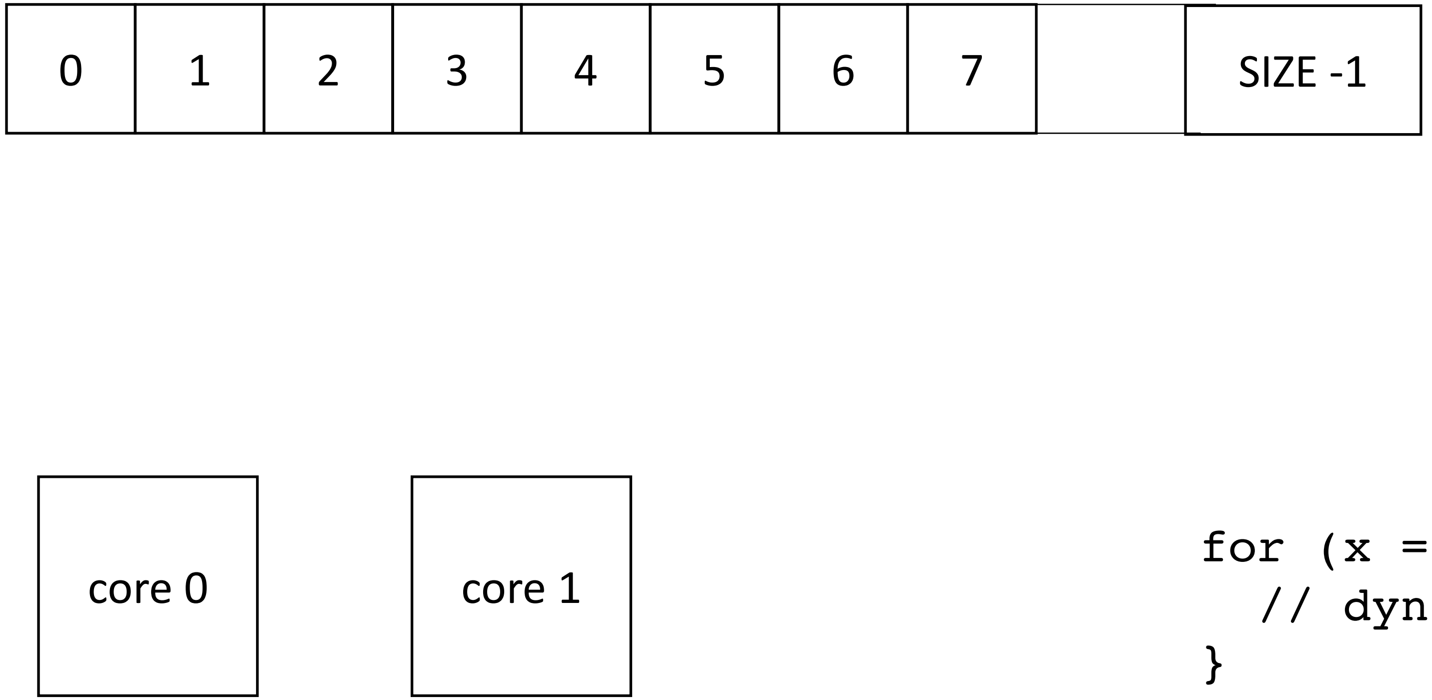
Work stealing

- Threads dynamically get assigned to loop iterations

- Two approaches:
  - global (pessimistic)
  - local (optimistic)

