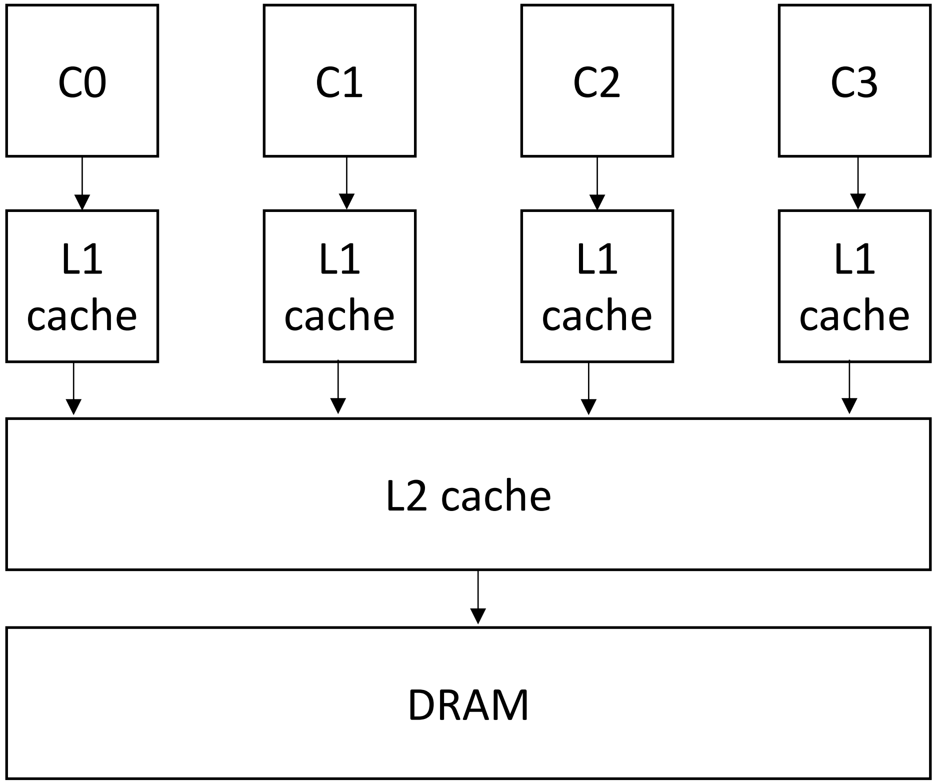
# CSE211: Compiler Design

Nov. 17, 2023

- **Topic**: SMP parallelism
  - Candidate DOALL loops
  - Safety checking

- **Discussion questions**:
  - Do compilers automatically make your code parallel?
  - What are some difficulties in SMP parallelism vs. ILP?



# Announcements

- Homework 3 is out
  - Due on Nov. 29 (2 weeks to do it)
  - Get a partner ASAP

- Start thinking about 2nd paper
- Getting close to the deadline to getting it approved
  - Approved in ~1 week (Nov. 27)!
  - Presentations must be ready by Dec. 6
  - Deadline is to get final project APPROVED, not start brainstorming

- One more homework

# Announcements

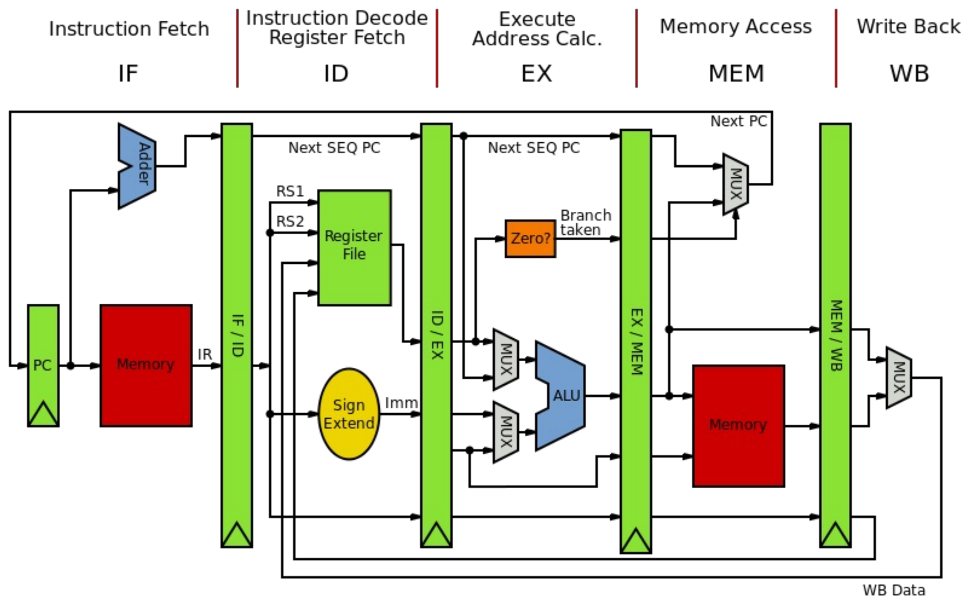
- Grading:
  - Working on grading HW 2

# Review SMP parallelism

# Limits of ILP?

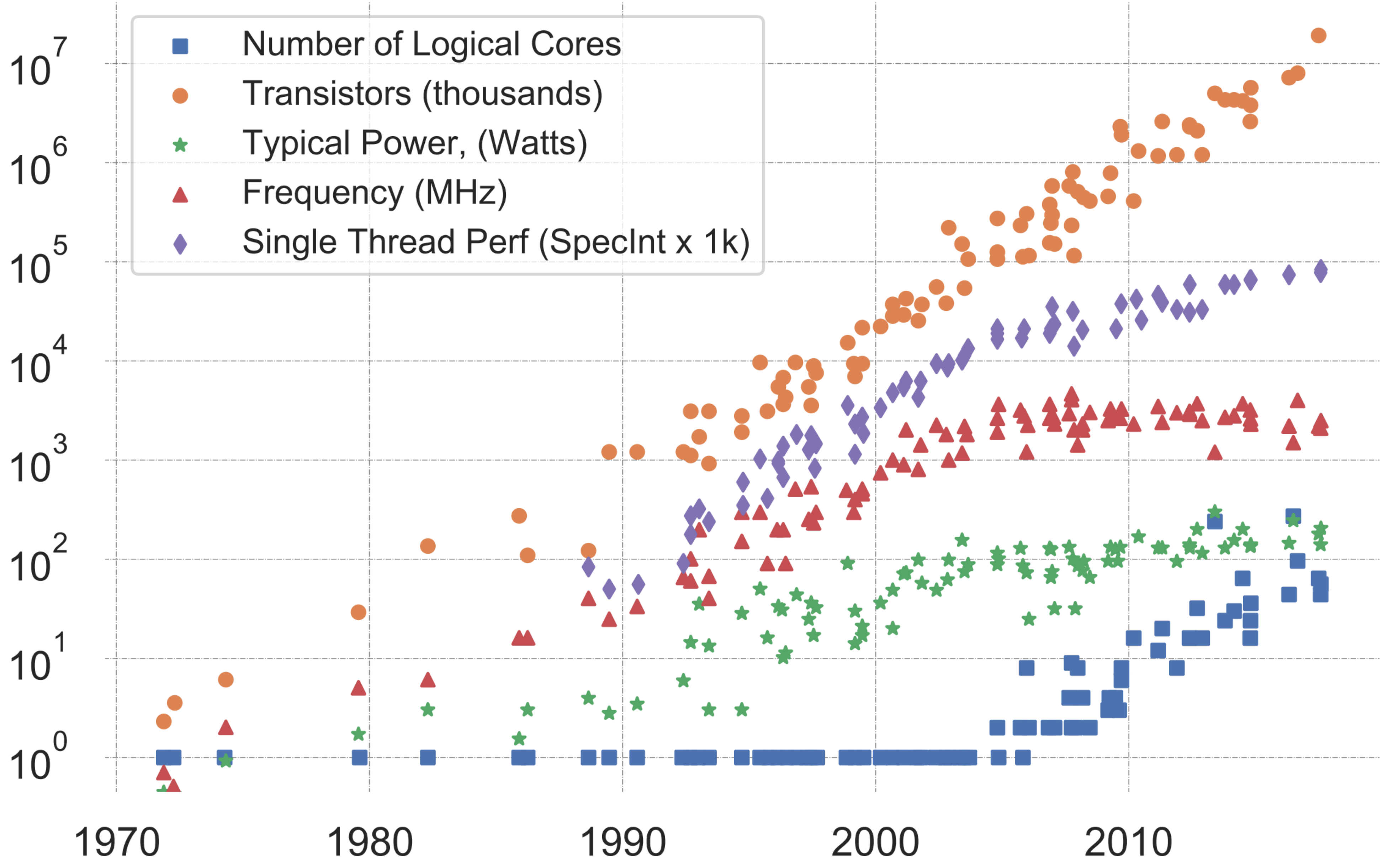
- Pipelines?
  - Only so much meaningful work to do per-stage.
  - Stage timing imbalance
  - Staging overhead

- Superscalar width?
  - Hardware checking becomes prohibitive:




*Collectively the power consumption, complexity and gate delay costs limit the achievable superscalar speedup to roughly eight simultaneously dispatched instructions.*

https://en.wikipedia.org/wiki/Superscalar_processor#Limitations




K. Rupp, "40 Years of Mircroprocessor Trend Data," https://www. karlrupp.net/2015/06/40-years-of-microprocessor-trend-data, 2015.

