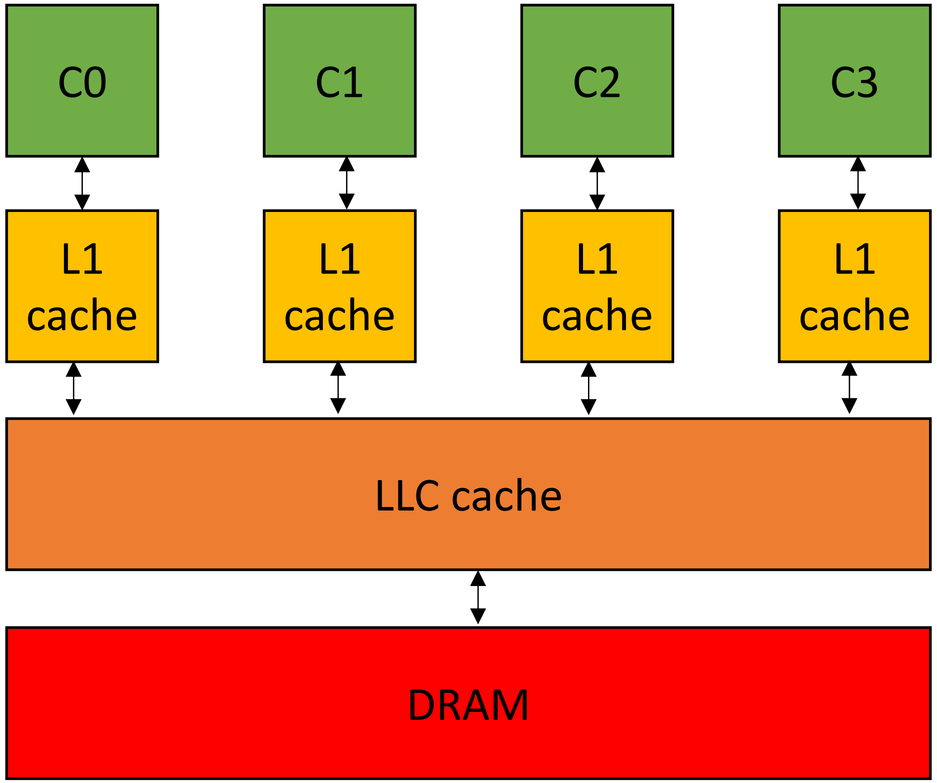
CSE113: Parallel Programming

Jan. 5, 2022

- **Topic**: Architecture and Compiler Overview
  - Programming Language to ISA compilation
  - 3-address code
  - multiprocessors
  - memory hierarchy



Announcements

- New room!
  - Previous room only sat 50 students
  - Moving to Kresge 327 (seats 90)
  - Let everyone on the wait list in

Asynchronous Forums

- Piazza is setup, Reese sent an announcement with a link
  - We will moderate and try to answer questions within 24 hours

- Unofficial discord:
  - we’re trusting you to moderate
  - be nice
  - don’t cheat

Office hours

- **Reese:**
  - Wednesday from 2:30 - 4:30 PM
  - Hybrid (remote or in person)
- **Sanya:**
  - Monday from 4:00 - 5:00 PM
  - Friday from 3:30 - 4:30 PM
  - ***Asynchronous until Jan. 10!***
  - primarily in person
  - room TBD
- **Tim**:
  - Tuesdays from 2:00 - 3:00 PM
  - Thursdays from 2:00 - 3:00 PM
  - primarily remote
- **Tyler:**
  - Thursday from 3:00 - 5:00 PM
  - Hybrid (remote or in person)
  - Room E2 233

Docker setup up

Instructions here:

https://sorensenucsc.github.io/CSE113-wi2022/homework-setup.html

Docker	27 respondents	43 %

Homeworks

- Homework 1 will posted on Friday
- Due in 2 weeks (Jan 21)
- It must run in the docker and adhere to the directory structure outlined in the assignment. We will provide a script to help you verify this.

Final TODOs on our end

- Find rooms for in-person office hours
- Update website with the new information
- Get a list of department resources for additional machines

Today’s Quiz

- Normally we’ll do quizzes at the beginning of class
- Remember, they are not graded, but please actually do your best
- With these classes being asynchronous, we’ll release the quiz after class (2:30 PM) and have it due at midnight tomorrow

Previous quiz

Some answers

Which of the following programming languages/frameworks do you have experience with?

Python	56 respondents	89 %
C	60 respondents	95 %
C++	51 respondents	81 %
Javascript	34 respondents	54 %
GPU Programming	5 respondents	8 %
Docker	27 respondents	43 %
Unix command line	59 respondents	94 %
console text editor (e.g. vim, emacs)	46 respondents	73 %

0% answered correctly

Some answers

- Previous experience with concurrency:
  - Some in CSE 130
  - *You're going to love C++ threads*
  - we're going to discuss implementations of primitives, e.g. locks
- No experience is fine too!
  - Could be helpful for when you take CSE 130

Some answers

- People are excited to learn about:
  - GPUs!
  - We can discuss how concepts we learn apply to GPUs
  - Module 5 will be about GPUs and we have a cool assignment planned!

- People are interested in:
  - machine learning
  - graphics/games
  - algorithms and problem solving

Some examples

Self driving cars:

- Requires a reaction speed of 1.6s
- How to make faster?
  - Algorithms
  - interconnects
  - **cores**

Nvidia's embedded device has increased from 256, to 384 to 512 cores

source: IEEESpectrum

Some examples

Just because something is parallel doesn’t mean it will go fast!

Some examples

Just because something is parallel doesn't mean it will go fast!

pretty straight
forward computation
for brightening

(do every pixel in parallel)

image processing example

from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/

Some examples

Just because something is parallel doesn't mean it will go fast!

This computation is known as the "Local Laplacian Filter". Requires visiting all pixels 99 times

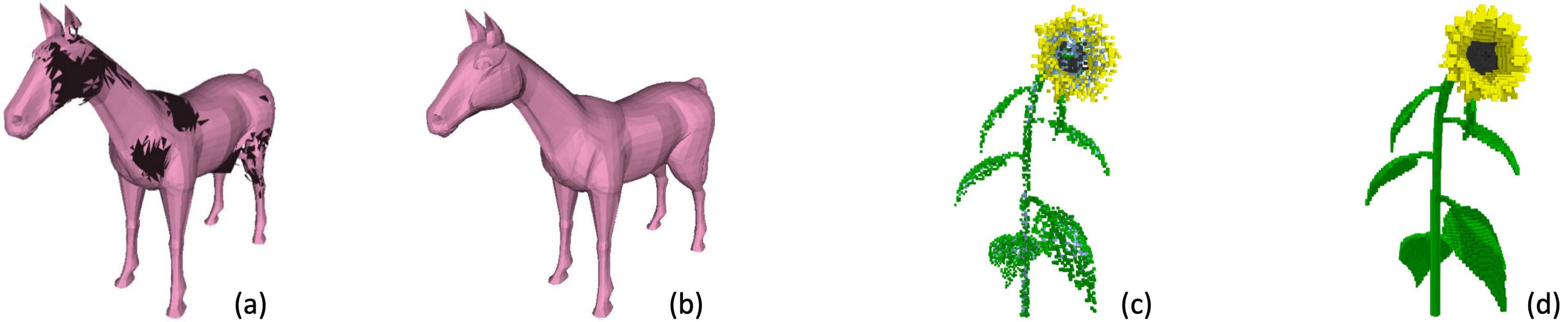
We want to be able to do this efficiently!

Recent work showed a 1.7x speedup vs. versions at Adobe

from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/

Some examples

But we need to be careful! Parallel programming is full of tricky corner cases!



Moving on to the lecture

Lecture Schedule

- Overview - why do we need a lecture on compilation and architecture?
- Compilation - How do we translate a program from a human-accessible language to a language that the processor understands
- Architecture - How do processors execute programs?
- Example

Lecture Schedule

- **Overview** - why do we need a lecture on compilation and architecture?

- Compilation - How do we translate a program from a human-accessible language to a language that the processor understands

- Architecture - How do processors execute programs?

- Example

In a perfect world...

- Programming languages provide an abstraction

Programmer: Writes Code

Hardware Designer: Makes Chips

In a perfect world…

- Programming languages provide an abstraction

Separation of concerns allows incredible progress

Programmer: Writes Code

Java

Programming

modern software:
~4.8 million lines of code
(Chromium)

Hardware Designer: Makes Chips

modern chip:
~16 billion transistors
(Apple M1)

In a perfect world...

- Programming languages provide an abstraction

Programmer: Writes Code

Hardware Designer: Makes Chips

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

In a perfect world…

- Historically this worked well

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

In a perfect world...

- Historically this worked well

Java

Programming

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2003

700 MHz

In a perfect world...

- Historically this worked well

- Dennard's scaling:
  - Computer speed doubles every 1.5 years.

Java

Programming

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2003

700 MHz

In a perfect world...

- Historically this worked well

Java

Programming

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

- Dennard's scaling:
  - Computer speed doubles every 1.5 years.

2003

700 MHz

2007

2.1 GHz

In a perfect world...

- Historically this worked well

Java

Programming

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

- Dennard's scaling:
  - Computer speed doubles every 1.5 years.

2003

700 MHz

3x ***increase***
over 4 years

2007

2.1 GHz

In a perfect world…

- Historically this worked well

Java

Programming

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

- Programming languages also evolved:
  - Garbage Collection
  - Memory Safety
  - Runtimes

However...

These trends slowed down in ~2007

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

However...

These trends slowed down in ~2007

2007
2.1 GHz

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

However...

These trends slowed down in ~2007

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2007
2.1 GHz

2017
2.5 GHz

However...

These trends slowed down in ~2007

Java

Programming

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2007
2.1 GHz

1.2x increase over 10 years

2017
2.5 GHz

However…

These trends slowed down in ~2007

Java

Programming

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2007
2.1 GHz

2 cores

1.2x increase
over 10 years

2017
2.5 GHz

4 cores

Reexamining the stack

Java

Programming

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

Optimized and designed over decades for single core.

Parallel programming breaks down these abstractions

Performance - memory contention
Safety - how to reason about shared data

Reexamining the stack

- Nowadays

Java Programming

To efficiently program parallel architectures, developers looking past the negotiators and more directly at hardware

Reexamining the stack

- Nowadays

We're going to pick a language that allows reasoning about how it is executed on the hardware

Reexamining the stack

- Nowadays

Heavy runtime, JIT

Java

Programming

Reexamining the stack

- Nowadays

often intuitive mappings to assembly
lean runtime

Modern trends

Mar 2021	Mar 2020	Change	Programming Language	Ratings	Change
1	2	^	C	15.33%	-1.00%
2	1	v	Java	10.45%	-7.33%
3	3		Python	10.31%	+0.20%
4	4		C++	6.52%	-0.27%
5	5		C#	4.97%	-0.35%
6	6		Visual Basic	4.85%	-0.40%
7	7		JavaScript	2.11%	+0.06%
8	8		PHP	2.07%	+0.05%
9	12	^	Assembly language	1.97%	+0.72%

source: Tiobe index

Reasons for C’s popularity

- There have always been reasons to program close to the hardware
  - Embedded systems
  - parallelism
  - diversity of architecture (especially recently)

- C/++ has a massive ecosystem, large and active community. It can keep up with hardware trends and allows extremely efficient code to be written while keeping a manageable level of abstraction

C/++ is not perfect

- **Downsides**: Security issues, bugs, pointers, complicated specification
- designing a fast, and safe programming language is ***difficult***. Very much an open problem. Many of you may be working on it in your career.
- Rust seems like an interesting development. Not yet to the place where I see it being viable to teach.
  - currently ranked 27
  - Overhead of learning a new language and parallelism...

Python?

- Great language for scripting
  - We will use it to automate experiments in this class
- The GIL (global interpreter lock) restricts parallelism significantly.
  - makes the language safe
- TensorFlow and Pytorch?
  - wrappers around low-level kernels that execute outside of the python interpreter

Lecture Schedule

- Overview - why do we need a lecture on compilation and architecture?
- **Compilation** - *How do we translate a program from a human-accessible language to a language that the processor understands*
- Architecture - How do processors execute programs?
- Example

Compilation:

Language

C++ Programming

ISA

x86

Compilation:

Language

Programming

ISA

x86

```
int add(int a, int b) {
    return a + b;
}
```

Compilation:

Language

Programming

ISA

x86

```
int add(int a, int b) {
    return a + b;
}
```

Officially defined by the specification

ISO standard: costs $200

~1400 pages

Compilation:

Language

ISA

```cpp
int add(int a, int b) {
   return a + b;
}
```

Officially defined by the specification
ISO standard: costs $200
~1400 pages

official specification
Intel provides a specification: *free*
2200 pages

Compilation:

Language

```
int add(int a, int b) {
    return a + b;
}
```

Officially defined by the specification

ISO standard: costs $200

~1400 pages

ISA

```
???
```

official specification

Intel provides a specification: *free*

2200 pages

Compilation:

Language

```cpp
int add(int a, int b) {
    return a + b;
}
```

Officially defined by the specification

ISO standard: costs $200

~1400 pages

```asm
add(int, int): # @add(int, int)
push rbp
mov rbp, rsp
mov dword ptr [rbp - 4], edi
mov dword ptr [rbp - 8], esi
mov eax, dword ptr [rbp - 4]
add eax, dword ptr [rbp - 8]
pop rbp
ret
```

official specification

Intel provides a specification: *free*

2200 pages

Compilation:

Language

Programming

```
int add(int a, int b) {
    return a + b;
}
```

Officially defined by the specification

ISO standard: costs $200

~1400 pages

arm

```
add(int, int):
sub sp, sp, #16
str w0, [sp, #12]
str w1, [sp, #8]
ldr w8, [sp, #12]
ldr w9, [sp, #8]
add w0, w8, w9
add sp, sp, #16
ret
```

How about a more complicated program?

Quadratic formula

How about a more complicated program?

Quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

How about a more complicated program?

Quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

How about a more complicated program?

Quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

x86

official specification

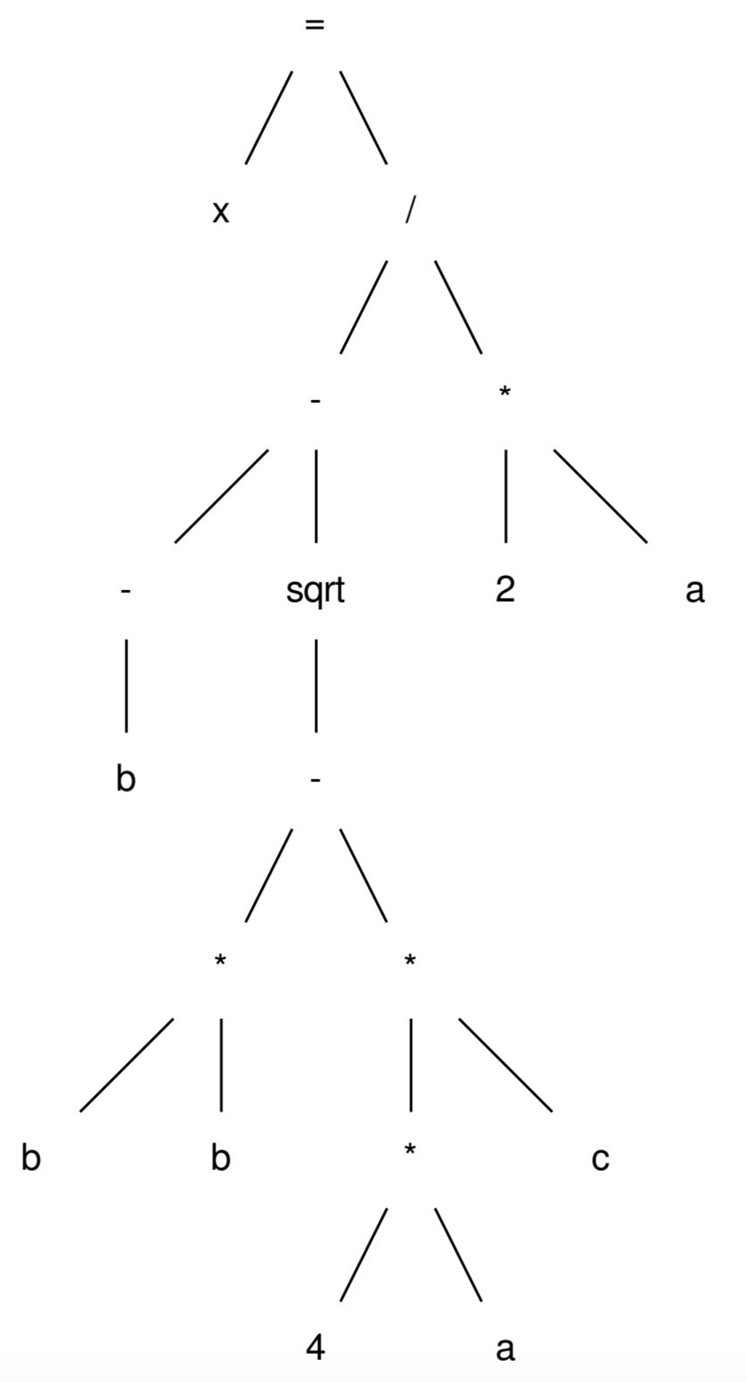
Intel provides a specification: *free*

2200 pages

There is not an ISA instruction that combines all these instructions!