Work stealing

- Global worklist: threads take tasks (iterations) dynamically



0	1	2	3	4	5	6	7		SIZE -1

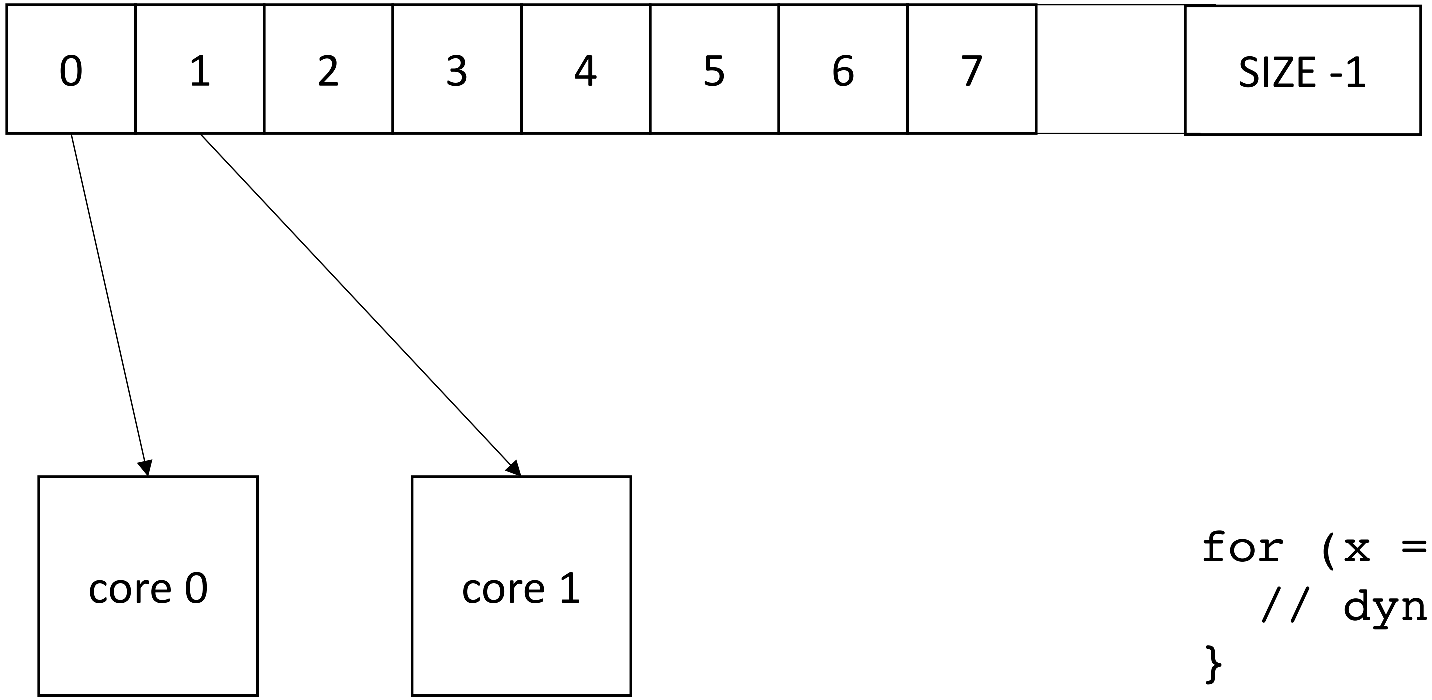
core 0

core 1

```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

- Global worklist: threads take tasks (iterations) dynamically



0 1 2 3 4 5 6 7 SIZE -1

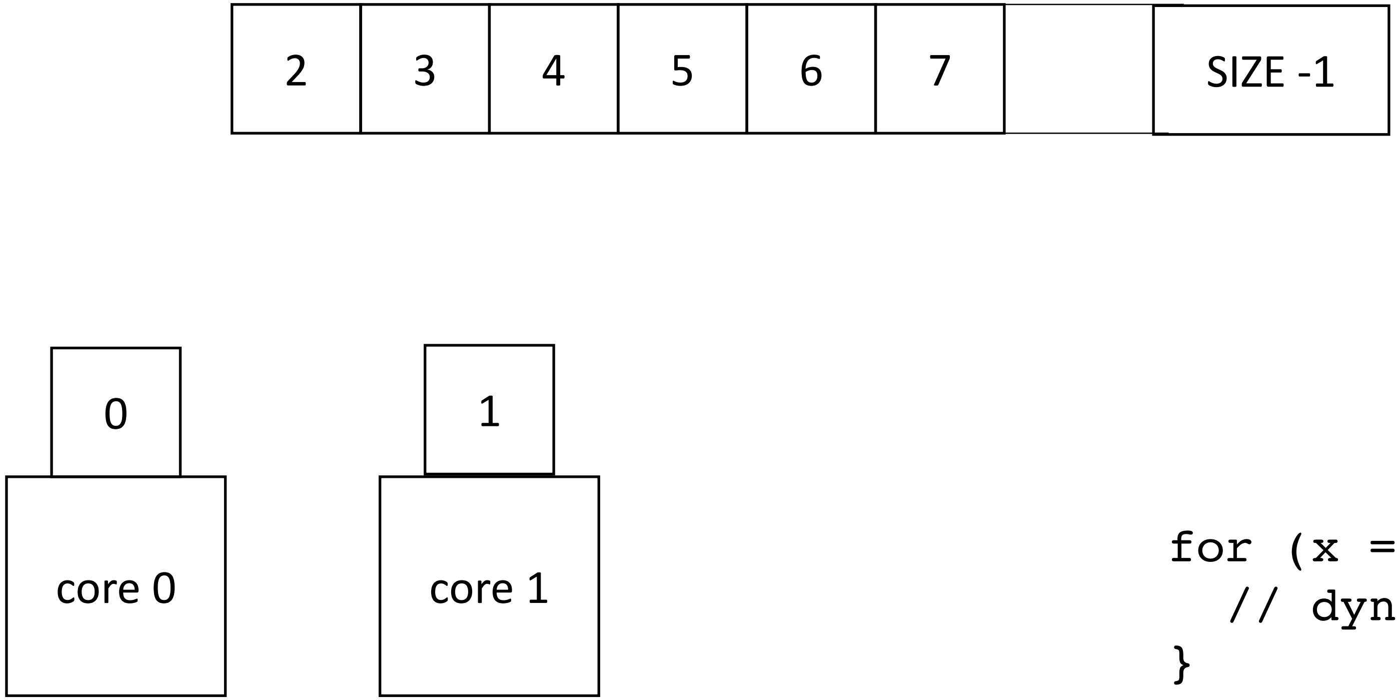
core 0

core 1

```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

- Global worklist: threads take tasks (iterations) dynamically



2 3 4 5 6 7 SIZE -1

0

core 0

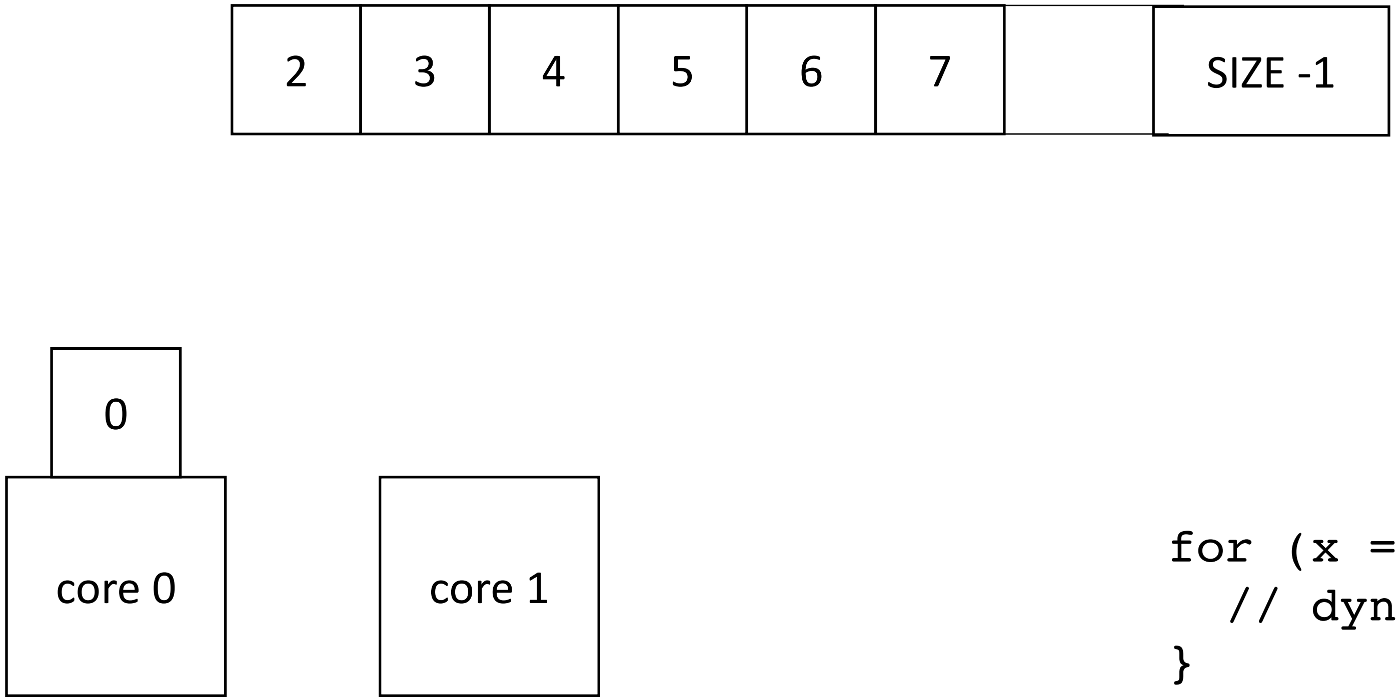
1

core 1

```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

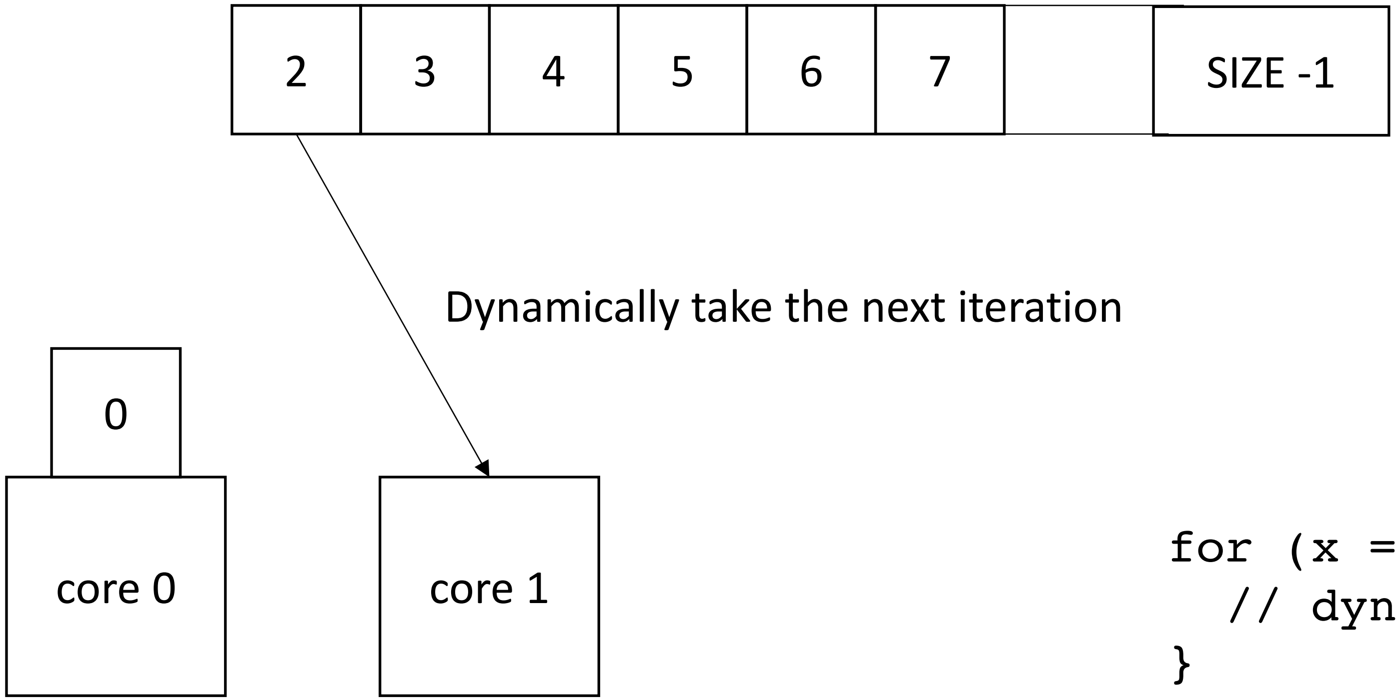
- Global worklist: threads take tasks (iterations) dynamically



