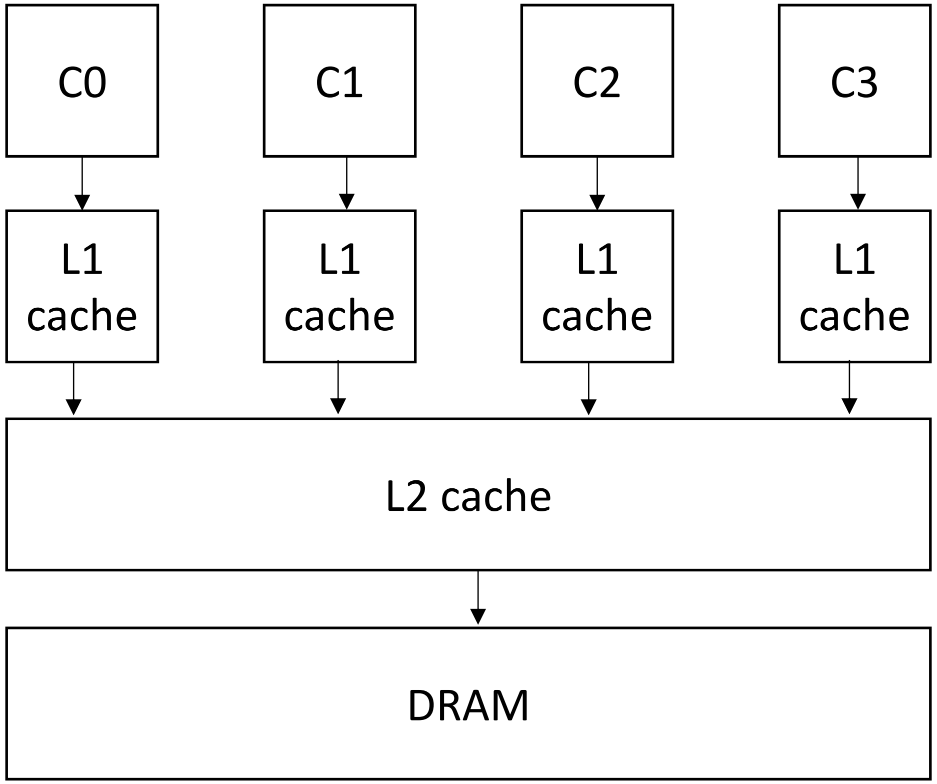
CSE211: Compiler Design

Nov. 12, 2020

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

- **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. Please write your answers on a separate piece of paper. You can do it on the computer, by hand, or a hybrid. As long as I can read your answers
- Homework 3 is posted. You have 3 weeks to complete.
- Homework 2 is Due today. I will collect early tomorrow morning.

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. 24

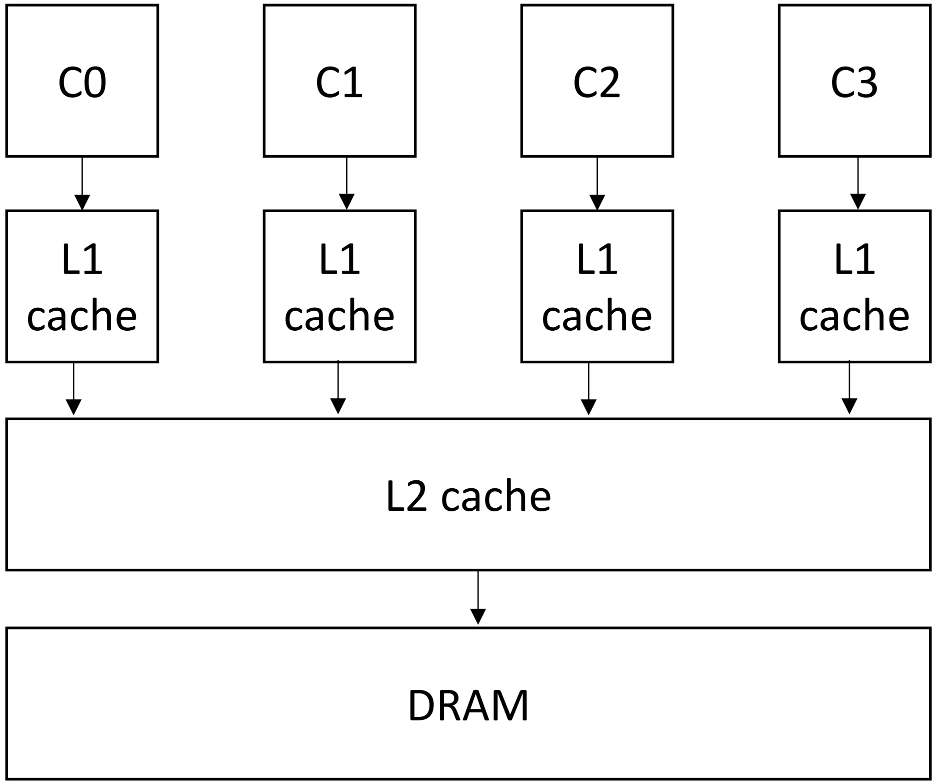
- Midterm is a good indicator for how the final will be.