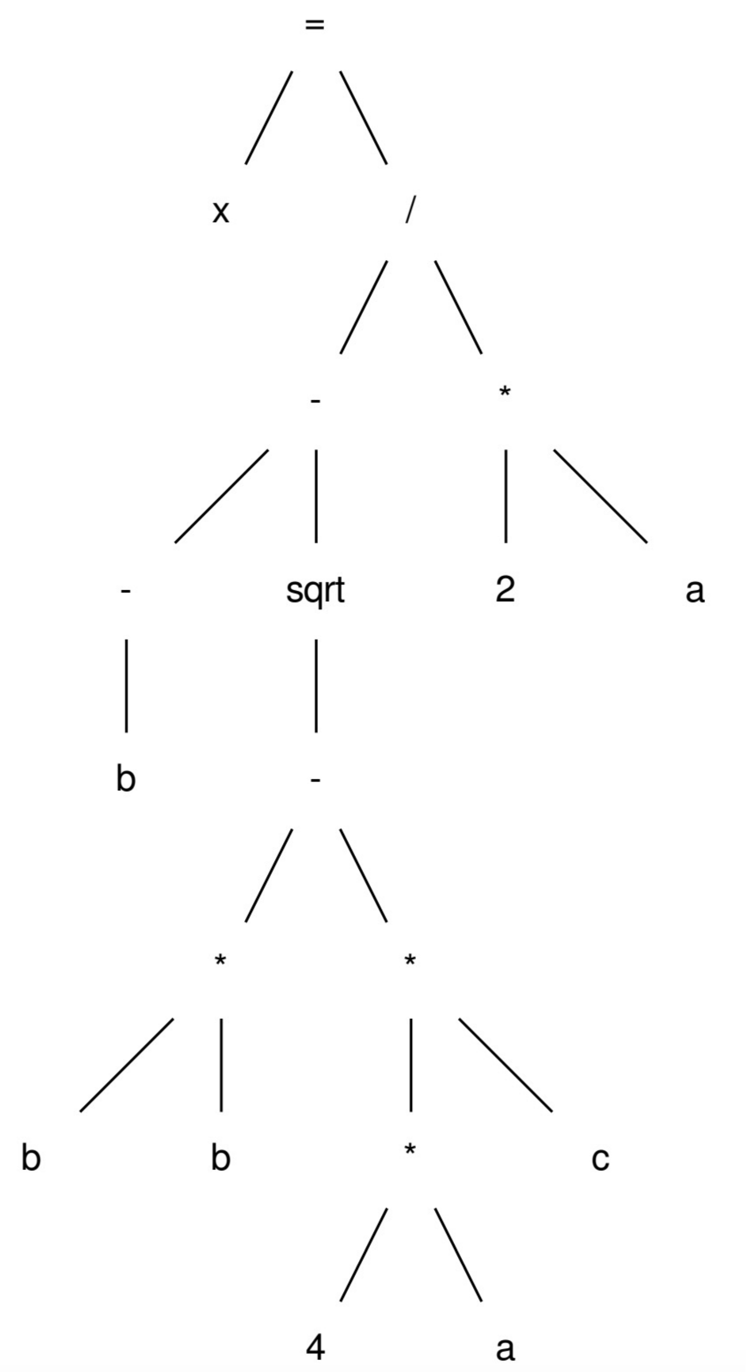
```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

A compiler will turn this into an *abstract syntax tree* (AST)



Simplify this code:

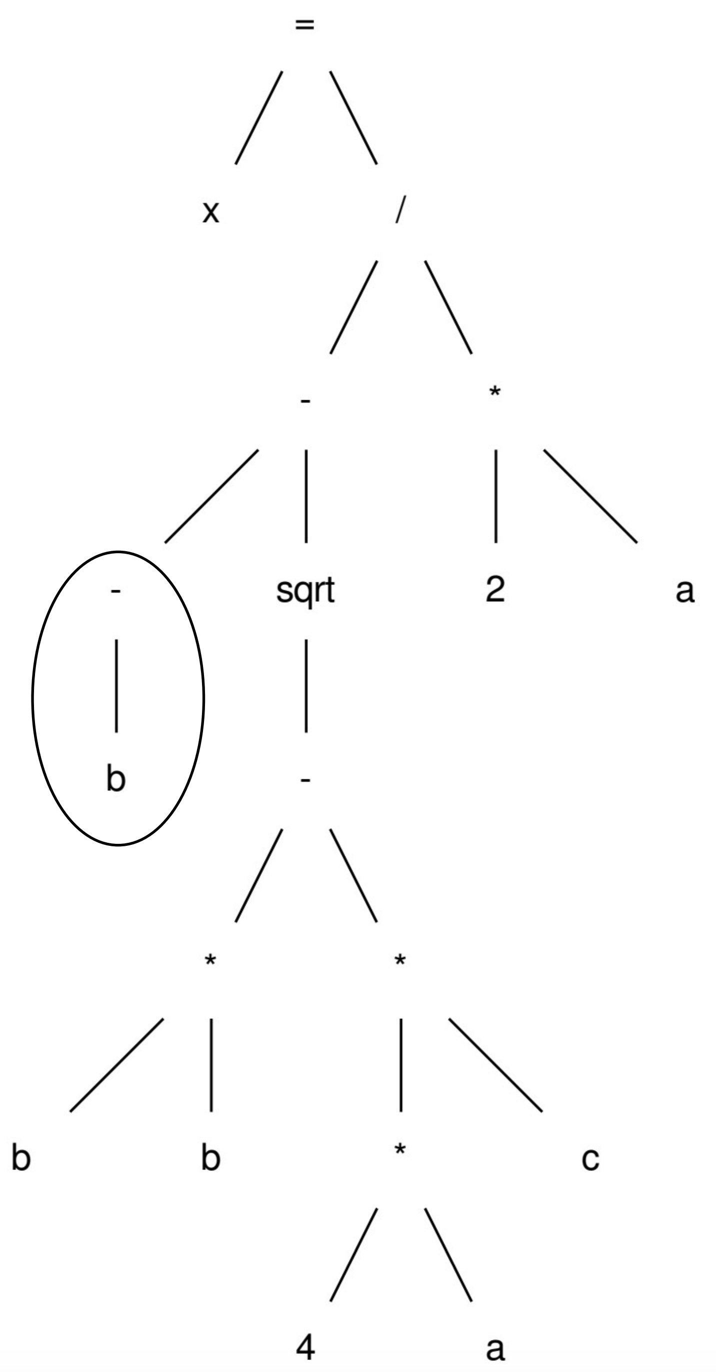
post-order traversal, using temporary variables



Simplify this code:

post-order traversal, using temporary variables

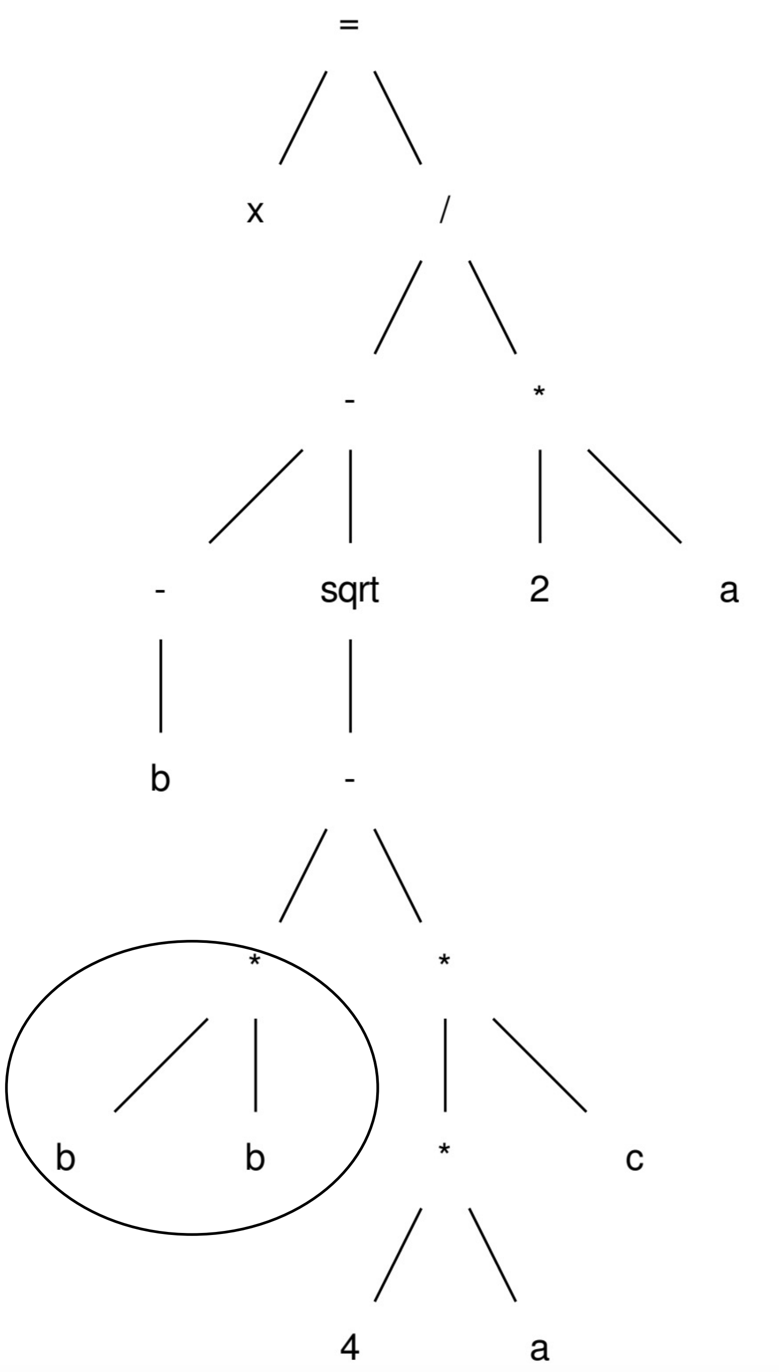
```
r0 = neg(b);
```



Simplify this code:

post-order traversal, using temporary variables

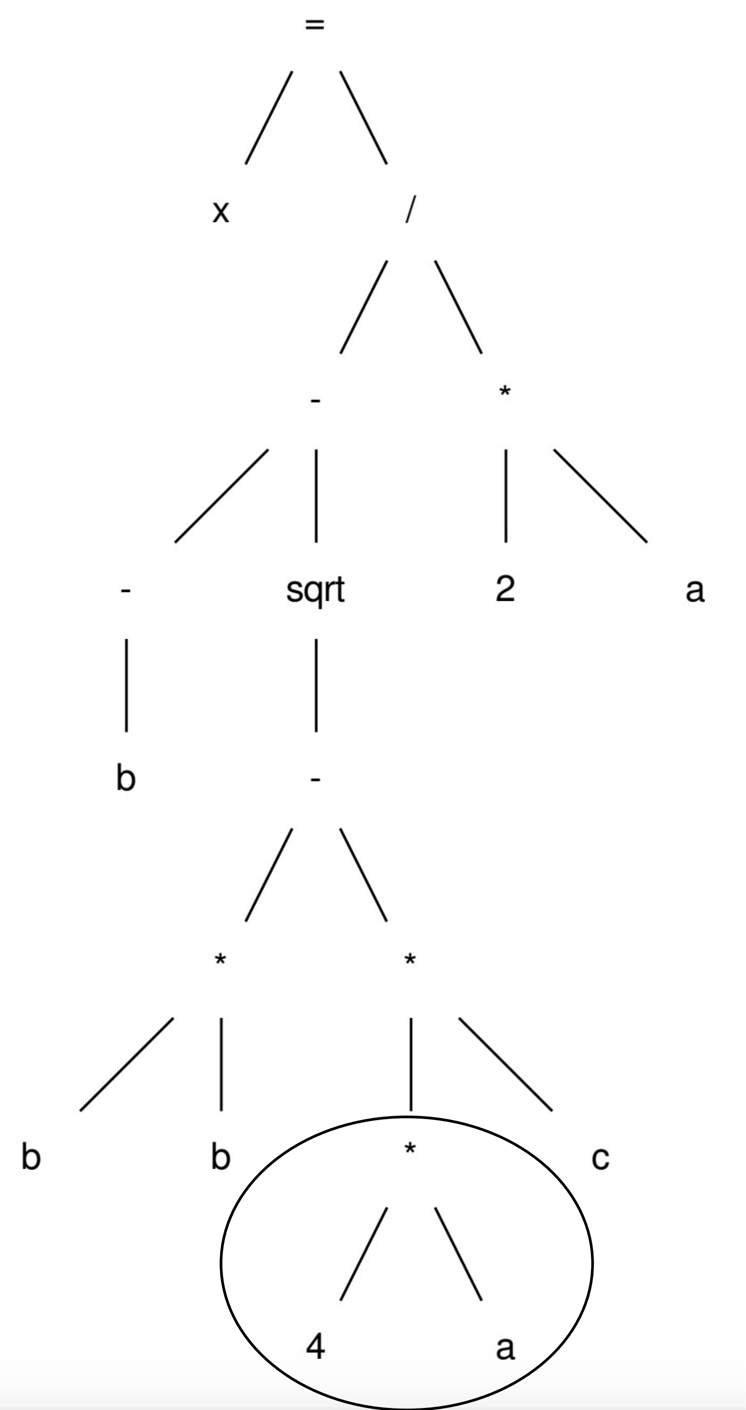
```
r0 = neg(b);
r1 = b * b;
```



Simplify this code:

post-order traversal, using temporary variables

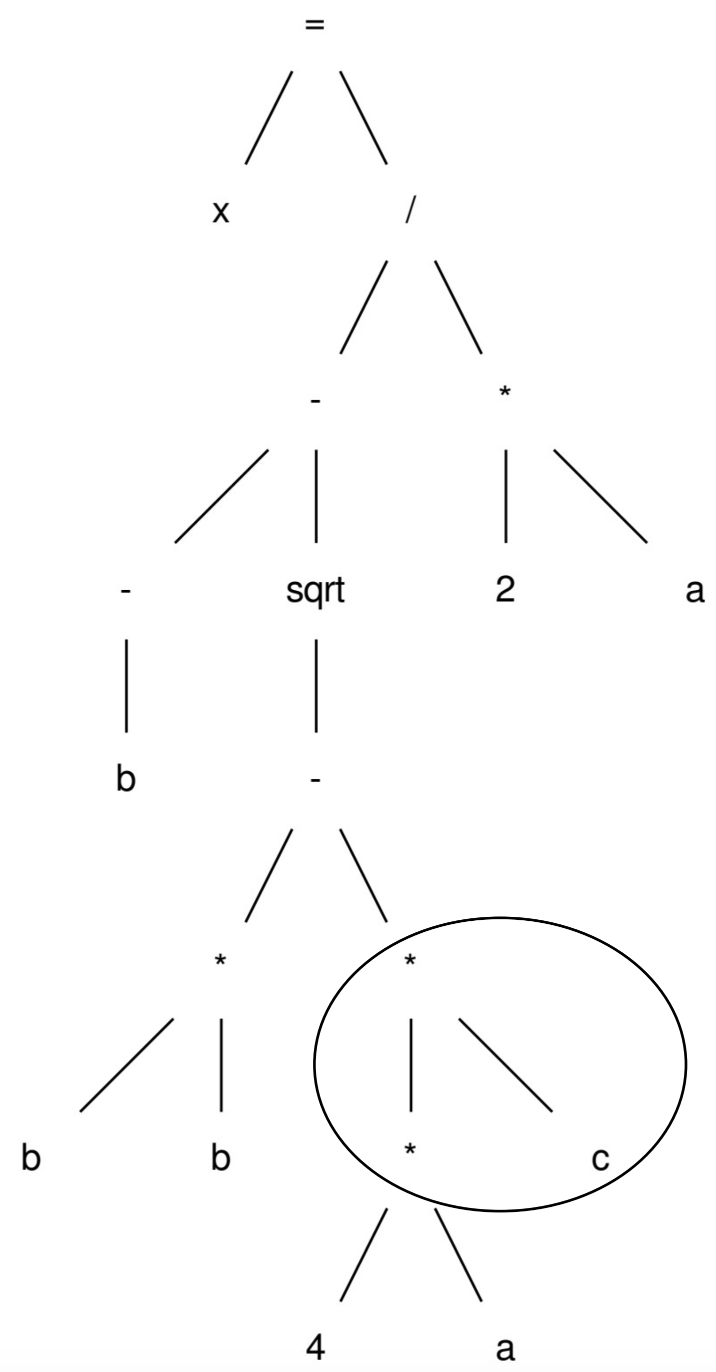
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
```



Simplify this code:

post-order traversal, using temporary variables

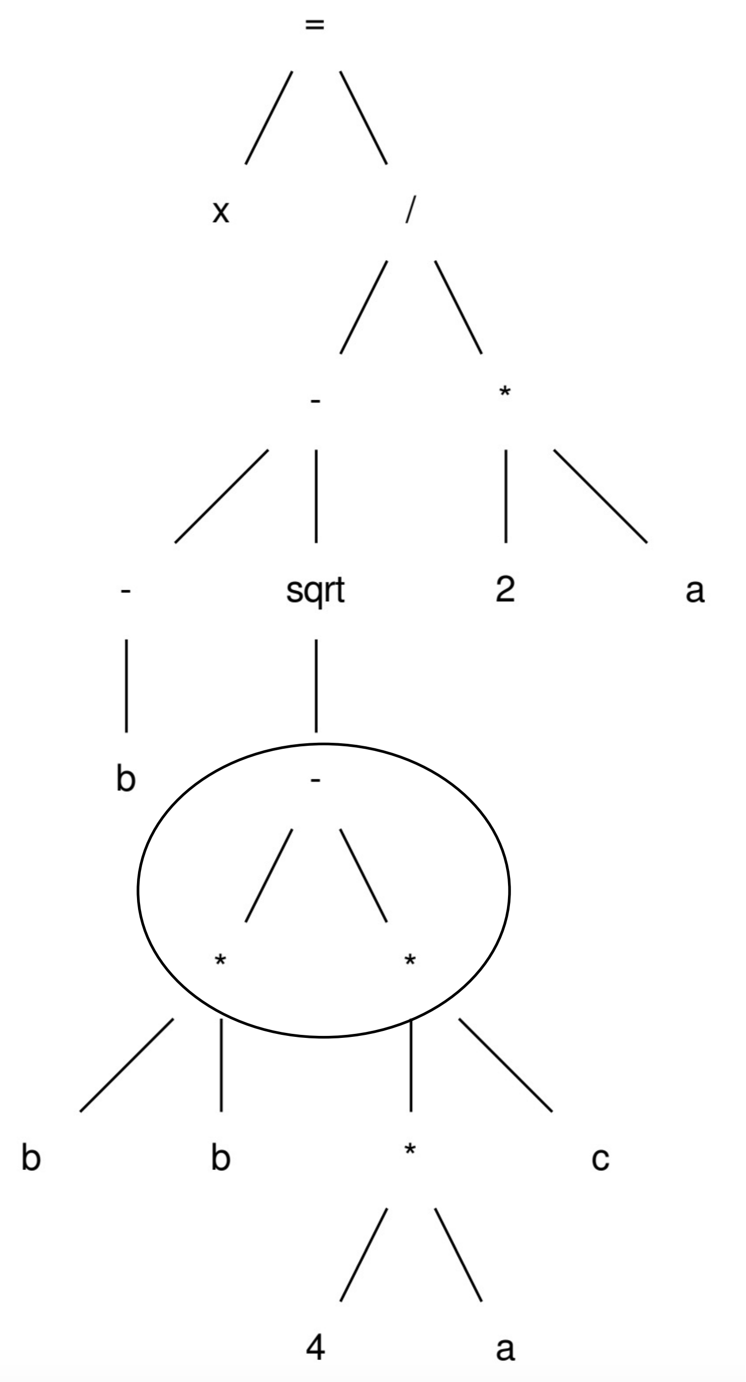
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
```



Simplify this code:

post-order traversal, using temporary variables

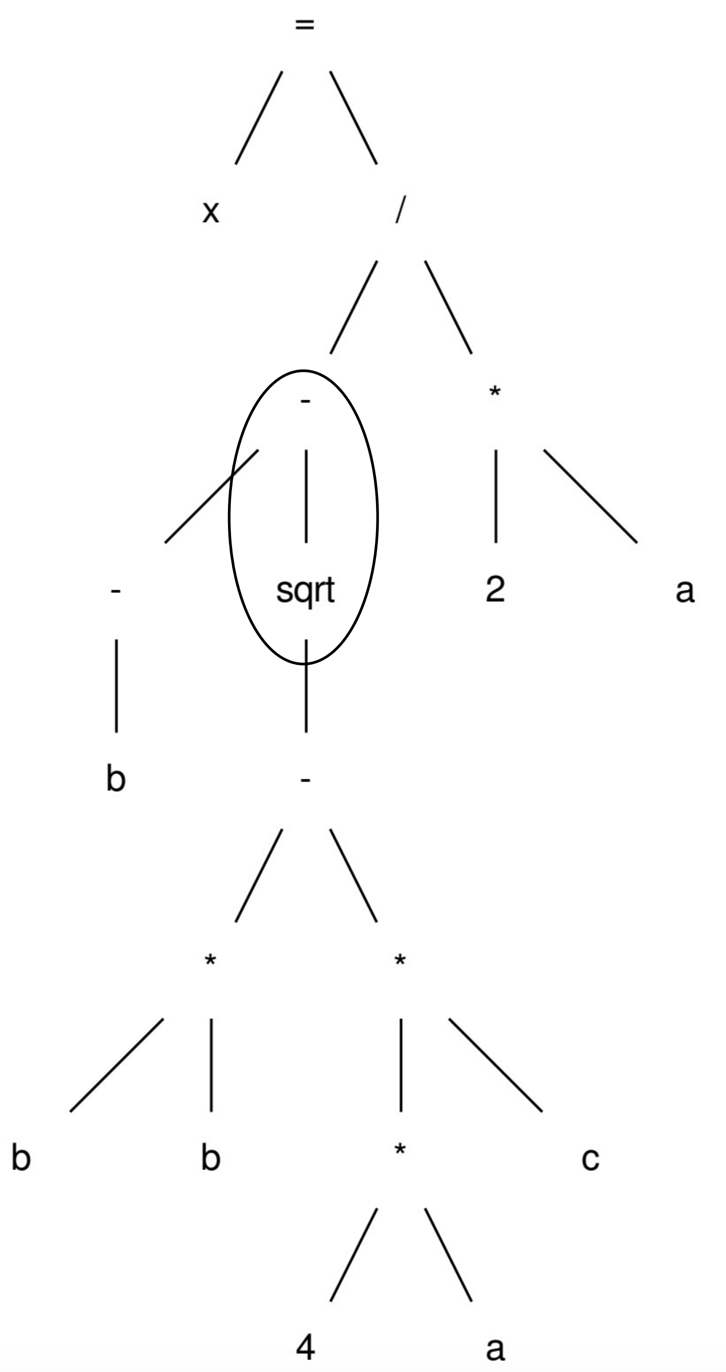
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
```



Simplify this code:

post-order traversal, using temporary variables

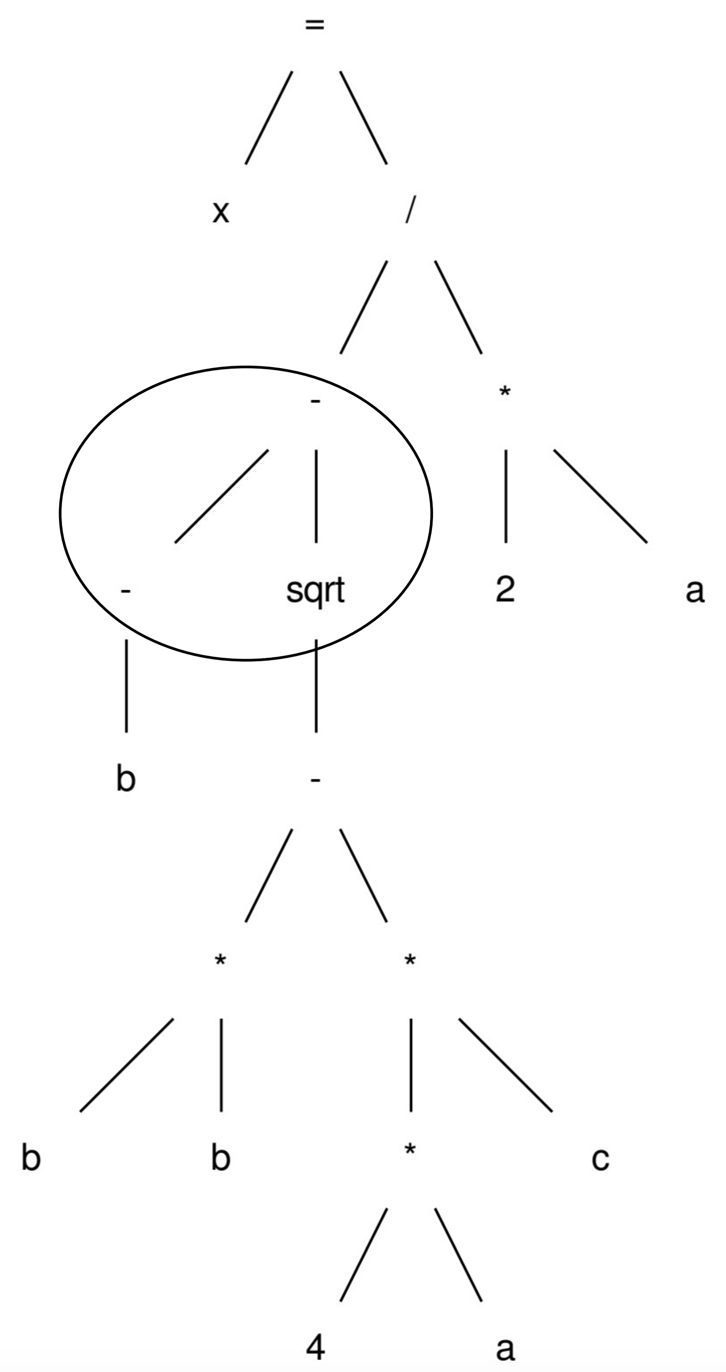
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
```



Simplify this code:

post-order traversal, using temporary variables

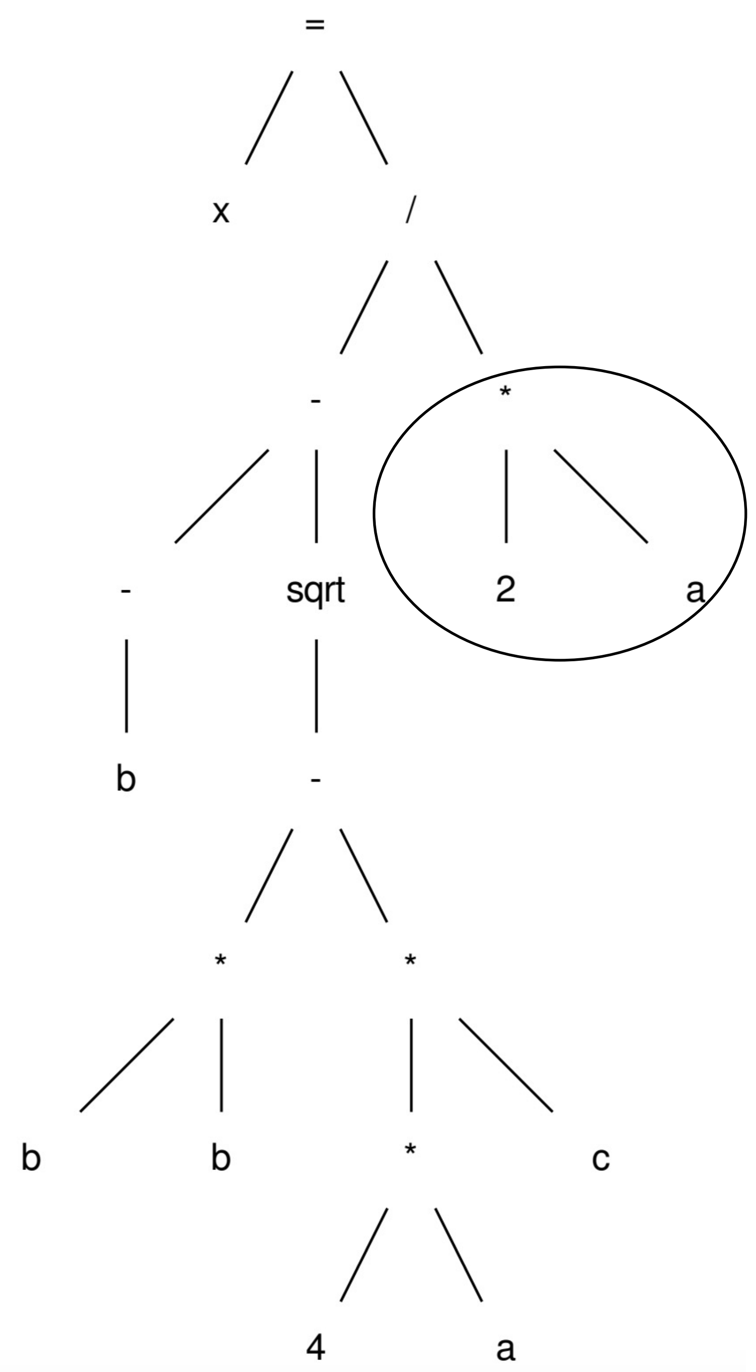
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
```



Simplify this code:

post-order traversal, using temporary variables

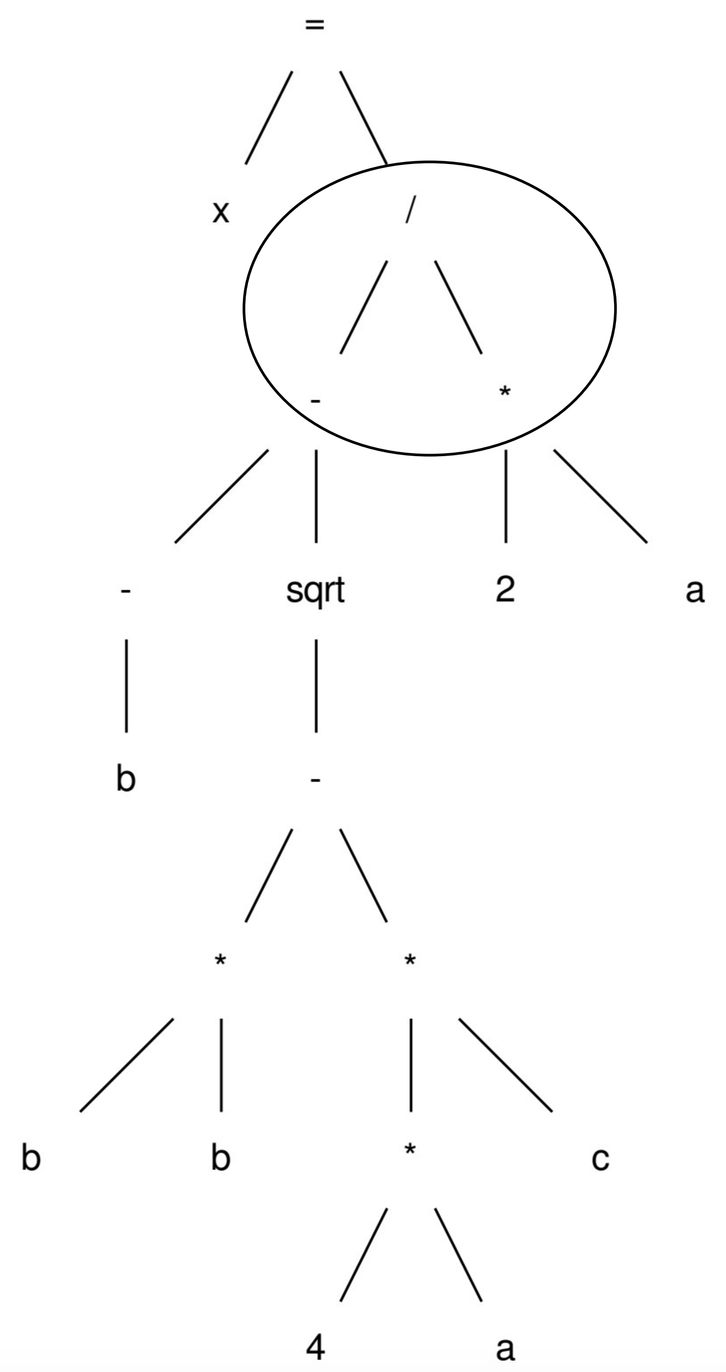
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
```



Simplify this code:

post-order traversal, using temporary variables

```
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;
```



Simplify this code:

post-order traversal, using temporary variables

```
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;
```

- This is not exactly an ISA
  - unlimited registers
  - not always a 1-1 mapping of instructions.
- but it is much easier to translate to the ISA
- We call this an intermediate representation, or IR
- Examples of IR: LLVM, SPIR-V

```c
// Type your code here, or load an example.
float sqrt(float x);

float add(float a, float b, float c) {
    return (-b - sqrt(b*b - 4 * a * c)) / (2*a);

}

```

A Output... Filter... Libraries Add new... Add tool...

```llvm

define dso_local float @_Z3addfff(float %0, float %1, float %2) #0 !dbg !
  %4 = alloca float, align 4
  %5 = alloca float, align 4
  %6 = alloca float, align 4
  store float %0, float* %4, align 4
  call void @llvm.dbg.declare(metadata float* %4, metadata !12, metadata
  store float %1, float* %5, align 4
  call void @llvm.dbg.declare(metadata float* %5, metadata !14, metadata
  store float %2, float* %6, align 4
  call void @llvm.dbg.declare(metadata float* %6, metadata !16, metadata
  %7 = load float, float* %5, align 4, !dbg !18
  %8 = fneg float %7, !dbg !19
  %9 = load float, float* %5, align 4, !dbg !20
  %10 = load float, float* %5, align 4, !dbg !21
  %11 = fmul float %9, %10, !dbg !22
  %12 = load float, float* %4, align 4, !dbg !23
  %13 = fmul float 4.000000e+00, %12, !dbg !24
  %14 = load float, float* %6, align 4, !dbg !25
  %15 = fmul float %13, %14, !dbg !26
  %16 = fsub float %11, %15, !dbg !27
  %17 = call float @_Z4sqrtf(float %16), !dbg !28
  %18 = fsub float %8, %17, !dbg !29
  %19 = load float, float* %4, align 4, !dbg !30
  %20 = fmul float 2.000000e+00, %19, !dbg !31
  %21 = fdiv float %18, %20, !dbg !32
  ret float %21, !dbg !33
}
```

C program

llvm IR

Memory accesses

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

Unless explicitly expressed in the programming language, loads and stores are split into multiple instructions!

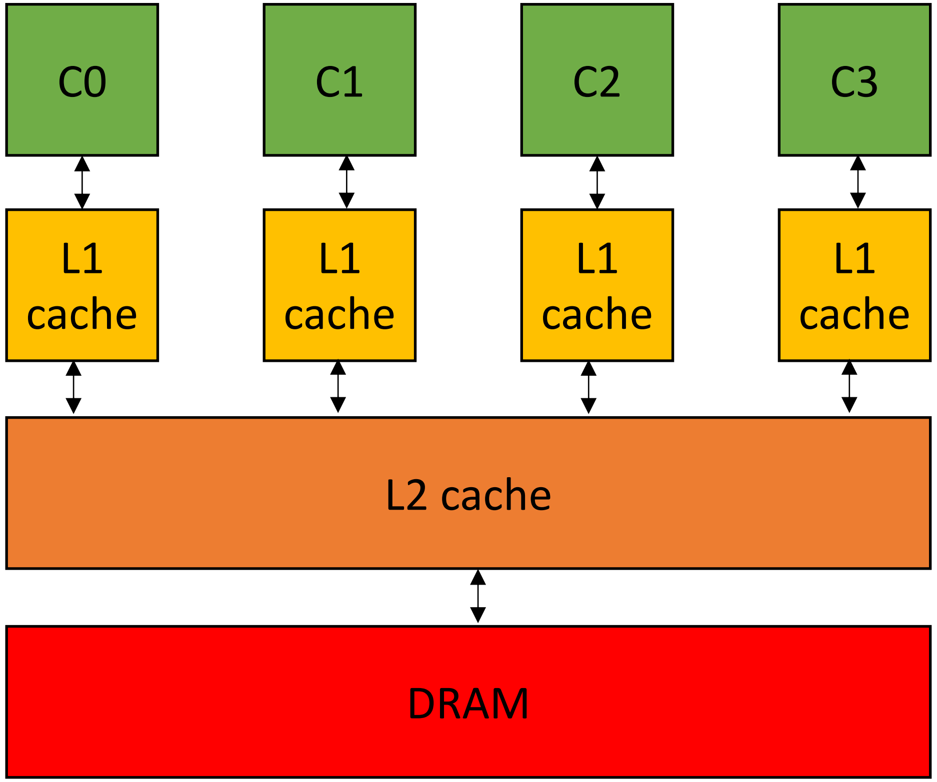
Zoom out

- This can be a lot if you don’t have a compiler background; don’t feel overwhelmed!
- To be successful in this class, you don’t need to be an expert on compilation, ISAs, or IRs.
- The important thing is to have a mental model of how your complex code is broken down into instructions that are executed on hardware, especially loads and stores

Lecture Schedule

- Overview - why do we need a lecture on compilation and architecture?
- Compilation - How do we translate a program from a human-accessible language to a language that the processor understands
- **Architecture** - How do processors execute programs?
- Example

Architecture visual



Core

A core executes a stream
of sequential ISA instructions

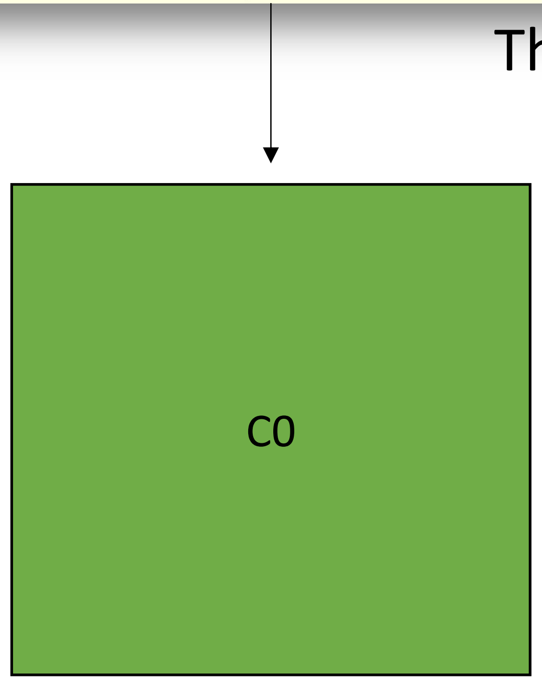
A good mental model executes
1 ISA instruction per cycle

3 Ghz means 3B cycles per second
1 ISA instruction takes .33 ns

Compiled function #0