```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

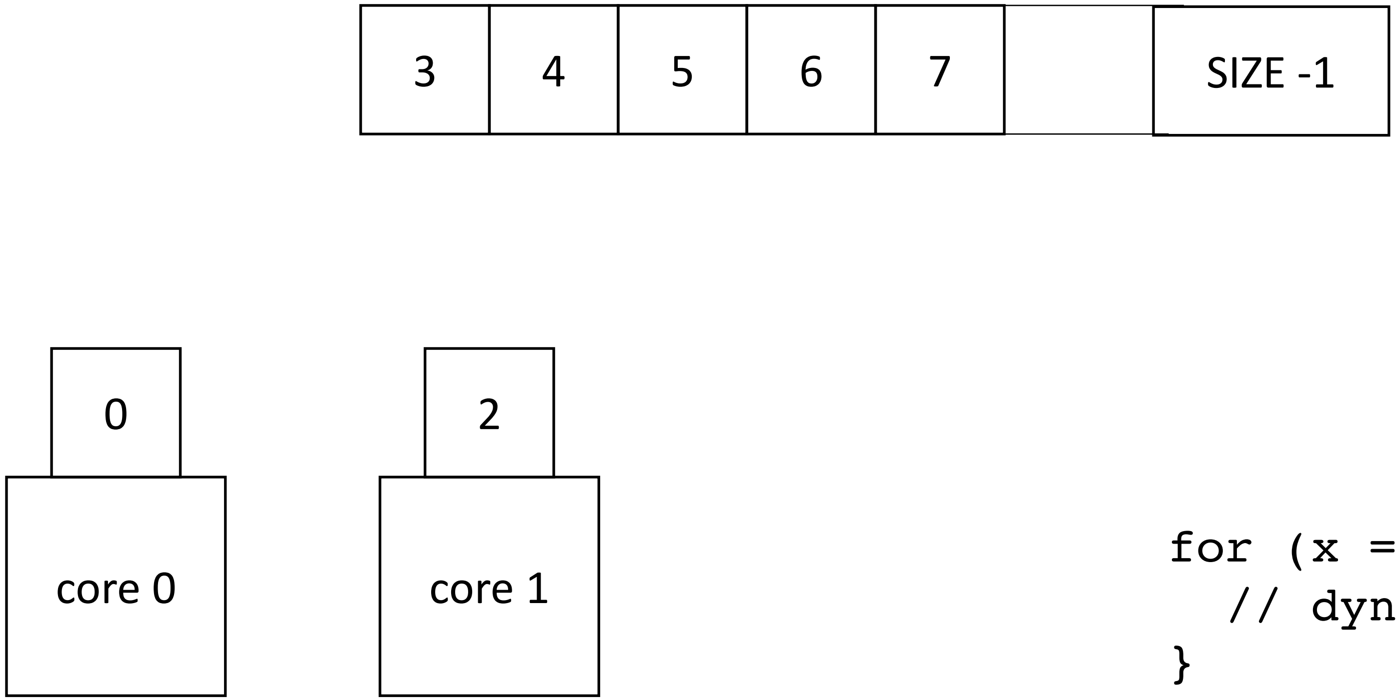
- Global worklist: threads take tasks (iterations) dynamically



```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

- Global worklist: threads take tasks (iterations) dynamically



3 4 5 6 7 SIZE -1

0

core 0

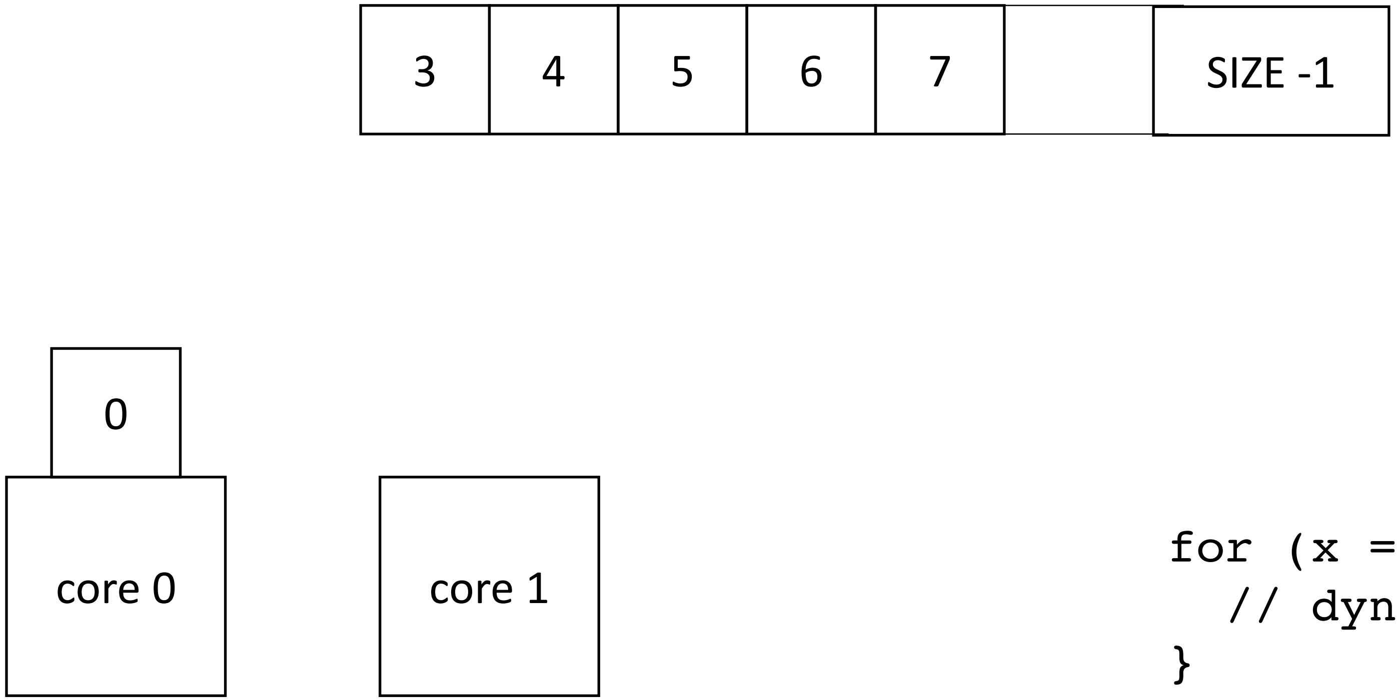
2

core 1