# Trends

- Frequency scaling: **Dennard's scaling**
  - Mostly agreed that this is over

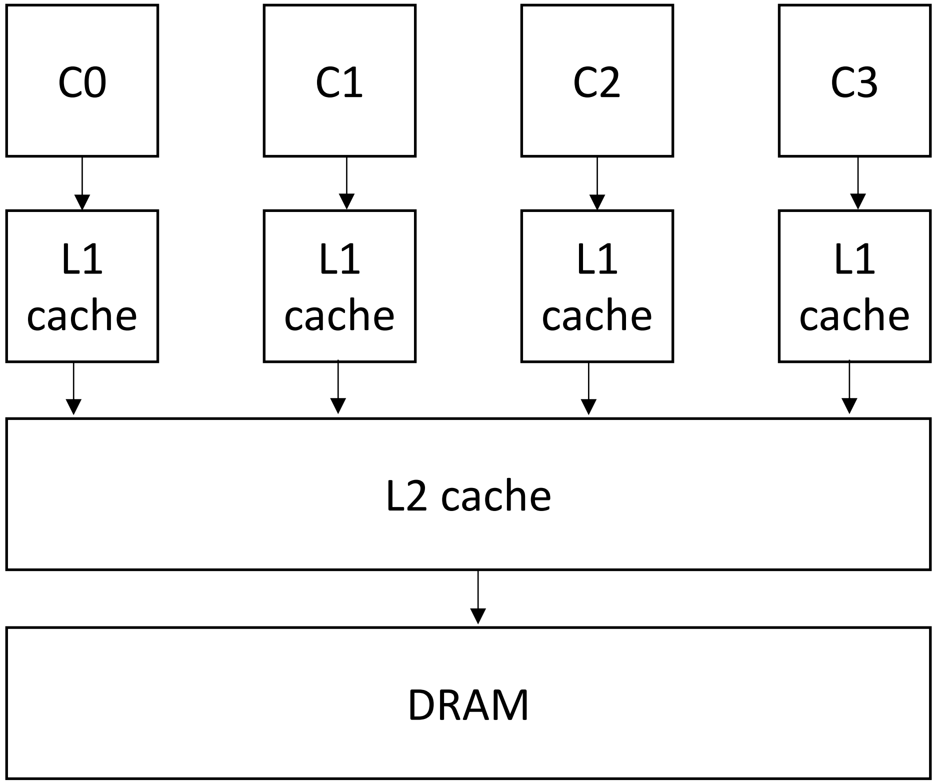
- Number of transistors: **Moore's law**
  - On its last legs?
  - Intel delayed 7nm chips (out now?). Apple has a 5nm. Roadmaps go to 3nm, or 1.8nm

- *Chips are not increasing in raw frequency, and space is becoming more valuable*

# Symmetric Multiprocessing (SMP)

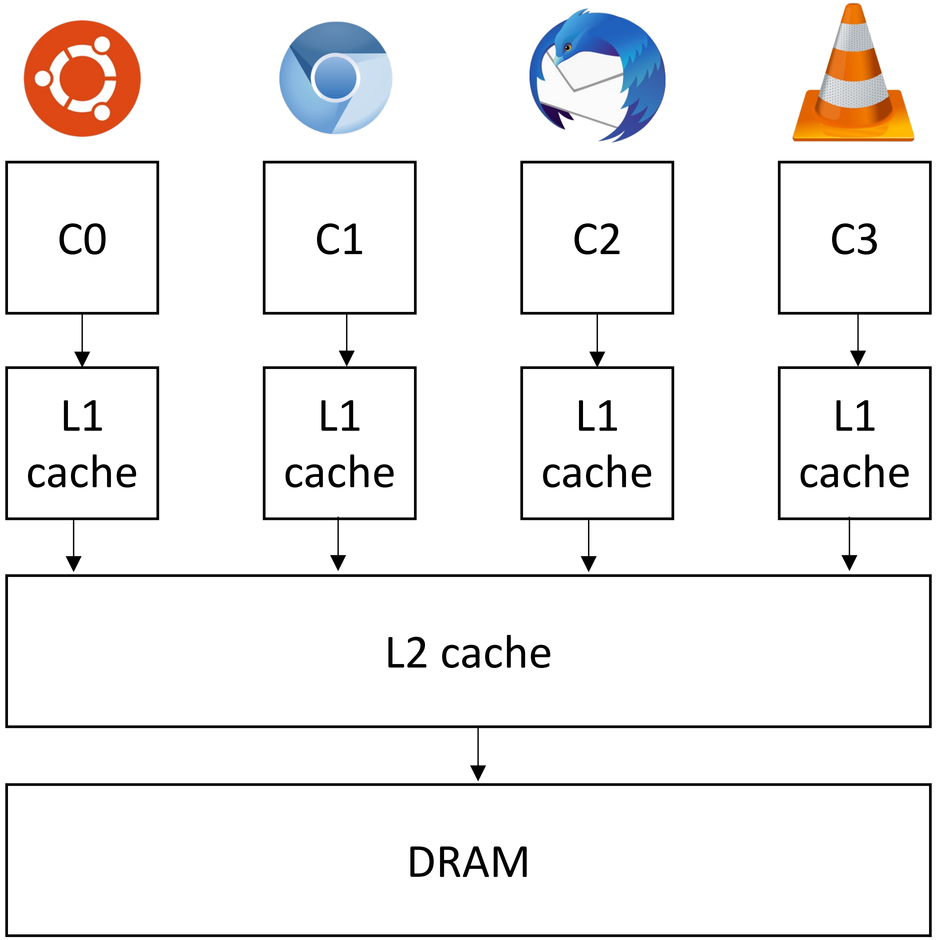
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS

- Pros:
  - Simple(r) HW design
  - Great for multitasking machines



# Symmetric Multiprocessing (SMP)

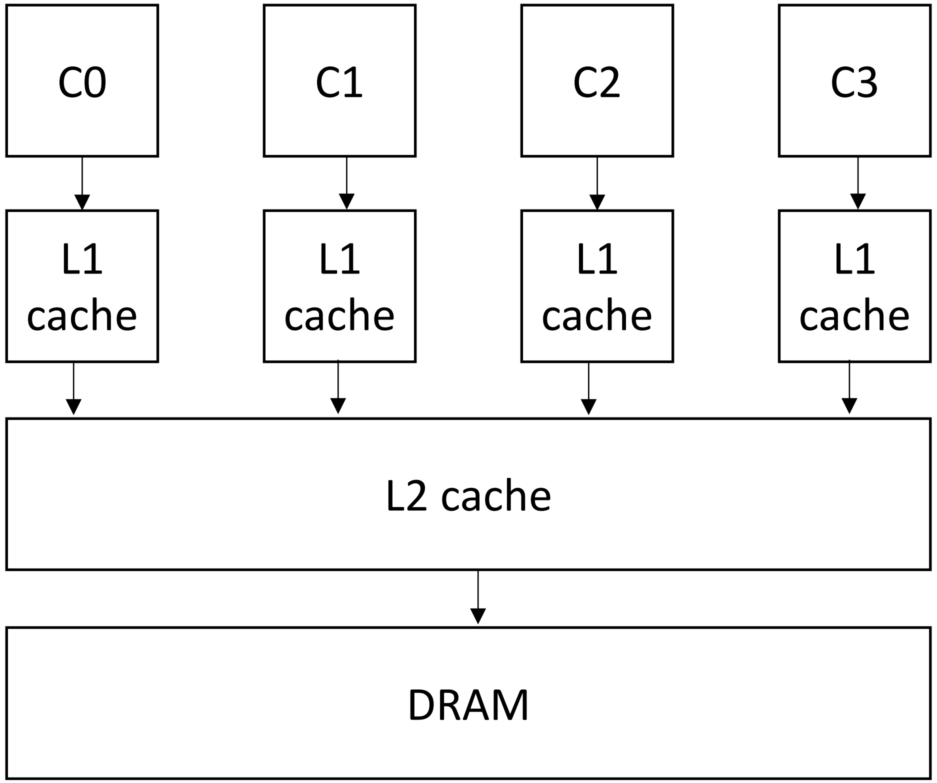
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines



# Symmetric Multiprocessing (SMP)

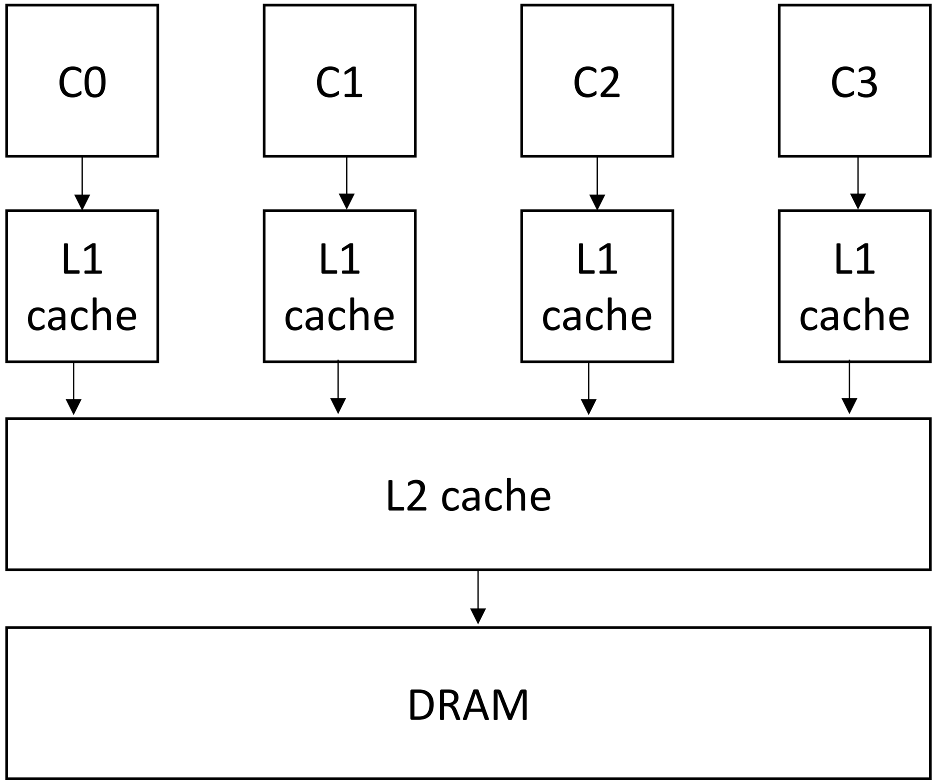
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines
  - Can provide (close to) linear speedups for parallel applications

PyTorch



# Symmetric Multiprocessing (SMP)

- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines
  - Can provide (close to) linear speedups for parallel applications
- Cons: difficult to program!