```
        movd    eax, xmm0
        xor     eax, 2147483648
        movd    xmm0, eax
        movss   dword ptr [rbp - 16], xmm0
        movss   xmm0, dword ptr [rbp - 8]
        mulss   xmm0, dword ptr [rbp - 8]
        movss   xmm1, dword ptr [rip + .LCPI0_1]
        mulss   xmm1, dword ptr [rbp - 4]
        mulss   xmm1, dword ptr [rbp - 12]
        subss   xmm0, xmm1
        call    sqrt(float)
        movaps  xmm1, xmm0
        movss   xmm0, dword ptr [rbp - 16]
        subss   xmm0, xmm1
        movss   xmm1, dword ptr [rip + .LCPI0_0]
        mulss   xmm1, dword ptr [rbp - 4]
        divss   xmm0, xmm1
```

Thread 0



Core

Core

Sometimes multiple programs want to share the same core.

Compiled function #0

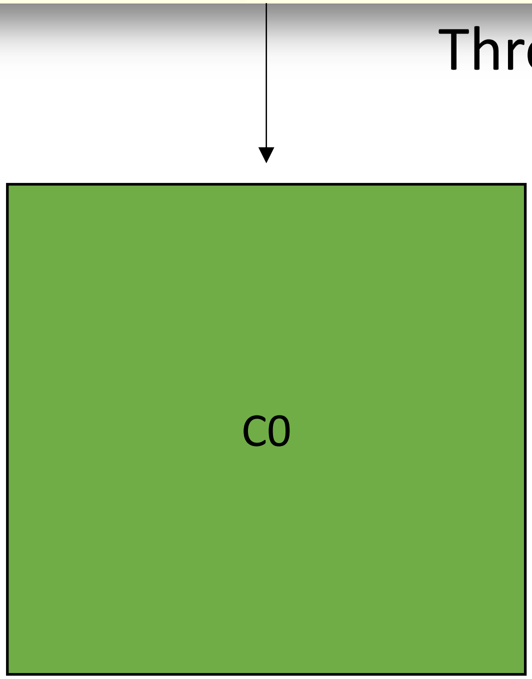
```
movd    eax, xmm0
xor     eax, 2147483648
movd    xmm0, eax
movss   dword ptr [rbp - 16], xmm0
movss   xmm0, dword ptr [rbp - 8]
mulss   xmm0, dword ptr [rbp - 8]
movss   xmm1, dword ptr [rip + .LCPI0_1]
mulss   xmm1, dword ptr [rbp - 4]
mulss   xmm1, dword ptr [rbp - 12]
subss   xmm0, xmm1
call    sqrt(float)
movaps  xmm1, xmm0
movss   xmm0, dword ptr [rbp - 16]
subss   xmm0, xmm1
movss   xmm1, dword ptr [rip + .LCPI0_0]
mulss   xmm1, dword ptr [rbp - 4]
divss   xmm0, xmm1
```

Thread 0

Compiled function #1

```
movss   xmm0, dword ptr [rbp - 8]          #
mulss   xmm0, dword ptr [rbp - 8]
movss   xmm1, dword ptr [rip + .LCPI0_1]
mulss   xmm1, dword ptr [rbp - 4]
mulss   xmm1, dword ptr [rbp - 12]
subss   xmm0, xmm1
call    sqrt(float)
movaps  xmm1, xmm0
movss   xmm0, dword ptr [rbp - 16]         #
subss   xmm0, xmm1
movss   xmm1, dword ptr [rip + .LCPI0_0]
mulss   xmm1, dword ptr [rbp - 4]
divss   xmm0, xmm1
add     rsp, 16
```

Thread 1



Core

Core

Sometimes multiple programs want to share the same core.

Compiled function #0

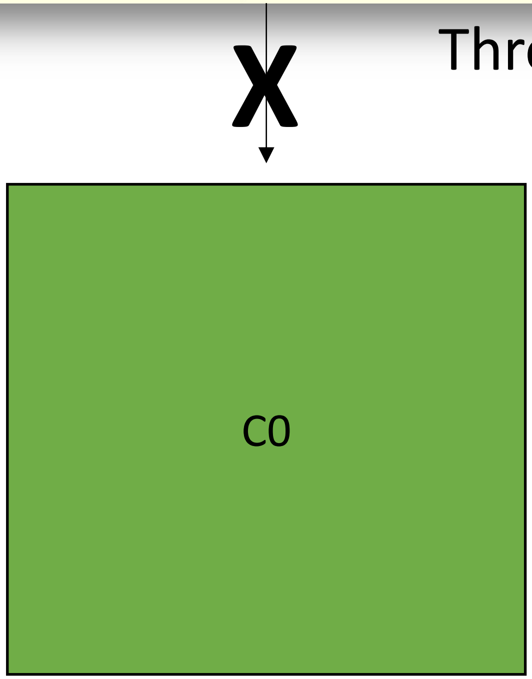
```
movd    eax, xmm0
xor     eax, 2147483648
movd    xmm0, eax
movss   dword ptr [rbp - 16], xmm0
movss   xmm0, dword ptr [rbp - 8]
mulss   xmm0, dword ptr [rbp - 8]
movss   xmm1, dword ptr [rip + .LCPI0_1]
mulss   xmm1, dword ptr [rbp - 4]
mulss   xmm1, dword ptr [rbp - 12]
subss   xmm0, xmm1
call    sqrt(float)
movaps  xmm1, xmm0
movss   xmm0, dword ptr [rbp - 16]
subss   xmm0, xmm1
movss   xmm1, dword ptr [rip + .LCPI0_0]
mulss   xmm1, dword ptr [rbp - 4]
divss   xmm0, xmm1
```

Thread 0

Compiled function #1

```
movss   xmm0, dword ptr [rbp - 8]          #
mulss   xmm0, dword ptr [rbp - 8]
movss   xmm1, dword ptr [rip + .LCPI0_1]
mulss   xmm1, dword ptr [rbp - 4]
mulss   xmm1, dword ptr [rbp - 12]
subss   xmm0, xmm1
call    sqrt(float)
movaps  xmm1, xmm0
movss   xmm0, dword ptr [rbp - 16]         #
subss   xmm0, xmm1
movss   xmm1, dword ptr [rip + .LCPI0_0]
mulss   xmm1, dword ptr [rbp - 4]
divss   xmm0, xmm1
add     rsp, 16
```

Thread 1



Core

The OS can preempt a thread
(remove it from the hardware resource)

Core

Compiled function #1

```
movss    dword ptr [rbp - 16], xmm0    #
movss    xmm0, dword ptr [rbp - 8]     #
mulss    xmm0, dword ptr [rbp - 8]
movss    xmm1, dword ptr [rip + .LCPI0_1] #
mulss    xmm1, dword ptr [rbp - 4]
mulss    xmm1, dword ptr [rbp - 12]
subss    xmm0, xmm1
call     sqrt(float)
movaps   xmm1, xmm0
movss    xmm0, dword ptr [rbp - 16]    #
subss    xmm0, xmm1
movss    xmm1, dword ptr [rip + .LCPI0_0] #
mulss    xmm1, dword ptr [rbp - 4]
divss    xmm0, xmm1
add      rsp, 16
```

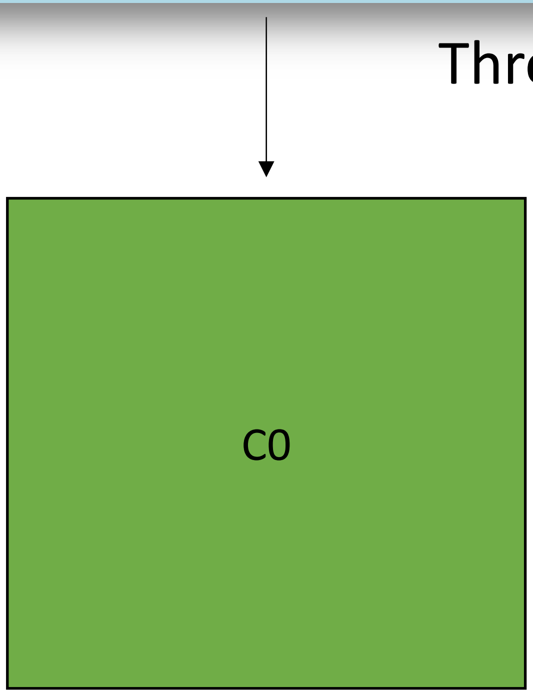
Thread 1

Compiled function #0

```
movd     eax, xmm0
xor      eax, 2147483648
movd     xmm0, eax
movss    dword ptr [rbp - 16], xmm0
movss    xmm0, dword ptr [rbp - 8]
mulss    xmm0, dword ptr [rbp - 8]
movss    xmm1, dword ptr [rip + .LCPI0_1]
mulss    xmm1, dword ptr [rbp - 4]
mulss    xmm1, dword ptr [rbp - 12]
subss    xmm0, xmm1
call     sqrt(float)
movaps   xmm1, xmm0
movss    xmm0, dword ptr [rbp - 16]
subss    xmm0, xmm1
movss    xmm1, dword ptr [rip + .LCPI0_0]
mulss    xmm1, dword ptr [rbp - 4]
divss    xmm0, xmm1
```

Thread 2

Sometimes multiple programs want to share the same core.



Core

And place another thread to execute

This is called concurrency: multiple threads taking turns executing on the same hardware resource

Core

Compiled function #1

```
movss    xmm0, dword ptr [rbp - 8]          #
mulss    xmm0, dword ptr [rbp - 8]
movss    xmm1, dword ptr [rip + .LCPI0_1]
mulss    xmm1, dword ptr [rbp - 4]
mulss    xmm1, dword ptr [rbp - 12]
subss    xmm0, xmm1
call     sqrt(float)
movaps   xmm1, xmm0
movss    xmm0, dword ptr [rbp - 16]         #
subss    xmm0, xmm1
movss    xmm1, dword ptr [rip + .LCPI0_0]
mulss    xmm1, dword ptr [rbp - 4]
divss    xmm0, xmm1
add      rsp, 16
```

Thread 1

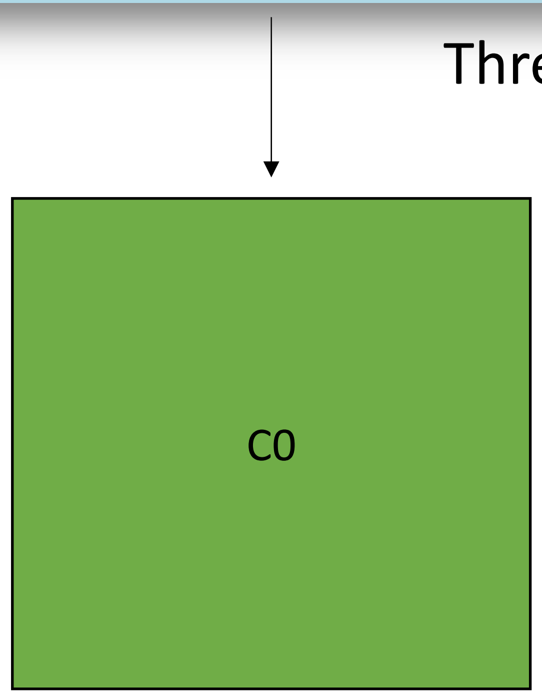
Compiled function #0

```
movd     eax, xmm0
xor      eax, 2147483648
movd     xmm0, eax
movss    dword ptr [rbp - 16], xmm0
movss    xmm0, dword ptr [rbp - 8]
mulss    xmm0, dword ptr [rbp - 8]
movss    xmm1, dword ptr [rip + .LCPI0_1]
mulss    xmm1, dword ptr [rbp - 4]
mulss    xmm1, dword ptr [rbp - 12]
subss    xmm0, xmm1
call     sqrt(float)
movaps   xmm1, xmm0
movss    xmm0, dword ptr [rbp - 16]
subss    xmm0, xmm1
movss    xmm1, dword ptr [rip + .LCPI0_0]
mulss    xmm1, dword ptr [rbp - 4]
divss    xmm0, xmm1
```

Thread 2

Preemption can occur:

- when a thread executes a long latency instruction
- periodically from the OS to provide fairness
- explicitly using sleep instructions



Core

And place another thread to execute

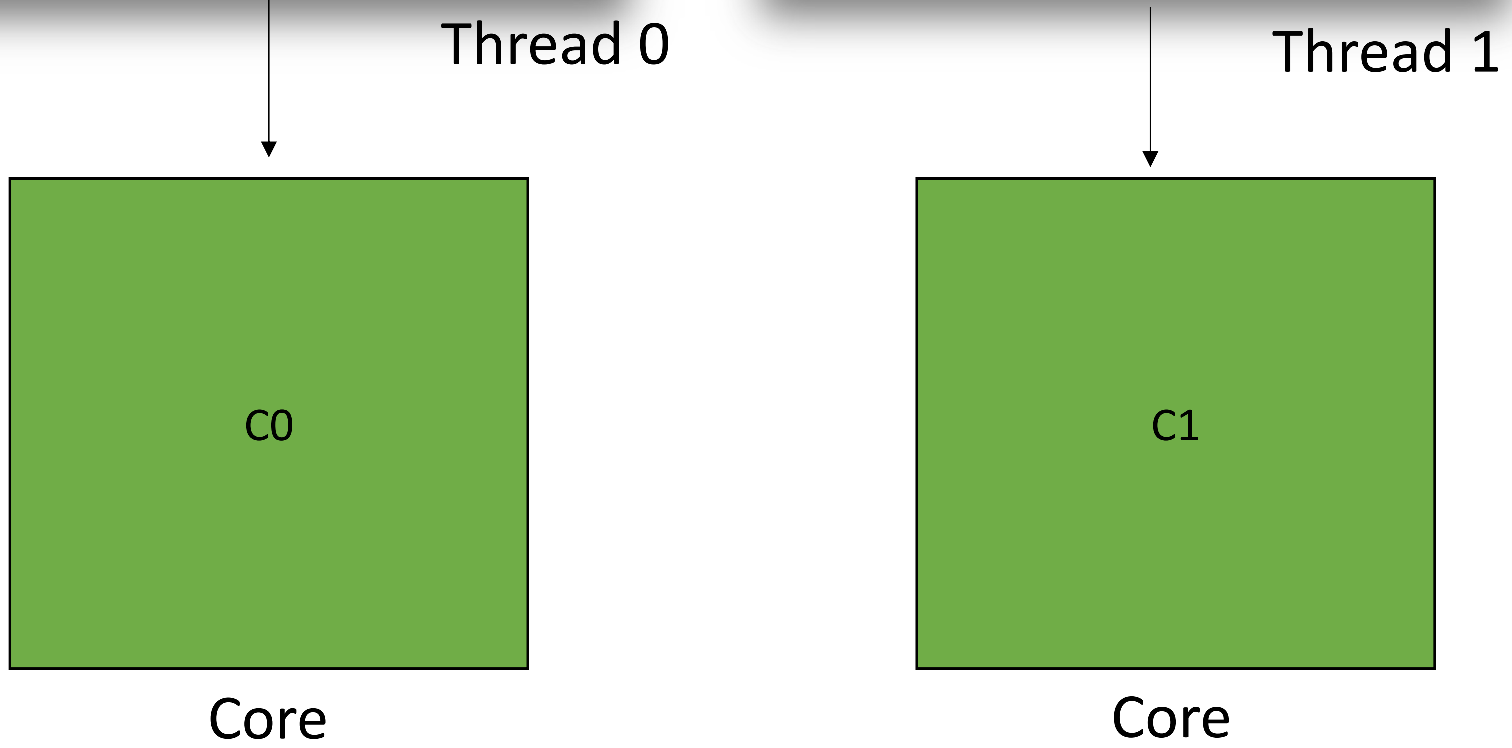
Multicores

Compiled function #0

