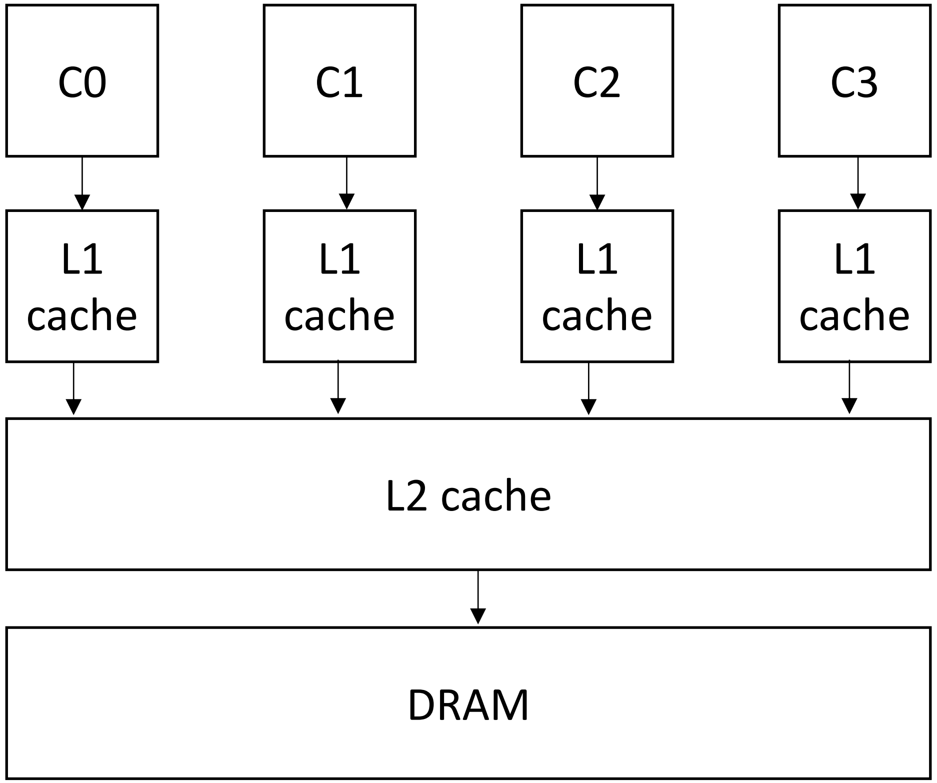
CSE211: Compiler Design

Nov. 1, 2022

- **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

- Back to in person!

Announcements

- Homework 2 is out
  - due Nov. 2 (tomorrow)
  - office hours tomorrow at normal time

Announcements

- Midterm
  - due Friday (Nov. 4) by midnight
  - Rules:
    - Open book, open internet, open notes.
    - Do not discuss the test with any other student while it is out.
    - Do not google (or otherwise search) for exact questions. It is fine to search for concepts.
    - Do not post questions to others on the internet (e.g. through discord or reddit)
    - Any question should be asked as a private post on Piazza. If it’s a clarification that needs to be made to the whole class, I will do it in a public Piazza thread

Announcements

- Midterm
  - Designed to take about 2 hours (not including studying)
  - Students report taking longer because they study while taking the test.
  - Students also report taking longer because they double check their answers and make the test nicely formatted.
  - Please look over the test as soon as it is released so that you roughly know how long it will take you.

  - LATE MIDTERMS WILL NOT BE ACCEPTED

Announcements

- Mark your attendance for today after you watch the recording (or if you are attending live)
  - Please try to keep on top of this.
  - We have more attendance put in, please let us know within 1 week if there are any issues

Review ILP

Finding dependencies in the compiler

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

```
x = z + w;
a = b + c;
```

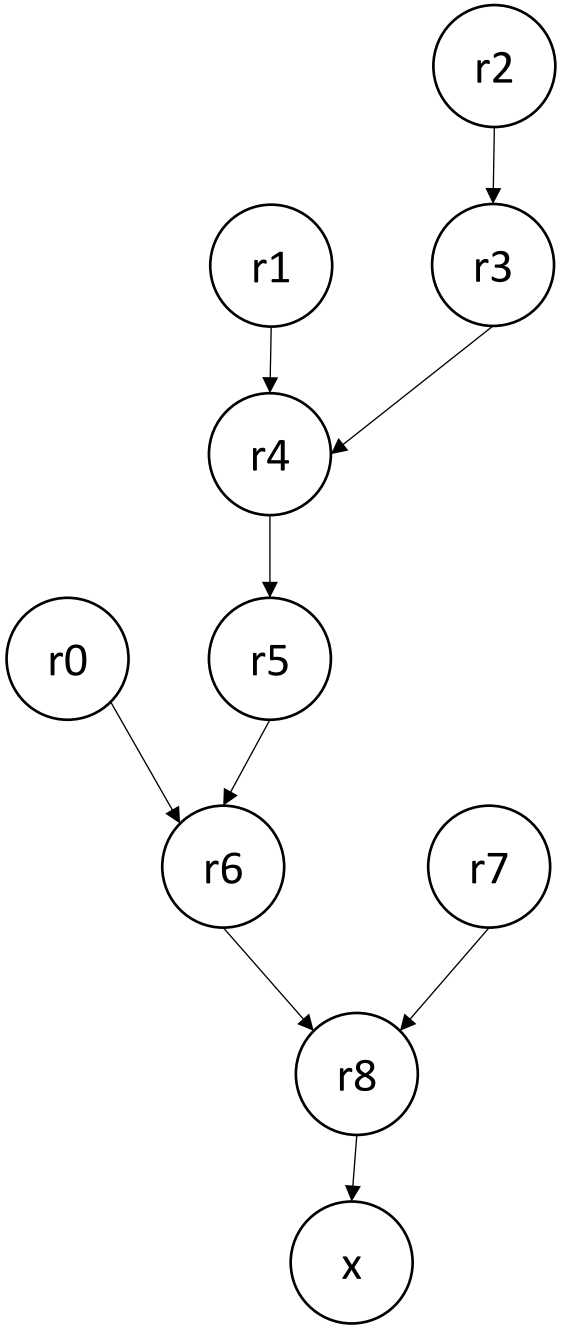
Two instructions are independent if the operand registers are disjoint from the result registers

instructions that are not independent cannot be executed in parallel

```
x = z + w;
a = b + x;
```

Data Dependencies

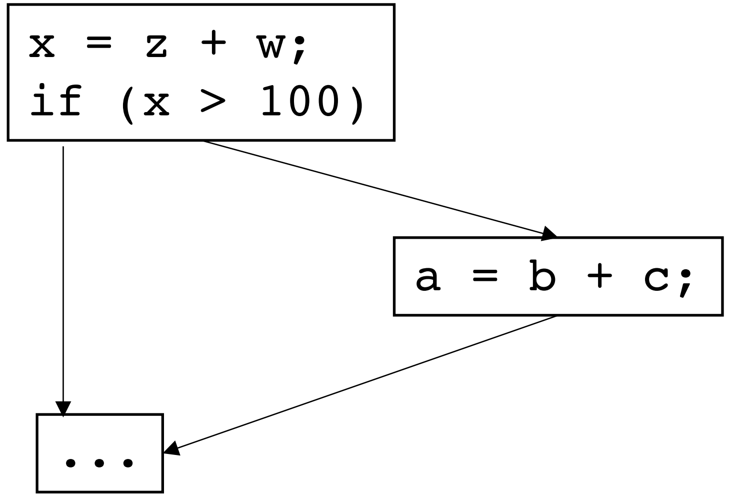
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