# SMP systems are widespread

- Laptops
  - My laptop has 8 cores
  - Most have at least 2
  - New Macbook: 16 core
- Workstations:
  - 2 - 64 cores (x86)
  - ARM racks: 128
- Phones:
  - iPhone: 2 big cores, 4 small cores
  - Samsung: 1 + 3 + 4

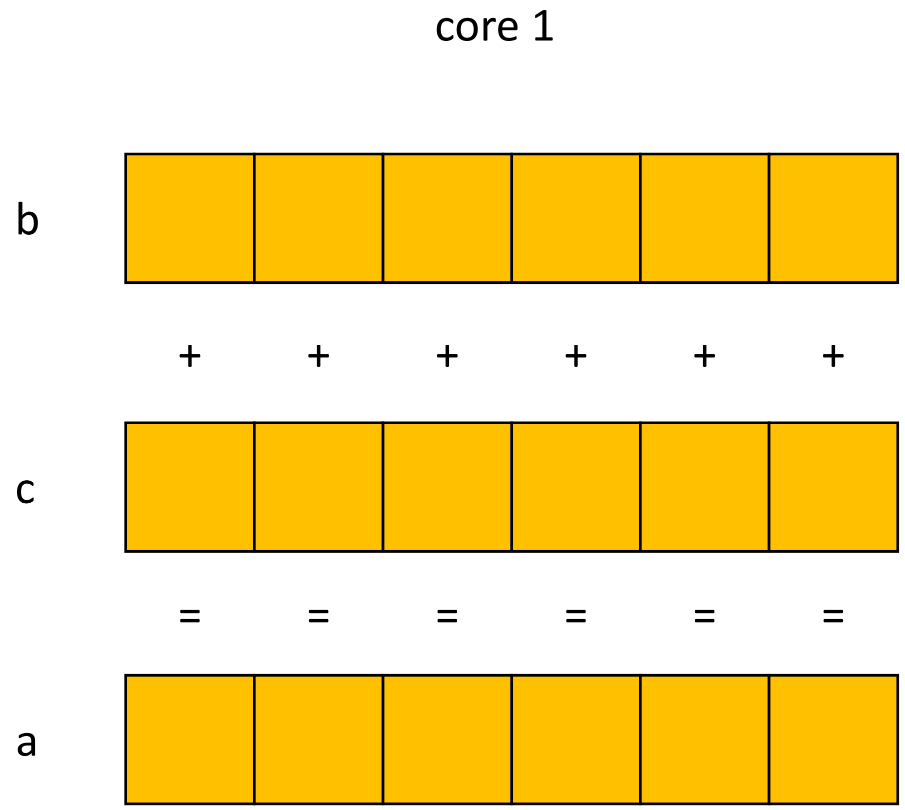
*https://www.crn.com/news/components-peripherals/ampere-s-new-128-core-altra-cpu-targets-intel-amd-in-the-cloud

# Can compilers help?

- Much like ILP: convert sequential streams of computation in to SMP parallel code.
- Much harder constraints
  - Correctness
  - Performance
- For loops are a good target for compiler analysis

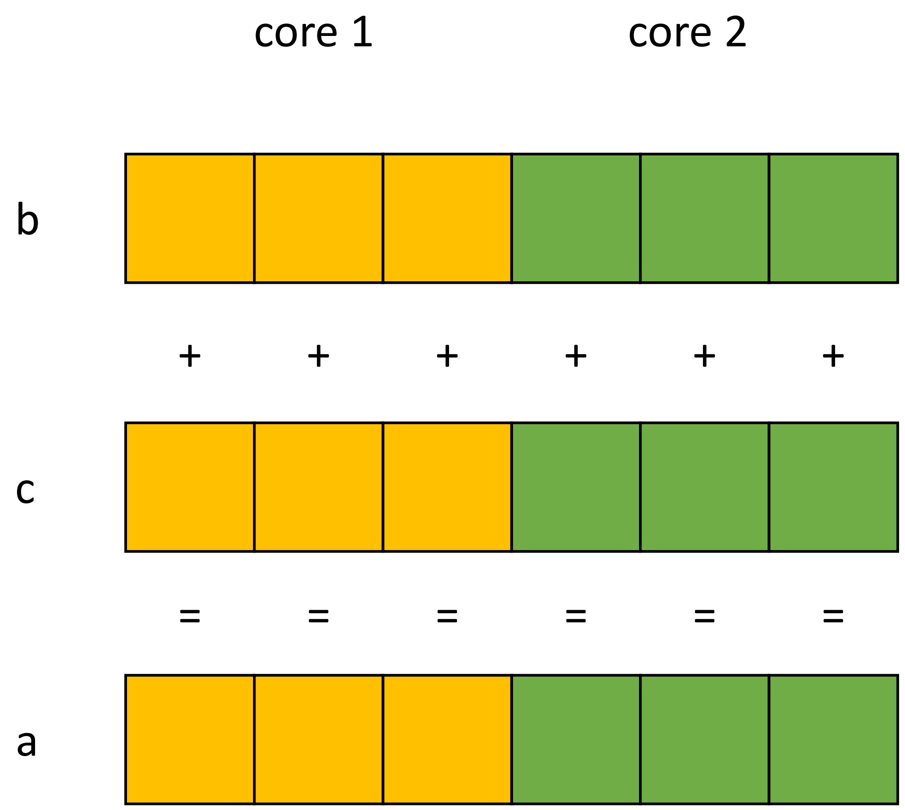
# For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {
   a[i] = b[i] + c[i]
}
```



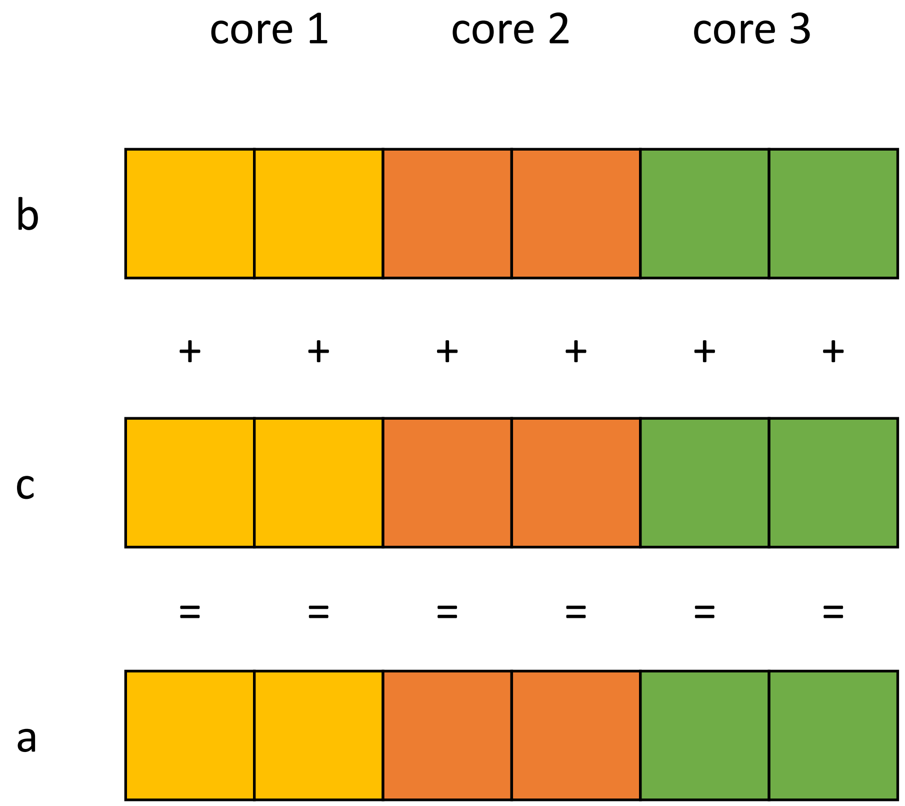
# For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {
   a[i] = b[i] + c[i]
}
```



# For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {
   a[i] = b[i] + c[i]
}
```



# SMP Parallelism in For Loops

- Given a nest of For loops, can we make the outer-most loop parallel?
  - Safely
  - Efficiently

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays
  - Only side-effects are array writes
  - Array bases are disjoint and constant
  - Bounds and array indexes are a function of loop variables, input variables and constants*
  - Loops increment by 1 and start at 0

*If the bounds and indexes are affine functions, then more analysis is possible, see dragon book*

# SMP Parallelism in For Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays
  - Only side-effects are array writes
  - Array bases are disjoint and constant
  - Bounds and array indexes are a function of loop variables, input variables and constants*
  - Loops Increment by 1 and start at 0

```
for (int i = 0; i < dim1; i++) {
  for (int j = 0; j < dim3; j++) {
    for (int k = 0; k < dim2; k++) {
      a[i][j] += b[i][k] * c[k][j];
    }
  }
}
```

# SMP Parallelism in For Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1 and start at 0

```
for (int i = 2; i < 100; i+=3) {
  a[i] = c[i + 128];
}
```

# SMP Parallelism in For Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1 and start at 0

Make new loop bounds:
i = j

```
for (int i = 2; i < 100; i+=3) {
  a[i] = c[i + 128];
}
```

# SMP Parallelism in For Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1 and start at 0

Make new loop bounds:
i = j*3 + 2

```
for (int j = 0; j < 32; j+=1) {
  a[j*3+2] = c[j*3+2 + 128];
}
```

subtract by constant to start at 0

# SMP Parallelism in For Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1 and start at 0

```
for (int i = 2; i < 100; i+=3) {
  a[i] = c[i + 128];
}
```

```
for (int j = 0; j < 32; j+=1) {
  a[3*j+2] = c[(3*j+2) + 128];
}
```

# SMP Parallelism in For Loops

- Given a nest of ***candidate*** *For* loops, determine if we can we make the outer-most loop parallel?
  - Safely
  - efficiently
- Criteria: every iteration of the outer-most loop must be *independent*
  - The loop can execute in any order, and produce the same result
- Such loops are called "DOALL" Loops. The can be flagged and handed off to another pass that can finely tune the parallelism (number of threads, chunking, etc)

# Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*

- 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

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

*First example: write-write conflict*

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

# Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

*First example: write-write conflict*

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

Calculate index based on i

# Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

*First example: write-write conflict*

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

Computation to store in the memory location

# Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

*First example: write-write conflict*

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

for two distinct iterations:

$i_x$ != $i_y$

Check:

index($i_x$) != index($i_y$)

# Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

*First example: write-write conflict*

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

for two distinct iterations:

$i_x$ != $i_y$

Check:

index($i_x$) != index($i_y$)

*Because we start at 0 and increment by 1, we can use i to refer to loop iterations*

# Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

*First example: write-write conflict*

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

for two distinct iterations:

$i_x$ != $i_y$

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

- 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);
}
```

*Examples:*

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

# Safety Criteria

- 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);
}
```

*Examples:*

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

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

# Safety Criteria

- 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

- 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;
}
```

# Automation?

- We have decent intuition about this, but if its going to be in a compiler, then it needs to be automatable

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

two integers: $i_x$ != $i_y$

$i_x$ >= 0

$i_x$ < 128

$i_y$ >= 0

$i_y$ < 128

*write-write conflict* write_index($i_x$) == write_index($i_y$)

*read-write conflict* write_index($i_x$) == read_index($i_y$)

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

# Automation?

- We have decent intuition about this, but if its going to be in a compiler, then it needs to be automatable

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

two integers: $i_x$ != $i_y$

$i_x >= 0$

$i_x < 128$

$i_y >= 0$

$i_y < 128$

$i_x == i_y$

$i_x == i_y$

# Automation?

- We have decent intuition about this, but if its going to be in a compiler, then it needs to be automatable

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

two integers: $i_x$ != $i_y$

$i_x$ >= 0

$i_x$ < 128

$i_y$ >= 0

$i_y$ < 128

$i_x$ == $i_y$

$i_x$ == $i_y$

*We can feed these constraints to an SMT Solver!*

# SMT Solver

- Satisfiability Modulo Theories (SMT)
  - Generalized SAT solver
- Solves many types of constraints over many domains
  - Integers
  - Reals
  - Bitvectors
  - Sets
- Complexity bounds are high (and often undecidable). In practice, they work pretty well

# Microsoft Z3

- State-of-the-art
- Python bindings
- Tutorials:
  - Python: https://ericpony.github.io/z3py-tutorial/guide-examples.htm
  - SMT LibV2: https://rise4fun.com/z3/tutorial

# Automation?

- We have decent intuition about this, but if its going to be in a compiler, then it needs to be automatable

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

two integers: $i_x \; != \; i_y$

$i_x >= 0$

$i_x < 128$

$i_y >= 0$

$i_y < 128$

$i_x == i_y$

$i_x == i_y$

*We can feed these constraints to an SMT Solver!*

# Another example:

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

Write-write

two integers: $i_x$ != $i_y$

$i_x$ >= 0

$i_x$ < 128

$i_y$ >= 0

$i_y$ < 128

$i_x$ % 64 == $i_y$ % 64

# Another example:

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

Write-read?

two integers: $i_x$ != $i_y$

$i_x$ >= 0

$i_x$ < 128

$i_y$ >= 0

$i_y$ < 128

$i_x$ % 64 == ?