```
movd     eax, xmm0
xor      eax, 2147483648
movd     xmm0, eax
movss    dword ptr [rbp - 16], xmm0
movss    xmm0, dword ptr [rbp - 8]
mulss    xmm0, dword ptr [rbp - 8]
movss    xmm1, dword ptr [rip + .LCPI0_1]
mulss    xmm1, dword ptr [rbp - 4]
mulss    xmm1, dword ptr [rbp - 12]
subss    xmm0, xmm1
call     sqrt(float)
movaps   xmm1, xmm0
movss    xmm0, dword ptr [rbp - 16]
subss    xmm0, xmm1
movss    xmm1, dword ptr [rip + .LCPI0_0]
mulss    xmm1, dword ptr [rbp - 4]
divss    xmm0, xmm1
```

Compiled function #1

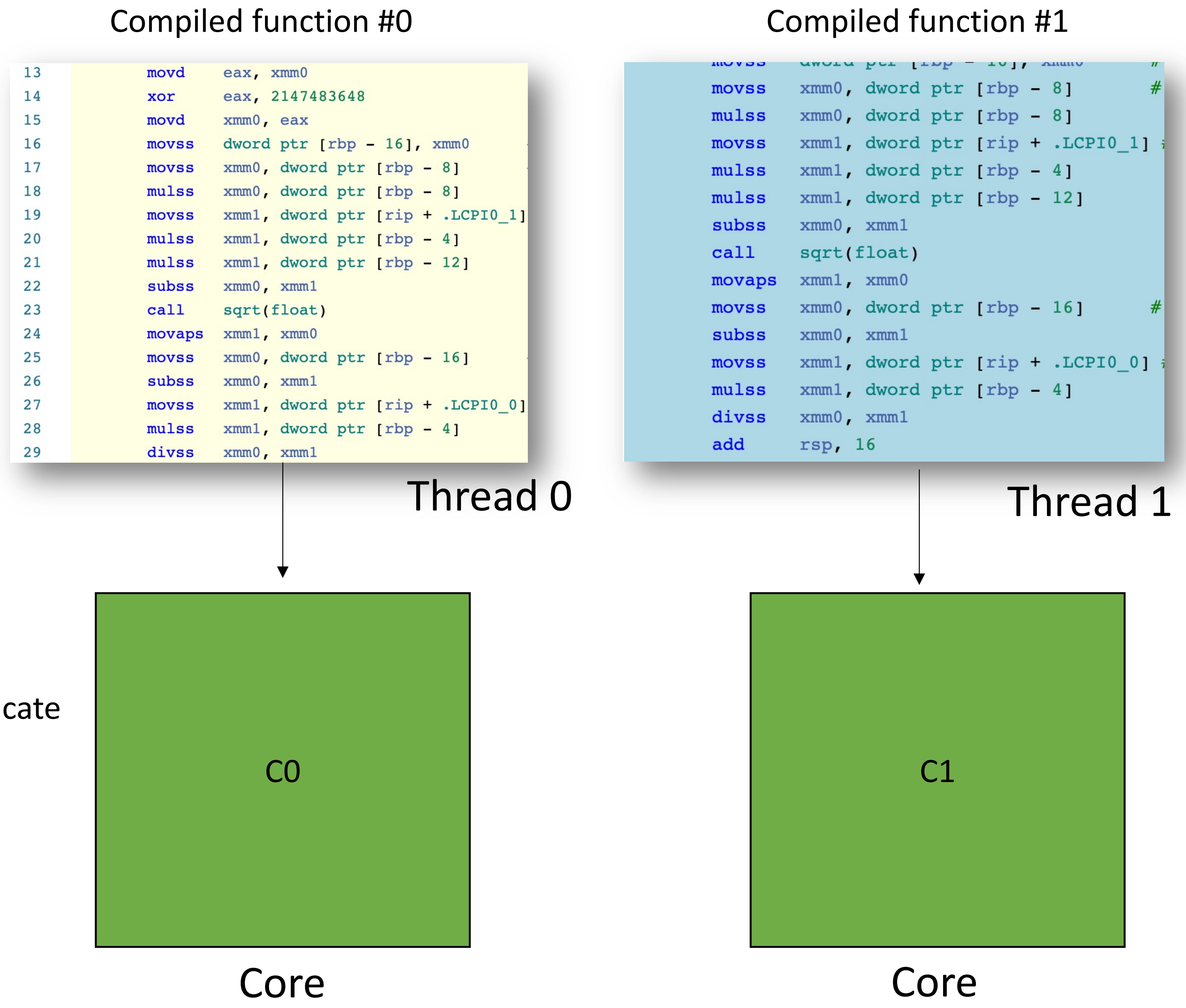
```
movss    xmm0, dword ptr [rbp - 8]          #
mulss    xmm0, dword ptr [rbp - 8]
movss    xmm1, dword ptr [rip + .LCPI0_1]
mulss    xmm1, dword ptr [rbp - 4]
mulss    xmm1, dword ptr [rbp - 12]
subss    xmm0, xmm1
call     sqrt(float)
movaps   xmm1, xmm0
movss    xmm0, dword ptr [rbp - 16]         #
subss    xmm0, xmm1
movss    xmm1, dword ptr [rip + .LCPI0_0]
mulss    xmm1, dword ptr [rbp - 4]
divss    xmm0, xmm1
add      rsp, 16
```



Threads can execute simultaneously.

This is also concurrency. But the simultaneously called parallelism.

Multicores

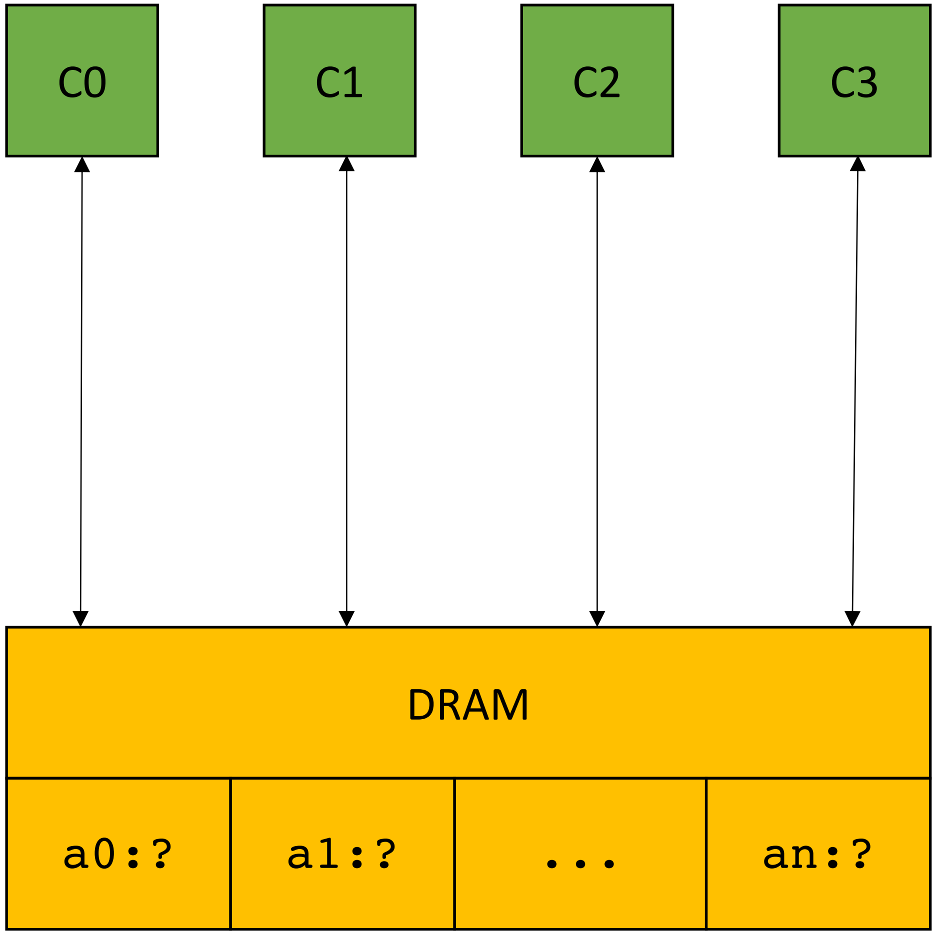


This is fine if threads are independent: e.g. running Chrome and Spotify at the same time.

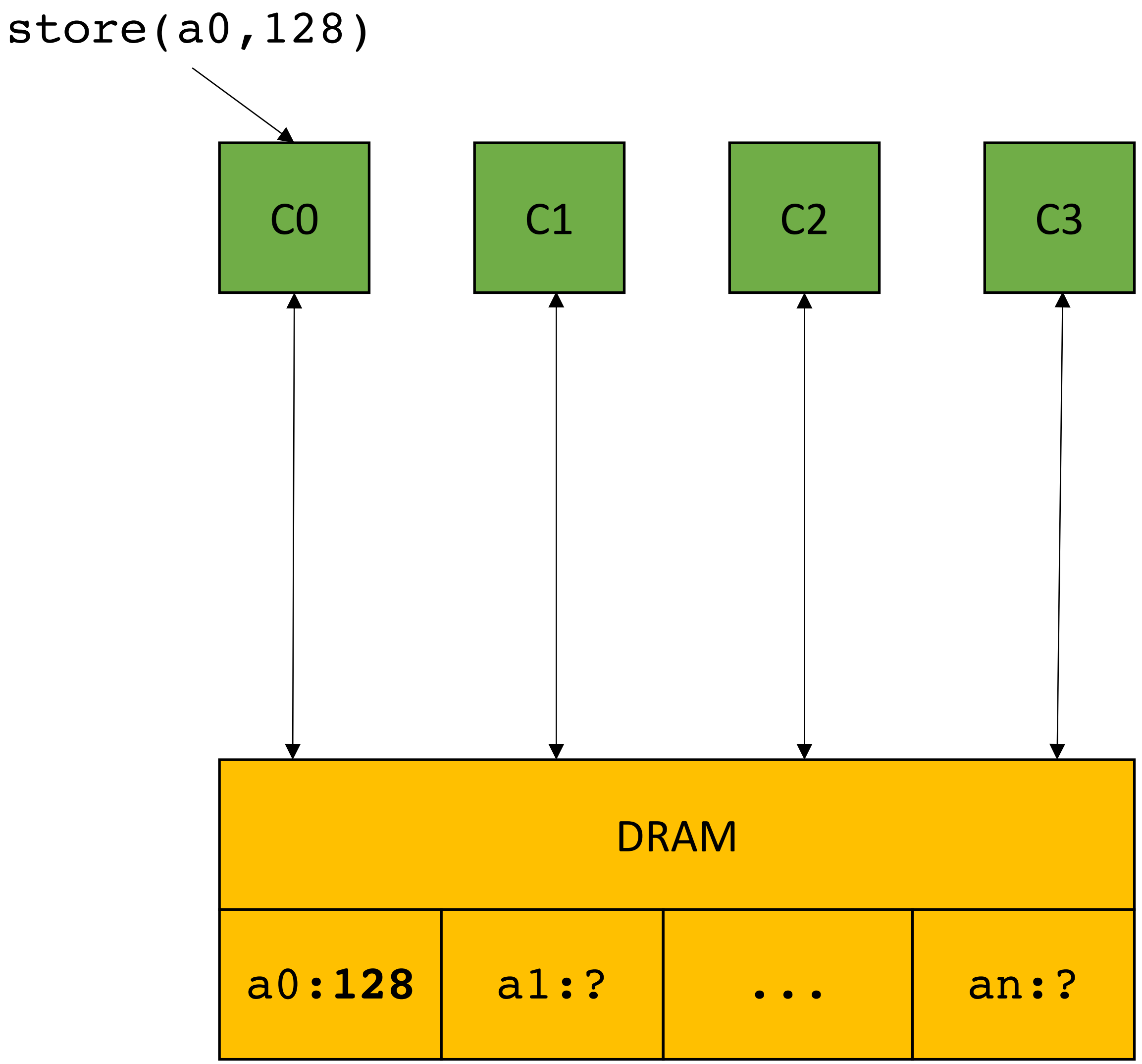
If threads need to cooperate to run the program, then they need to communicate through memory

Main memory

`store(a0,128)`

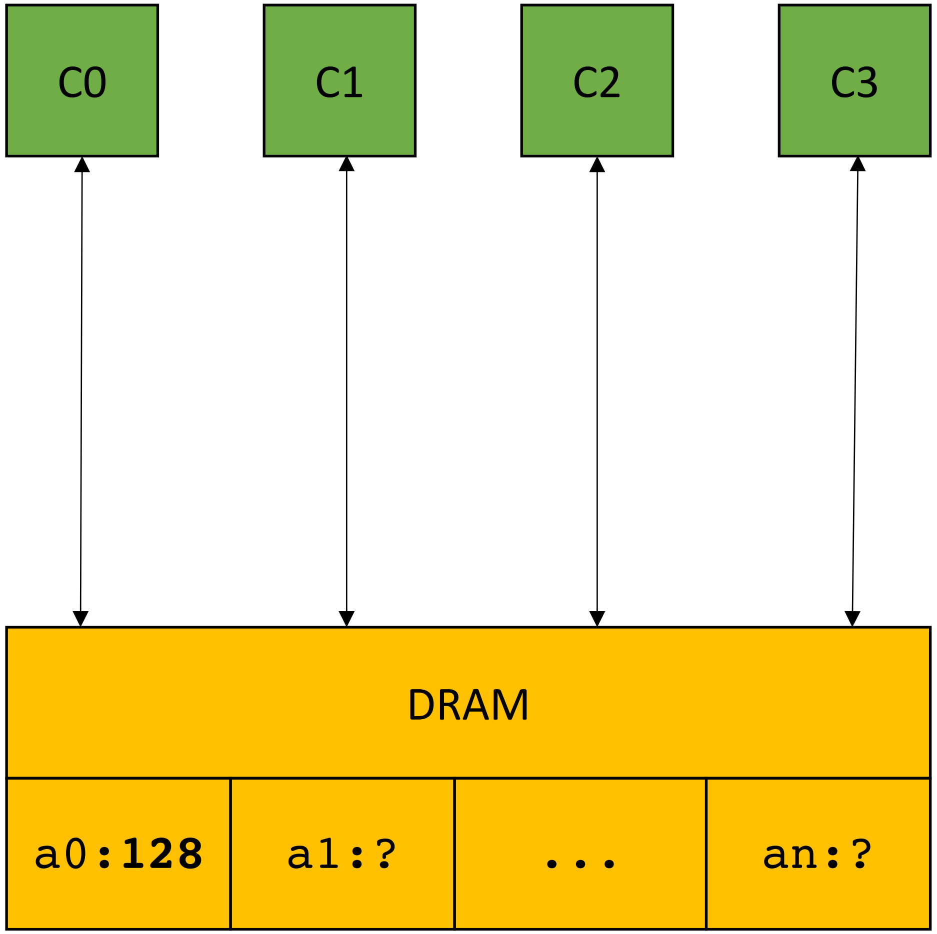


Main memory

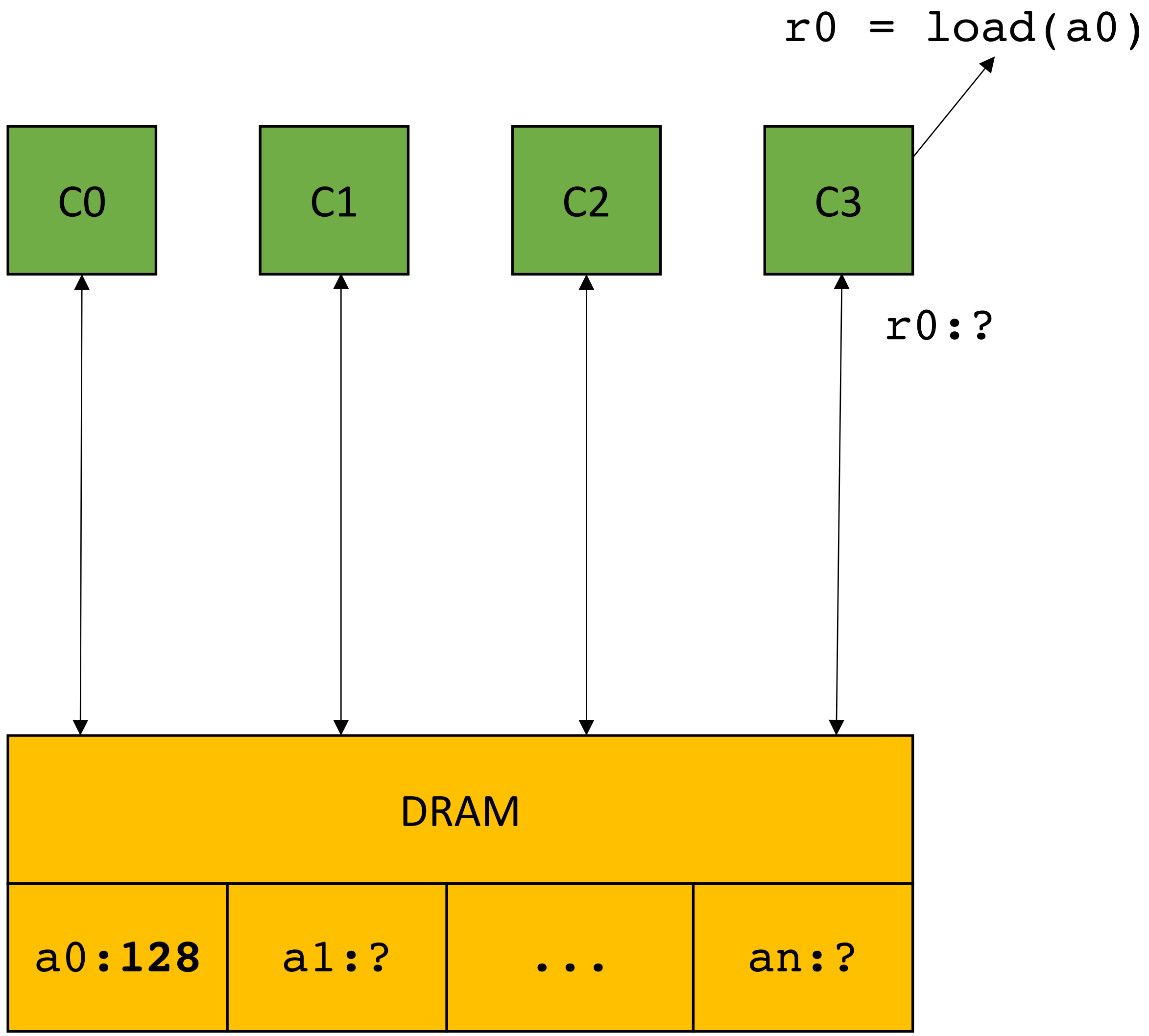


Main memory

`r0 = load(a0)`

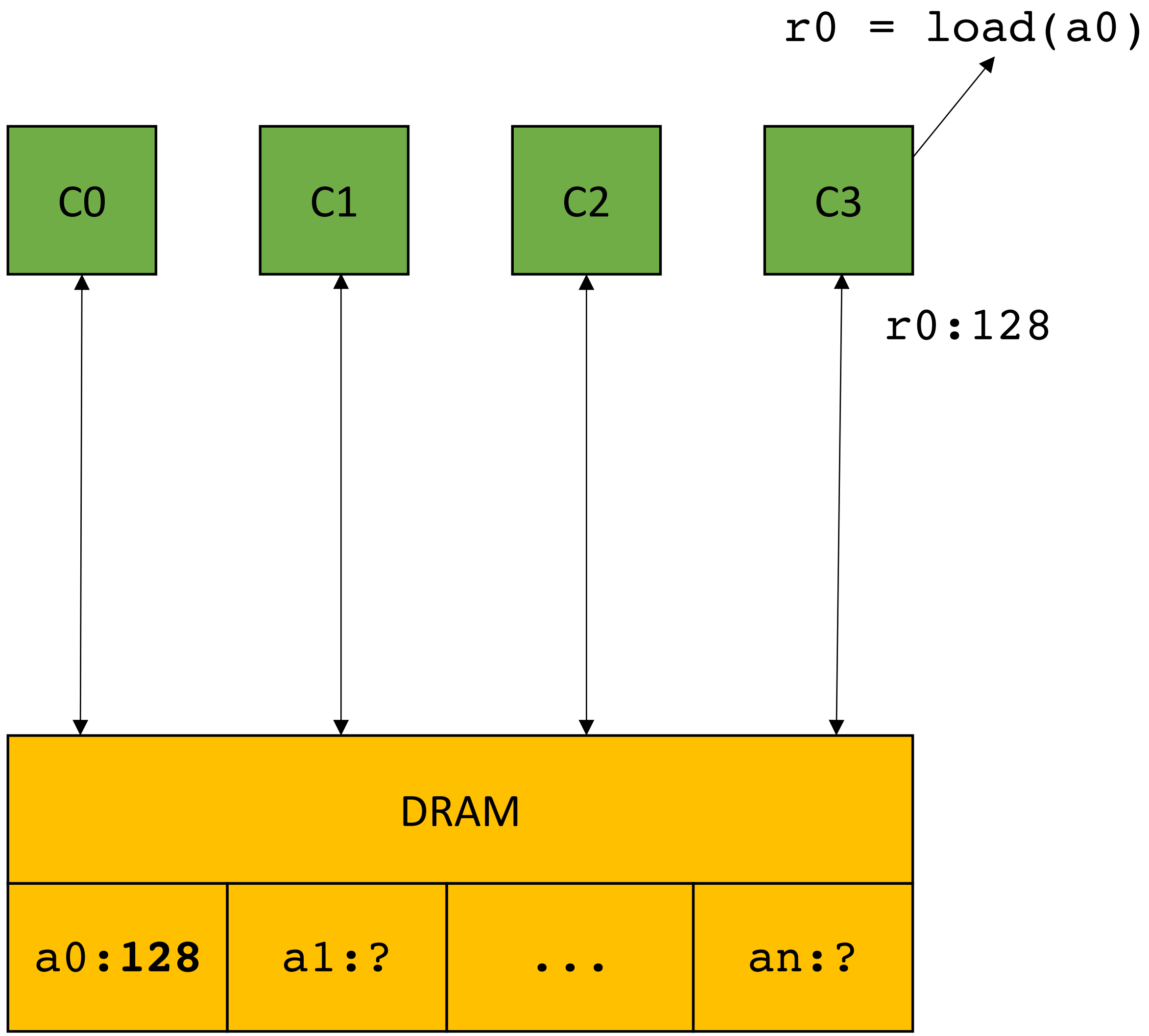


Main memory



Main memory

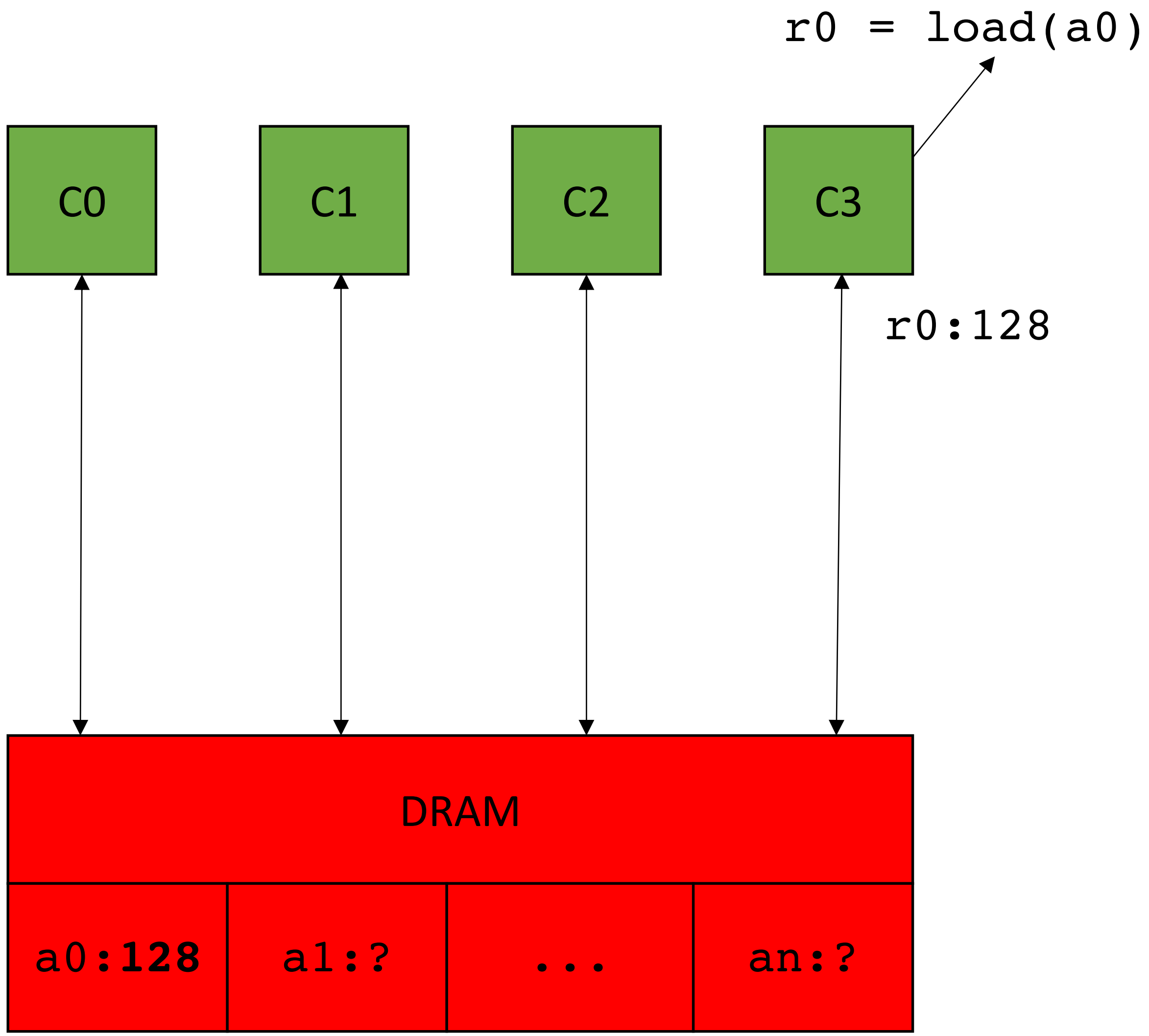
Problem solved!
Threads can communicate!



Main memory

Problem solved!
Threads can communicate!

reading a value takes ~200 cycles

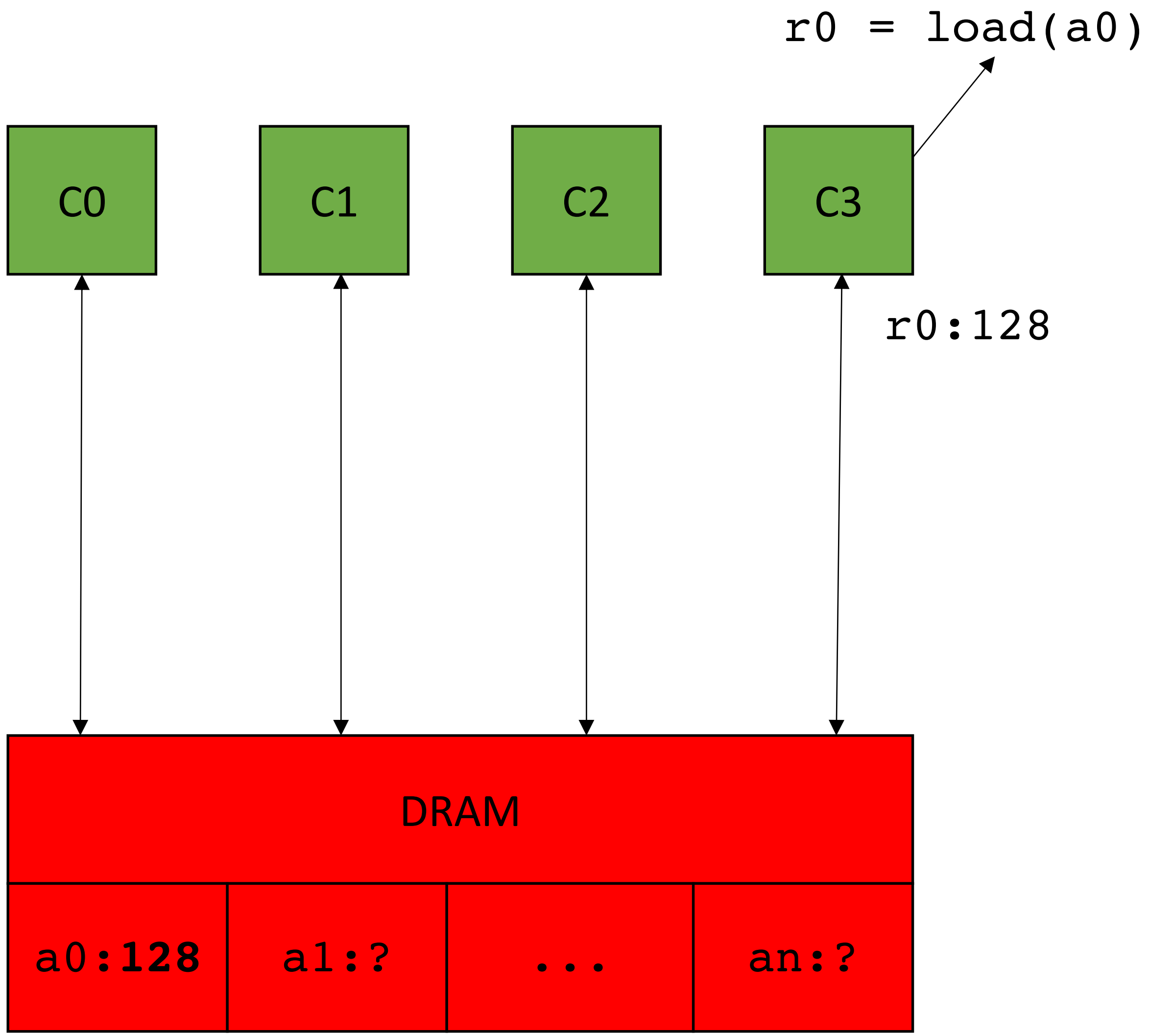


Main memory

Problem solved!
Threads can communicate!

reading a value takes ~200 cycles

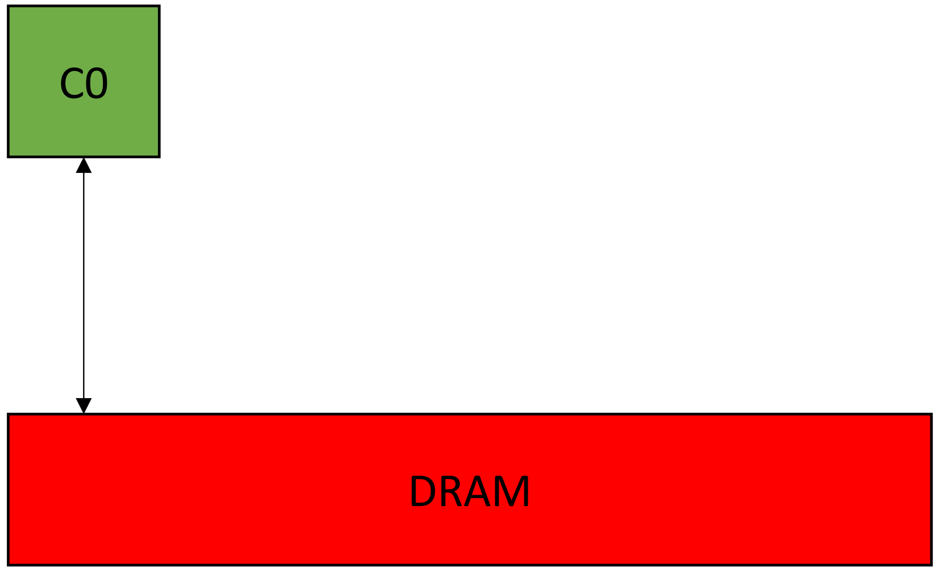
Bad for parallelism, even worse for sequential programs



Main memory

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

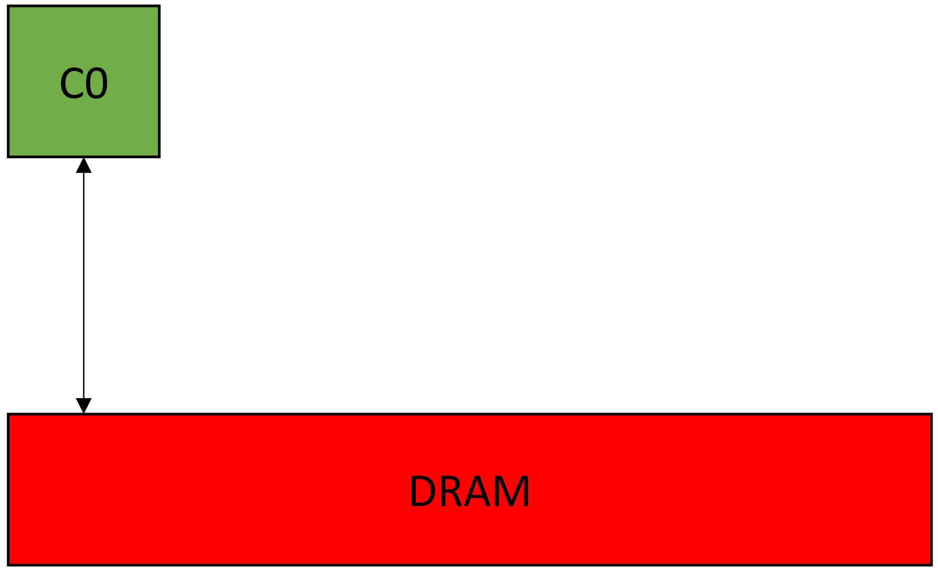


Main memory