Control dependencies

```
x = z + w;
if (x > 100)
   a = b + c;
```

Instructions in different CFG nodes have control-dependencies



Memory dependencies

True dependence:
Read-after-write

```
a[i] = z + w;
x = a[i]
```

Output dependence:
Write-after-write

```
a[i] = z + w;
a[i] = a + b;
```

Dependencies can be removed

```
reg_a_i = z + w;
a[i] = a + b;
```

anti-dependence:
Write-after-read

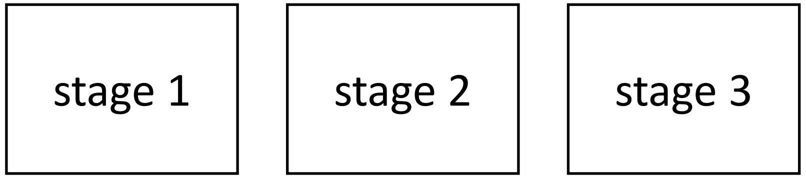
```
x = a[i]
a[i] = z + w;
```

Dependencies can be delayed

```
x = a[i]
reg_a_i = z + w;
...
a[i] = reg_a_i;
```

How can hardware execute ILP?

- Pipeline parallelism



- Abstract mental model for compiler:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline

```
instr1;
instrX0;
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

How can hardware execute ILP?

- Executing multiple instructions at once:

- Superscalar architecture:
  - Several sequential operations are issued in parallel
  - hardware detects dependencies

issue-width is maximum number of instructions that can be issued in parallel

```
instr0;
instr1;
instr2;
```

if instr0 and instr1 are independent, they will be issued in parallel

What does this look like in the real world?

- Intel Haswell (2013):
  - Issue width of 4
  - 14-19 stage pipeline
  - OoO execution
- Intel Nehalem (2008)
  - 20-24 stage pipeline
  - Issue width of 2-4
  - OoO execution
- ARM
  - V7 has 3 stage pipeline; Cortex V8 has 13
  - Cortex V8 has issue width of 2
  - OoO execution
- RISC-V
  - Ariane and Rocket are In-Order
  - 3-6 stage pipelines
  - some super scaler implementations (BOOM)

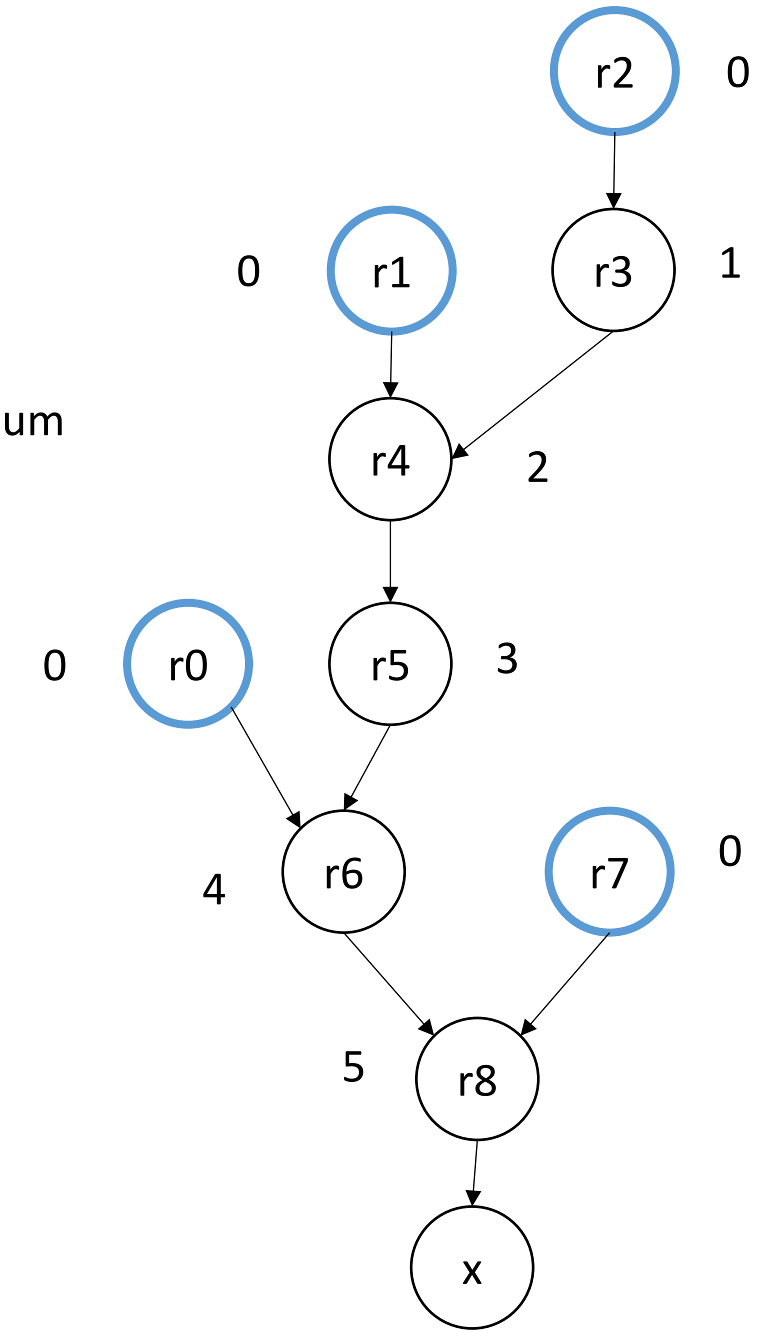
Other examples?

Priority Topological Ordering of DDGs for Superscalar

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

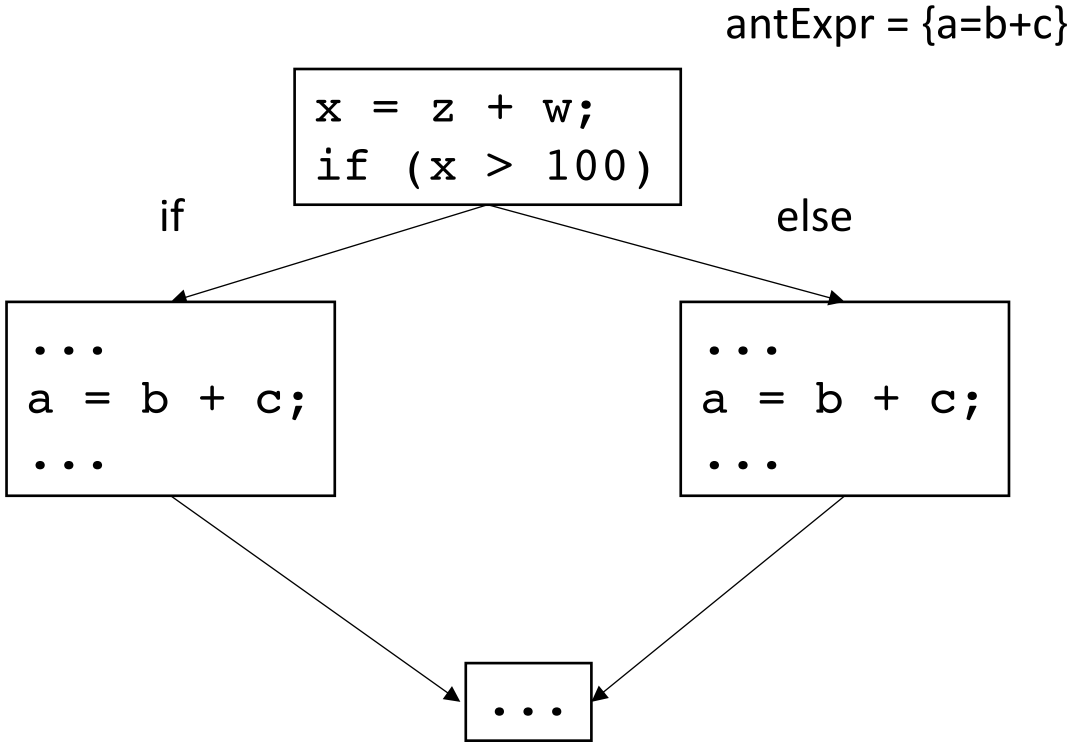
Label nodes with the maximum distance to a source

Break ties in topological order using this number



Anticipable Expressions

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

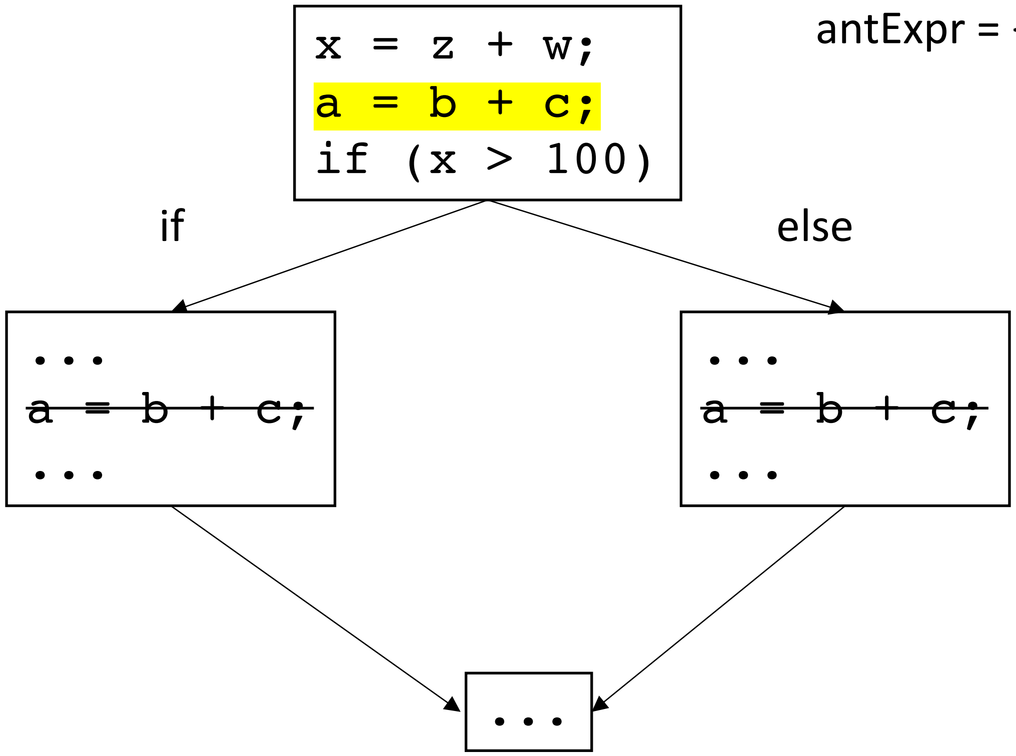


Anticipable Expressions

also called ”Upward code motion”

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

antExpr = {a=b+c}



Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
    SEQ(i,1);
    SEQ(i,2);
    ...
    SEQ(i,N); // end iteration for i
    SEQ(i+1,1);
    SEQ(i+1,2);
    ...
    SEQ(i+1, N); // end iteration for i + 1
}
```