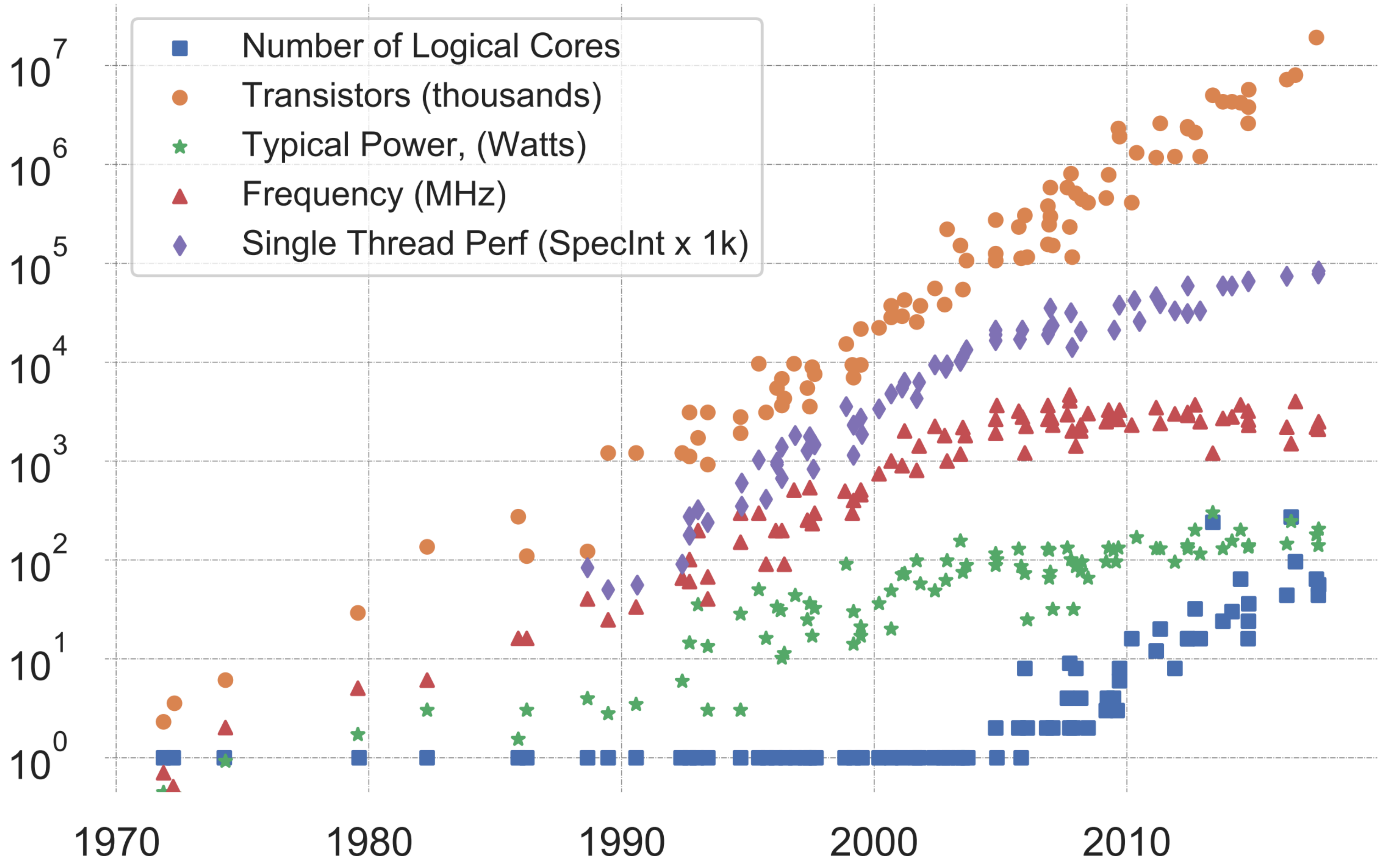
CSE211: Compiler Design

Nov. 12, 2020

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

- **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.

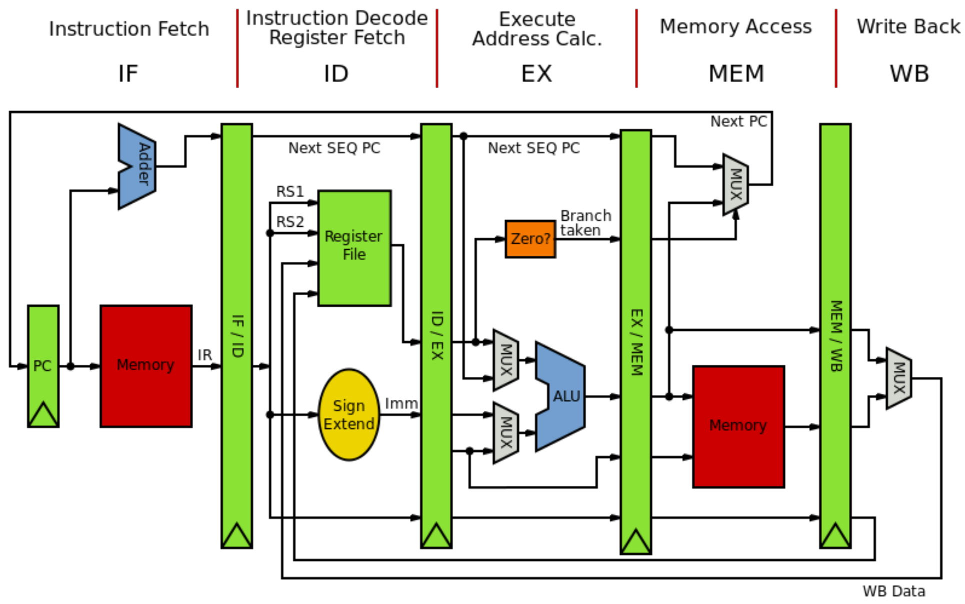
Trends

- 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?