```
int increment(int *a) {
   a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

200 cycles



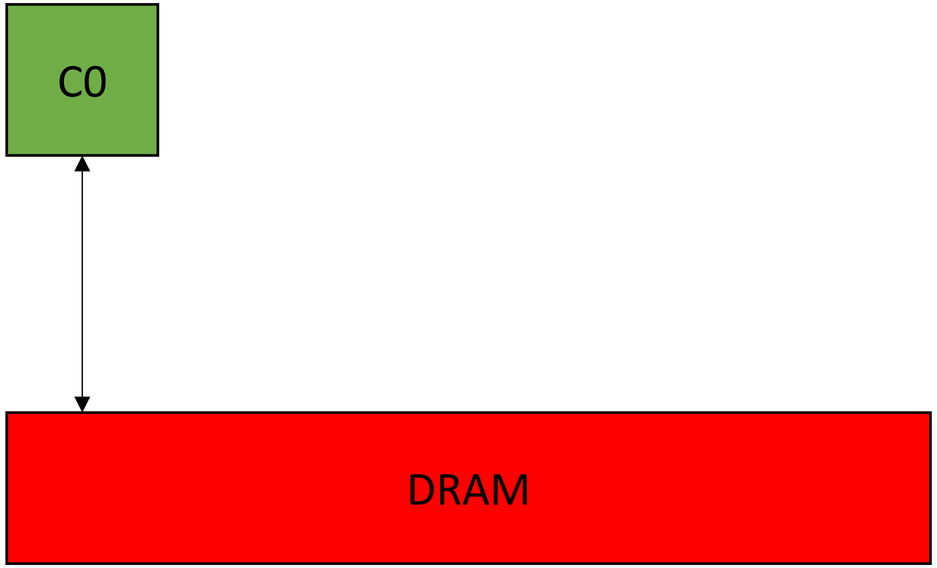
Main memory

```
int increment(int *a) {
   a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

200 cycles

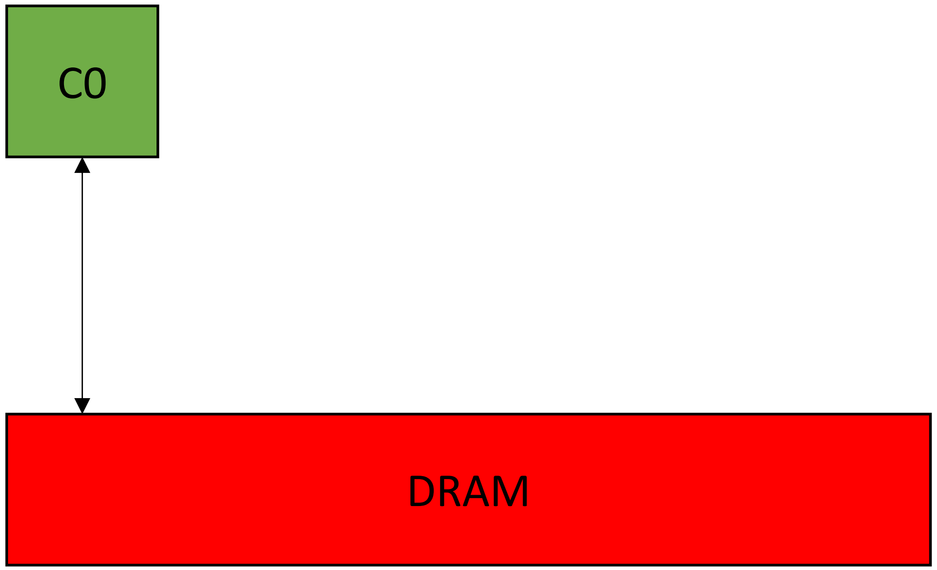
1 cycles



Main memory

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4          200 cycles
%6 = add nsw i32 %5, 1          1 cycles
store i32 %6, i32* %4           200 cycles
```



