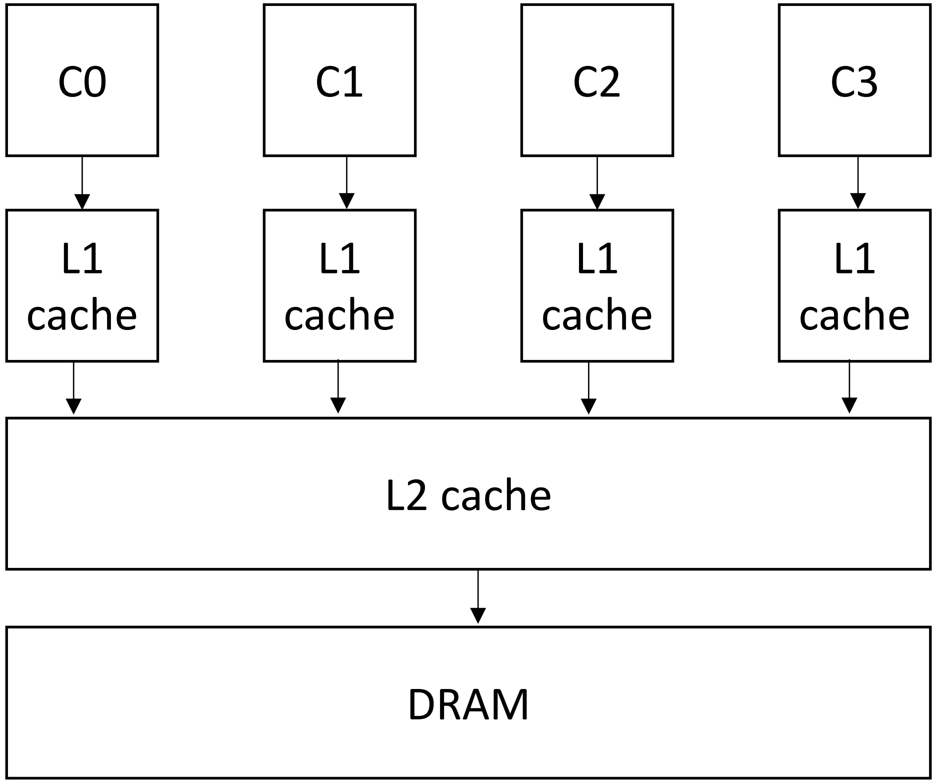
CSE211: Compiler Design

Nov. 1, 2021

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

- **Discussion questions**:
    - What parallel frameworks have you used?
    - Do you achieve linear speedup?
    - When is it safe to parallelize for loops?



Announcements

- Midterm is posted
    - Questions can be sent via email or dm or slack
    - Due on Wednesday at midnight
    - Do not expect replies after the work day

- Homework 2 is due today
    - Turn in on canvas

- Homework 3 will be assigned on Wednesday

Paper/Project proposals

- Please start thinking about these.
  - Message me for recommendations
  - Tell me what you’re interested in so we can find a good fit!

- Proposals due on Nov. 14
  - Please be pro-active about this. If you don’t have one in mind, please send me an email with some of your interests ASAP

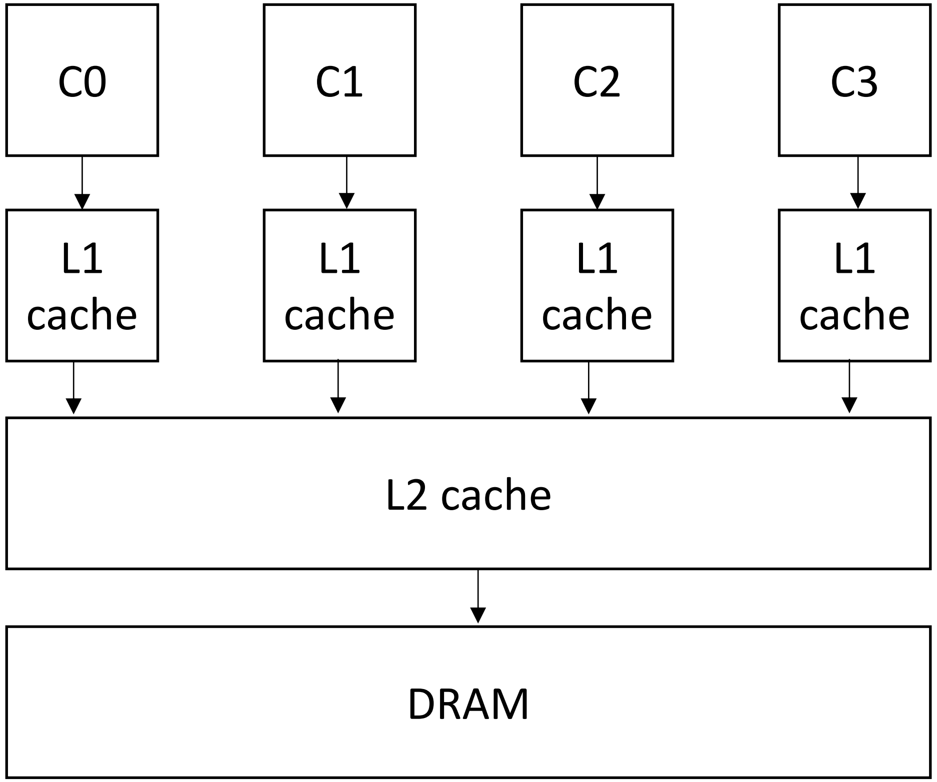
- Midterm is a good indicator for how the final will be.

