Lecture 1:

Why Parallelism?

Why Efficiency?

Parallel Computing
Stanford CS149, Fall 2023
Hello!

Prof. Kayvon

Prof. Olukotun

James

Minfei

Yasmine

Senyang

Zhenbang

Neha

Michael

Jensen

Shiv

Tom
One common definition

A parallel computer is a collection of processing elements that cooperate to solve problems quickly.

We care about performance and we care about efficiency.

We’re going to use multiple processors to get it.
DEMO 1

(CS149 Fall 2023’s first parallel program)
Speedup

One major motivation of using parallel processing: achieve a speedup

For a given problem:

\[
\text{speedup (using P processors)} = \frac{\text{execution time (using 1 processor)}}{\text{execution time (using P processors)}}
\]
Class observations from demo 1

- Communication limited the maximum speedup achieved
  - In the demo, the communication was telling each other the partial sums

- Minimizing the cost of communication improved speedup
  - Moved students (“processors”) closer together (or let them shout)
DEMO 2

(scaling up to four “processors”)
Class observations from demo 2

- Imbalance in work assignment limited speedup
  - Some students ("processors") ran out work to do (went idle), while others were still working on their assigned task

- Improving the distribution of work improved speedup
DEMO 3

(massively parallel execution)
Class observations from demo 3

- The problem I just gave you has a significant amount of communication compared to computation.

- Communication costs can dominate a parallel computation, severely limiting speedup.
Course theme 1:
Designing and writing parallel programs ... that scale!

- Parallel thinking
  1. Decomposing work into pieces that can safely be performed in parallel
  2. Assigning work to processors
  3. Managing communication/synchronization between the processors so that it does not limit speedup

- Abstractions/mechanisms for performing the above tasks
  - Writing code in popular parallel programming languages
Course theme 2:
Parallel computer hardware implementation: how parallel computers work

- Mechanisms used to implement abstractions efficiently
  - Performance characteristics of implementations
  - Design trade-offs: performance vs. convenience vs. cost

- Why do I need to know about hardware?
  - Because the characteristics of the machine really matter
    (recall speed of communication issues in earlier demos)
  - Because you care about efficiency and performance
    (you are writing parallel programs after all!)
Course theme 3: Thinking about efficiency

- **FAST ≠ EFFICIENT**

- Just because your program runs faster on a parallel computer, it does not mean it is using the hardware efficiently
  - Is 2x speedup on computer with 10 processors a good result?

- Programmer’s perspective: make use of provided machine capabilities

- HW designer’s perspective: choosing the right capabilities to put in system (performance/cost, cost = silicon area?, power?, etc.)
Course logistics
Getting started

- The course web site
  - https://cs149.stanford.edu

- Textbook
  - There is no course textbook (the internet is plenty good these days), also see the course web site for suggested references
Four programming assignments

Assignment 1: ISPC programming on multi-core CPUs

Assignment 2: scheduling a task graph

Assignment 3: Writing a renderer in CUDA on NVIDIA GPUs

Assignment 4: chat149: flash-attention transformers for a mini language model

Optional assignment 5: (Can be used to boost a prior grade)

- Topics TBD
- programming FPGAs,
- multi-core graph processing

Programming assignments can (optionally) be done with a partner.

We realize finding a partner can be stressful.

Fill out our partner request form by Thursday 11:59pm and we will find you a partner!
Written assignments

- Every two-weeks we will have a take-home written assignment graded on effort only

- Written assignments contain modified versions of previous exam questions, so they:
  - Give you practice with key course concepts
  - Provide practice for the style of questions you will see on an exam
Commenting and contributing to lectures

Why Parallelism? Why Efficiency?