```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

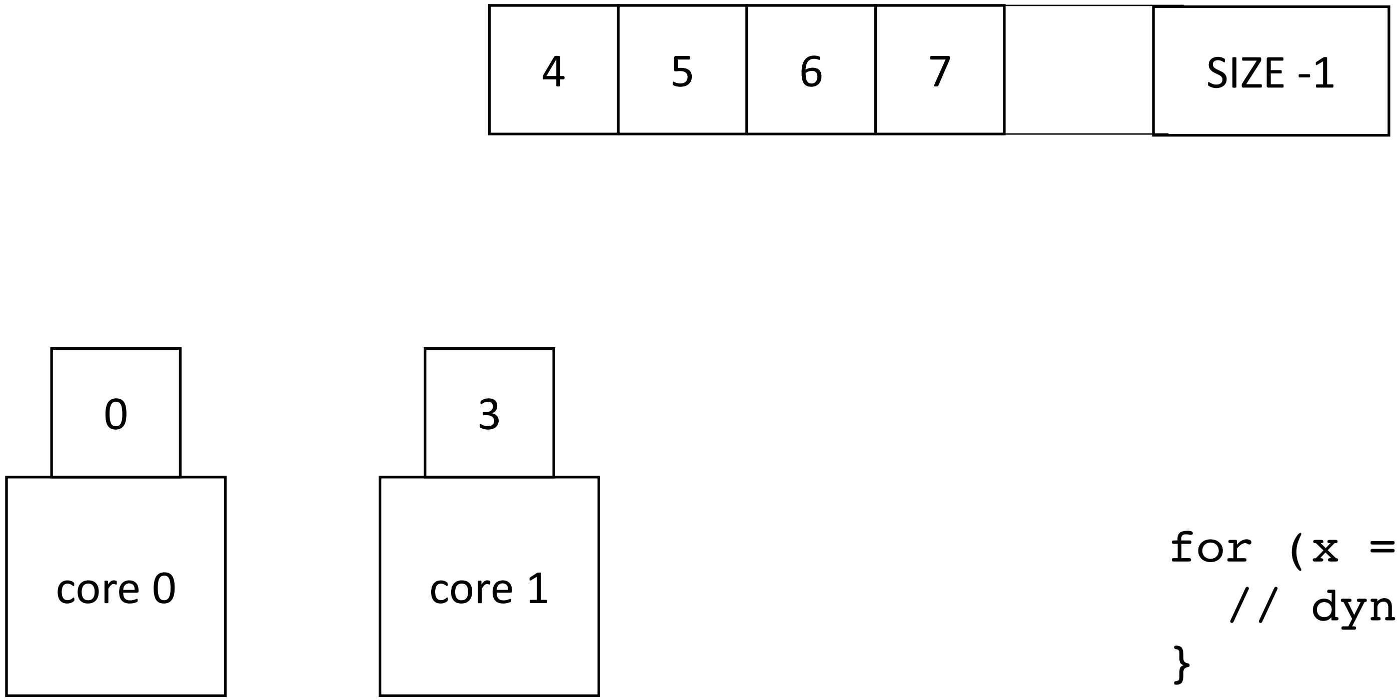
- Global worklist: threads take tasks (iterations) dynamically



```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

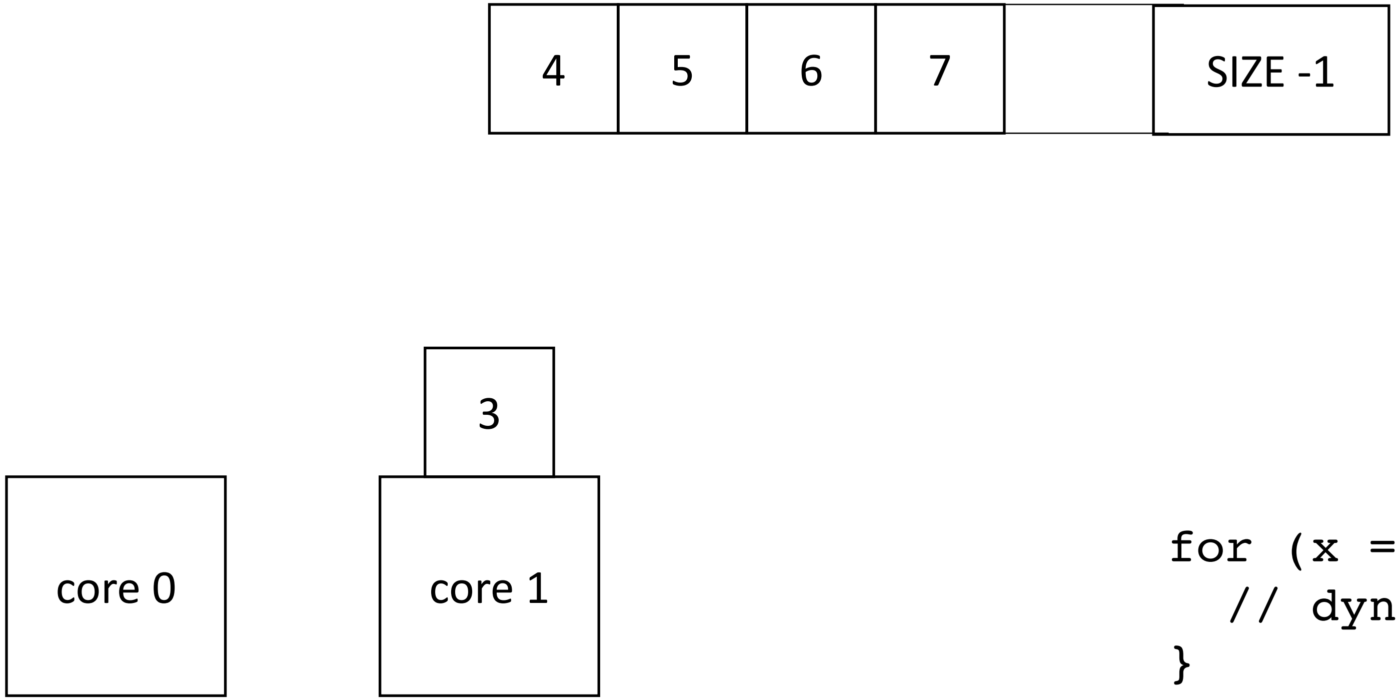
- Global worklist: threads take tasks (iterations) dynamically



```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

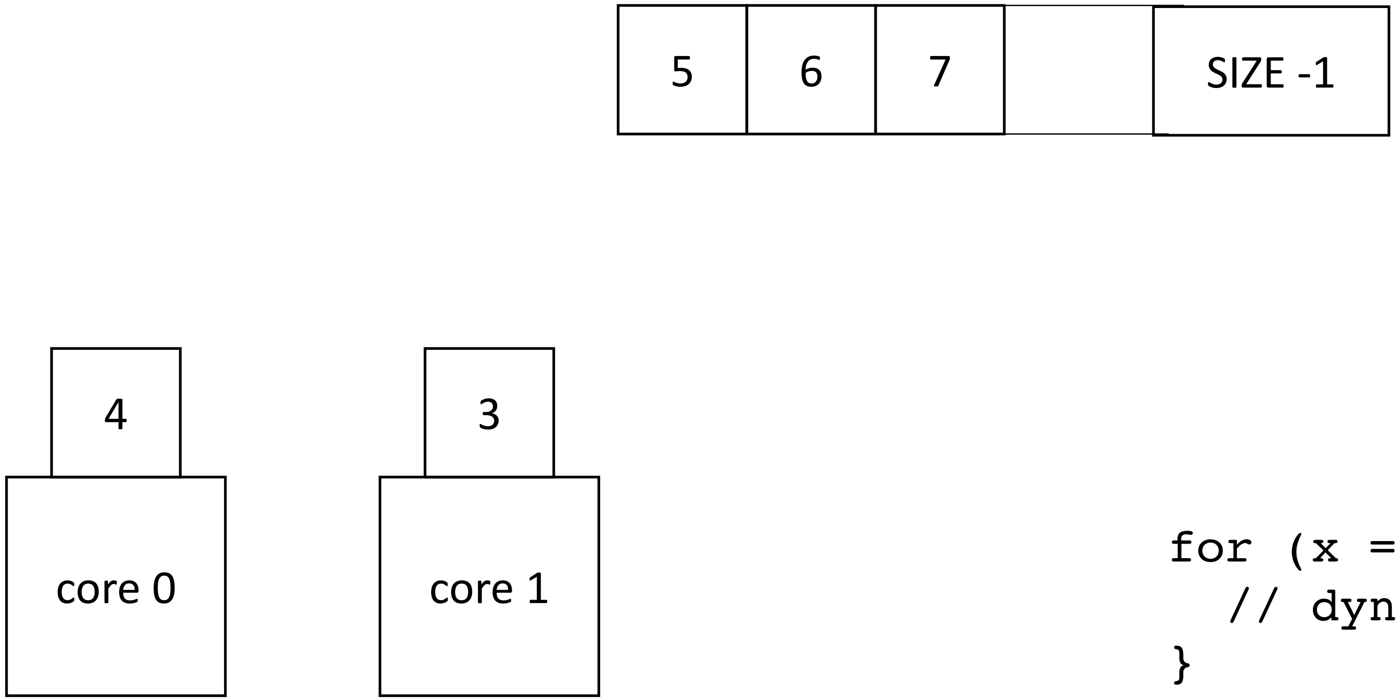
- Global worklist: threads take tasks (iterations) dynamically



```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

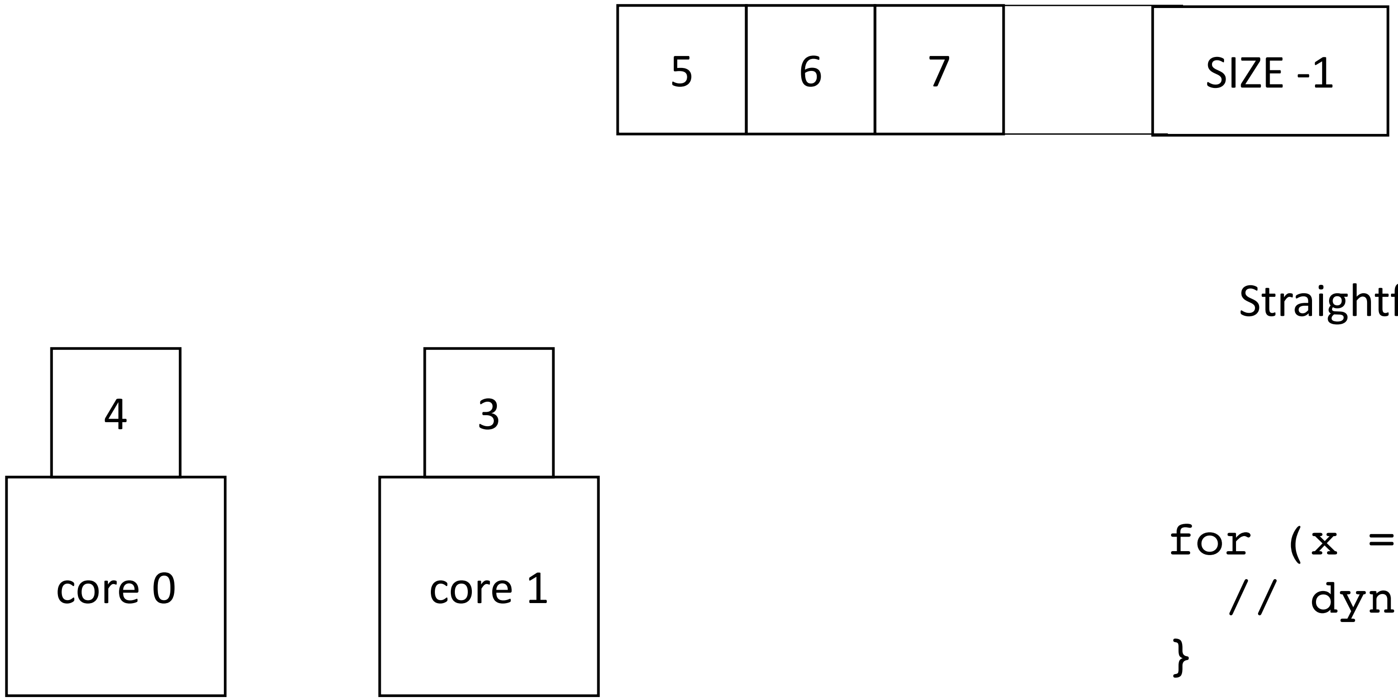
- Global worklist: threads take tasks (iterations) dynamically