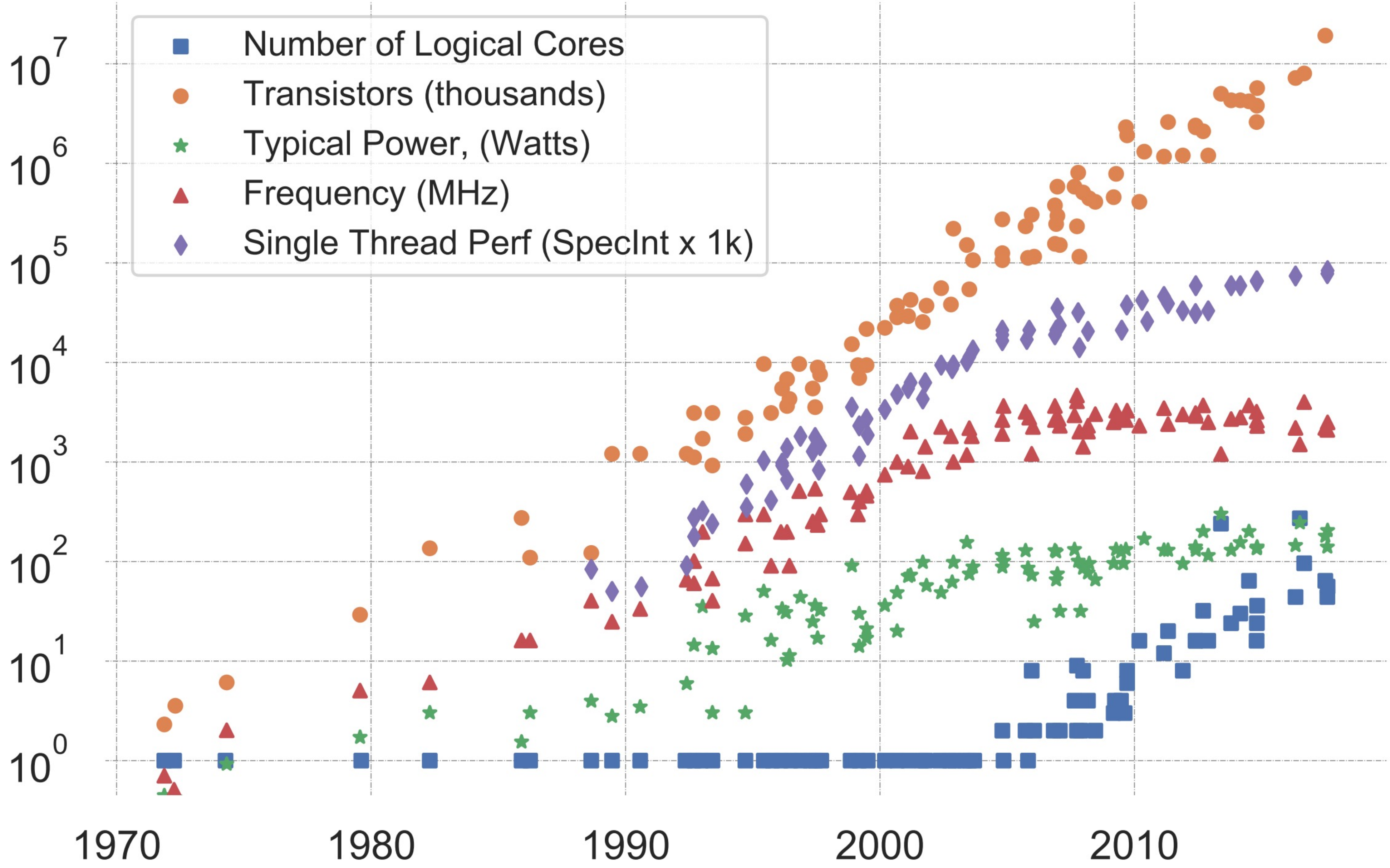
CSE211: Compiler Design

Nov. 1, 2021

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

- **Discussion questions**:
  - What parallel frameworks have you used?
  - Do you achieve linear speedup?
  - When is it safe to parallelize for loops?





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

Trends

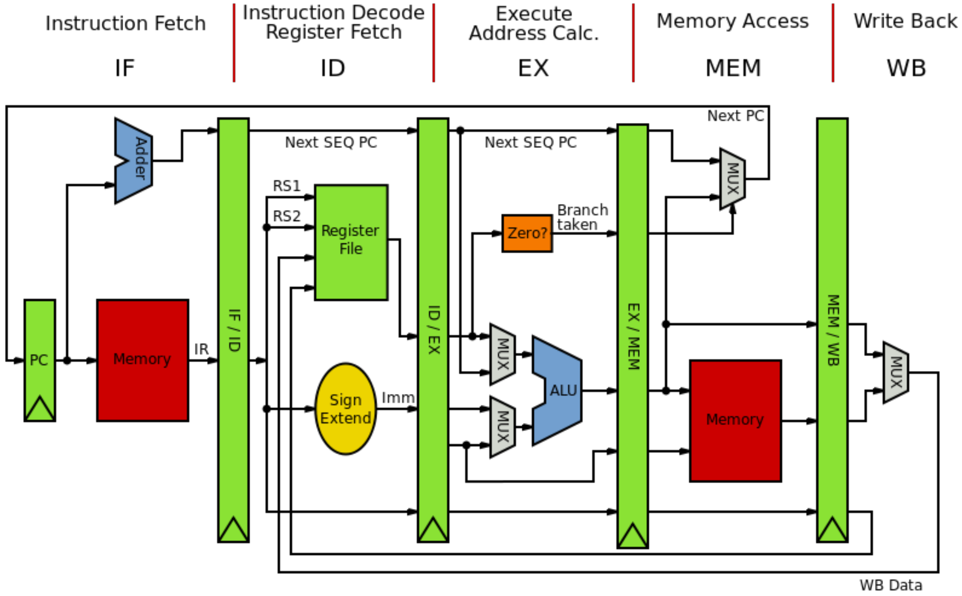
- Frequency scaling: **Dennard's scaling**
  - Mostly agreed that this is over