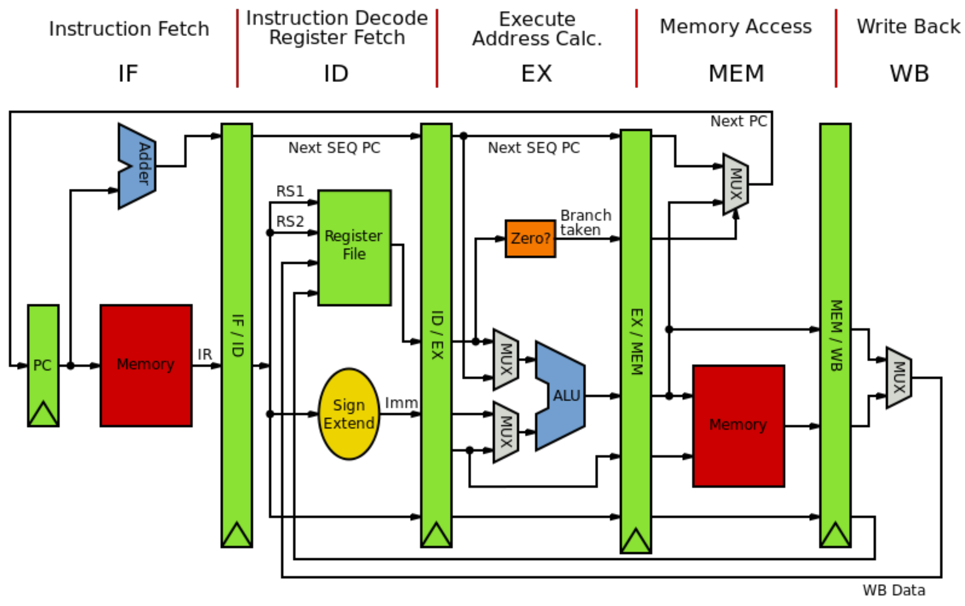
- Pipelines?
  - Only so much meaningful work to do per-stage.
  - Stage timing imbalance
  - Staging overhead

- 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:

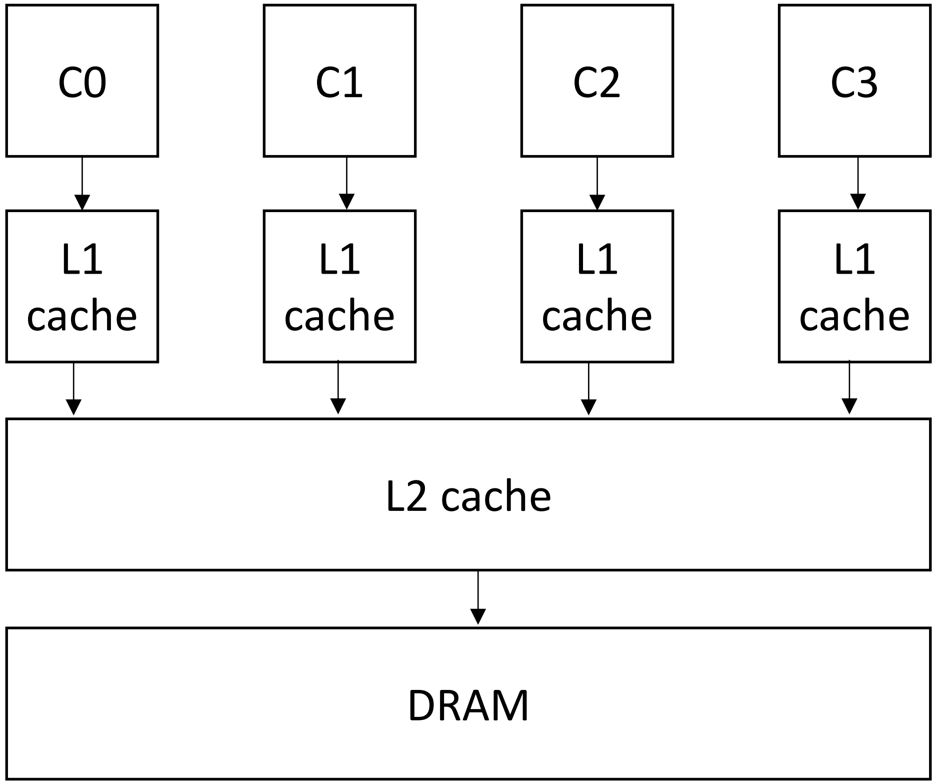
Collectively the power consumption, complexity and gate delay costs limit the achievable superscalar speedup to roughly eight simultaneously dispatched instructions.