Instruction level parallelism (ILP)

- Processors did in fact leverage parallel execution to make programs run faster, it was just invisible to the programmer

- Instruction level parallelism (ILP)
  - Idea: Instructions must appear to be executed in program order. BUT independent instructions can be executed simultaneously by a processor without impacting program correctness
  - Superscalar execution: processor dynamically finds independent instructions in an instruction sequence and executes them in parallel

The website supports commenting on a per-slide basis

It is computationally expensive for the processor to determine dependencies between instructions. The following PPT (slides 9/10) provides an example of how the number of checks grows with the number of instructions that are simultaneously dispatched: http://www.cs.cmu.edu/afs/cs/academic/class/16740-f15/www/lectures/11-superscalar-pipelining.pdf

This additional cost is likely one of the predominant reasons that ILP has plateaued at 4 simultaneous instructions. To circumvent this issue, architects have tried to force the compiler to solve the dependency issue using VLIW (very long instruction word). To summarize VLIW, if a processor contains 5 independent execution units, the compiler will have 5 operations in the "very long instruction word" that the processor will map to the 5 execution units: https://en.wikipedia.org/wiki/Very_long_instruction_word. This way dependency checking is the responsibility of software and not hardware.

I am not sure if VLIW has helped significantly pushed the four simultaneous instruction threshold through. If somebody knows, please share.

Question: The key phrase on this slide is that a processor must execute instructions in a manner "appears" as if they were executed in program order. This is a key idea in this class. What is program order?

And what does it mean for the results of a program's execution to appear as if instructions were executed in program order?

And finally... Why is the program order guarantee a useful one? (What if the results of execution were inconsistent with the results that would be obtained if the instructions were executed in program order?)

A programmer might write something like the code below.

```c
void main()
{
    x = a + b;
    print(x);
    y = c + d;
    print(y);
}
```
Participation (comments)

- You are asked to submit one well-thought-out comment per lecture
  - Only two comments per week
  - We expect you to submit “within the same calendar week” as the lectures (no credit for submitting all comments at the end of the quarter when you are studying for the final)

- Why do we ask you to write?
  - Because writing is a way many good architects and systems designers force themselves to think (explaining clearly and thinking clearly are highly correlated!)

- But take it seriously, there is a participation component to the final grade
What we are looking for in comments

- **Try to explain the slide (as if you were trying to teach your classmate while studying for an exam)**
  - “The instructor said this, but if you think about it this way instead it makes much more sense…”

- **Explain what is confusing to you:**
  - “What I’m totally confused by here was…”

- **Challenge classmates with a question**
  - For example, make up a question you think might be on an exam.

- **Provide a link to an alternate explanation**
  - “This site has a really good description of how multi-threading works…”

- **Mention real-world examples**
  - For example, describe all the parallel hardware components in the PS5

- **Constructively respond to another student’s comment or question**
  - “@segfault23, are you sure that is correct? I thought that Prof. Kayvon said…”

- **It is OKAY (and even encouraged) to address the same topic (or repeat someone else’s summary, explanation or idea) in your own words**
  - “@funkysenior23’s point is that the overhead of communication…”
Grades

58% Programming assignments (4)

8% Written assignments (5)

16% Midterm exam
  - An evening in-person exam on Nov 14th

16% Final exam
  - During the university-assigned slot: Dec 14th, 3:30pm

2% Asynchronous participation (website comments)
Late days

- You get **eight late days** for the quarter
  - For use on programming and written assignments

- The idea of late days is to give you the flexibility to handle almost all events that arise throughout the quarter
  - Work from other classes, failing behind, most illnesses, athletic/extra curricular events...
  - We expect to give extra late days only under exceptional circumstances

- Requests for additional late days for exceptional circumstances should be made days in advance if possible.
Why parallelism?
Some historical context: why avoid parallel processing?

- Single-threaded CPU performance doubling ~ every 18 months
- Implication: working to parallelize your code was often not worth the time
  - Software developer does nothing, code gets faster next year. Woot!