- Number of transistors: **Moore's law**
  - On its last legs.
  - Intel delaying 7nm chips. Apple has a 5nm. Some roadmaps project up to 3nm

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

How do chips exploit parallelism?

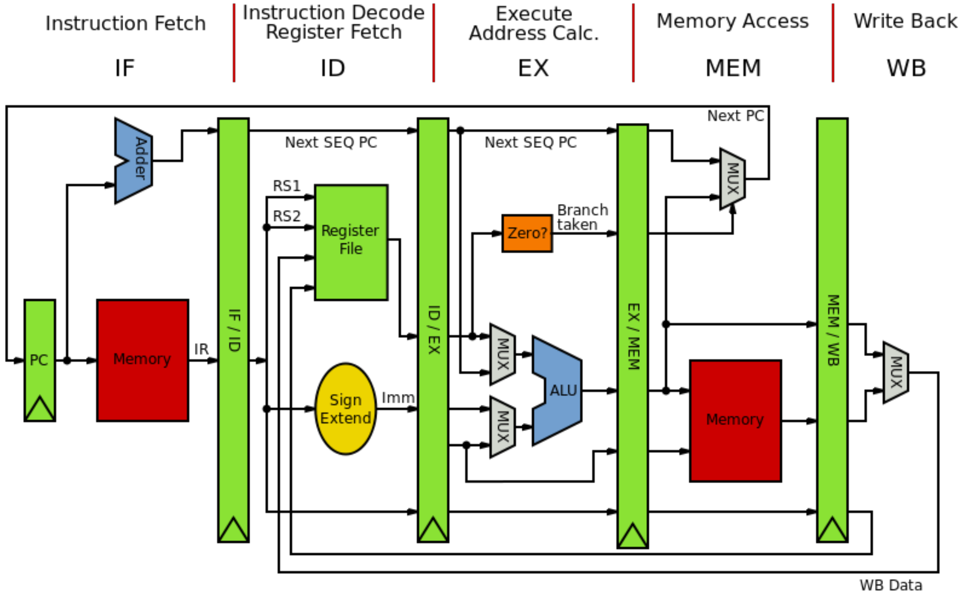
- Pipelines?
  - Only so much meaningful work to do per-stage.
  - Stage timing imbalance
  - Staging overhead



- Superscalar width?
  - Hardware checking becomes prohibitive:

How do chips exploit parallelism?

- Pipelines?
  - Only so much meaningful work to do per-stage.
  - Stage timing imbalance
  - Staging overhead
- Superscalar width?
  - Hardware checking becomes prohibitive:



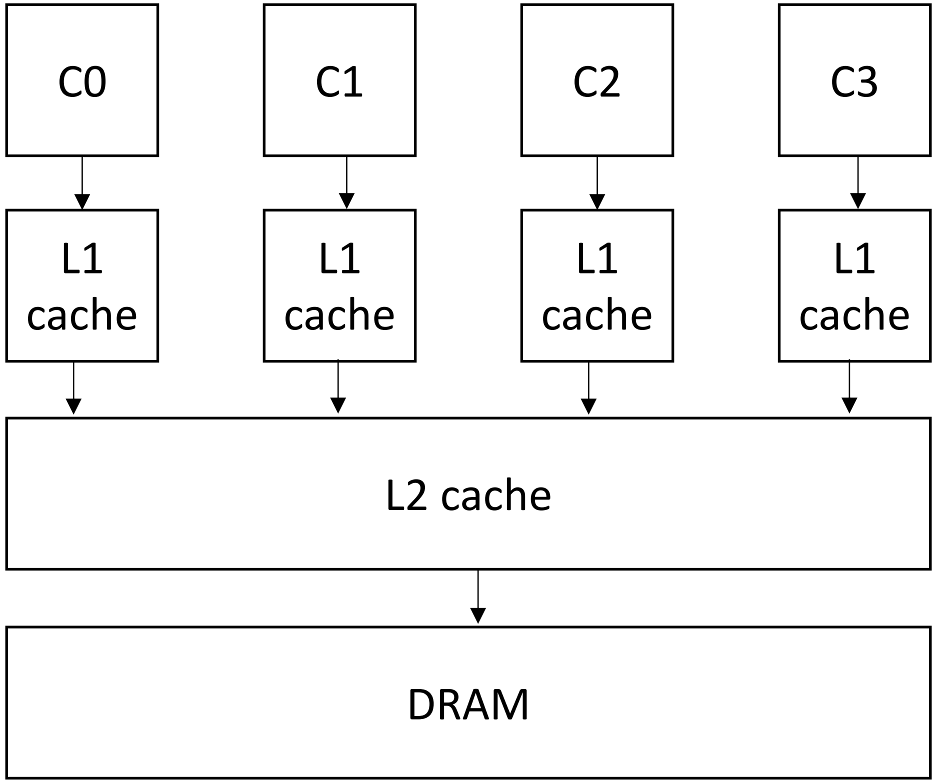
Collectively the power consumption, complexity and gate delay costs limit the achievable superscalar speedup to roughly eight simultaneously dispatched instructions.