```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

- Global worklist: threads take tasks (iterations) dynamically

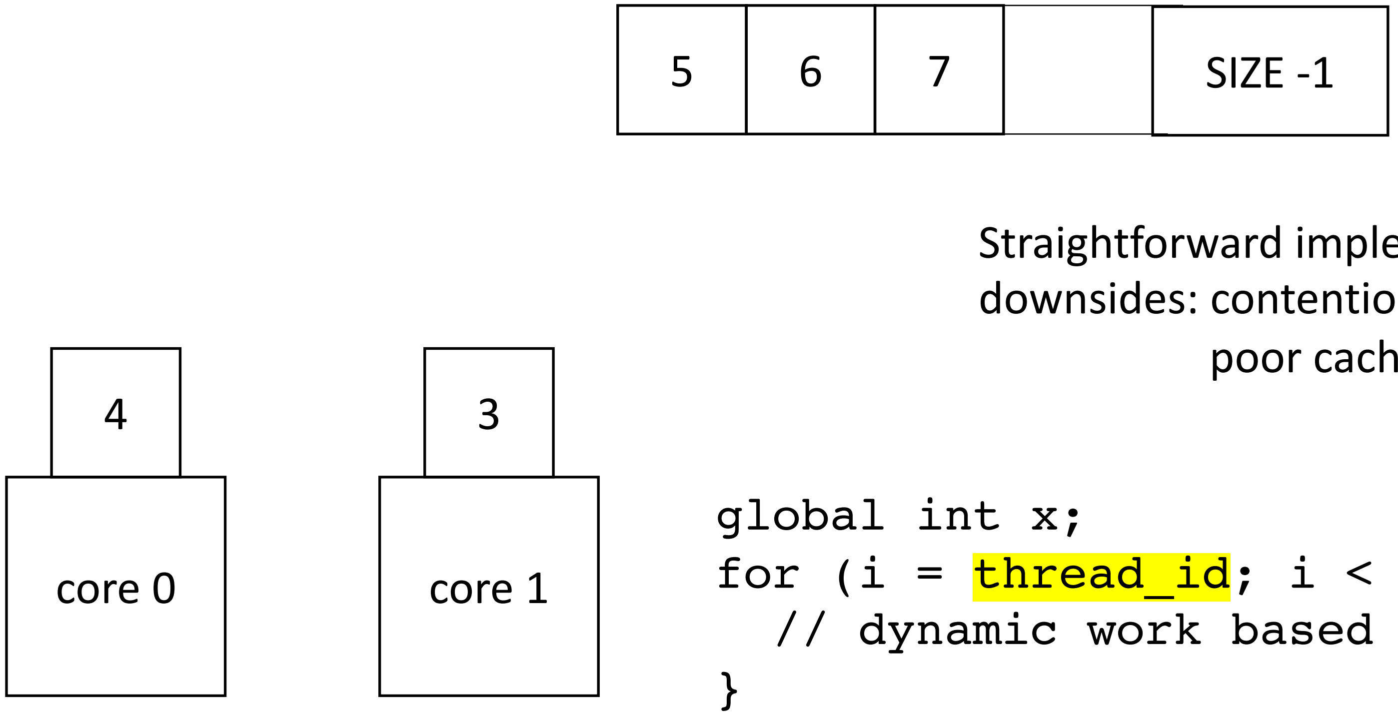


Straightforward implementation

```
for (x = 0; x < SIZE; x++) {
  // dynamic work based on x
}
```