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

Symmetric Multiprocessing (SMP)

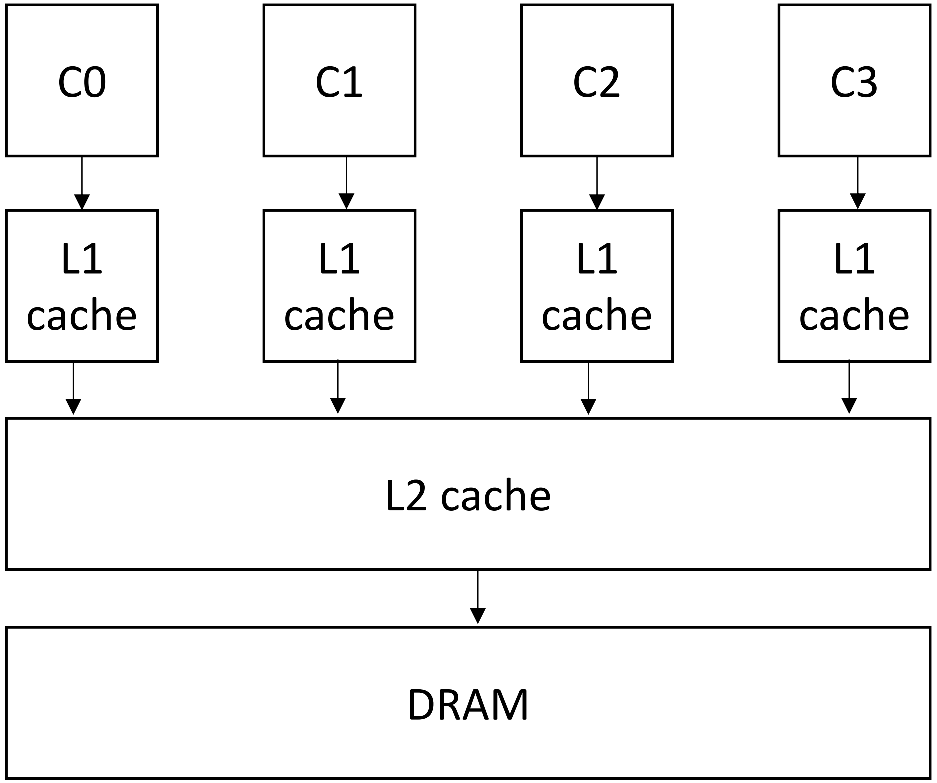
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS

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



Symmetric Multiprocessing (SMP)

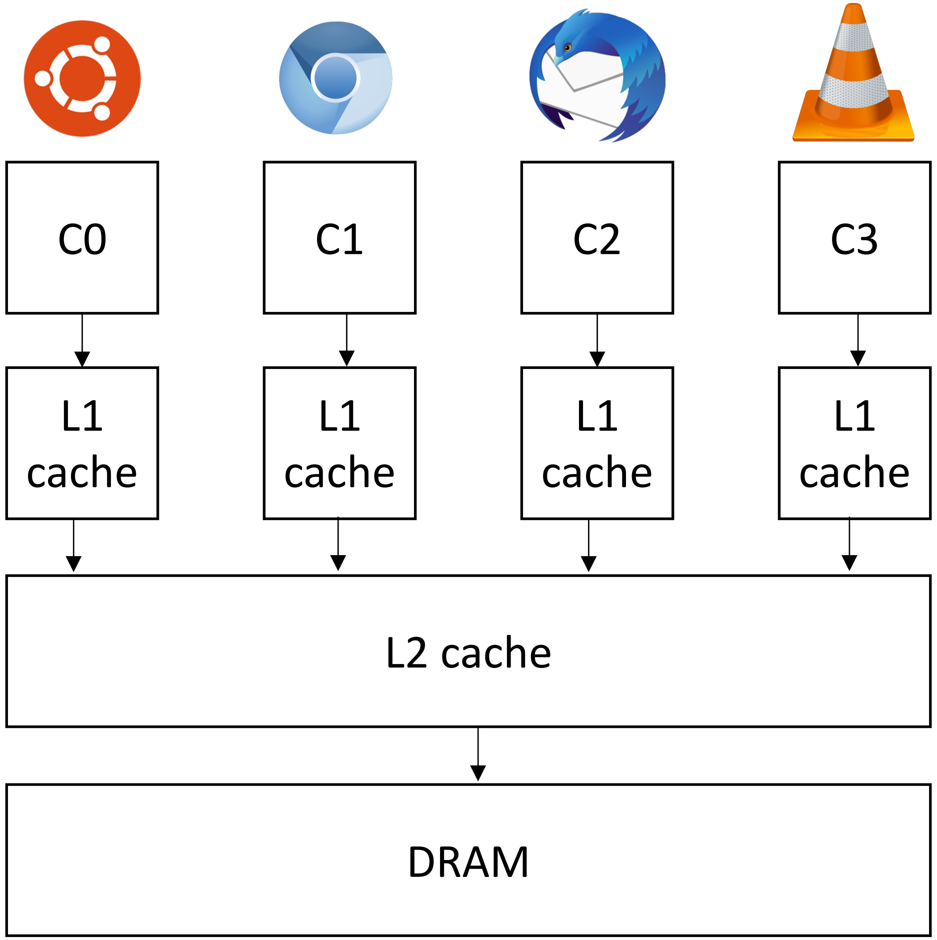
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines



Symmetric Multiprocessing (SMP)