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

Symmetric Multiprocessing (SMP)

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

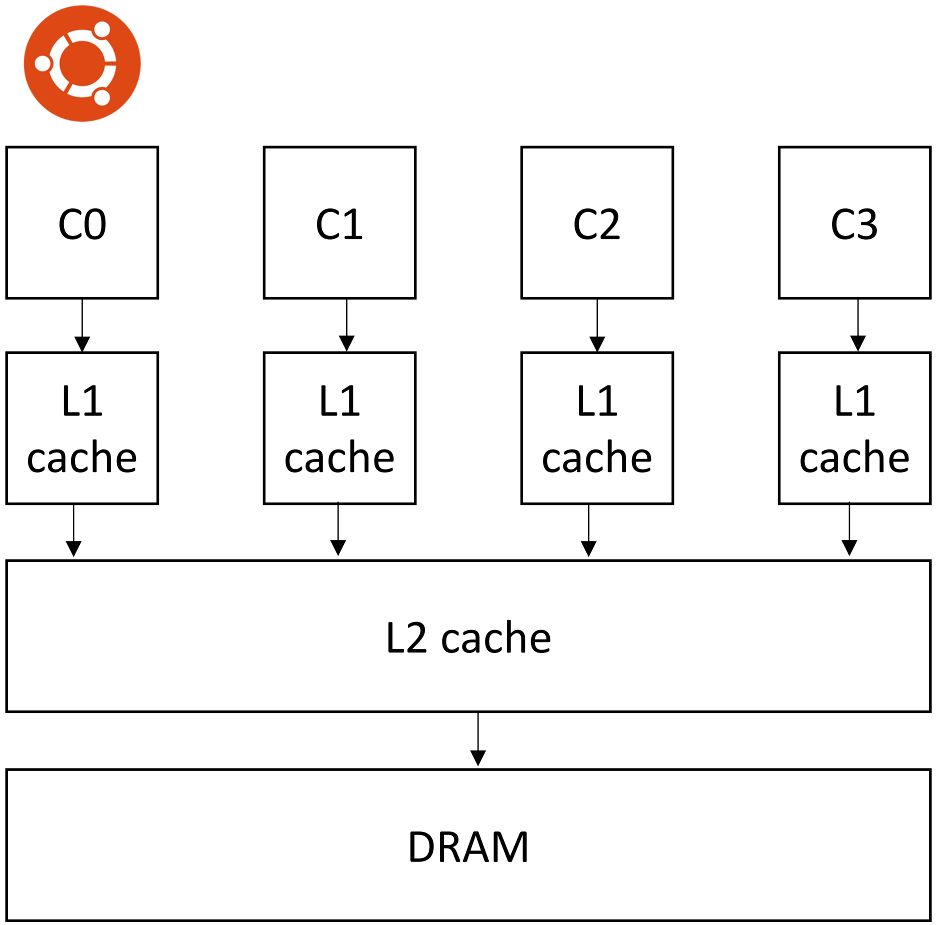
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines



Symmetric Multiprocessing (SMP)

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

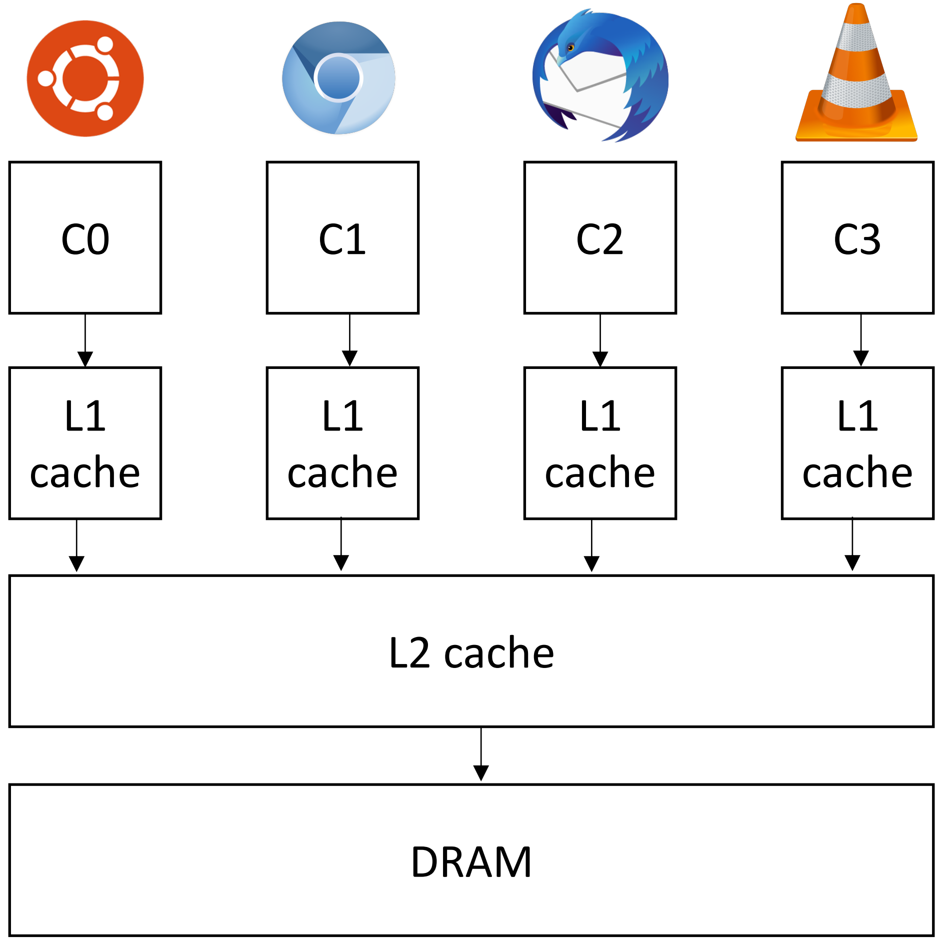
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines



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



C0

C1

C2

C3

L1 cache

L1 cache

L1 cache

L1 cache

L2 cache

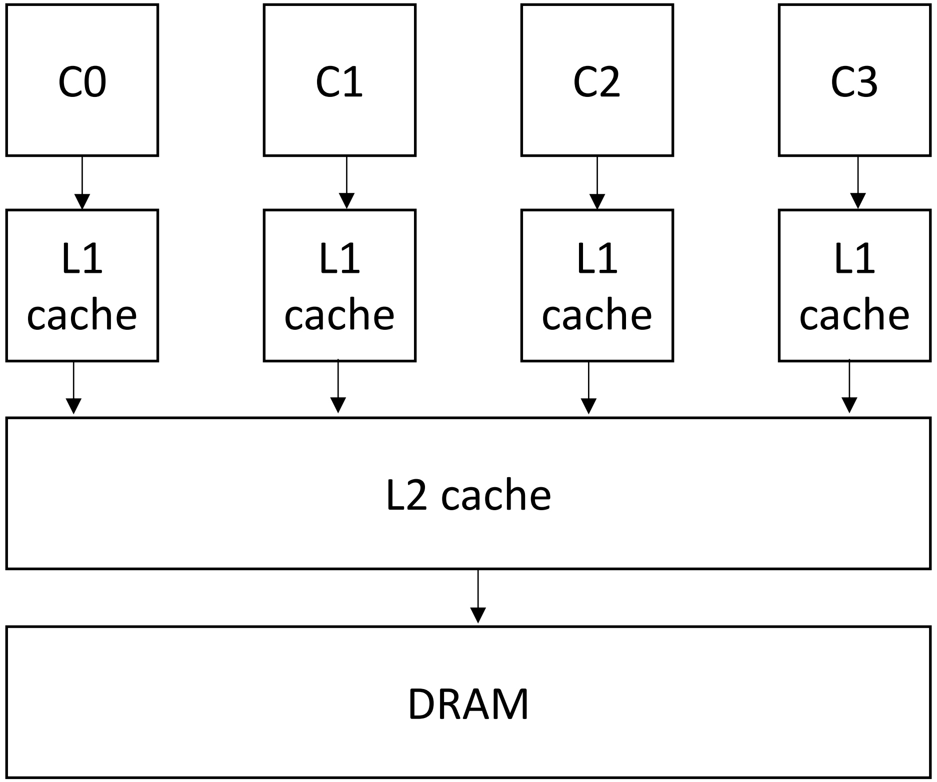
DRAM

Symmetric Multiprocessing (SMP)