Image credit: Olukutun and Hammond, ACM Queue 2005
Until ~15 years ago: two significant reasons for processor performance improvement

1. Exploiting instruction-level parallelism (superscalar execution)

2. Increasing CPU clock frequency
What is a computer program?
Here is a program written in C

```c
int main(int argc, char** argv) {
    int x = 1;
    for (int i=0; i<10; i++) {
        x = x + x;
    }
    printf("%d\n", x);
    return 0;
}
```
What is a program? (from a processor’s perspective)

A program is just a list of processor instructions!

```c
int main(int argc, char** argv) {
    int x = 1;
    for (int i=0; i<10; i++) {
        x = x + x;
    }
    printf(“%d\n”, x);
    return 0;
}
```

Compile code

```
_main:
100000f10:  pushq  %rbp
100000f11:  movq %rsp, %rbp
100000f14:  subq $32, %rsp
100000f18:  movl $0, -4(%rbp)
100000f1f:  movl %edi, -8(%rbp)
100000f22:  movq %rsi, -16(%rbp)
100000f26:  movl $1, -20(%rbp)
100000f2d:  movl $0, -24(%rbp)
100000f34:  cmpl $10, -24(%rbp)
100000f38:  jge 23 <_main+0x45>
100000f3e:  movl -20(%rbp), %eax
100000f41:  addl -20(%rbp), %eax
100000f44:  movl %eax, -20(%rbp)
100000f47:  movl -24(%rbp), %eax
100000f4a:  addl $1, %eax
100000f4d:  movl %eax, -24(%rbp)
100000f50:  jmp -33 <_main+0x24>
100000f55:  leaq 58(%rip), %rdi
100000f5c:  movl -20(%rbp), %esi
100000f5f:  movb $0, %al
100000f61:  callq 14
100000f66:  xorl %esi, %esi
100000f68:  movl %eax, -28(%rbp)
100000f6b:  movl %esi, %eax
100000f6d:  addq $32, %rsp
100000f71:  popq %rbp
100000f72:  rets
```
Kind of like the instructions in a recipe for your favorite meals

Mmm, carne asada

Instructions

1. In a large mixing bowl combine orange juice, olive oil, cilantro, lime juice, lemon juice, white wine vinegar, cumin, salt and pepper, jalapeno, and garlic; whisk until well combined.
2. Reserve ½ cup of the marinade; cover the rest and refrigerate.
3. Combine remaining marinade and steak in a large resealable freezer bag; seal and refrigerate for at least 2 hours, or overnight.
4. Preheat grill to HIGH heat.
5. Remove steak from marinade and lightly pat dry with paper towels.
6. Add steak to the preheated grill and cook for another 6 to 8 minutes per side, or until desired doneness. **Note that flank steak tastes best when cooked to rare or medium rare because it's a lean cut of steak.**
7. Remove from heat and let rest for 10 minutes. Thinly slice steak against the grain, garnish with reserved cilantro mixture, and serve.
What does a processor do?
A processor executes instructions

Professor Kayvon’s Very Simple Processor

Fetch/Decode

Determine what instruction to run next

ALU (Execution Unit)

Execution unit: performs the operation described by an instruction, which may modify values in the processor’s registers or the computer’s memory

Execution Context

Registers: maintain program state: store value of variables used as inputs and outputs to operations

Register 0 (R0)
Register 1 (R1)
Register 2 (R2)
Register 3 (R3)
One example instruction: add two numbers

Step 1:
Processor gets next program instruction from memory (figure out what the processor should do next)

\[ \text{add } R0 \leftarrow R0, R1 \]

"Please add the contents of register R0 to the contents of register R1 and put the result of the addition into register R0"

Step 2:
Get operation inputs from registers

Contents of R0 input to execution unit: 32
Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:
Execution unit performs arithmetic, the result is: 96
One example instruction: add two numbers

Step 1:
Processor gets next program instruction from memory
(figure out what the processor should do next)

```
add R0 ← R0, R1
```

"Please add the contents of register R0 to the contents of register R1 and put the result of the addition into register R0"

Step 2:
Get operation inputs from registers

- Contents of R0 input to execution unit: 32
- Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:

Execution unit performs arithmetic, the result is: 96

Step 4:
Store result 96 back to register R0
Execute program

My very simple processor: executes one instruction per clock

```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
...
...
...
...
st   addr[r2], r0
```
Execute program

My very simple processor: executes one instruction per clock
Execute program

My very simple processor: executes one instruction per clock

ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
...
...
...
st addr[r2], r0
Execute program

My very simple processor: executes one instruction per clock

Fetch/Decode

Execution Unit (ALU)

Execution Context

ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...  
...  
...  
...  
...  
...  
...  
st   addr[r2], r0
Review of how computers work...