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

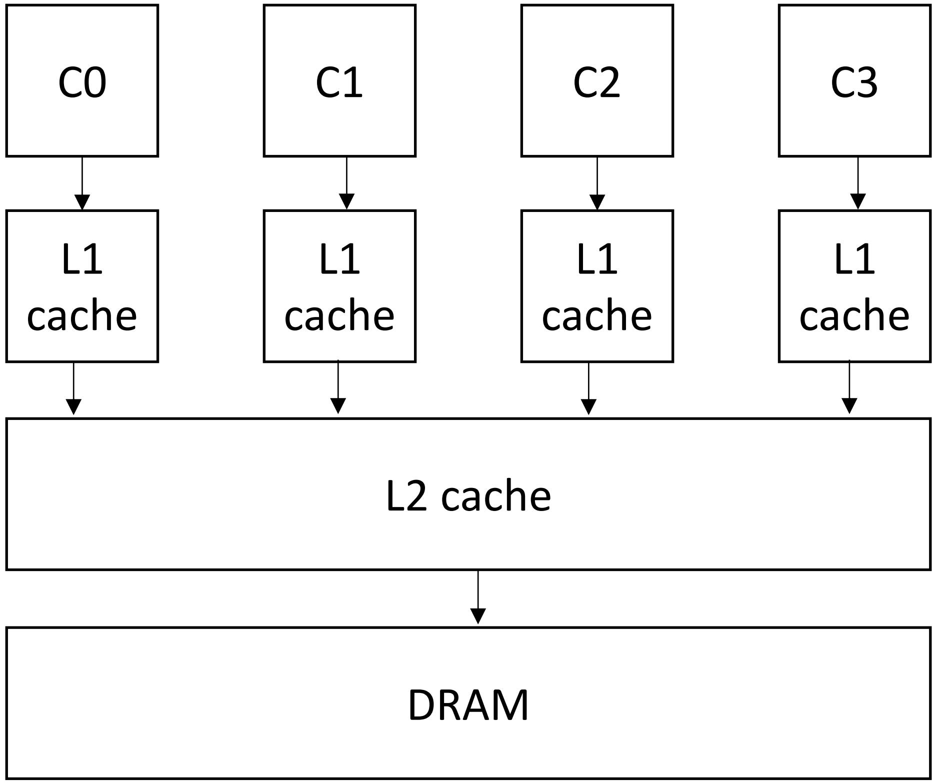
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines



Symmetric Multiprocessing (SMP)

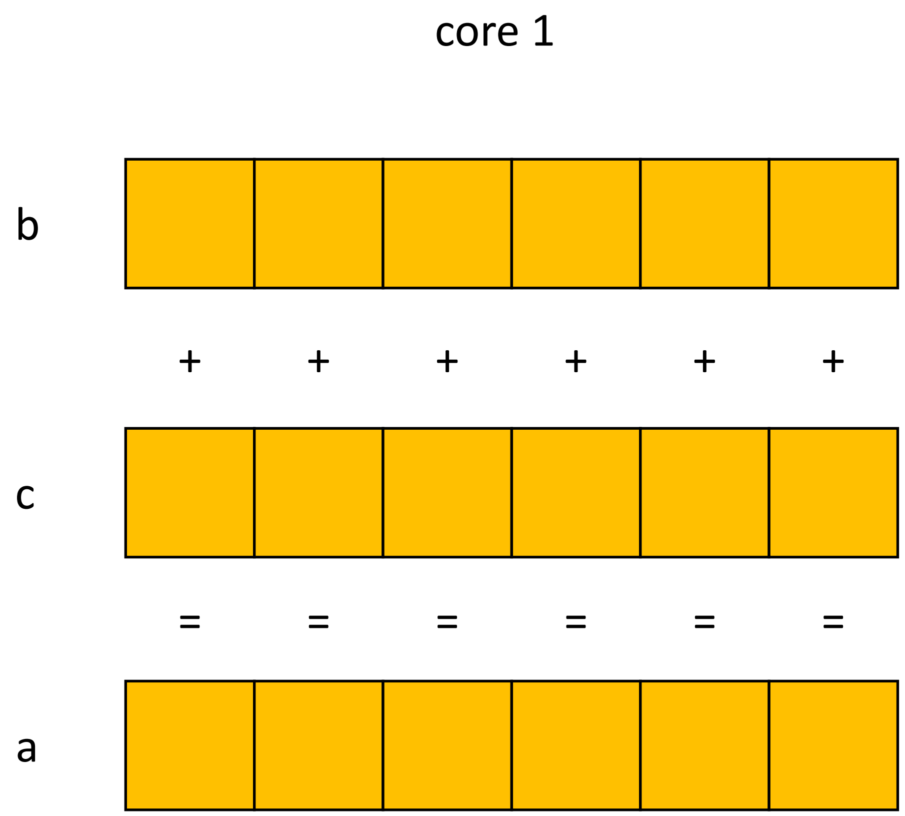
- Collection of "identical" cores
  - Shared memory (access to all system resources)
  - Managed by a single OS
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines
  - Can provide (close to) linear speedups for parallel applications