Let `SEQ(i,j)` be the jth instruction of `SEQ(i)`.

Let each instruction chain have N instructions

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
      SEQ(i,1);
      SEQ(i+1,1);
      SEQ(i,2);
      SEQ(i+1,2);
      ...
      SEQ(i,N);
      SEQ(i+1, N);
}
```

They can be interleaved

Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE; i++) {
    a[0] = REDUCE(a[0], a[i]);
}
```

If the reduction operator is associative, we can do better!

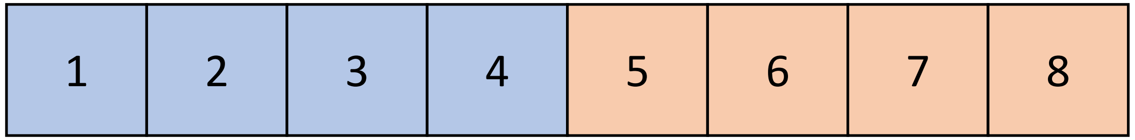
Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:

1	2	3	4	5	6	7	8

Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Loop Unrolling for Reduction Loops

- Simple implementation:

```
for (int i = 1; i < SIZE/2; i++) {
    a[0] = REDUCE(a[0], a[i]);
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2)+i]);
}

a[0] = REDUCE(a[0], a[SIZE/2])
```

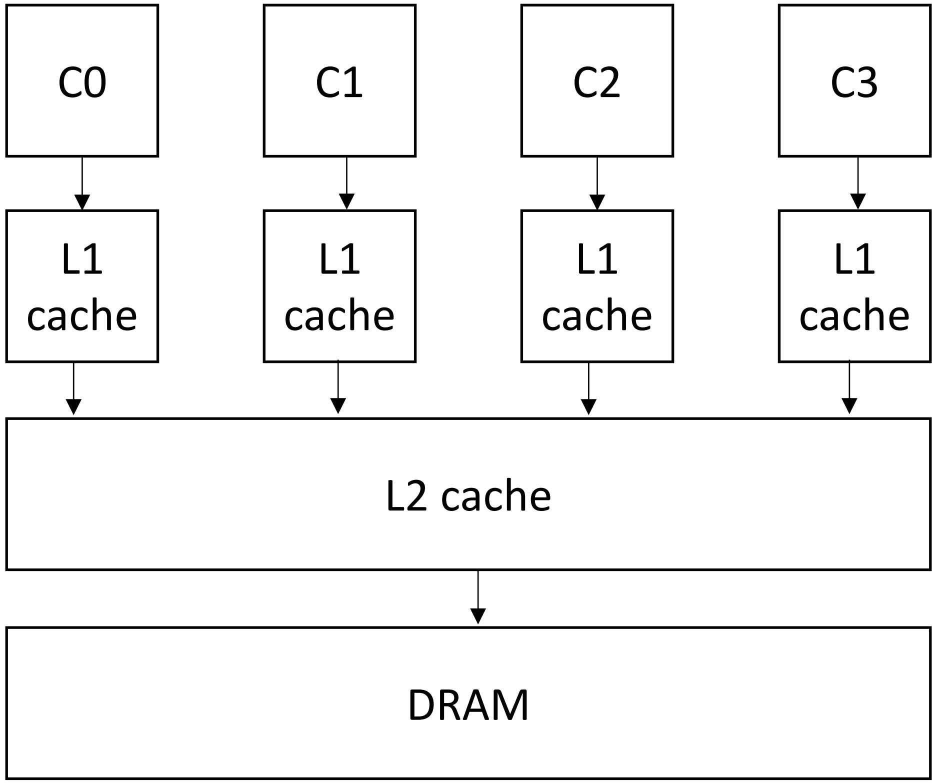
independent instructions can be done in parallel!