# Another 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

$i_x$ % 64 == ?

Write-write? Write-read?

# Another 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

$i_x$ % 64 == $i_y$ + 64

Write-read

# 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));
        }
    }
}
```

# 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: $i0_x \neq i0_y$

**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($i0_x$, $i1_x$ ...) == write_index ($i0_y$, $i1_y$ ...)

**4.** Do the same for write-read conflicts

# 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));
        }
    }
}
```

*What if we want to parallelize an inner loop?*

**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(i0`$_x$ `,i1`$_x$ `...) == write_index (i0`$_y$ `,i1`$_y$ `...)`

**4.** Do the same for write-read conflicts

# Are data races ever okay?

- Thoughts?

# Are data races ever okay?

- Consider this program:

```
int x = 0;
for (int i = 0; i < 1024; i++) {
    int tmp = *(&x);
    tmp += 1;
    *(&x) = tmp;
}
```

What can go wrong if we run the loop in parallel?

December 28, 2011
Volume 9, issue 12

PDF

# You Don’t Know Jack about Shared Variables or Memory Models

## Data races are evil.

Hans-J. Boehm, HP Laboratories, Sarita V. Adve, University of Illinois at Urbana-Champaign

The final count can also be too high. Consider a case in which the count is bigger than a machine word. To avoid dealing with binary numbers, assume we have a decimal machine in which each word holds three digits, and the counter x can hold six digits. The compiler translates x++ to something like

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;
x_hi = tmp_hi;
x_lo = tmp_lo;
```

Now assume that x is 999 (i.e., x_hi = 0, and x_lo = 999), and two threads, a blue and a red one, each increment x as follows (remember that each thread has its own copy of the machine registers tmp_hi and tmp_lo):

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;   //tmp_hi = 1, tmp_lo = 0
x_hi = tmp_hi;        //x_hi = 1, x_lo = 999, x = 1999
          x++;        //red runs all steps
                      //x_hi = 2, x_lo = 0, x = 2000
x_lo = tmp_lo;        //x_hi = 2, x_lo = 0
```

# Horrible data races in the real world

Therac 25: a radiation therapy machine

- Between 1987 and 1989 a software bug caused 6 cases where radiation was massively overdosed
- Patients were seriously injured and even died.
- Bug was root caused to be a data race.
- https://en.wikipedia.org/wiki/Therac-25

# Horrible data races in the real world

2003 NE power blackout

- second largest power outage in history: 55 million people were effected

- NYC was without power for 2 days, estimated 100 deaths

- Root cause was a data race

- https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

# But checking for data conflicts is hard...

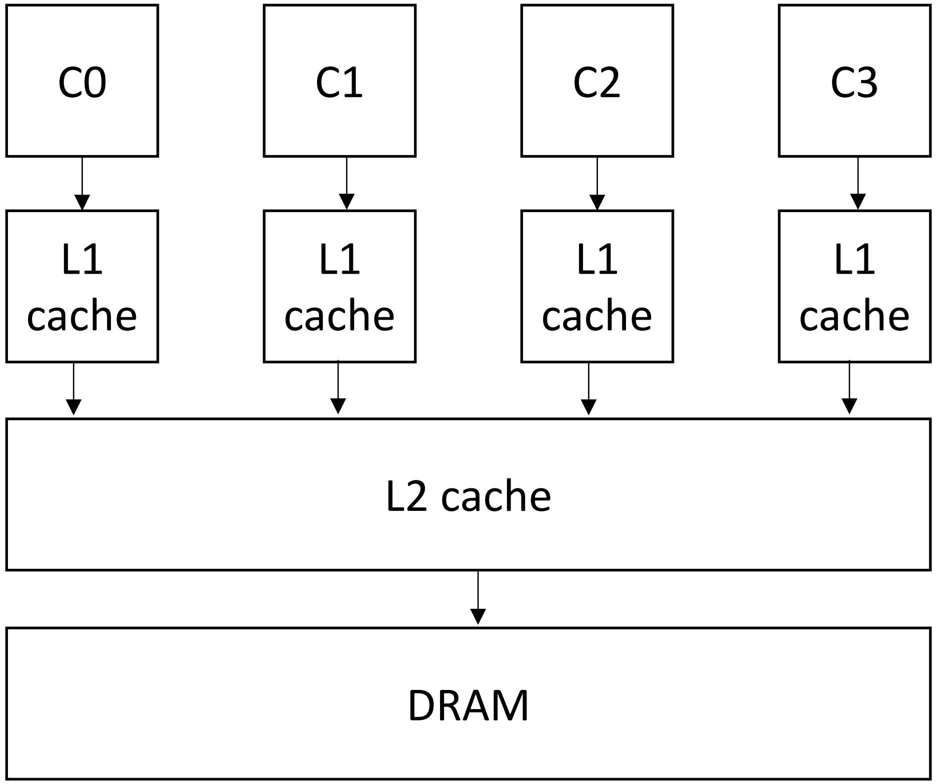
- Tools are here to help (Professor Flanagan is famous in this area)

- My previous group:
  - “Dynamic Race Detection for C++11” Lidbury and Donaldson
  - Scalable (complete) race detection
    - Firefox has ~40 data races
    - Chromium has ~6 data races

# Moving on to DSLs

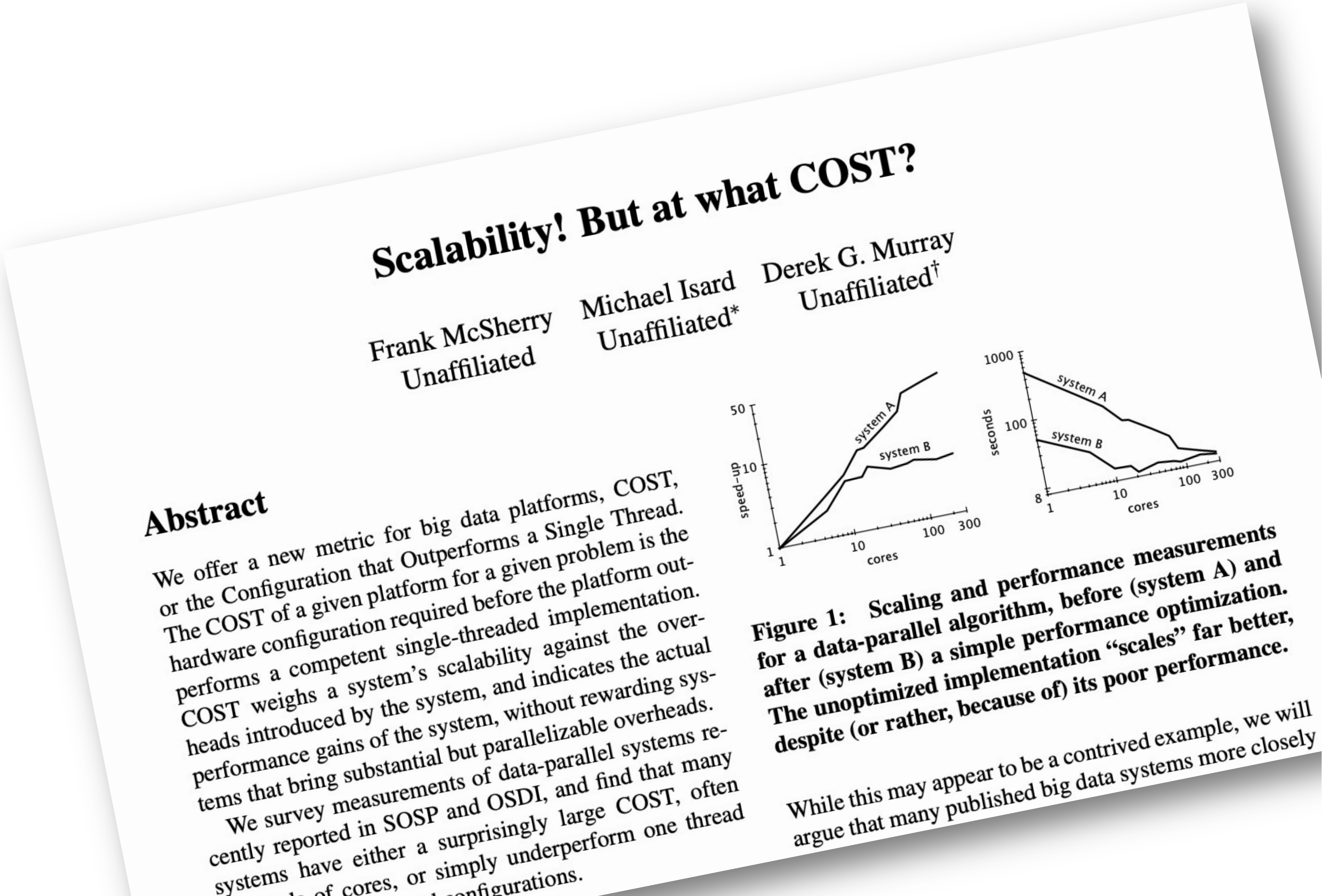
# Shifting our focus back to a single core

- Why?



# Shifting our focus back to a single core

- Why?



# Scalability! But at what COST?

Frank McSherry
Unaffiliated

Michael Isard
Unaffiliated*

Derek G. Murray
Unaffiliated†

## Abstract

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

We survey measurements of data-parallel systems recently reported in SOSP and OSDI, and find that many systems have either a surprisingly large COST, often ... of cores, or simply underperform one thread ... configurations.



**Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.**

While this may appear to be a contrived example, we will argue that many published big data systems more closely

# Shifting our focus back to a single core

- Why?

## 1 Introduction

"You can have a second computer once you've shown you know how to use the first one."