PyTorch



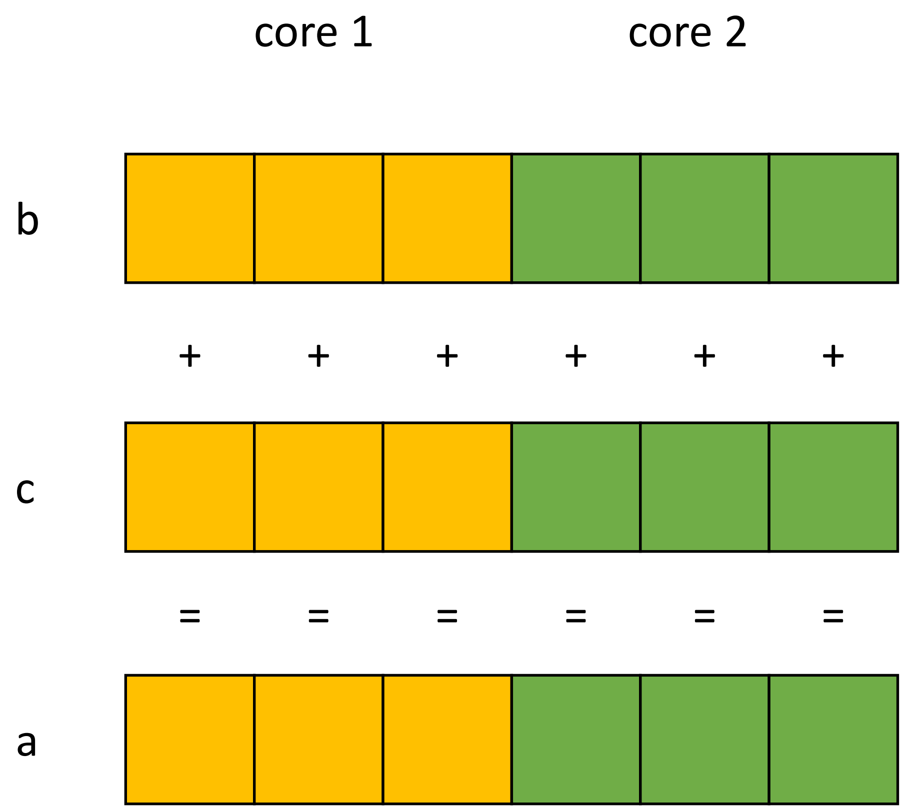
For loops are great candidates for SMP parallelism

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



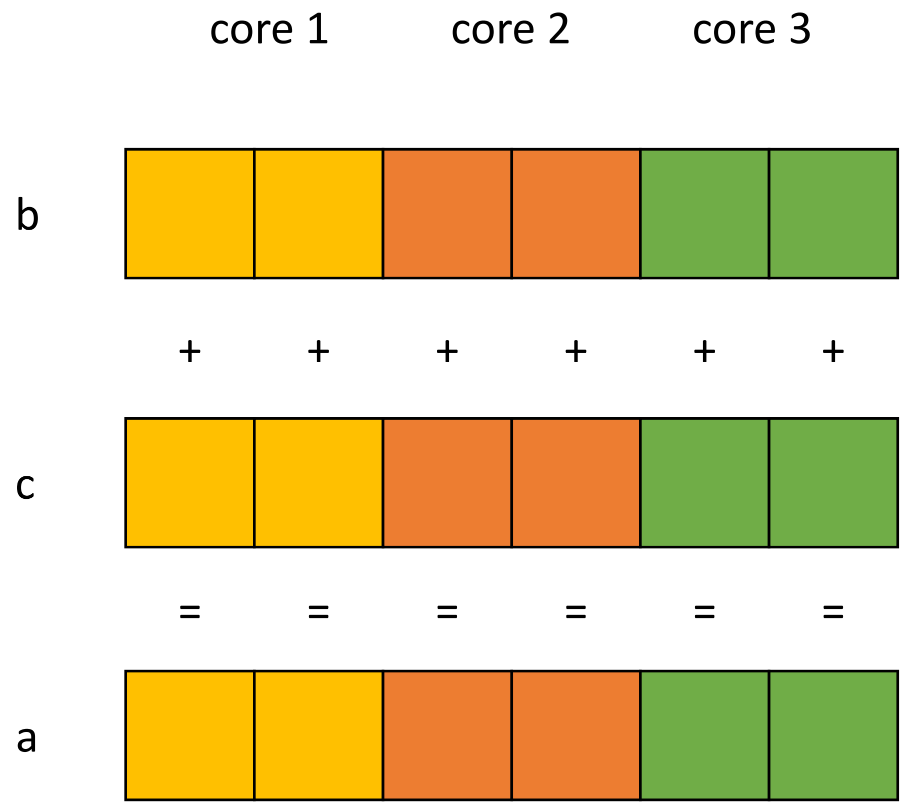
For loops are great candidates for SMP parallelism

```
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]
}
```