CSE211: Compiler Design

Nov. 1, 2022

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

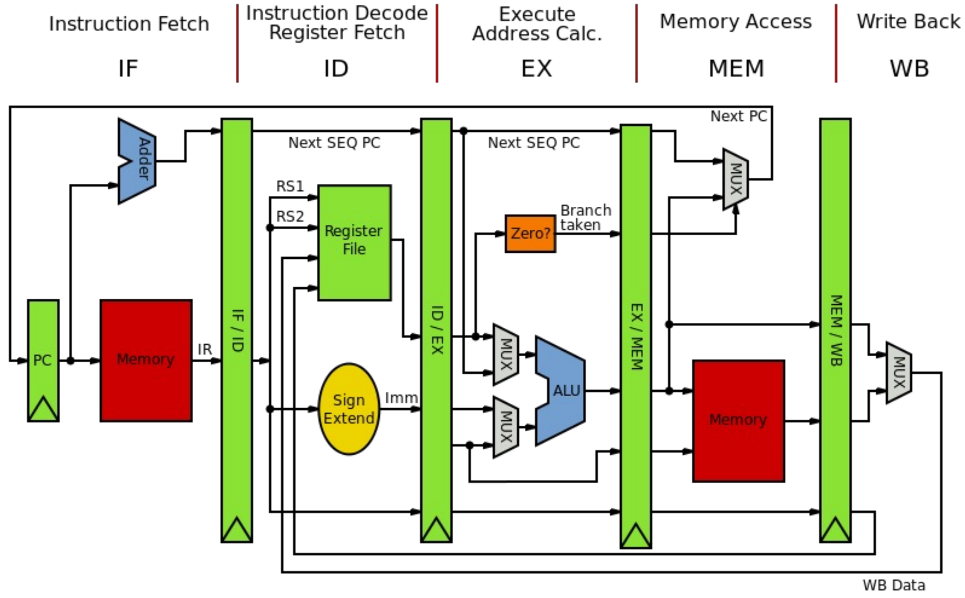
- **Discussion questions**:
  - What parallel frameworks have you used?
  - Do you achieve linear speedup?
  - When is it safe to parallelize for loops?



Limits of ILP?

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

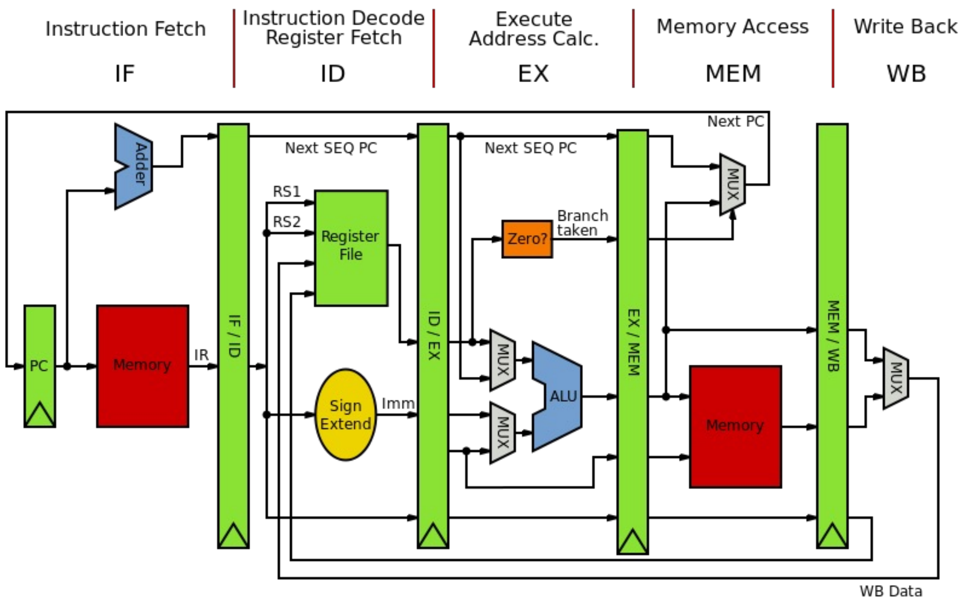
- Superscalar width?
  - Hardware checking becomes prohibitive:



Limits of ILP

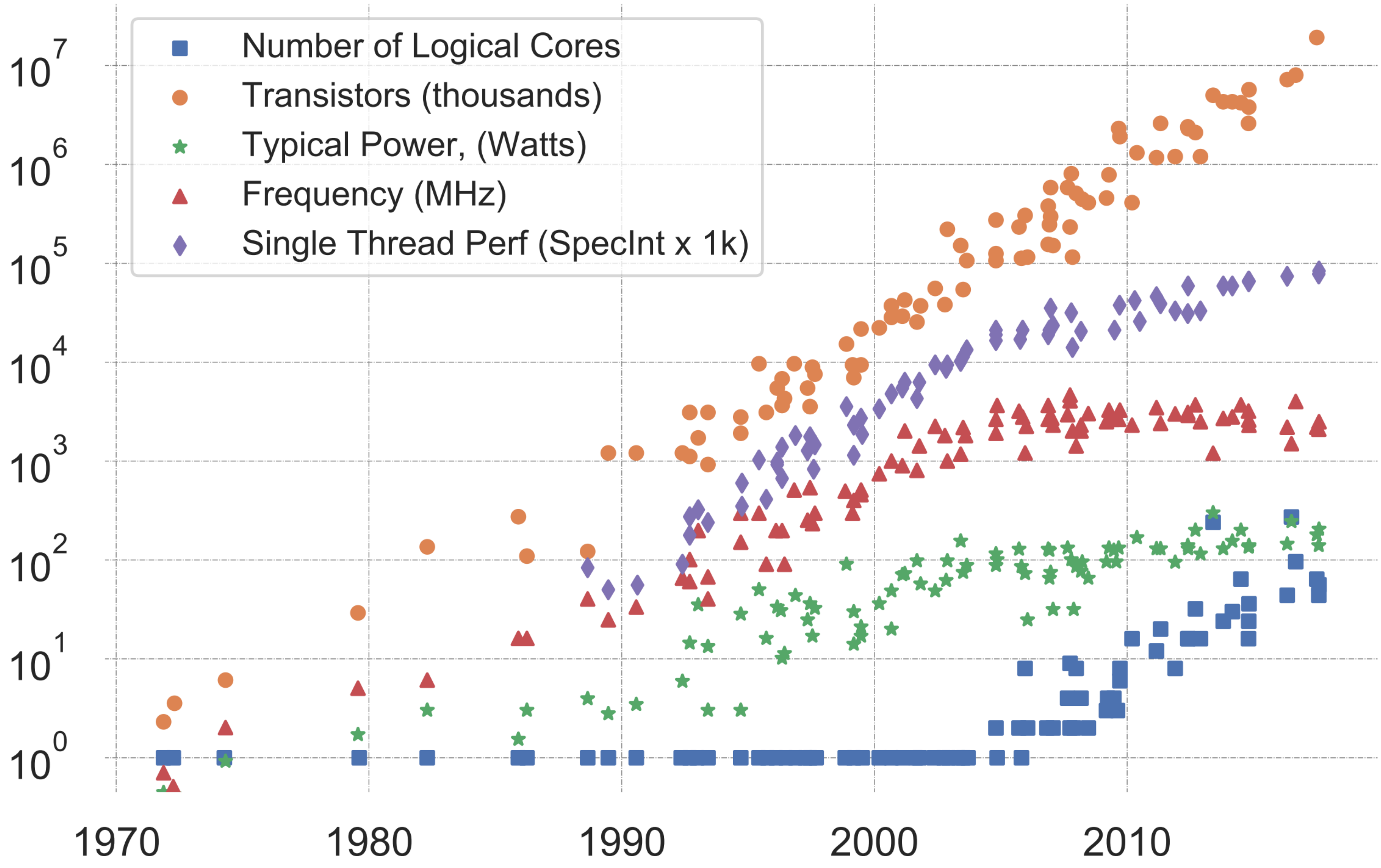
- Pipelines?
  - Only so much meaningful work to do per-stage.
  - Stage timing imbalance
  - Staging overhead