Work stealing

- Global worklist: threads take tasks (iterations) dynamically

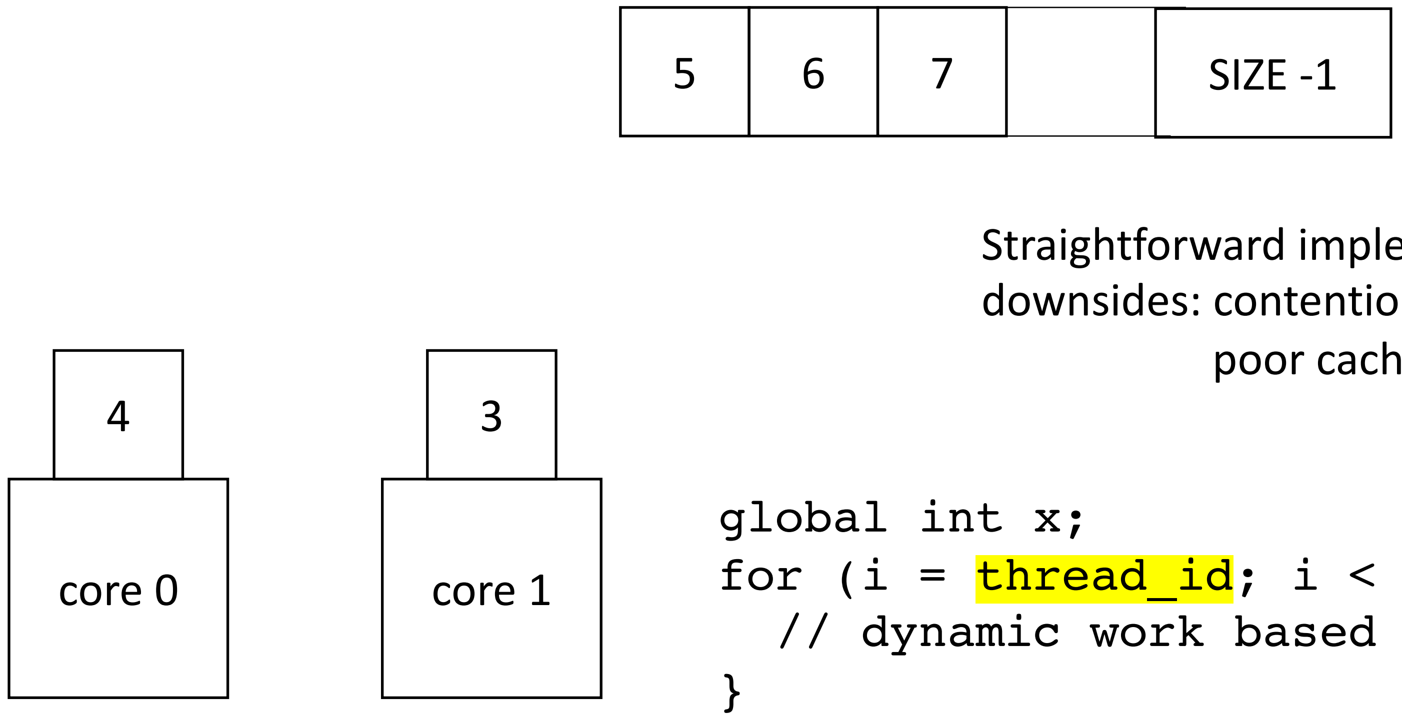


Straightforward implementation
downsides: contentious atomic operation for every task
poor cache locality from the work list

```
global int x;
for (i = thread_id; i < SIZE; i = atomic_increment(x)) {
  // dynamic work based on i
}
```

Work stealing

- Global worklist: threads take tasks (iterations) dynamically



Straightforward implementation
downsides: contentious atomic operation for every task
poor cache locality from the work list

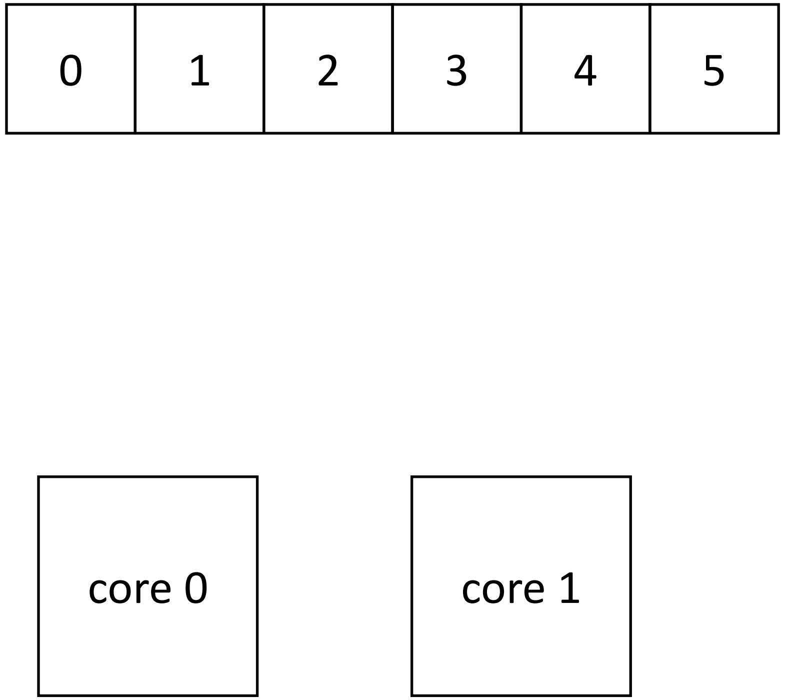
```
global int x;
for (i = thread_id; i < SIZE; i = atomic_increment(x)) {
  // dynamic work based on i
}
```

Work stealing

- Local worklists: threads optimistically are assigned an even sized chunk of work. Threads that finish early steal from unfinished threads

Work stealing

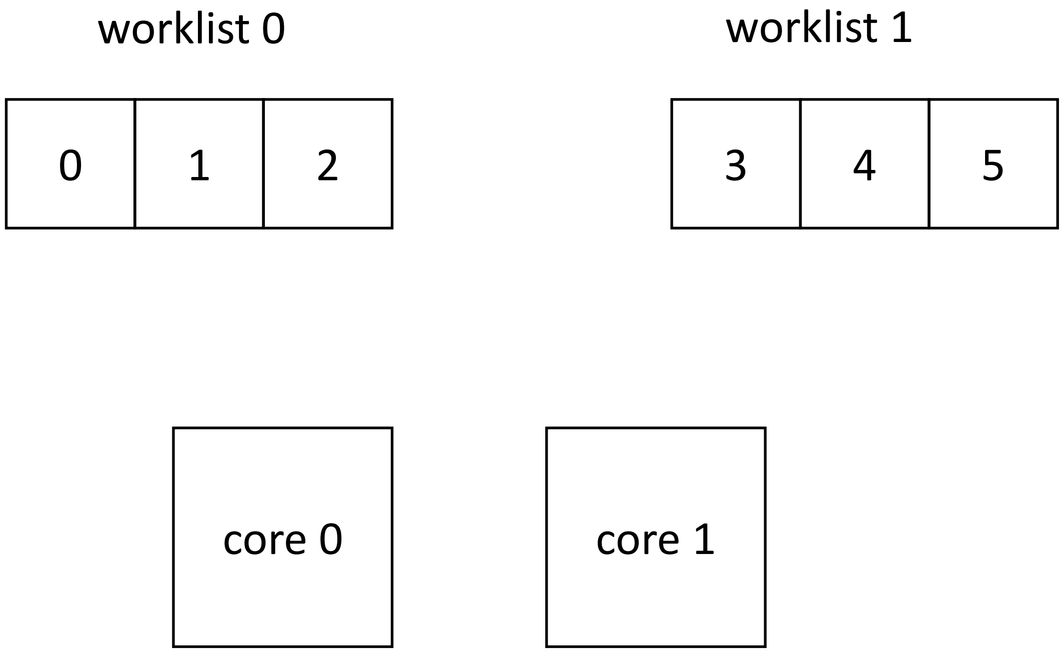
- local worklists: divide tasks into different worklists for each thread



```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

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



worklist 0

0	1	2

worklist 1

3	4	5

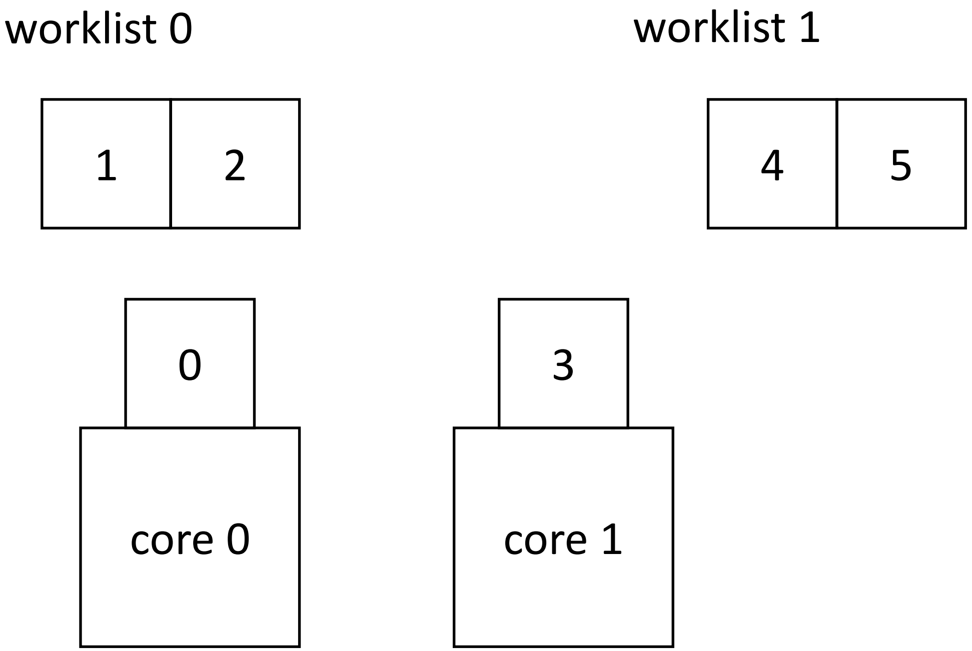
core 0

core 1