Main memory

```
int increment(int *a) {
   a[0]++;
}
```

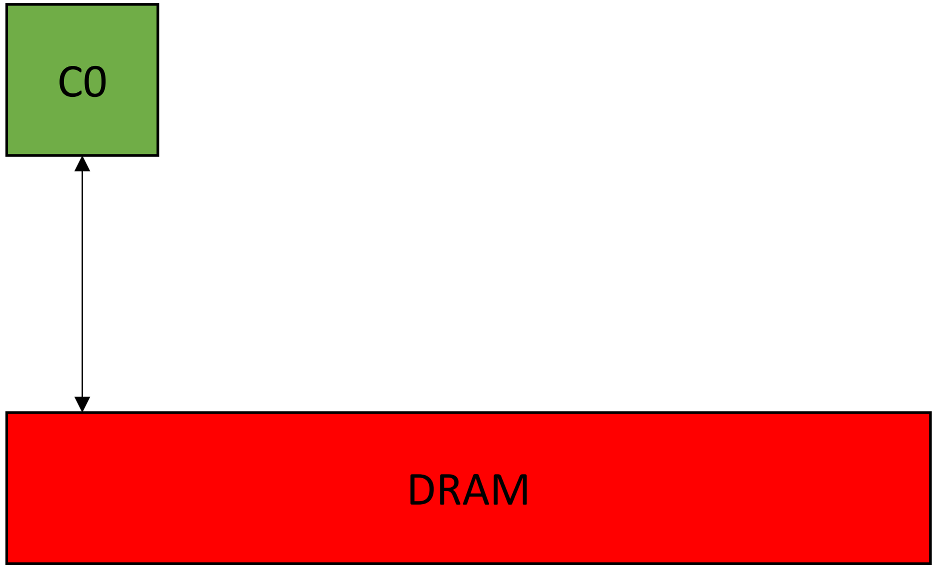
```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

200 cycles

1 cycles

200 cycles

401 cycles



Main memory

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

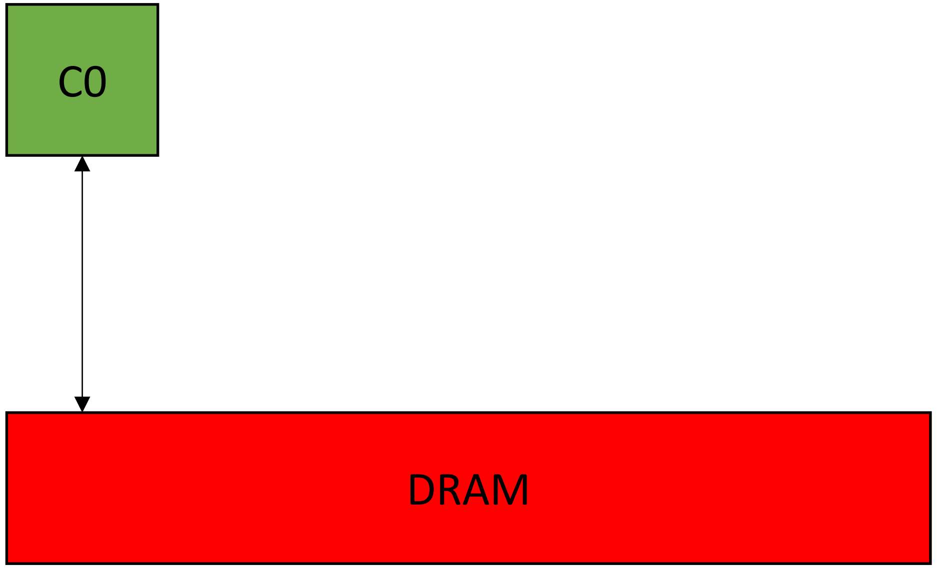
200 cycles

1 cycles

200 cycles

401 cycles

```
int x = 0;
for (int i = 0; i < 100; i++) {
    increment(&x);
}
```



Main memory

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

200 cycles

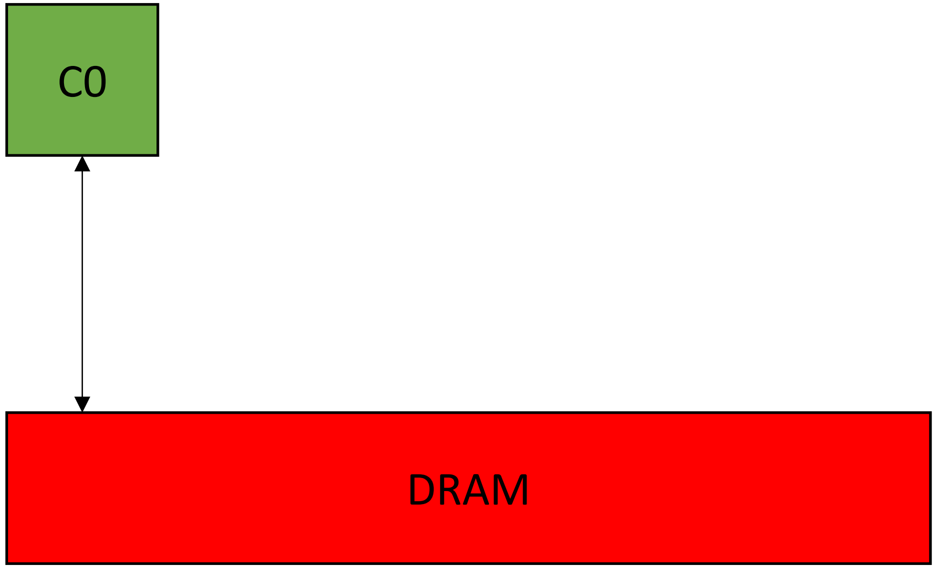
1 cycles

200 cycles

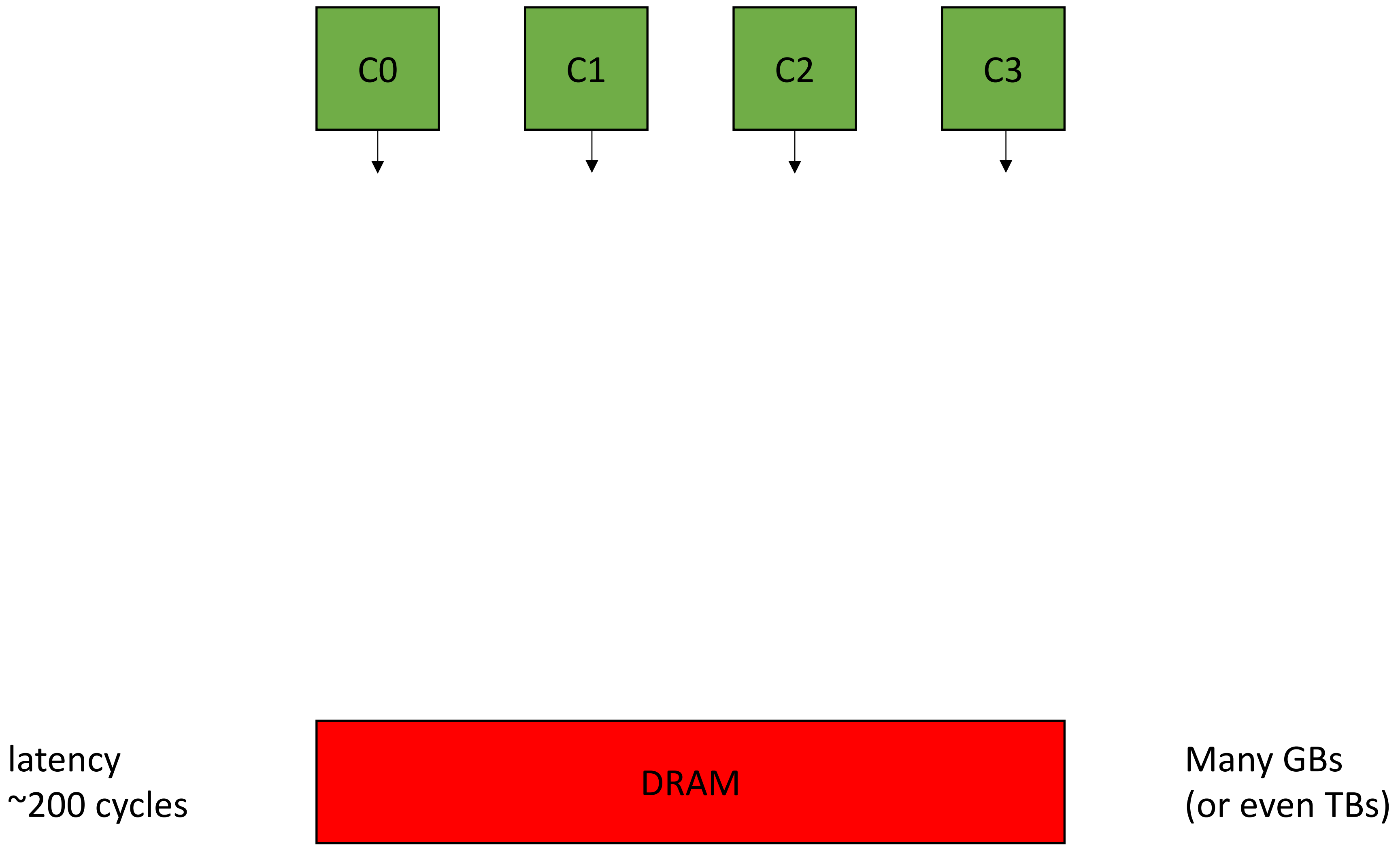
401 cycles