- 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

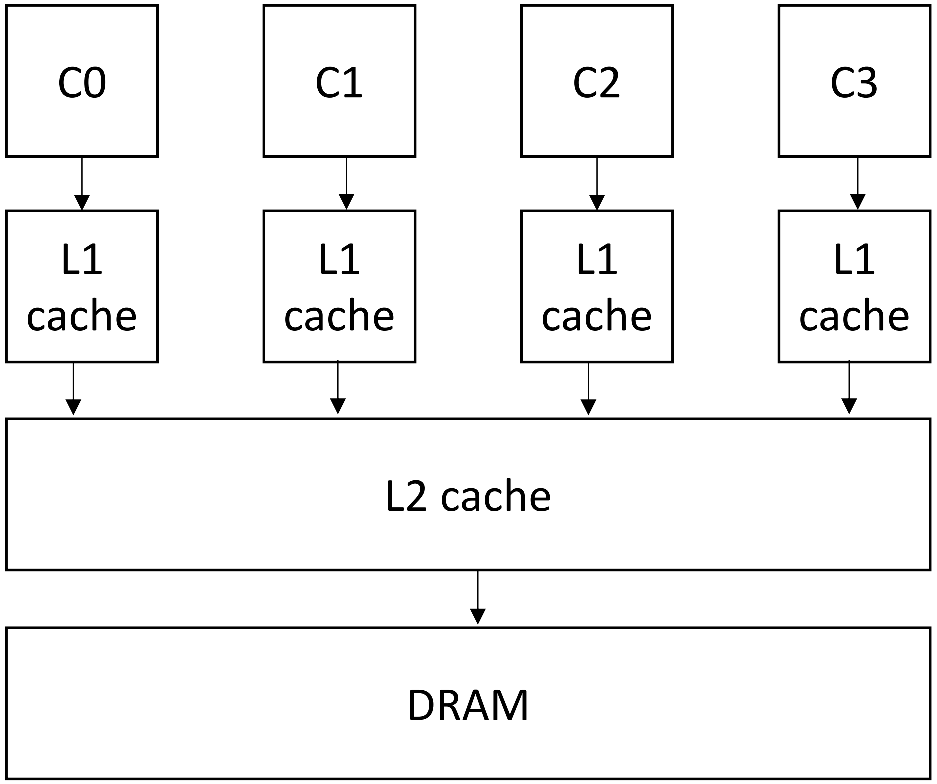
- Number of transistors: **Moore's law**
  - On its last legs.
  - Intel delaying 7nm chips. Apple has a 5nm. Some roadmaps project up to 3nm

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

Symmetric Multiprocessing (SMP)

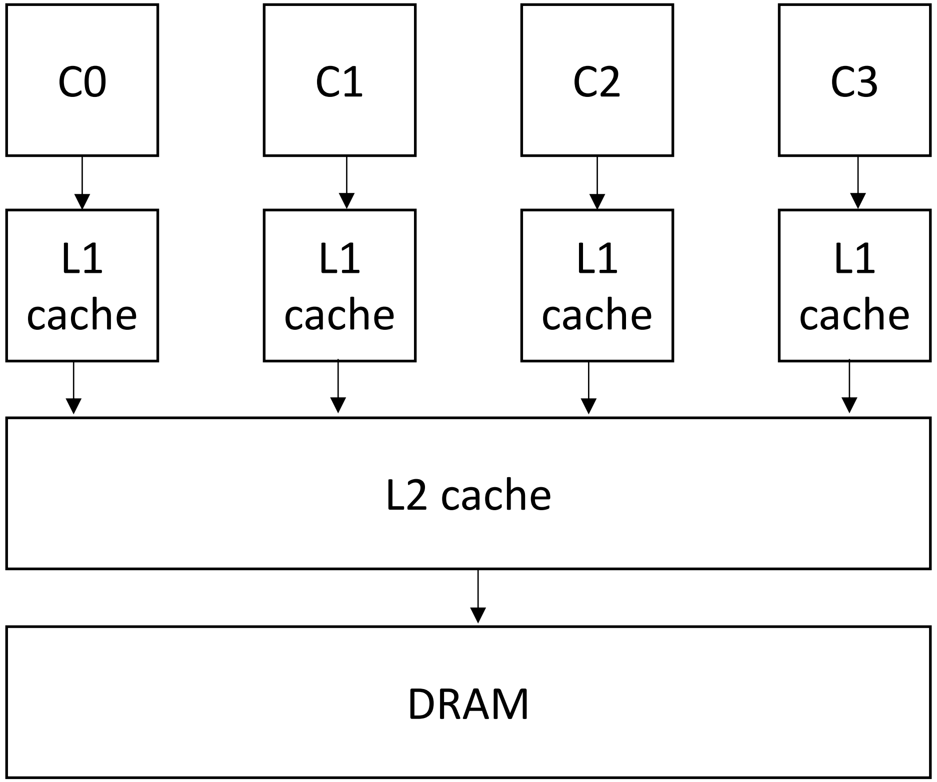
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS

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



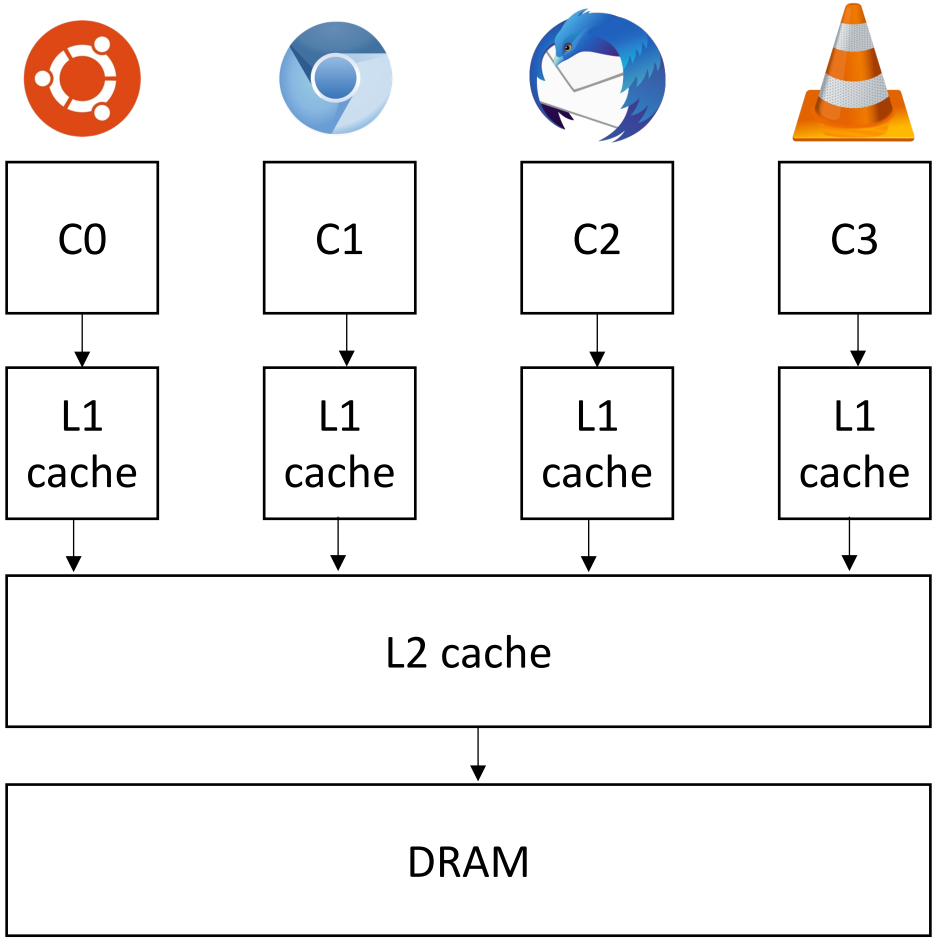
Symmetric Multiprocessing (SMP)

- 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)

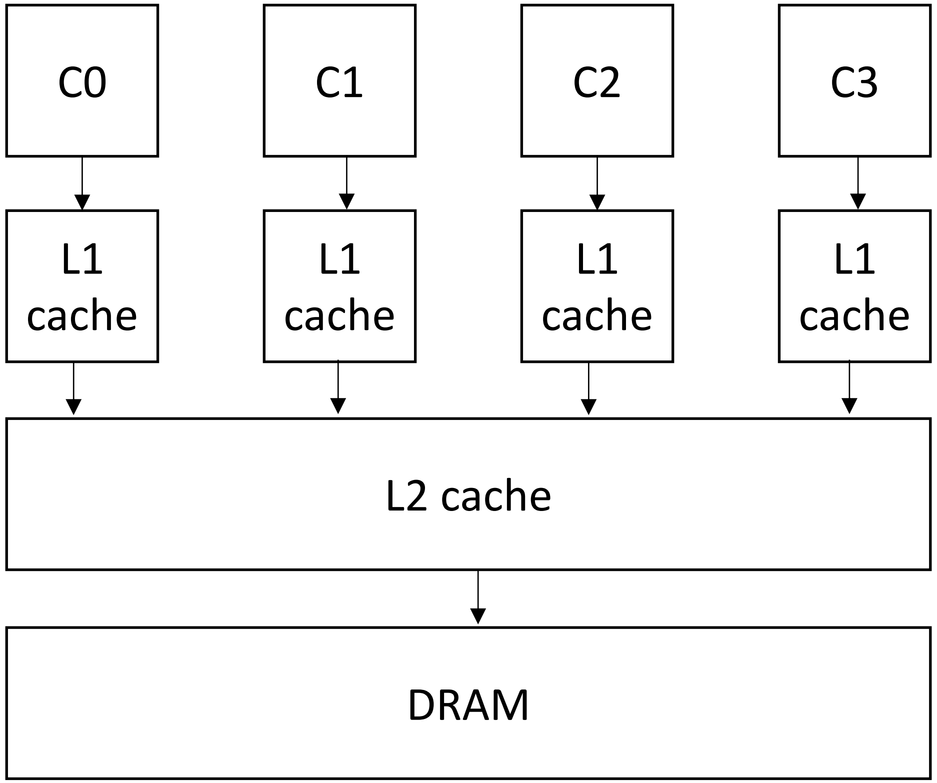
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines



Symmetric Multiprocessing (SMP)

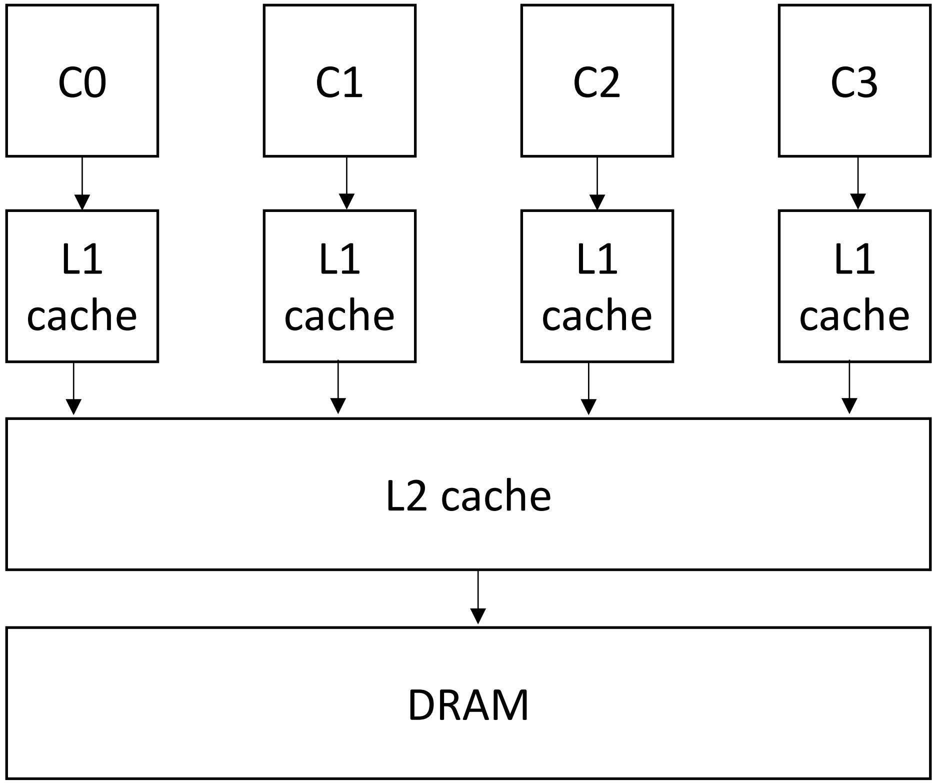
- Collection of “identical” cores
  - Shared memory (access to all system resources)
  - Managed by a single OS
- Pros:
  - Simple(r) HW design
  - Great for multitasking machines
  - Can provide (close to) linear speedups for parallel applications