–Paul Barham

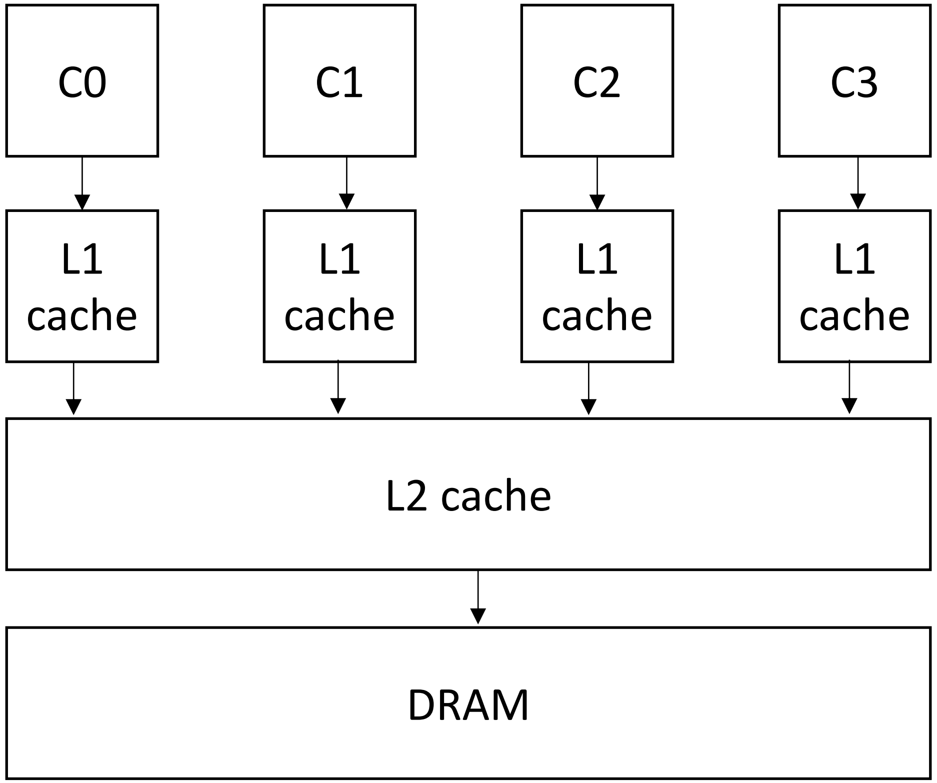
| scalable system | cores | twitter | uk-2007-05 |
|---|---|---|---|
| GraphChi [12] | 2 | 3160s | 6972s |
| Stratosphere [8] | 16 | 2250s | - |
| X-Stream [21] | 16 | 1488s | - |
| Spark [10] | 128 | 857s | 1759s |
| Giraph [10] | 128 | 596s | 1235s |
| GraphLab [10] | 128 | 249s | 833s |
| GraphX [10] | 128 | 419s | 462s |
| Single thread (SSD) | 1 | 300s | 651s |
| Single thread (RAM) | 1 | 275s | - |

**Table 2: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. GraphChi and X-Stream report times for 5 PageRank iterations, which we multiplied by four.**

# Shifting our focus back to a single core

- We need to consider single threaded performance

- Good single threaded performance can enable better parallel performance
  - **Memory locality** is key to good parallel performance.



# Transforming Loops

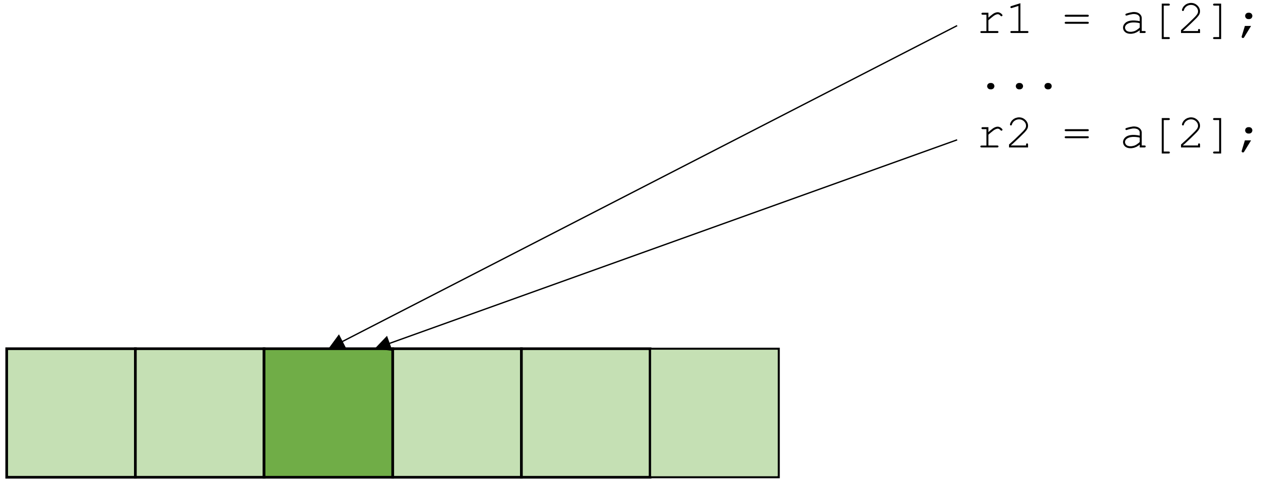
- Locality is key for good (parallel) performance:
- What kind of locality are we talking about?

# Transforming Loops

- Locality is key for good parallel performance:

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

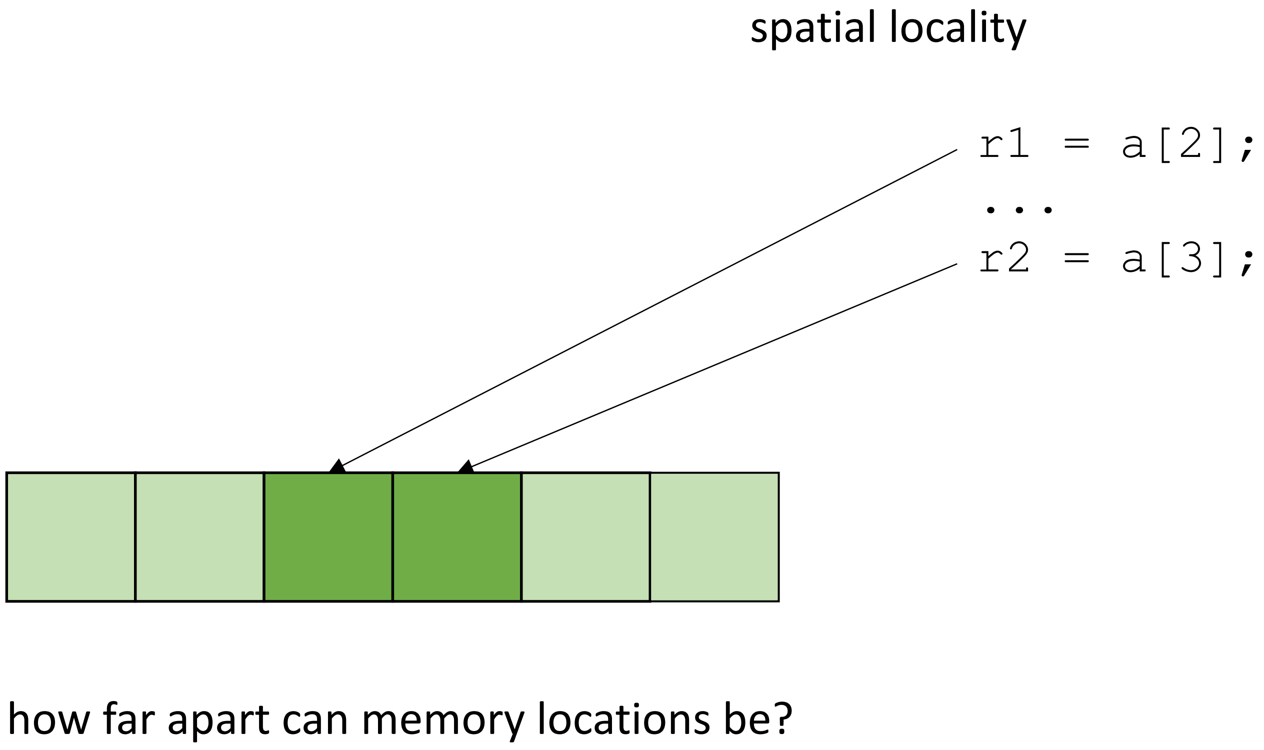
temporal locality



# Transforming Loops

- Locality is key for good parallel performance:

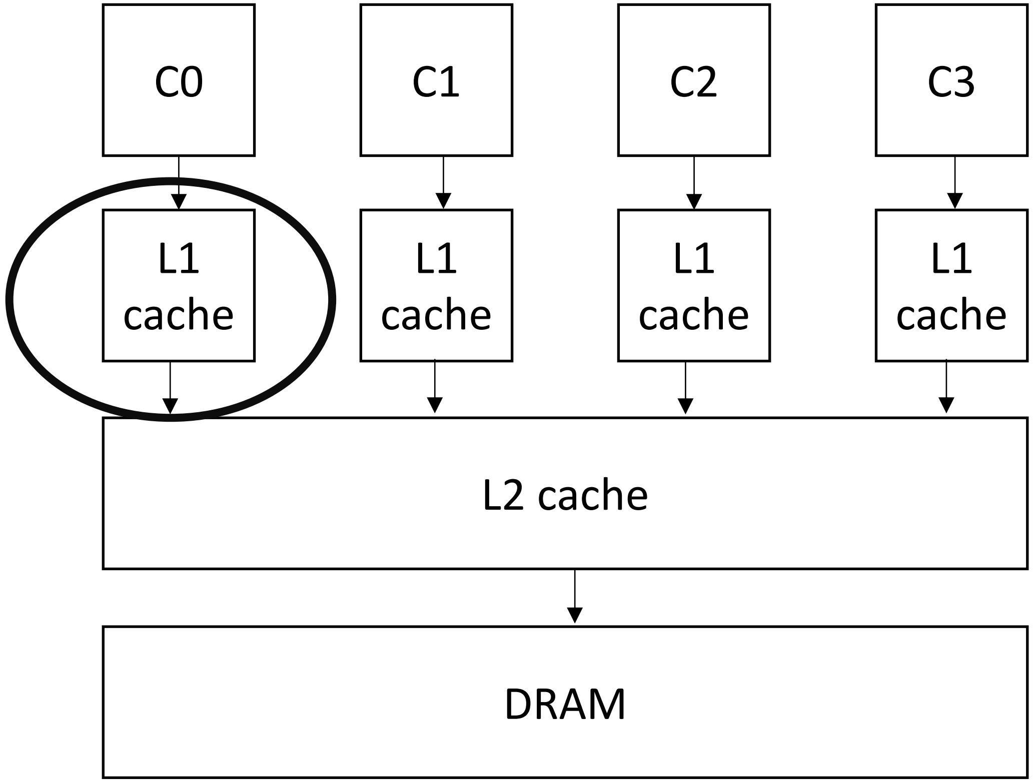
- Two types of locality:
  - Temporal locality
  - Spatial locality