PyTorch

- 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



C0 C1 C2 C3

L1 cache L1 cache L1 cache L1 cache

L2 cache

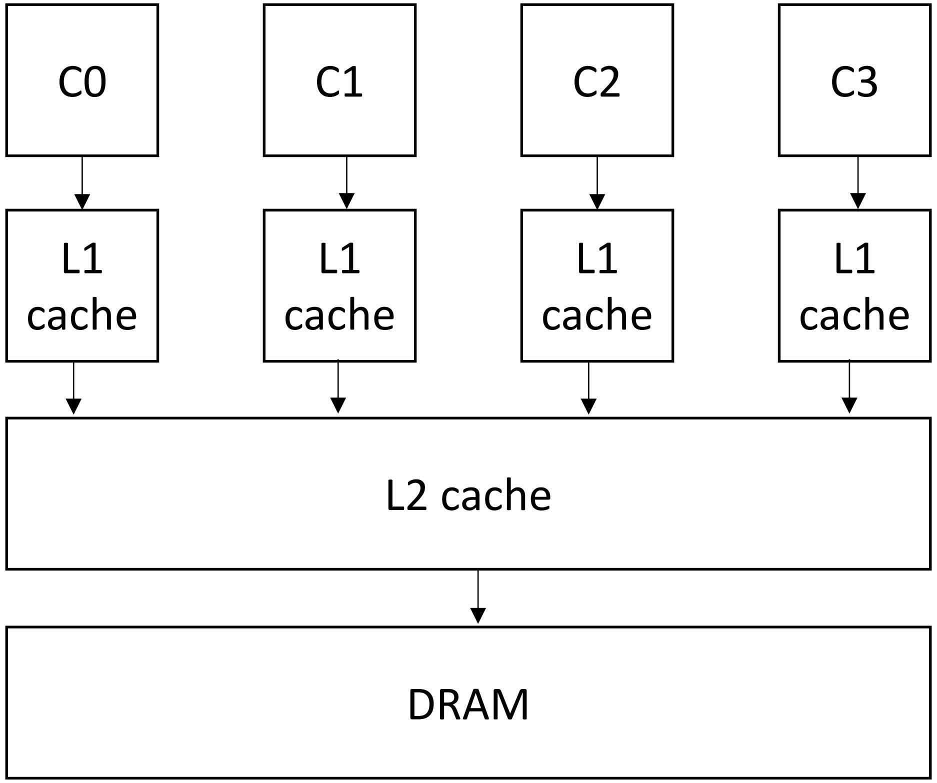
DRAM

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: 10 core

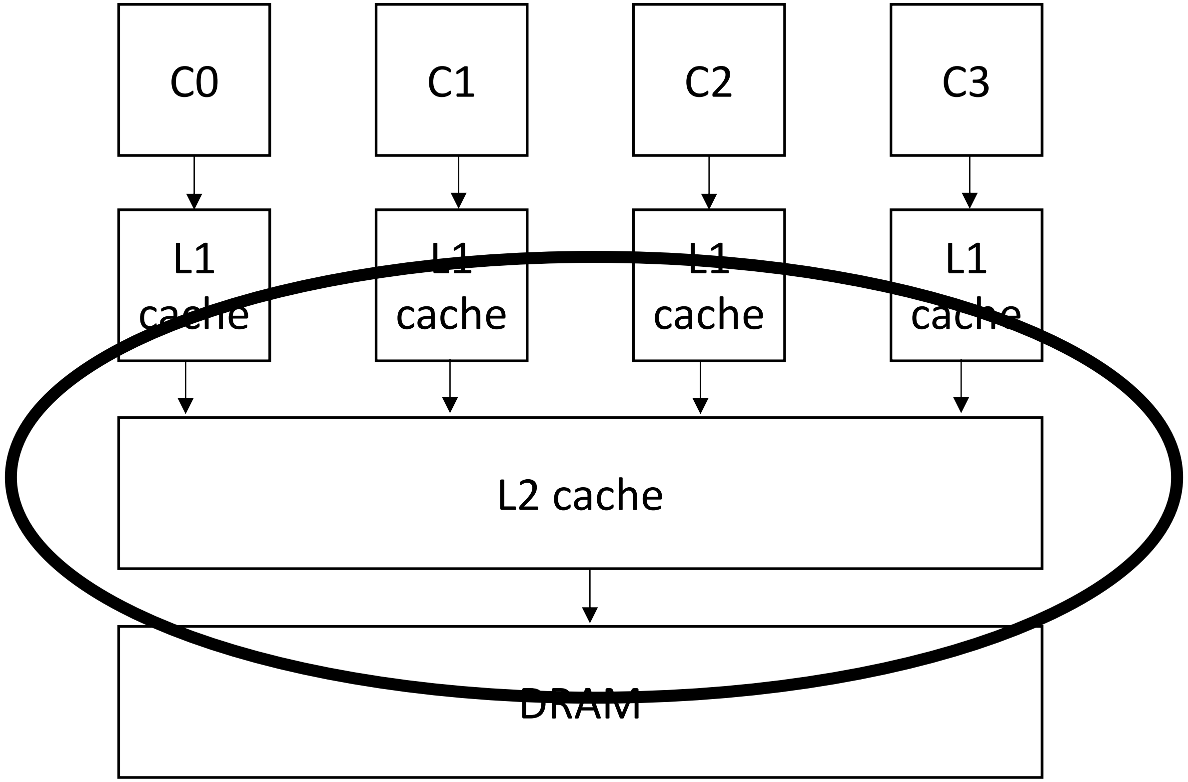
- Workstations:
    - 2 - 64 cores
    - ARM racks: 128

- 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

SMP systems are widespread

- Laptops
  - My laptop has 8 cores
  - Most have at least 2
  - New Macbook: 10 core
- Workstations:
  - 2 - 64 cores
  - 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