PyTorch



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

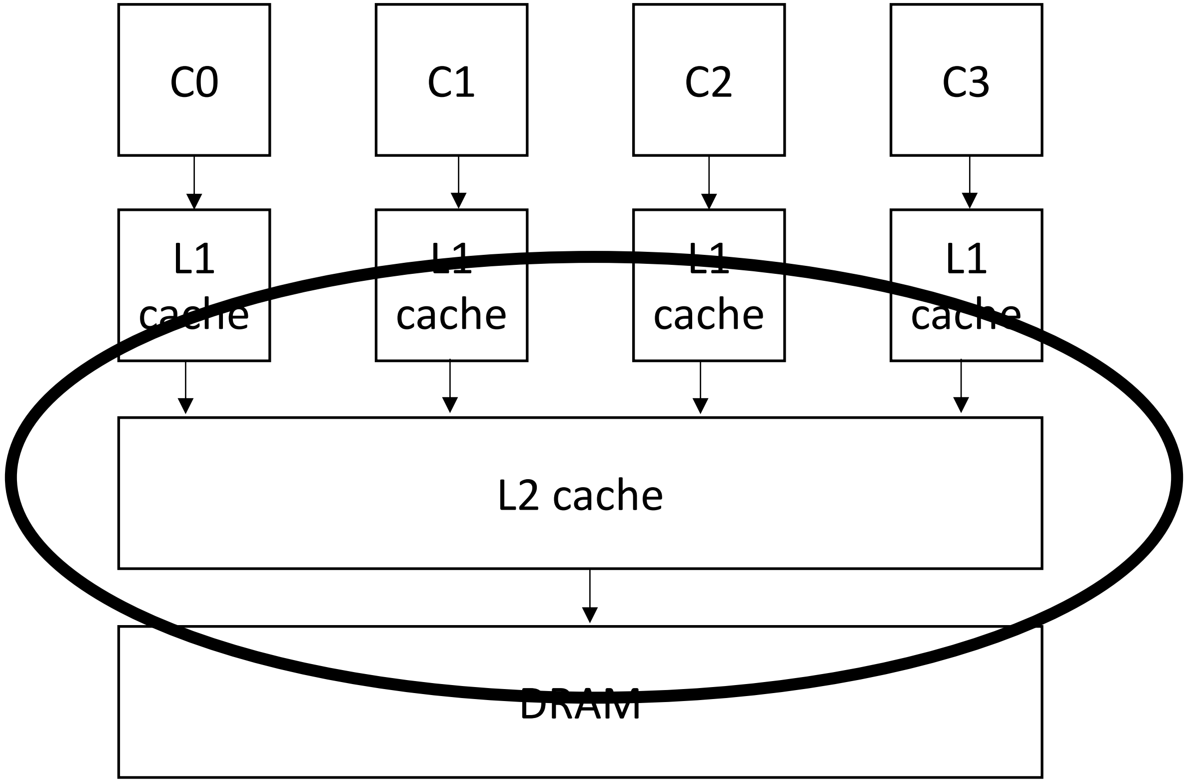
*https://www.crn.com/news/components-peripherals/ampere-s-new-128-core-altra-cpu-targets-intel-amd-in-the-cloud

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

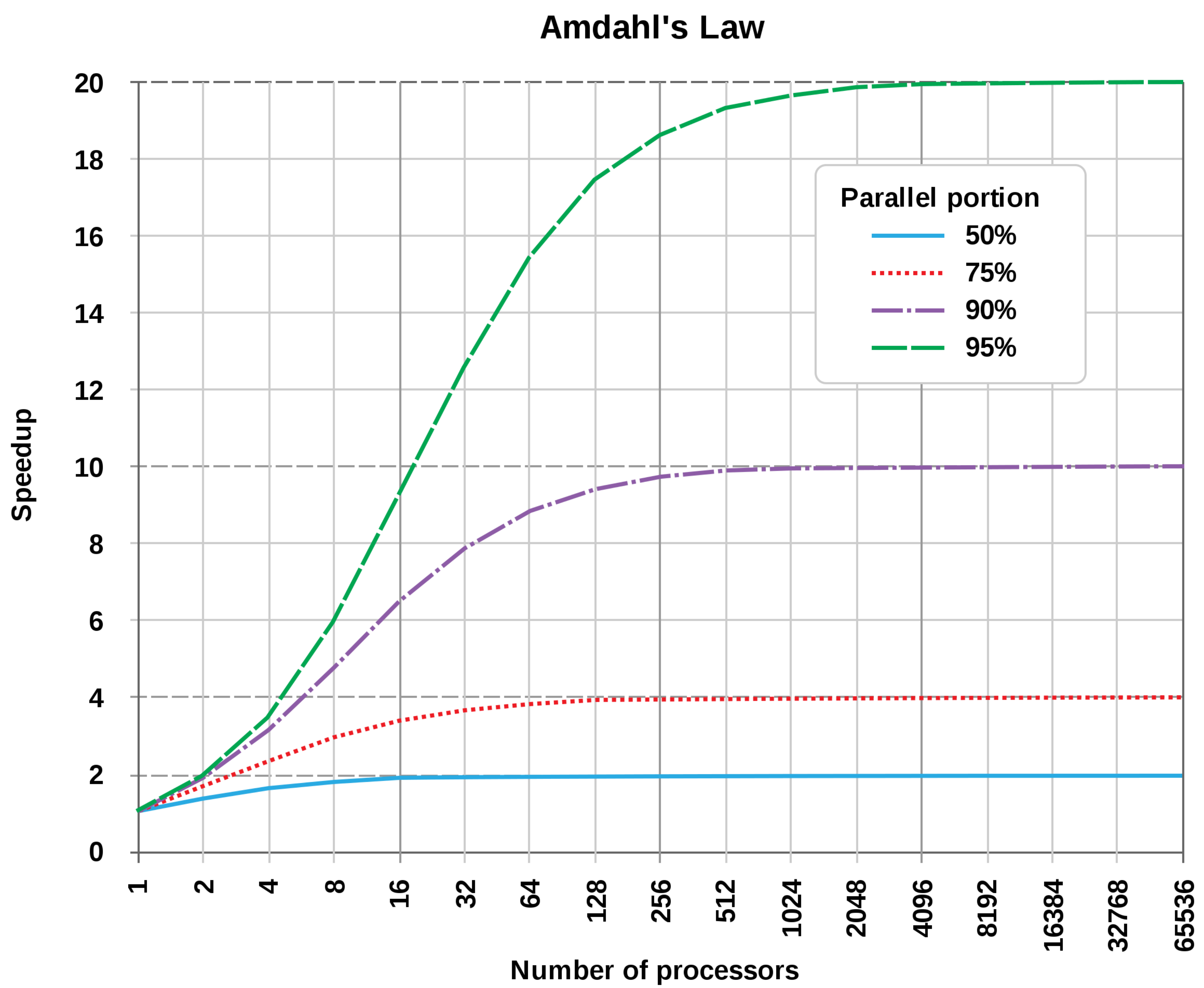
Potential for Parallel Speedup

- 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?