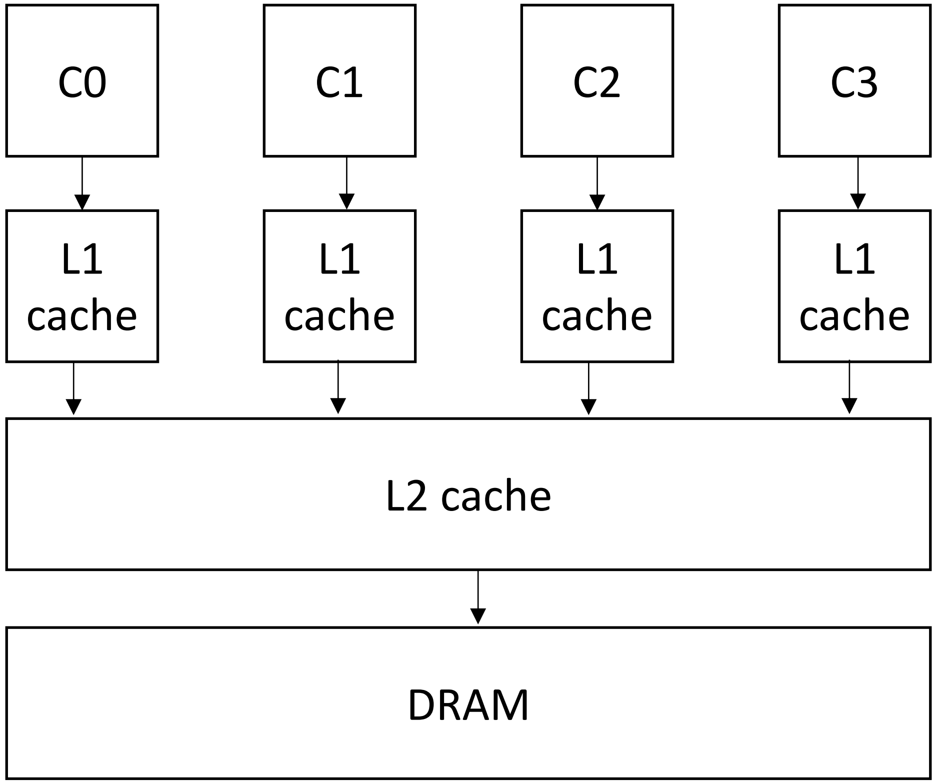


Demo

- Vector addition

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!



Demo

- Overhead

- Safety

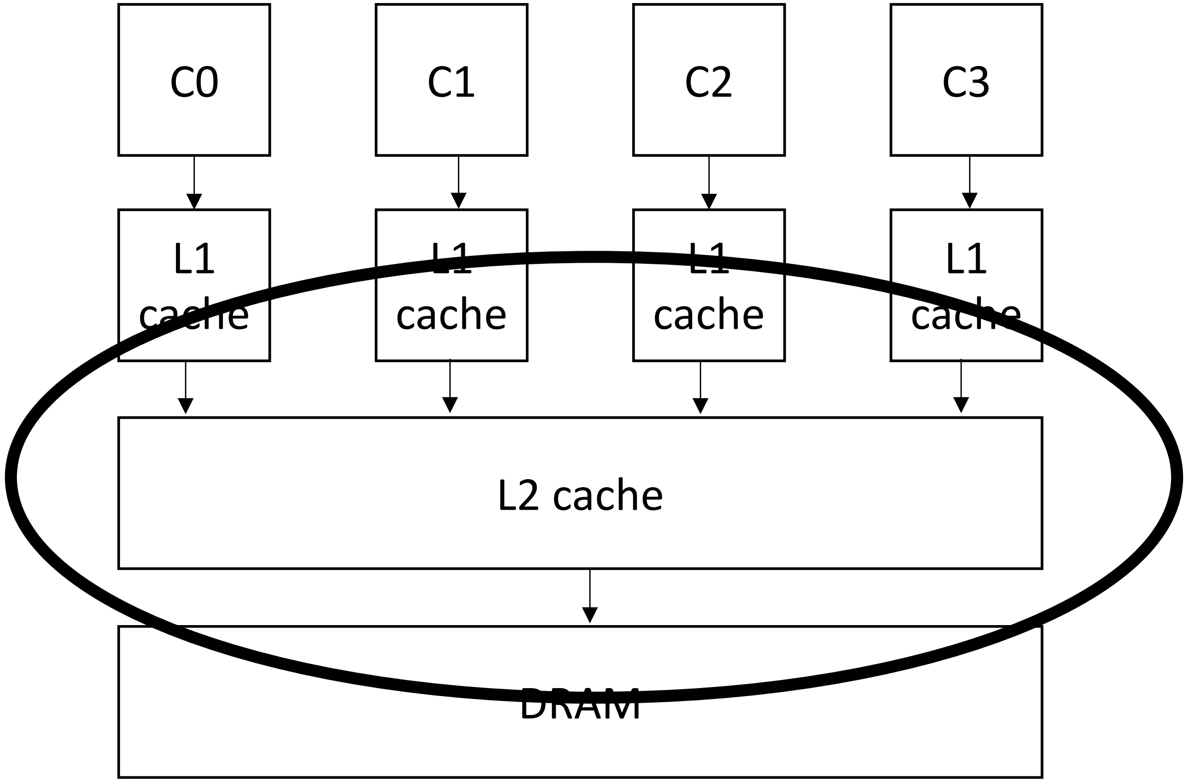
SMP systems are widespread

- Our server has 4 cores.
  - Most workstations have more; ~32 (up to 52 Intel Xeon)
  - New products: 128 core ARM system*
- My laptop: 8 cores (symmetric)
- Phones:
  - iPhone: 2 big cores, 4 small cores
  - Samsung: 2 + 4 + 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

- Our server has 4 cores.
  - Most workstations have more; ~32 (up to 52 Intel Xeon)
  - New products: 128 core ARM system*
- My laptop: 8 cores (symmetric)
- Phones:
  - iPhone: 2 big cores, 4 small cores
  - Samsung: 2 + 4 + 4