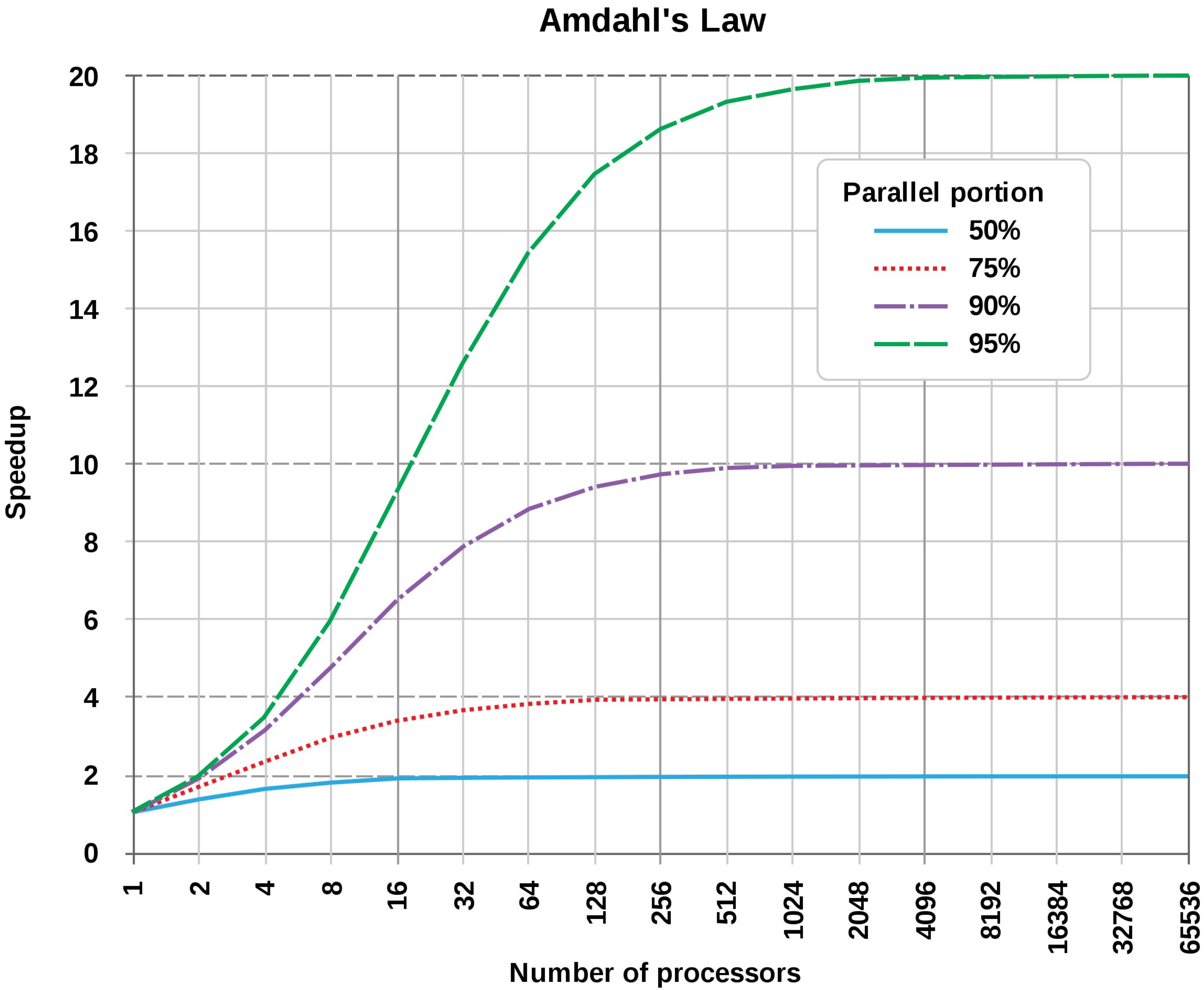
Potential for Parallel Speedup

- Amdahl's law

- *Speedup*(c) = $\frac{1}{(1-p)+\frac{p}{c}}$

- Where c is the number of cores and p is the percentage of the program execution time that would be improved by parallelism