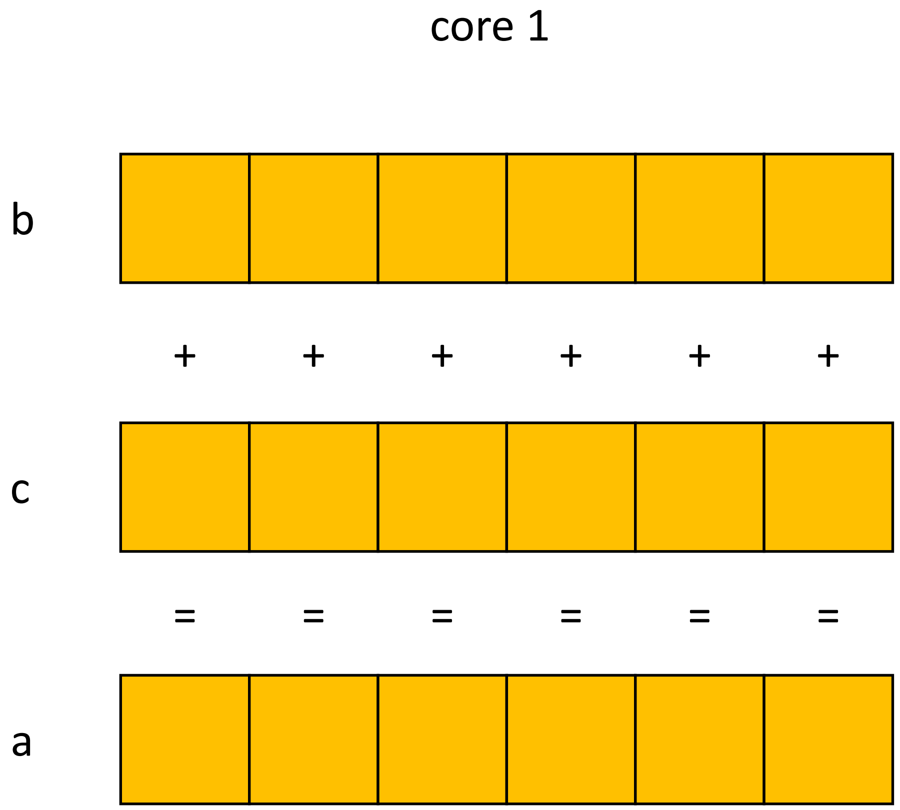
- Much like ILP: convert sequential streams of computation in to SMP parallel code.

- Much harder constraints
  - Correctness
  - Performance

- For loops are a good target for compiler analysis

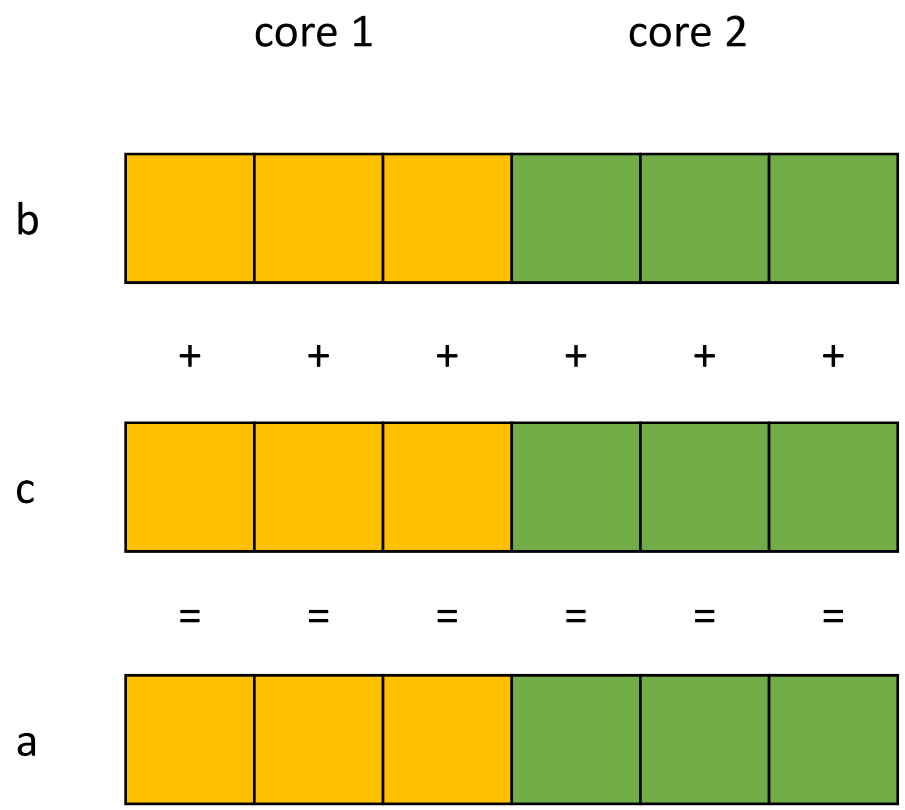
For loops are great candidates for SMP parallelism

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



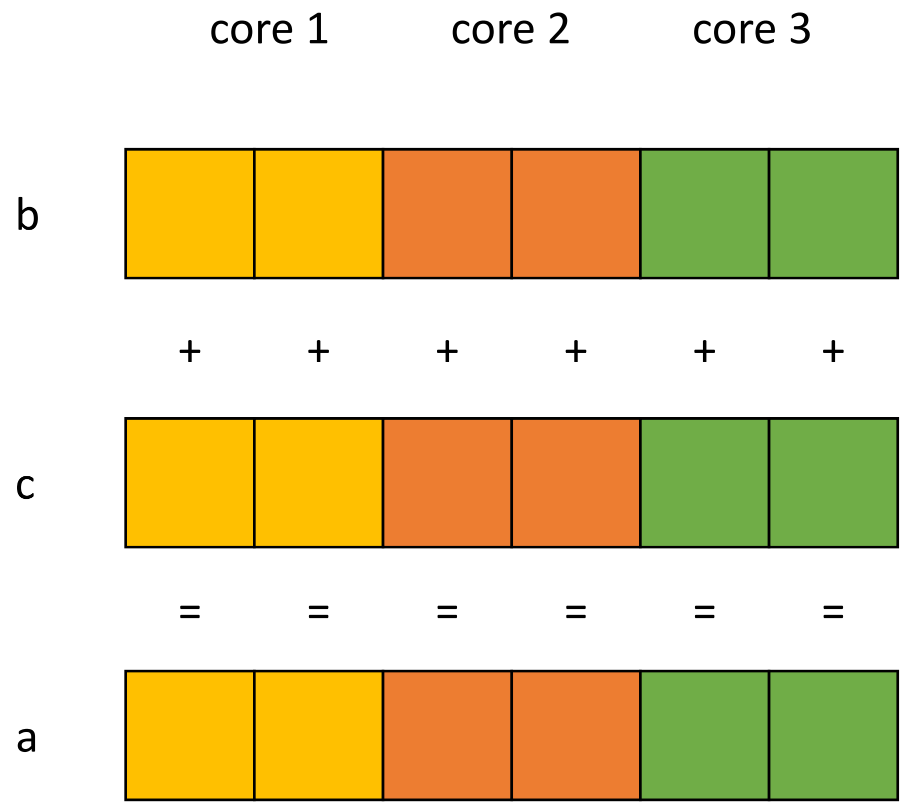
For loops are great candidates for SMP parallelism

```
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:
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+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

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

what about 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

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: $i0_x \ne i0_y$

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));
        }
    }
}
```

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

What if we want to parallelize an inner loop?

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

Next class

- Topics:
  - Restructuring loops

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