```
int x = 0;
for (int i = 0; i < 100; i++) {
  increment(&x);
}
```

40100 cycles!

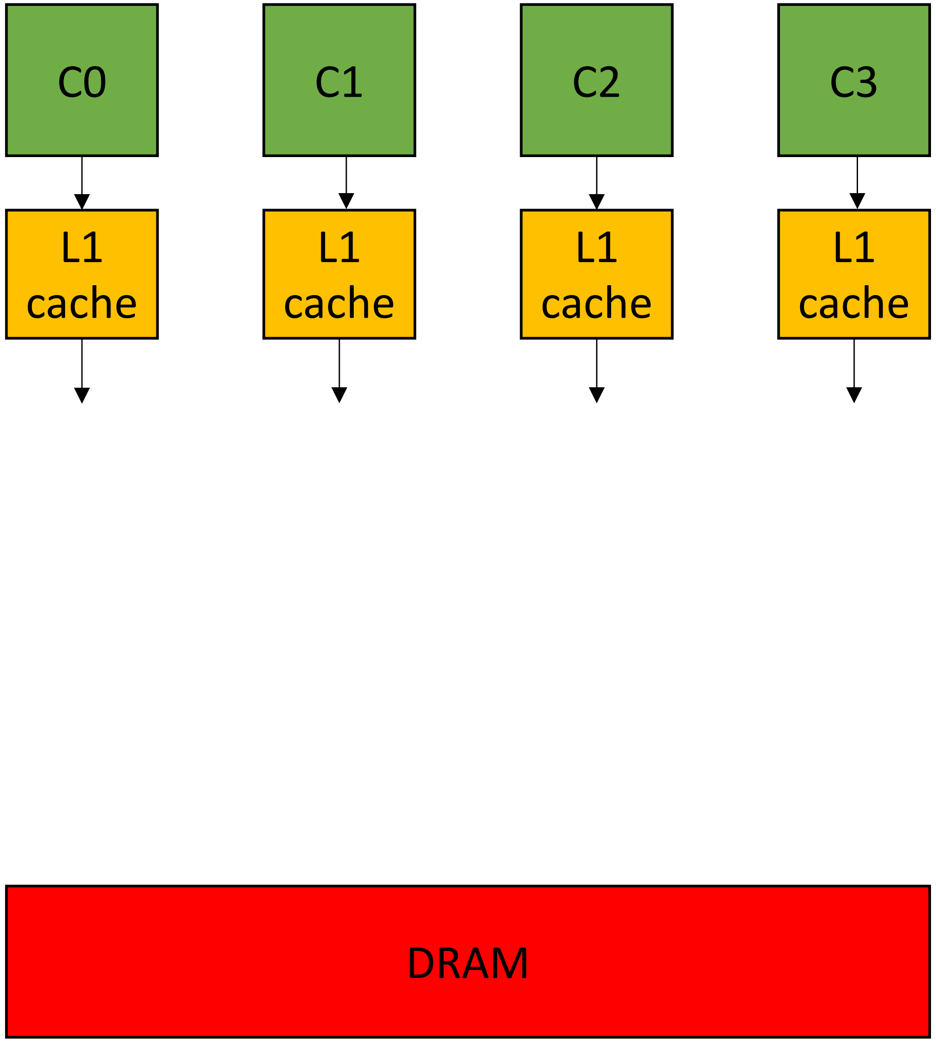


Caches



Caches

latency
~4 cycles

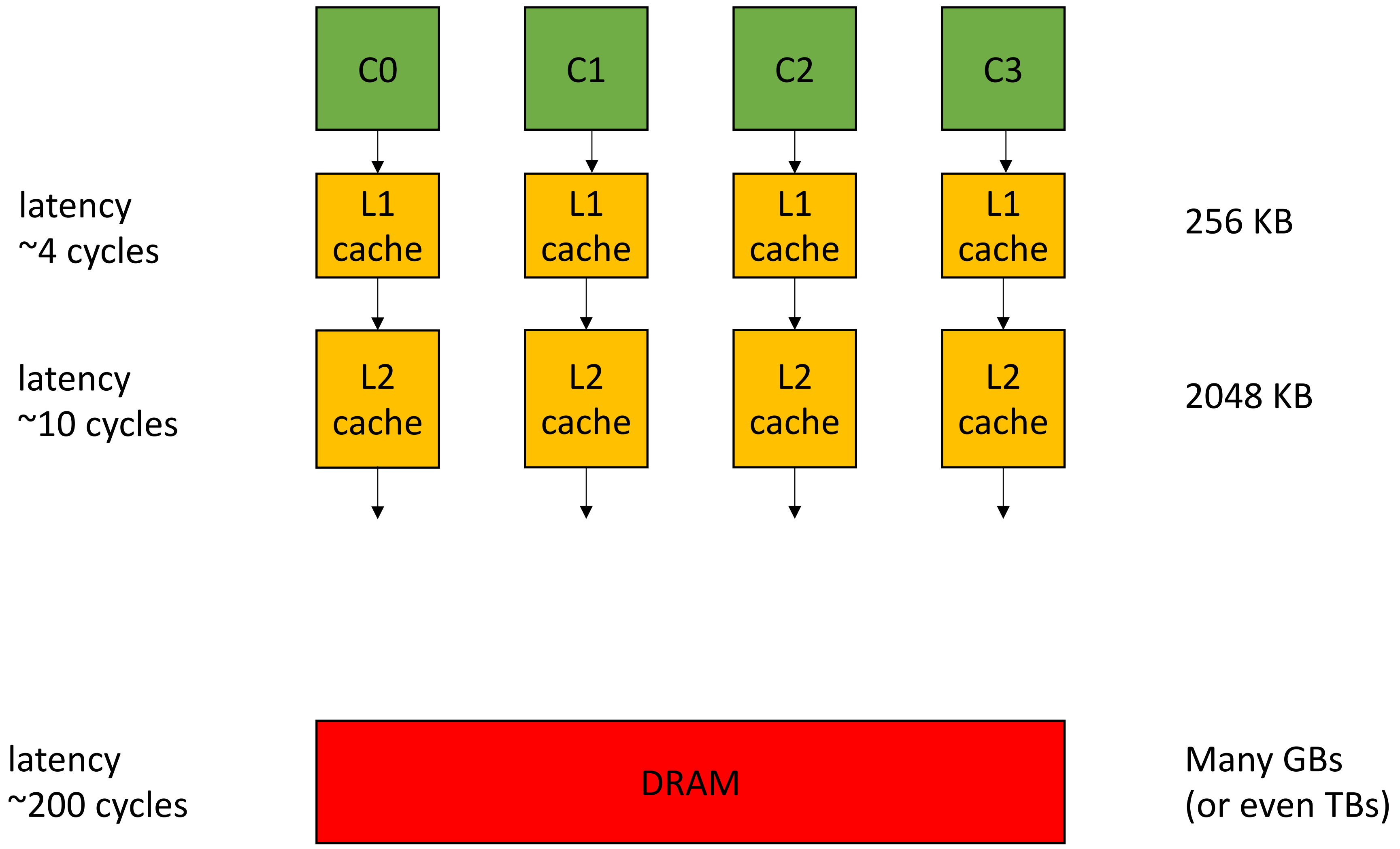


256 KB

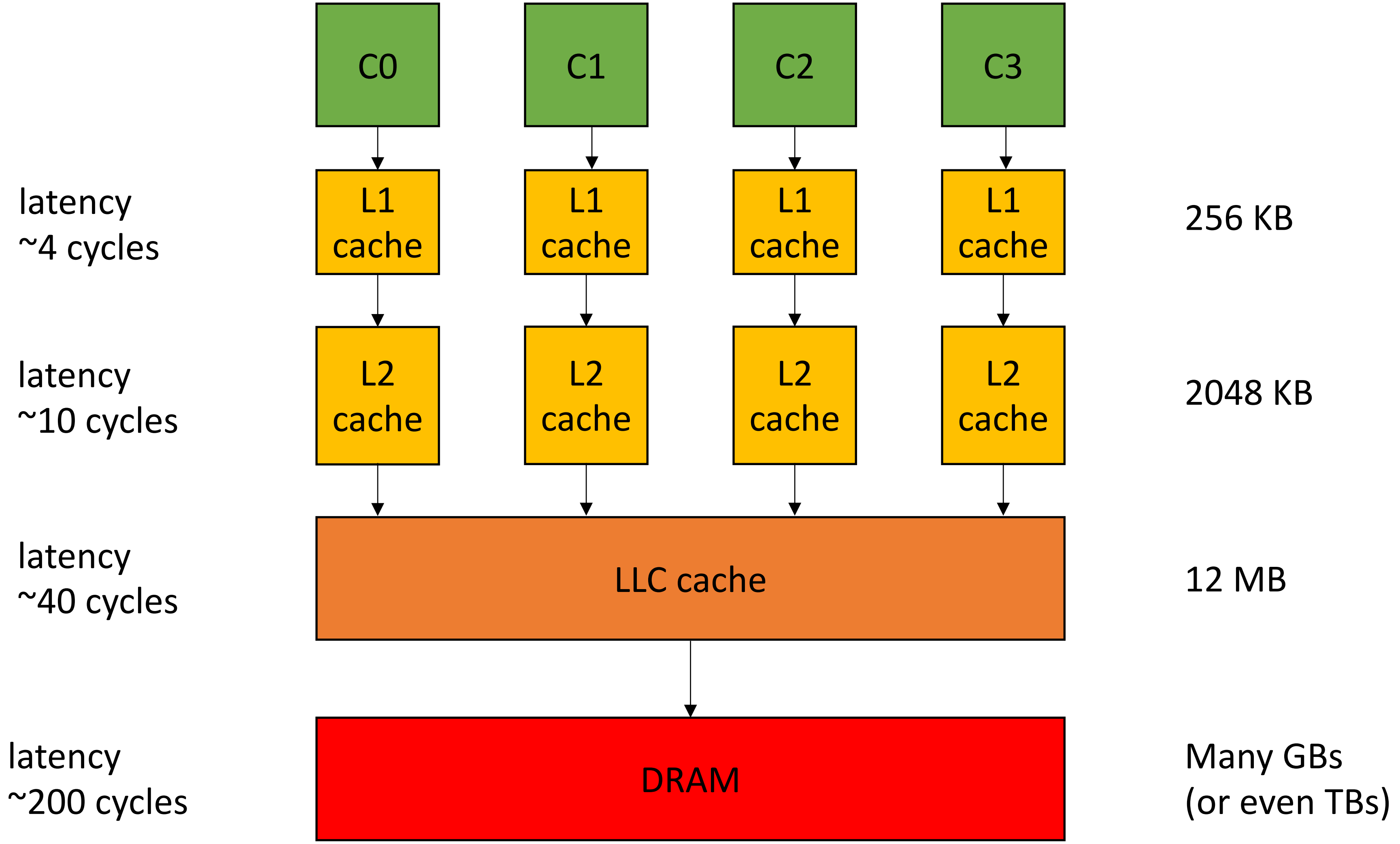
latency
~200 cycles

Many GBs
(or even TBs)

Caches


latency
~4 cycles
latency
~10 cycles
latency
~200 cycles
C0
C1
C2
C3
L1 cache
L1 cache
L1 cache
L1 cache
L2 cache
L2 cache
L2 cache
L2 cache
DRAM
256 KB
2048 KB
Many GBs
(or even TBs)

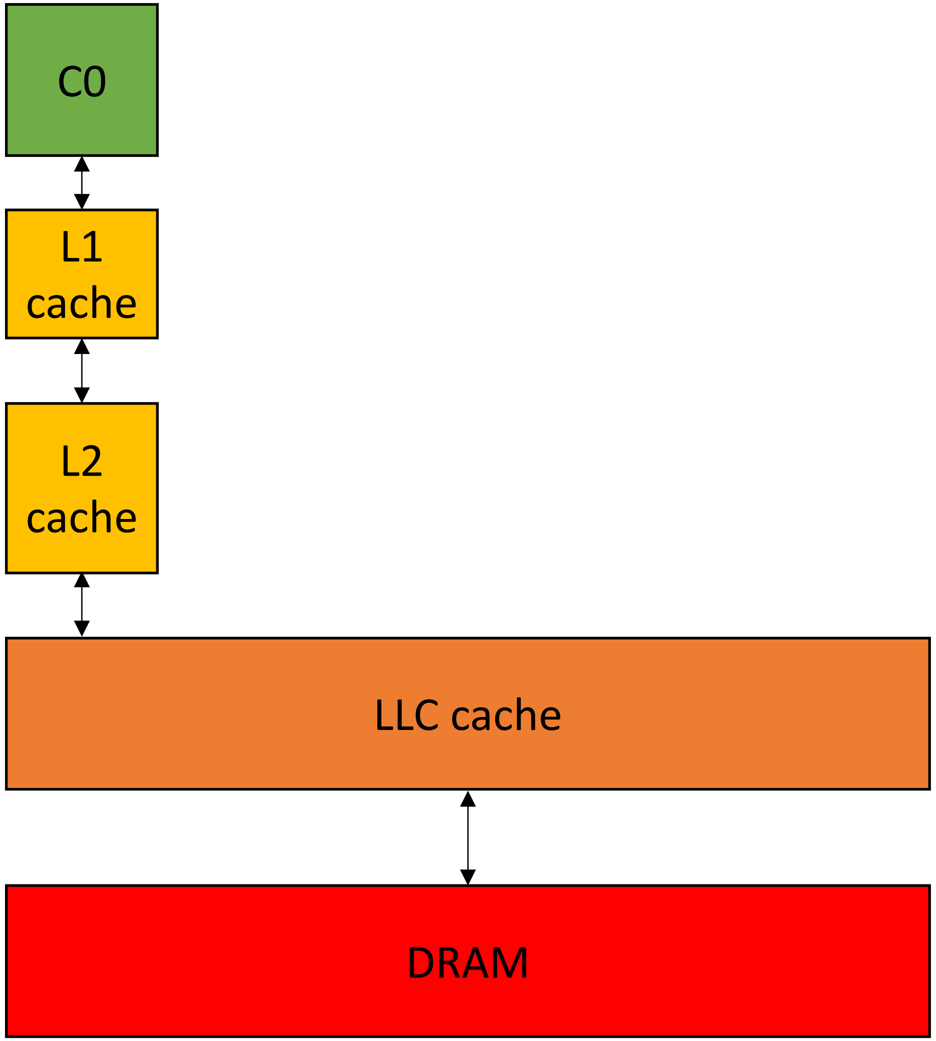
Caches


C0
C1
C2
C3
latency
~4 cycles
L1 cache
L1 cache
L1 cache
L1 cache
256 KB
latency
~10 cycles
L2 cache
L2 cache
L2 cache
L2 cache
2048 KB
latency
~40 cycles
LLC cache
12 MB
latency
~200 cycles
DRAM
Many GBs
(or even TBs)

Caches

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```



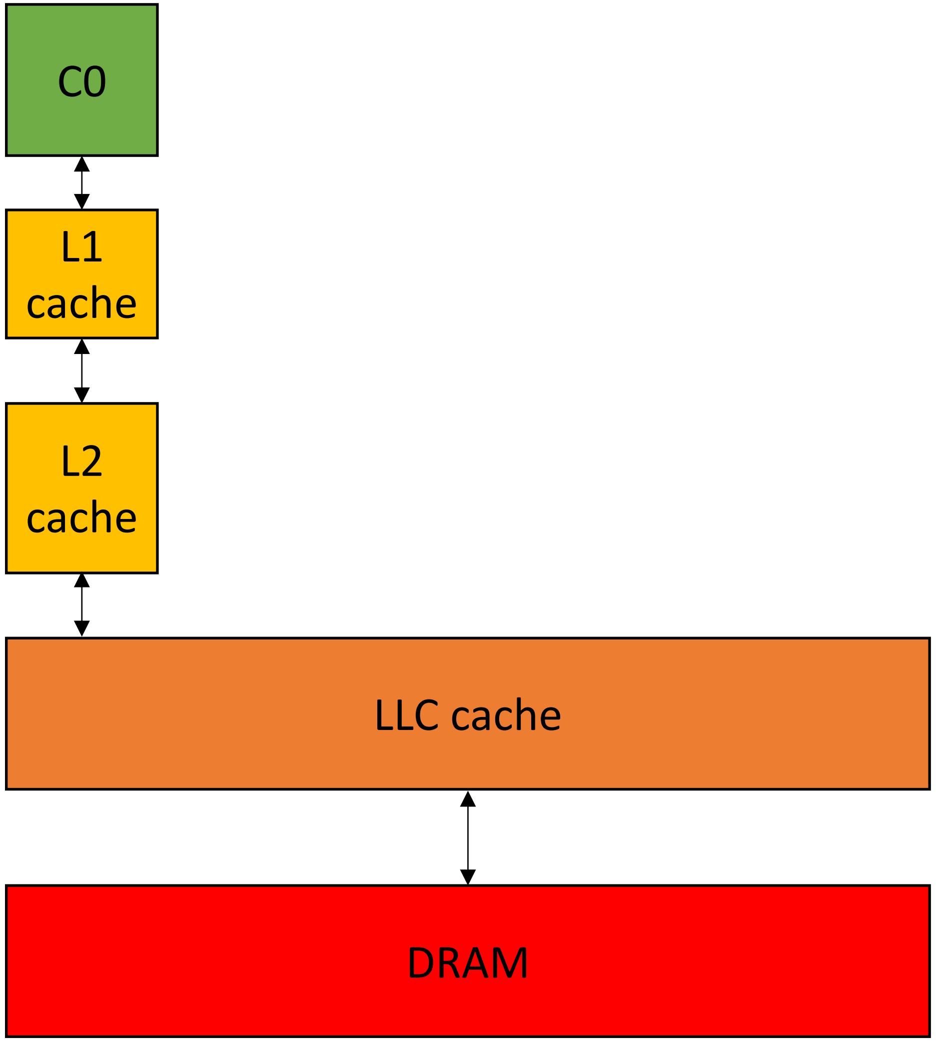
Caches

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

4 cycles

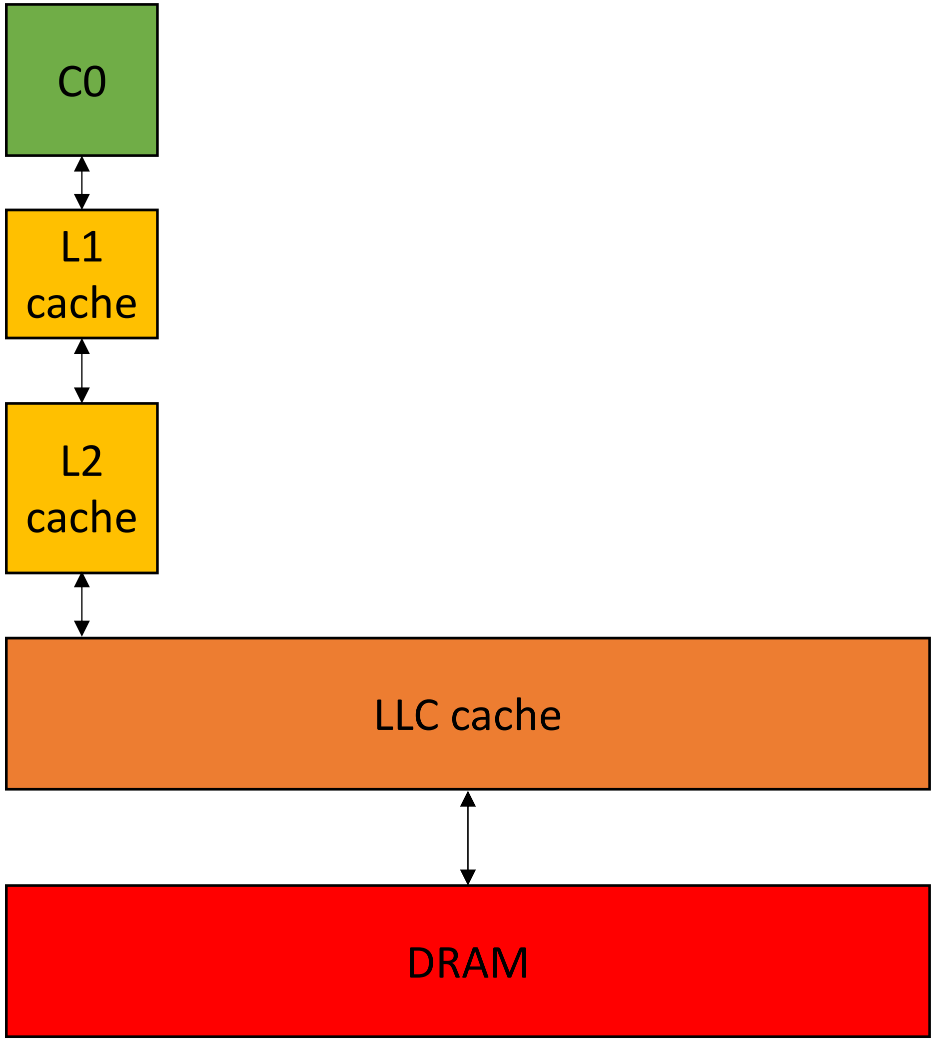
Assuming the value is in the cache!



Caches

```
int increment(int *a) {
    a[0]++;
}
```

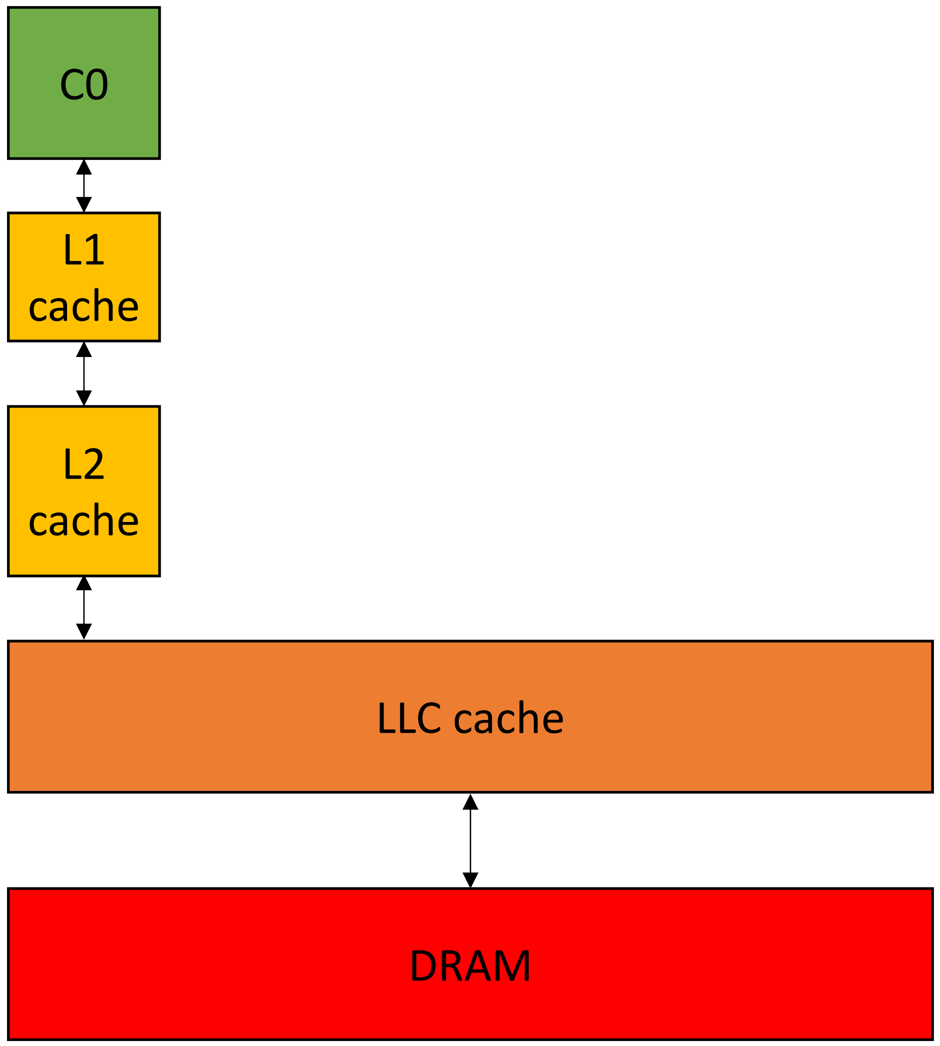
```
%5 = load i32, i32* %4          4 cycles
%6 = add nsw i32 %5, 1          1 cycles
store i32 %6, i32* %4
```



Caches

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4        4 cycles
%6 = add nsw i32 %5, 1        1 cycles
store i32 %6, i32* %4         4 cycles
```



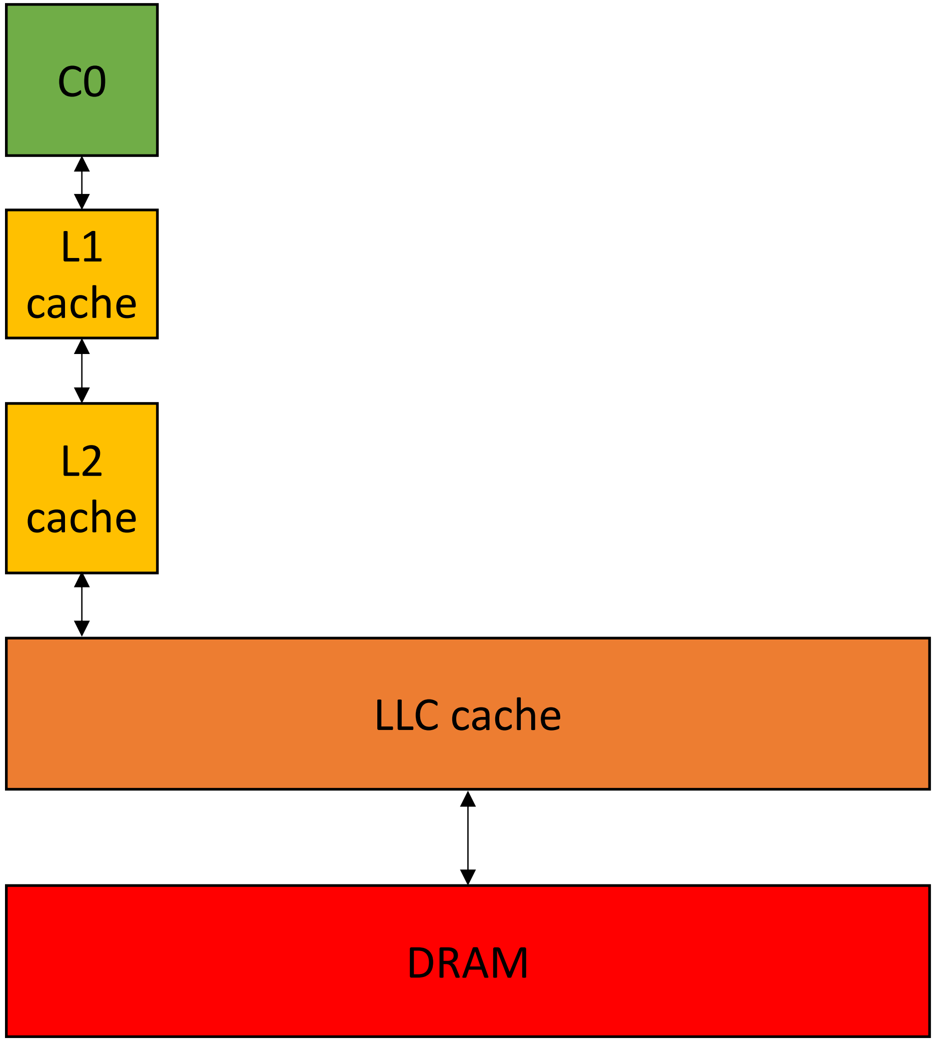
Caches

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

4 cycles
1 cycles
4 cycles

9 cycles!



Next lecture

- Cache associativity
- Cache coherence
- False Sharing

- Homework 1 released on Friday
- Make sure to do the quiz!!!