- Assumes linear speedups



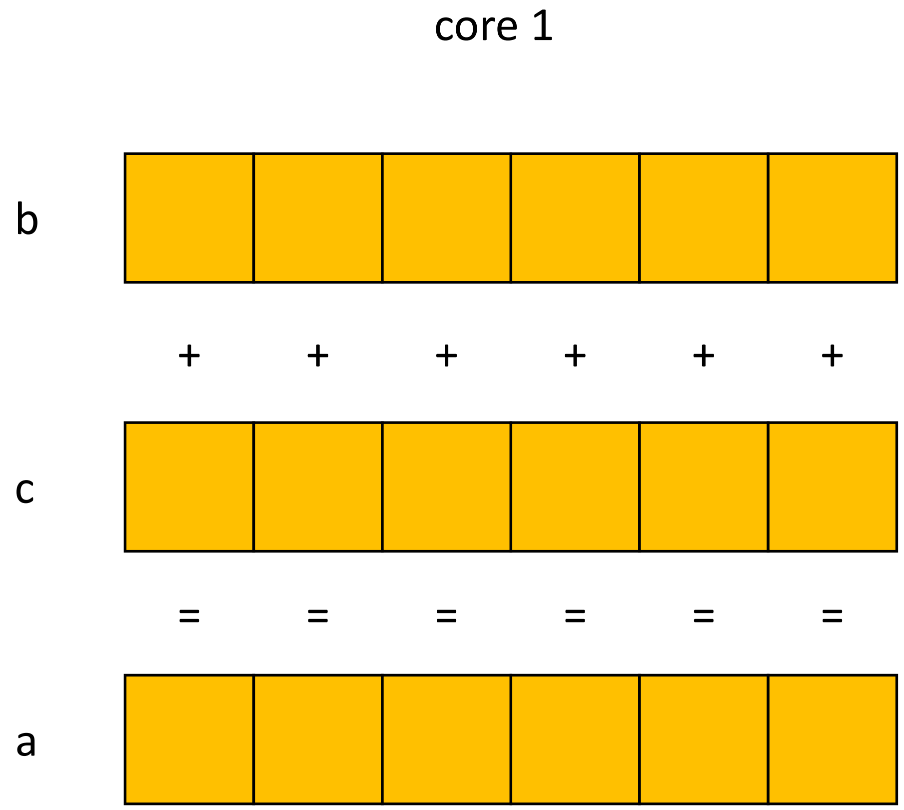
from wikipedia

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

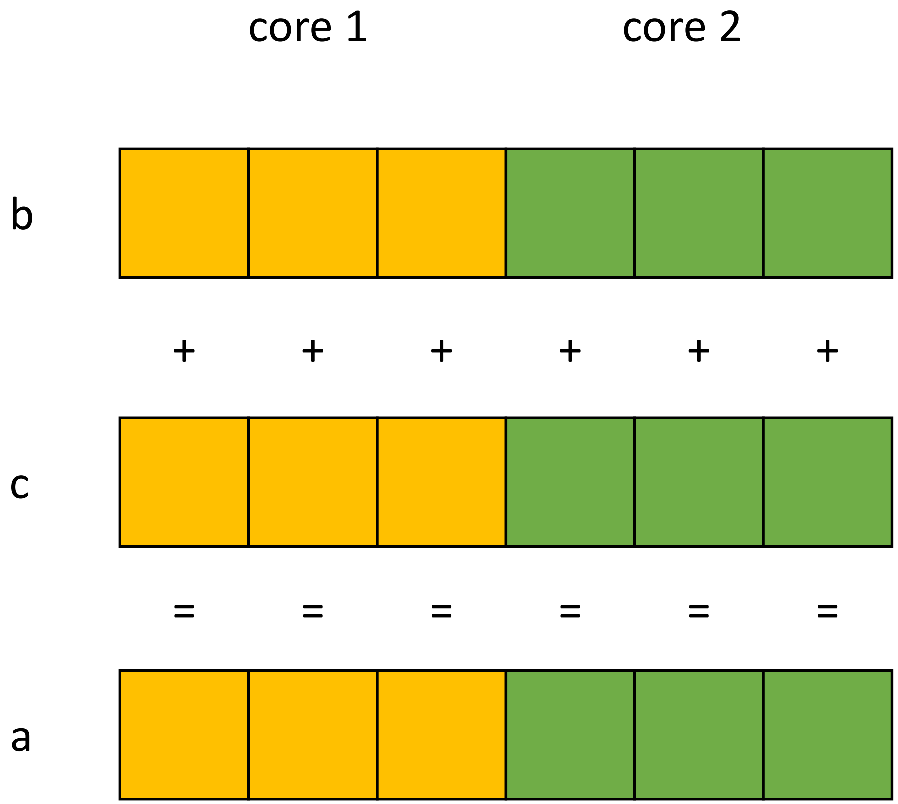
For loops are great candidates for SMP parallelism

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



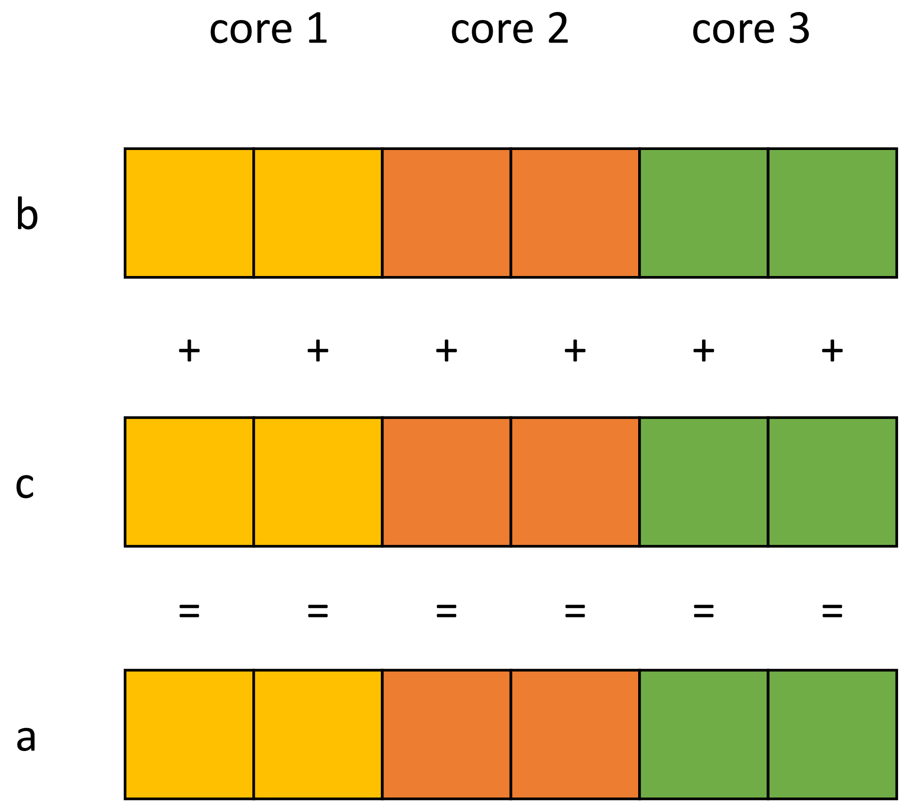
For loops are great candidates for SMP parallelism

```
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]
}
```



Demo

- Vector addition

Demo

- Safety

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 + 2)*3

```
for (int j = 0; j < 98; j+=1) {
  a[(j+2)*3] = c[j+2 + 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

```
for (int j = 2; j < 34; j+=1) {
  a[j*3] = c[j*3 + 128];
}
```

Multiply by a constant to make increment by 1. update loop body, update and bounds

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:

$\texttt{i}_\texttt{x} \mathrel{!=} \texttt{i}_\texttt{y}$

Check:

$\texttt{write\_index}(\texttt{i}_\texttt{x}) \mathrel{!=} \texttt{read\_index}(\texttt{i}_\texttt{y})$

Why?

if $\texttt{i}_\texttt{x}$ iteration happens first, then iteration $\texttt{i}_\texttt{y}$ reads an updated value.

if $\texttt{i}_\texttt{y}$ happens first, then it reads the original value

Next class

- Topics:
    - Reasoning about loop conflicts

- Remember:
    - Homework 2 due today
    - Midterm due on Wednesday (by midnight!)