```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

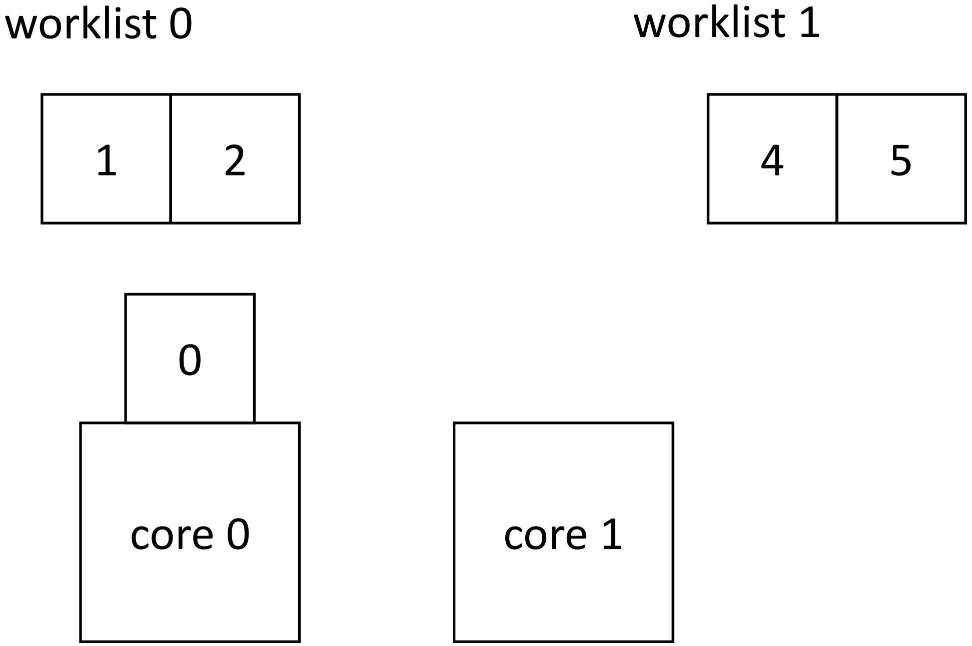
- local worklists: divide tasks into different worklists for each thread



```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

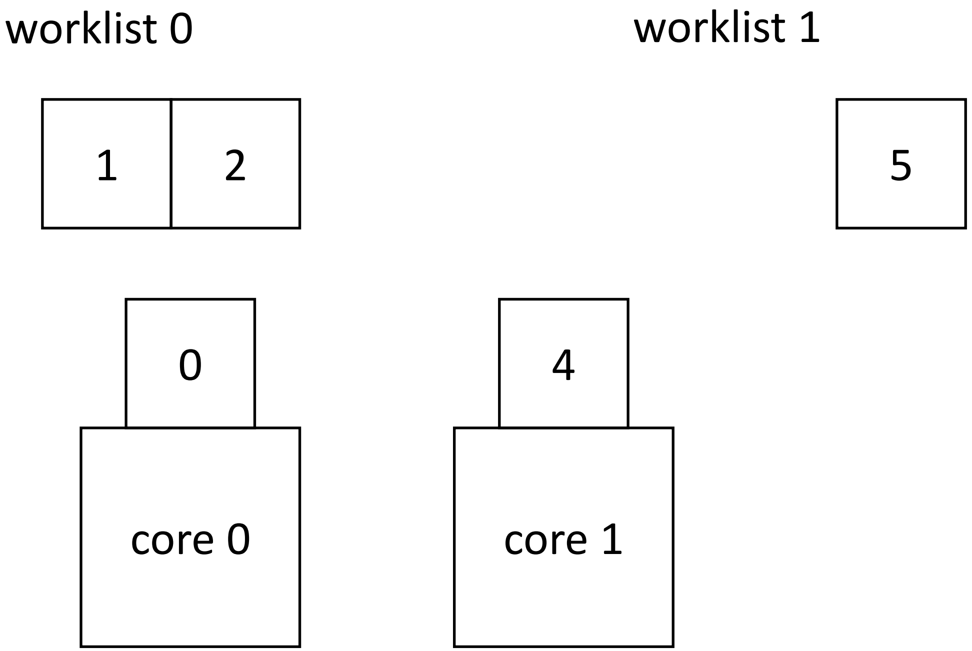
- local worklists: divide tasks into different worklists for each thread



```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

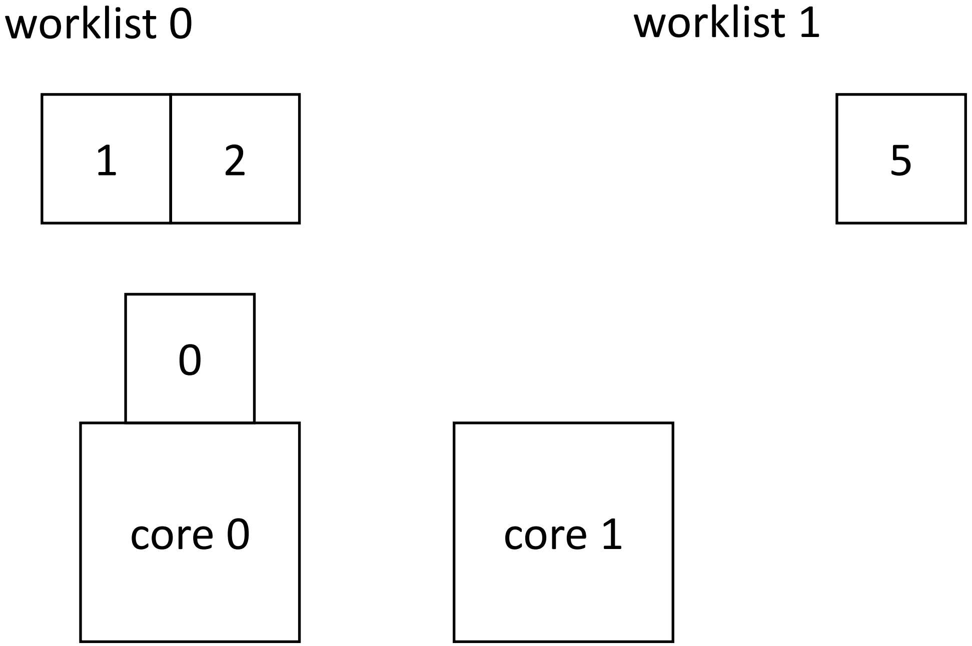
- local worklists: divide tasks into different worklists for each thread



```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

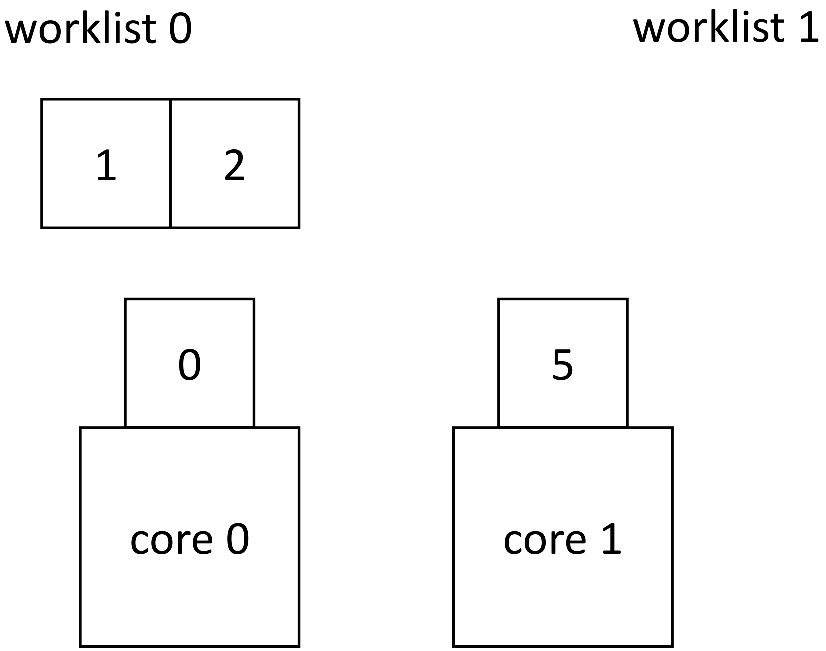
- local worklists: divide tasks into different worklists for each thread