# Transforming Loops

- Locality is key for good (parallel) performance:

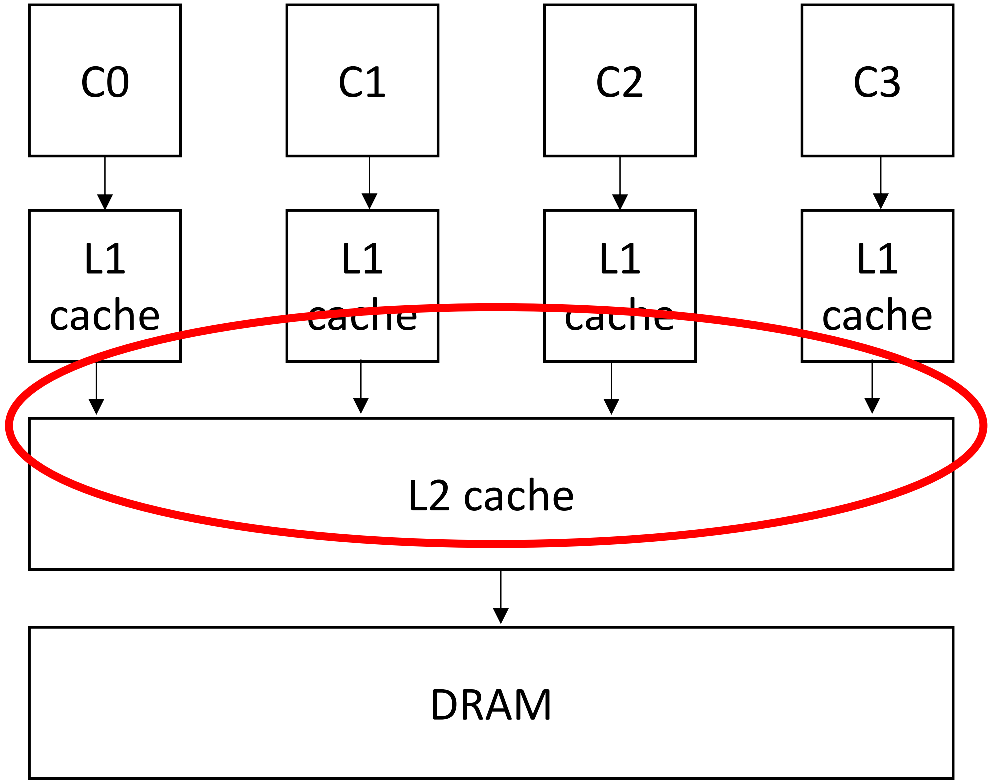
good data locality: cores will spend most of their time accessing private caches



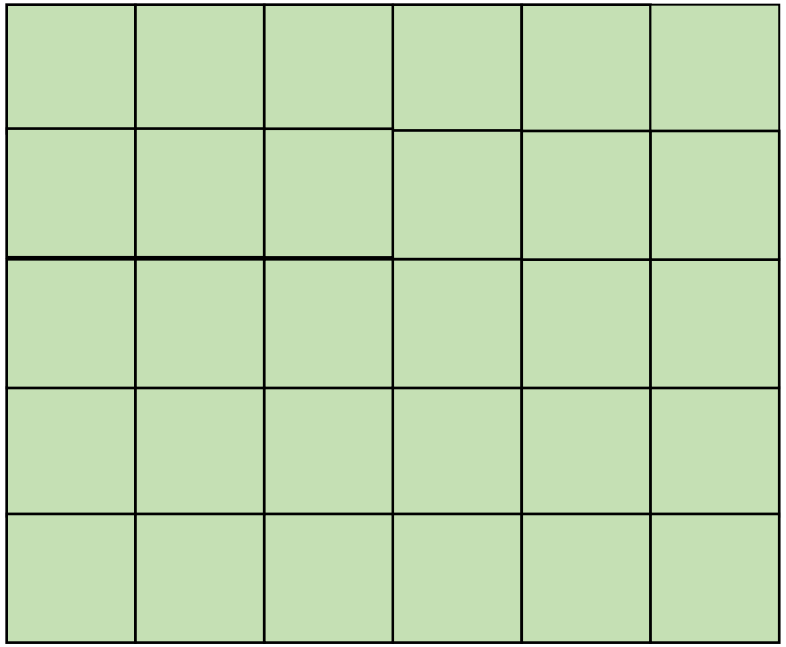
# Transforming Loops

- Locality is key for good (parallel) performance:

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

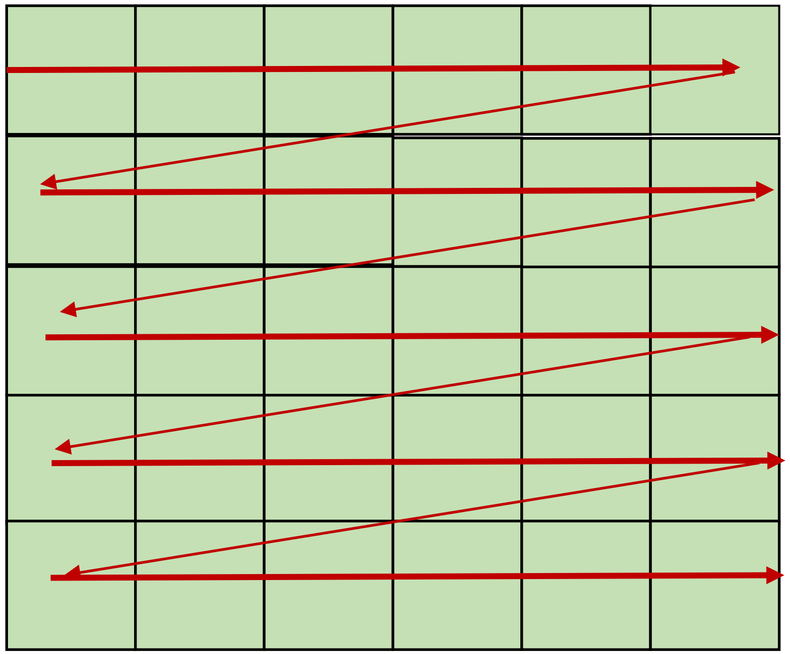


# How multi dimensional arrays are stored:



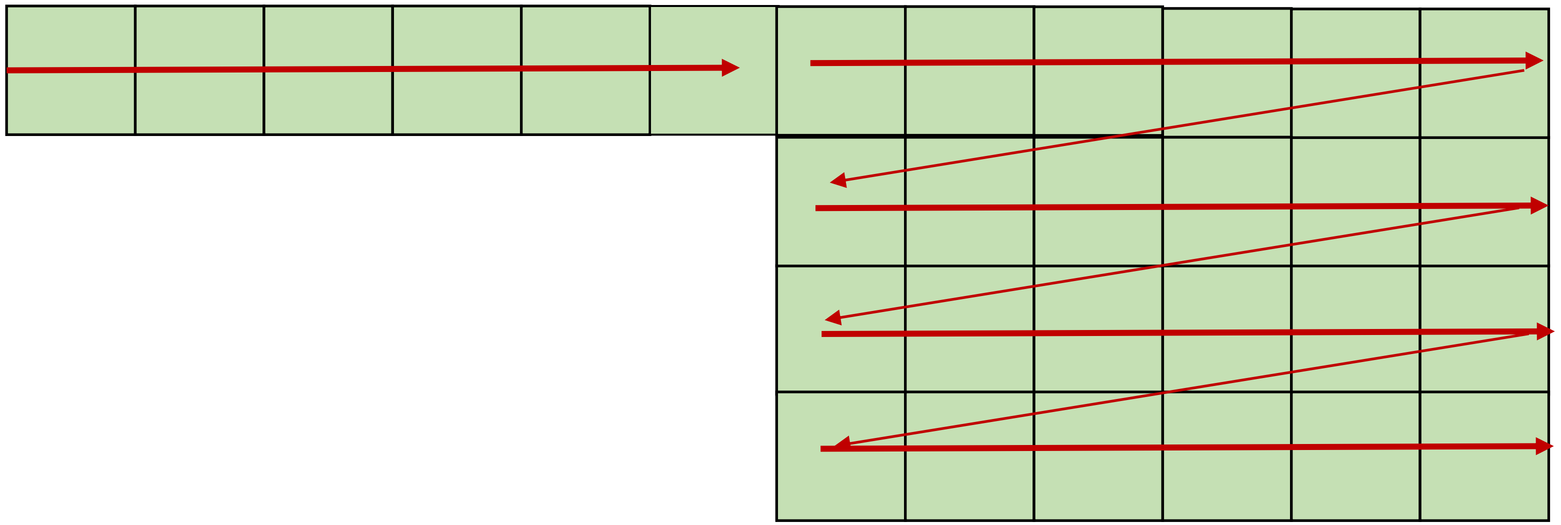
# How multi dimensional arrays are stored:

Row major



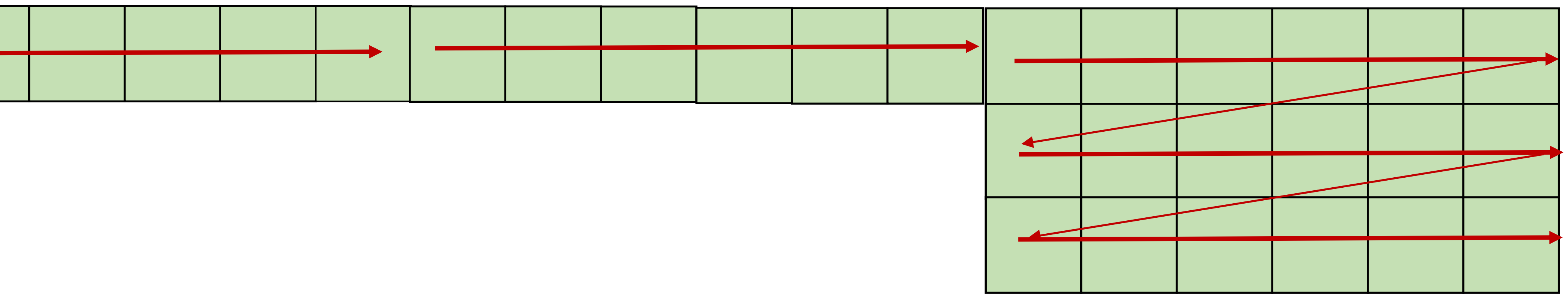
# How multi dimensional arrays are stored:

Row major



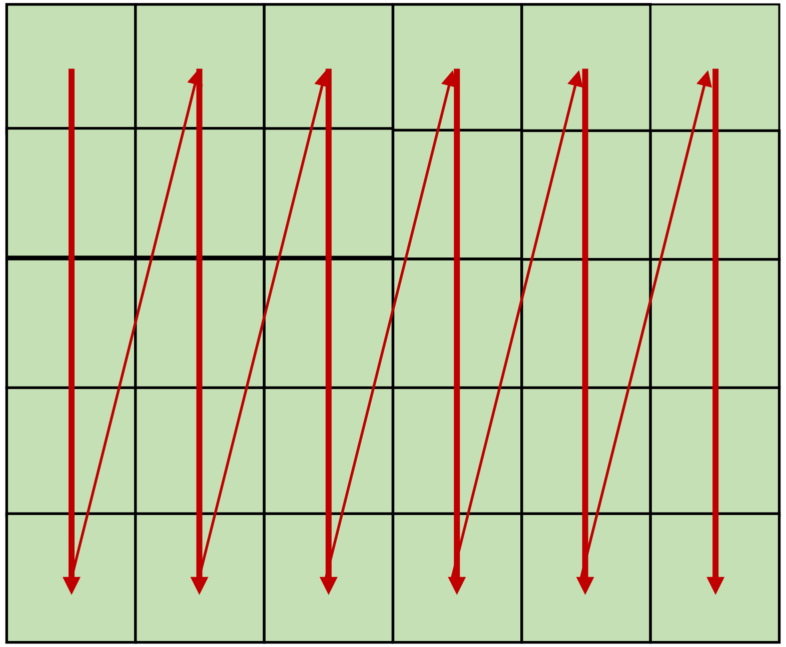
# How multi dimensional arrays are stored:

Row major



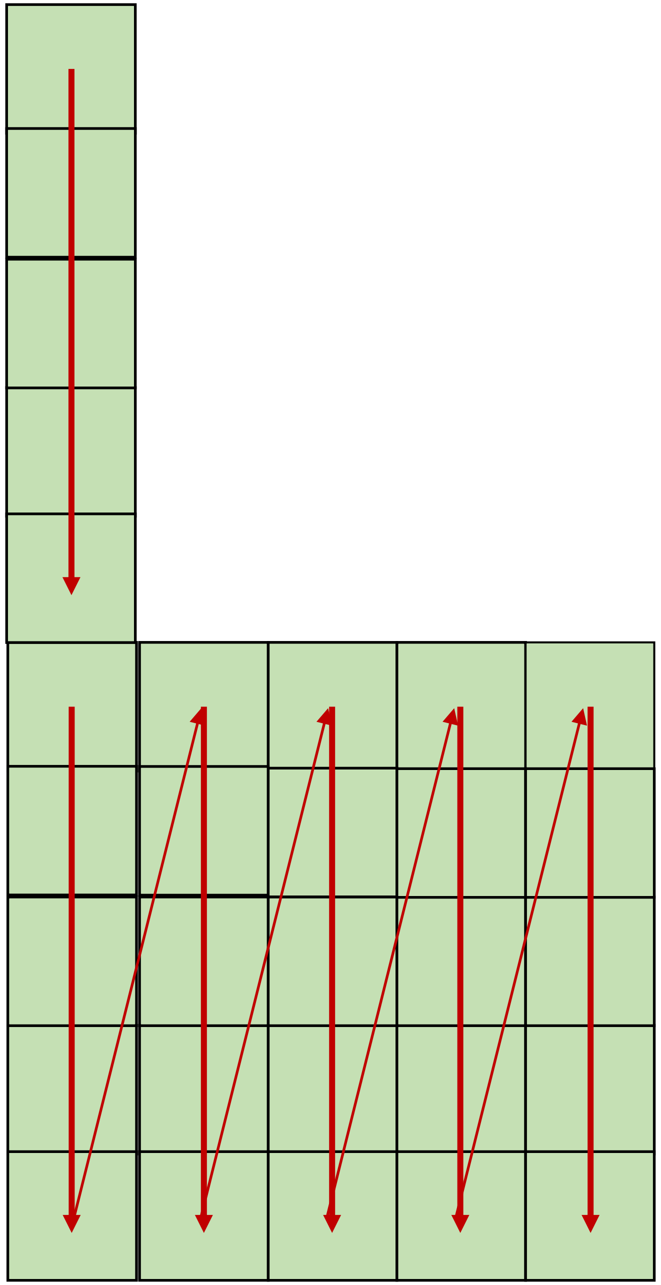
# How multi dimensional arrays are stored:

Column major?
Fortran
Matlab
R



# How multi dimensional arrays are stored:

Column major?
Fortran
Matlab
R

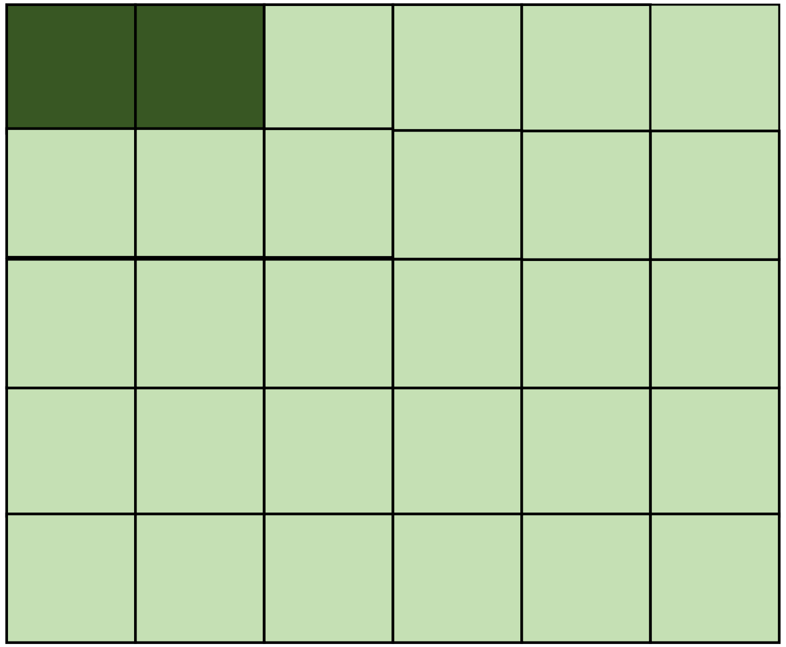


# How multi dimensional arrays are stored:

say x == y == 0

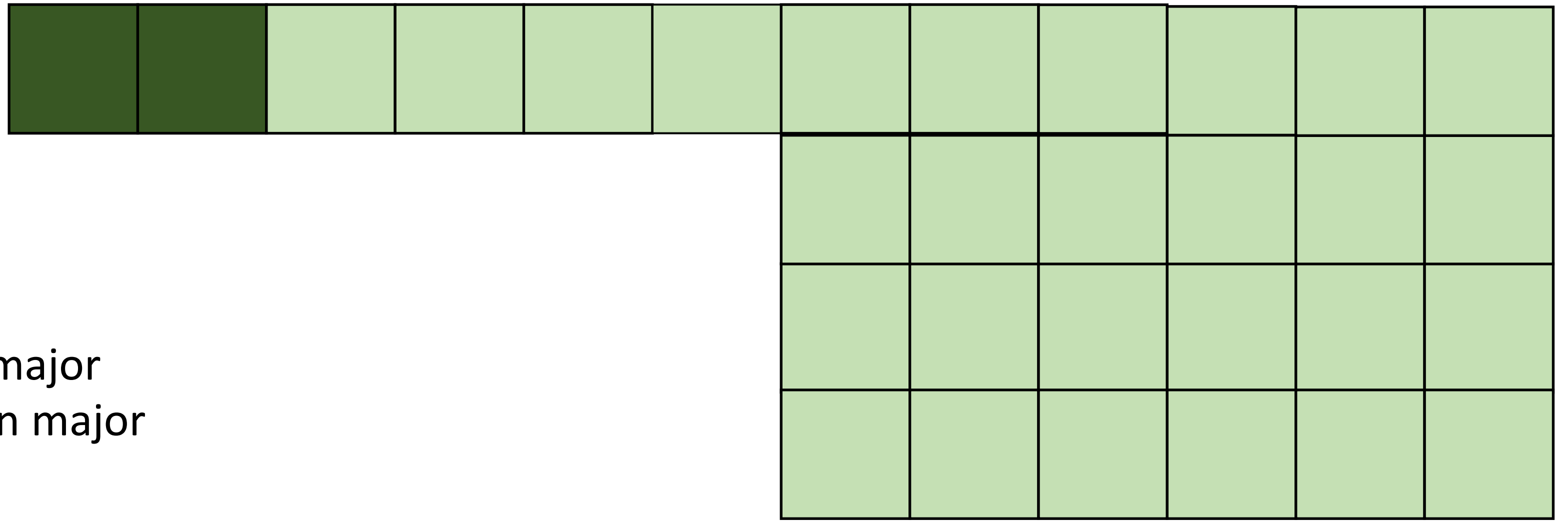
```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major



# How multi dimensional arrays are stored:

unrolled row major: still has locality



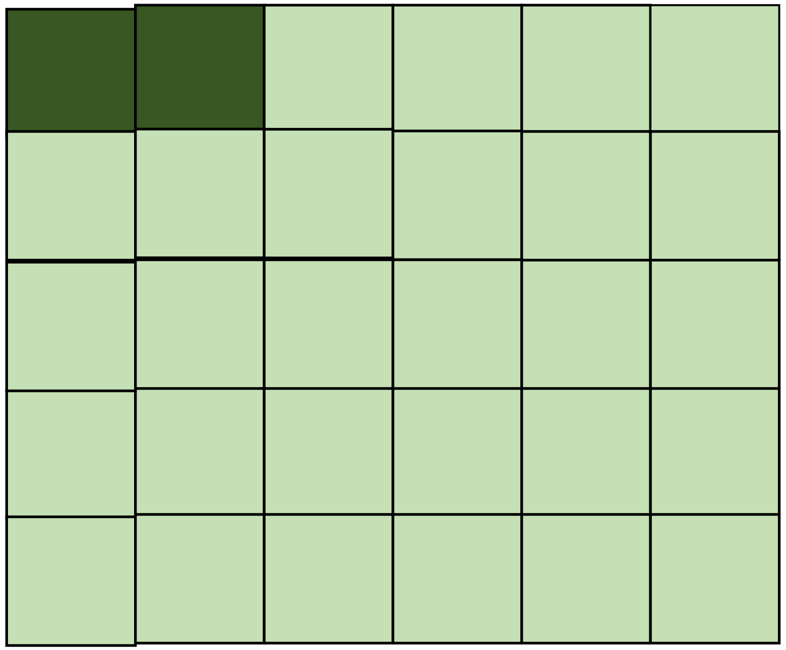
```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major