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

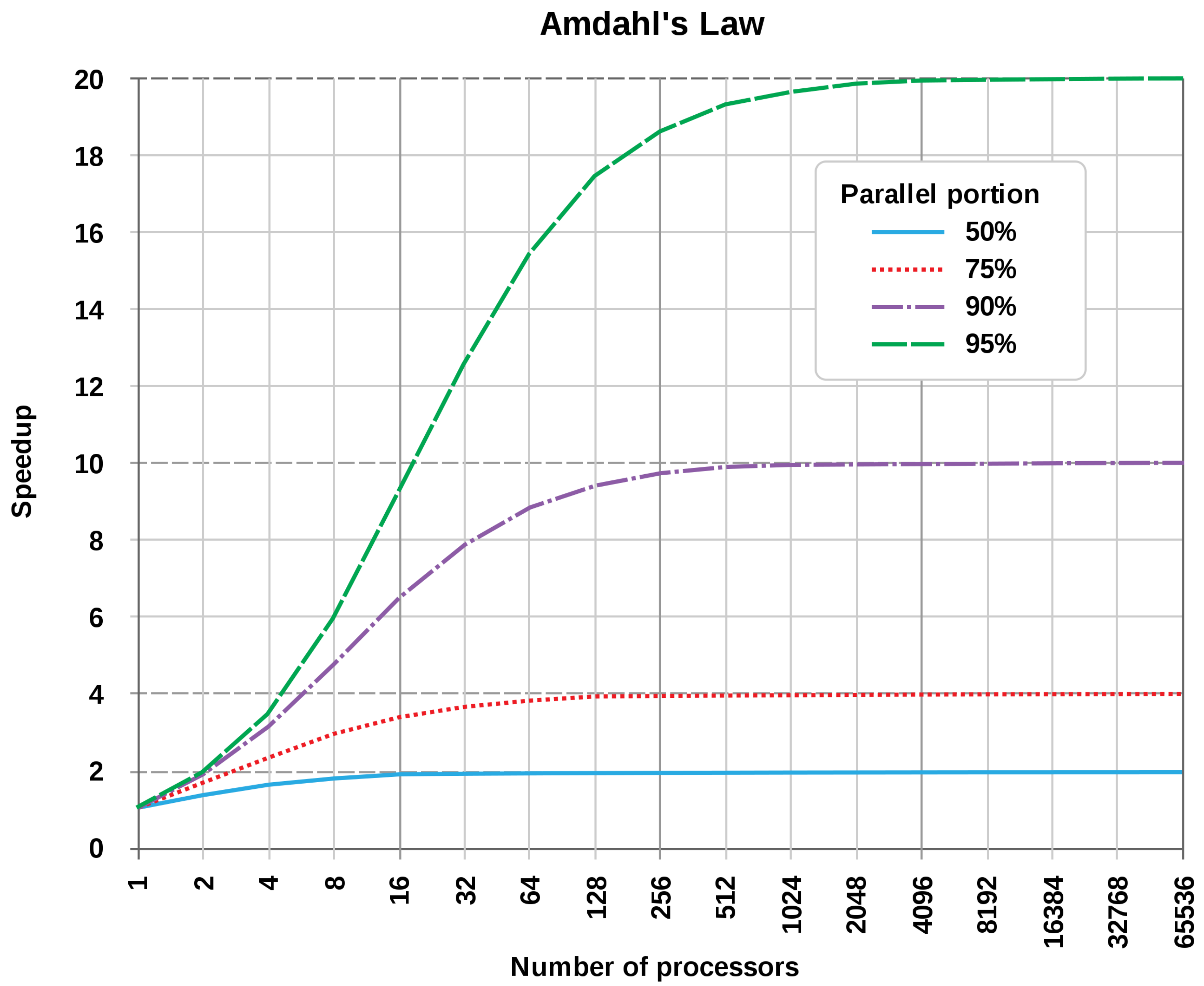
Potential for Parallel Speedup

- 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



Amdahl's Law
Speedup
20
18
16
14
12
10
8
6
4
2
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of processors
Parallel portion
50%
75%
90%
95%

Compiler applications

- 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

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, indexes are a function of loop variables, input variables and constants*
  - Loops Increment by 1

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, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

```
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

```
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

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

substitute:

```
i = 3*j + 2

double check
upperbound/lowe
```

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

```
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

```
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

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

index calculation based on the loop variable

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);
}
```

index calculation based on the loop variable
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

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

Write-write conflicts:

for two distinct iteration variables:

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

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

Write-write conflicts:

for two distinct iteration variables:
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*

```
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

SMT Solver

Jean Yang
@jeanqasaur

THE Z3 SMT SOLVER AND SMT SOLVERS IN GENERAL

It is the assembly of automated reasoning. 🧰

Hillel @hillelogram · 3h
What's your favorite software tool/topic/whatever that

1. Most people don't know,
2. Most people would benefit from knowing, and
3. Can be learned in an afternoon or two?

5:16 PM · Nov 11, 2020 · TweetDeck

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+64]**2;
}
```

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

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: $i0_x$!= $i0_y$

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

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: $i0_x == i0_y$ $i1_x != i1_y$

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(i0x, i1x, ... iNy)`

4. Do the same for write-read conflicts

Next week

- Reordering loop nestings
- irregular parallelism