```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

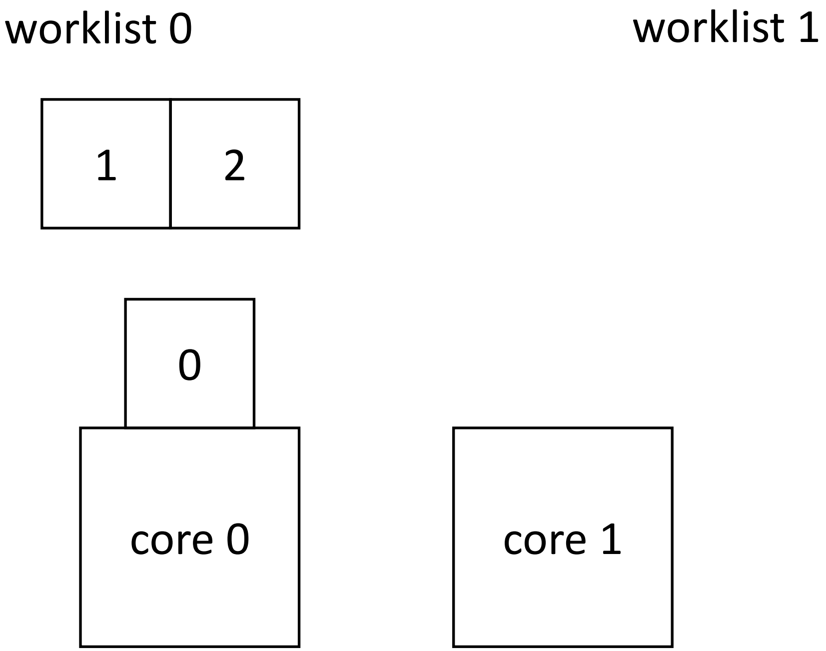
- local worklists: divide tasks into different worklists for each thread



```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

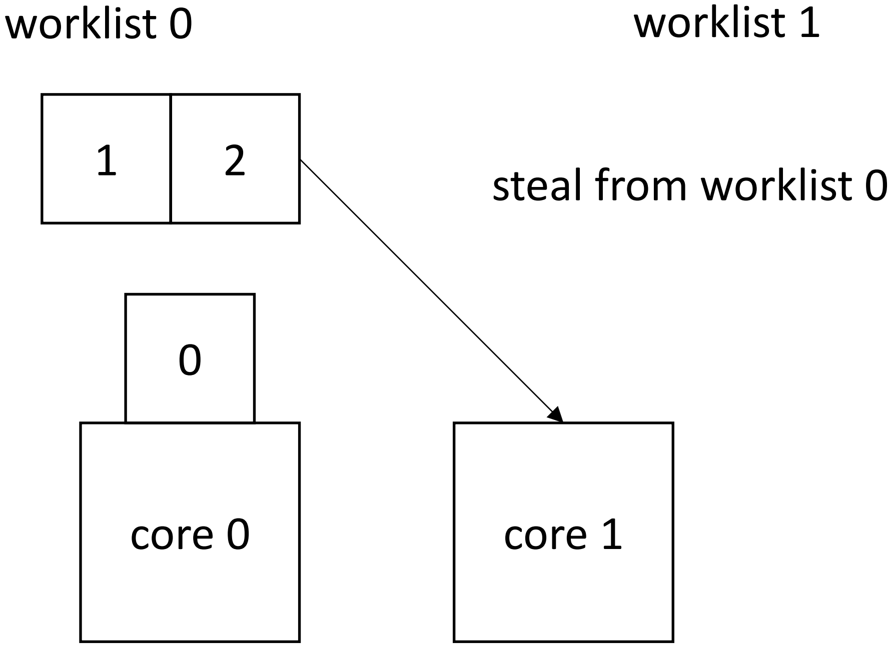
- local worklists: divide tasks into different worklists for each thread



```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

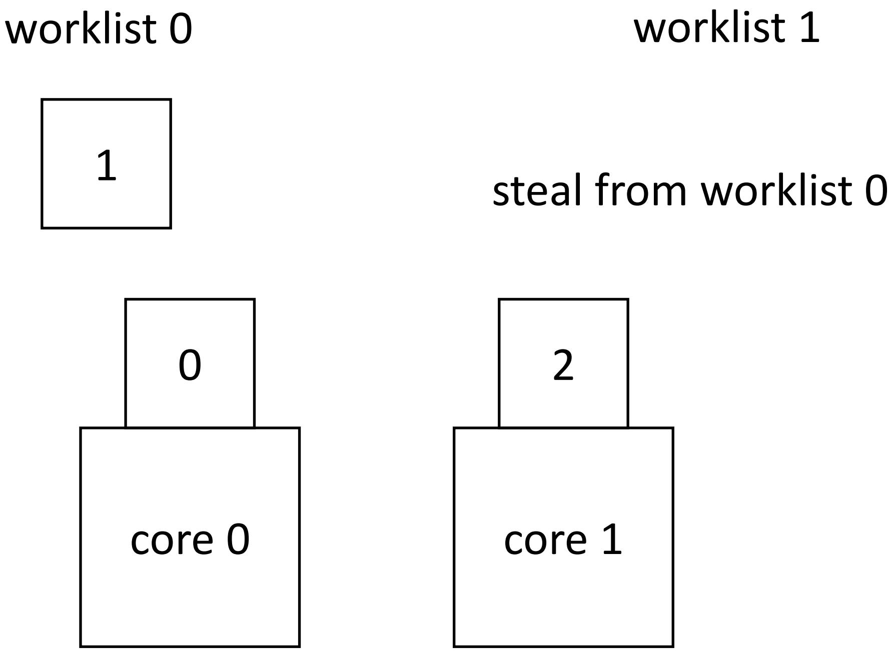
- local worklists: divide tasks into different worklists for each thread



```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

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



worklist 0

worklist 1

1

steal from worklist 0

0

2

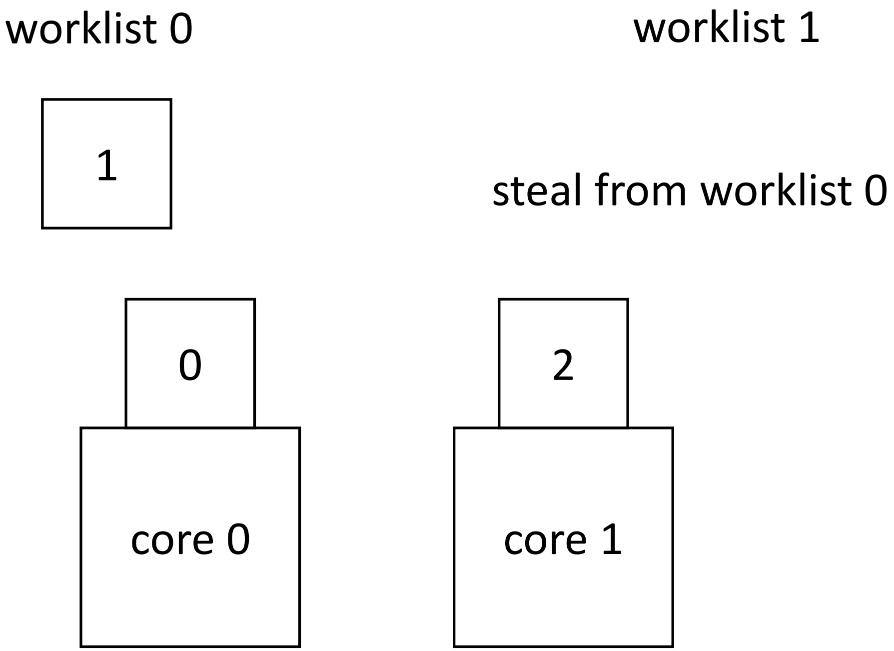
core 0

core 1

```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

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



worklist 0

worklist 1

1

steal from worklist 0

0

2

core 0

core 1

Implementation more difficult. Requires efficient concurrent data-structures, stealing strategies, etc.

Pros: less contention, better cache locality

```
for (x = 0; x < 6; x++) {
  // dynamic work based on x
}
```

Work stealing

- Well-studied, available (e.g. OpenMP)
- Requires fine-grained synchronization (concurrent data-structures, or atomic read-modify-write)
- Demo

Next class:

- Inspect/Execute load balancing
- Decoupled Access Execute