# How multi dimensional arrays are stored:

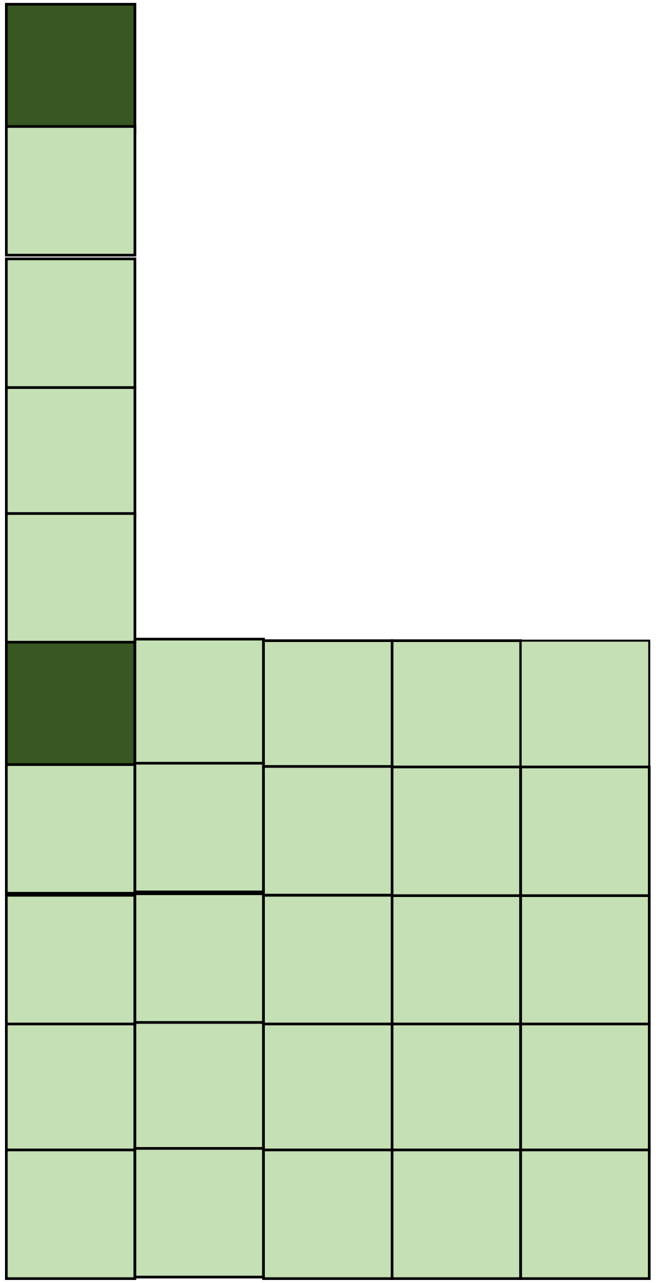
```
x1 = a[x,y];
x2 = a[x, y+1];
```

good pattern for row major
bad pattern for column major



# How multi dimensional arrays are stored:

unrolled
column
major:
Bad locality



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

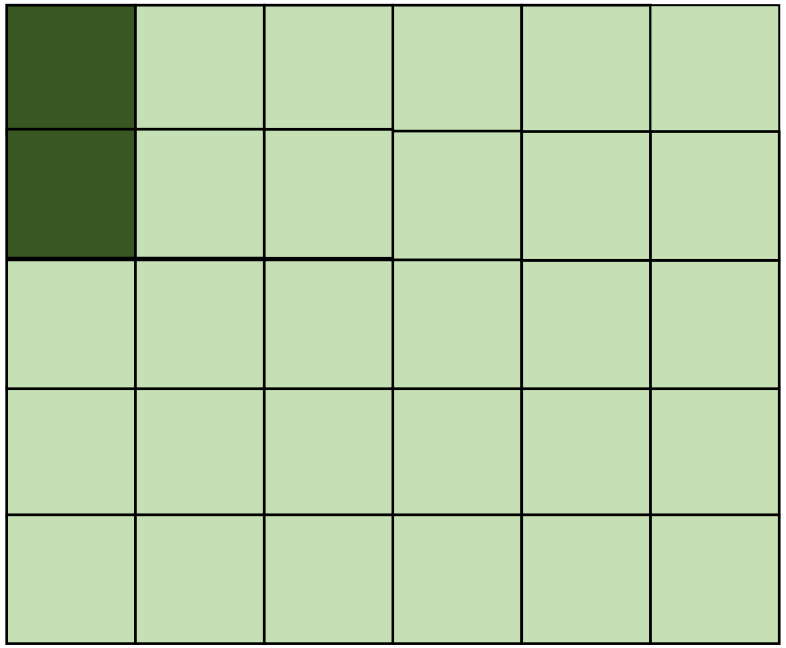
good pattern for row major
bad pattern for column major

# How multi dimensional arrays are stored:

say x == y == 0

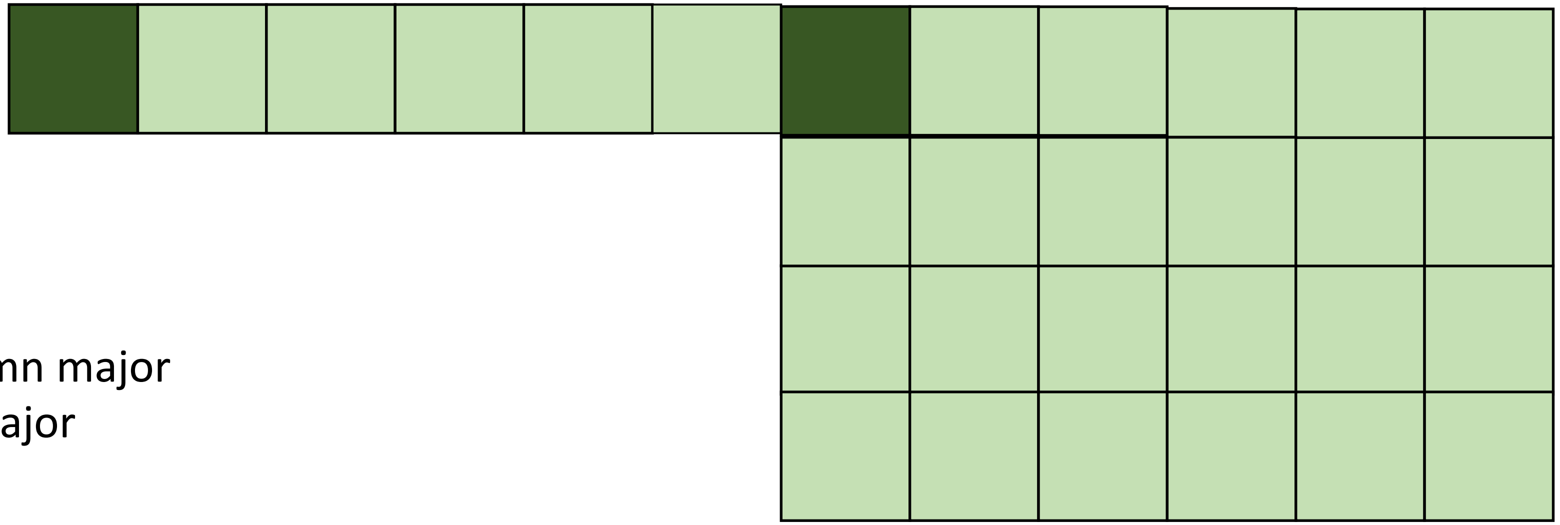
```
x1 = a[x,y];
x2 = a[x+1, y];
```

good pattern for column major
bad pattern for row major



# How multi dimensional arrays are stored:

row major unrolled: bad spatial locality

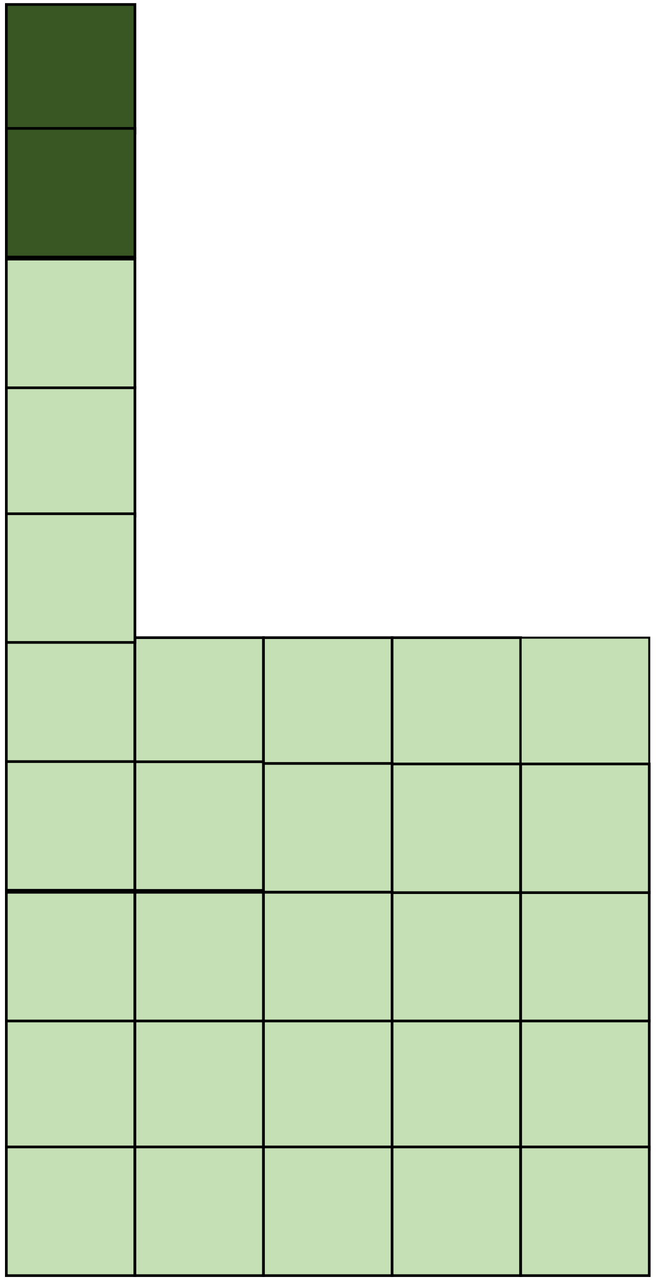


```
x1 = a[x,y];
x2 = a[x+1, y];
```

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

# Next class

- Topics:
  - Restructuring loops

- Remember:
  - Homework 2 due tomorrow
  - Midterm due on Friday
  - Office hours tomorrow 3-5