What is a computer program? (from a processor’s perspective)

*It is a list of instructions to execute!*

What is an instruction?

*It describes an operation for a processor to perform.*

*Executing an instruction typically modifies the computer’s state.*

What do I mean when I talk about a computer’s “state”?

*The values of program data, which are stored in a processor’s registers or in memory.*
Let's consider a very simple piece of code

\[ a = x^2 + y^2 + z^2 \]

Assume register \( R0 = x, R1 = y, R2 = z \)

1. \( \text{mul } R0, R0, R0 \)  
2. \( \text{mul } R1, R1, R1 \)  
3. \( \text{mul } R2, R2, R2 \)  
4. \( \text{add } R0, R0, R1 \)  
5. \( \text{add } R3, R0, R2 \)

This program has five instructions, so it will take five clocks to execute, correct? Can we do better?

\( R3 \) now stores value of program variable ‘\( a \)’
What if up to two instructions can be performed at once?

\[ a = x^2 + y^2 + z^2 \]

Assume register 
\( R0 = x, R1 = y, R2 = z \)

1. \( \text{mul } R0, R0, R0 \)
2. \( \text{mul } R1, R1, R1 \)
3. \( \text{mul } R2, R2, R2 \)
4. \( \text{add } R0, R0, R1 \)
5. \( \text{add } R3, R0, R2 \)

R3 now stores value of program variable ‘a’
What if up to two instructions can be performed at once?

\[ a = x^2 + y^2 + z^2 \]

Assume register
\( R0 = x, R1 = y, R2 = z \)

1. \( \text{mul } R0, R0, R0 \)
2. \( \text{mul } R1, R1, R1 \)
3. \( \text{mul } R2, R2, R2 \)
4. \( \text{add } R0, R0, R1 \)
5. \( \text{add } R3, R0, R2 \)

Time
1. 1. \( \text{mul } R0, R0, R0 \)
2. 2. \( \text{mul } R1, R1, R1 \)
3. 3. \( \text{mul } R2, R2, R2 \)
4. 4. \( \text{add } R0, R0, R1 \)
5. 5. \( \text{add } R3, R0, R2 \)

\( R3 \) now stores value of program variable ‘a’
What does it mean for our parallel to scheduling to “respect program order”?
What about three instructions at once?

\[ a = x^2 + y^2 + z^2 \]

Assume register
\[ R0 = x, R1 = y, R2 = z \]

1. mul R0, R0, R0
2. mul R1, R1, R1
3. mul R2, R2, R2
4. add R0, R0, R1
5. add R3, R0, R2

R3 now stores value of program variable ‘a’
What about three instructions at once?

\[
a = x^2 + y^2 + z^2
\]

Assume register

\[R_0 = x, R_1 = y, R_2 = z\]

1. mul R0, R0, R0
2. mul R1, R1, R1
3. mul R2, R2, R2
4. add R0, R0, R1
5. add R3, R0, R2

R3 now stores value of program variable ‘a’
Instruction level parallelism (ILP) example

- ILP = 3

\[ a = x^2 + y^2 + z^2 \]
Superscalar processor execution

\[ a = x^2 + y^2 + z^2 \]

Assume register
\[ R0 = x, R1 = y, R2 = z \]

\begin{verbatim}
1 mul R0, R0, R0
2 mul R1, R1, R1
3 mul R2, R2, R2
4 add R0, R0, R1
5 add R3, R0, R2
\end{verbatim}

Idea #1:

**Superscalar execution**: processor automatically finds independent instructions in an instruction sequence and executes them in parallel on multiple execution units!

In this example: instructions 1, 2, and 3 can be executed in parallel without impacting program correctness (on a superscalar processor that determines that the lack of dependencies exists)

But instruction 4 must be executed after instructions 1 and 2

And instruction 5 must be executed after instruction 4

*Or the compiler finds independent instructions at compile time and explicitly encodes dependencies in the compiled binary.*
Superscalar processor

This processor can decode and execute up to two instructions per clock
Aside:
Old Intel Pentium 4 CPU

A more complex example

Program (sequence of instructions)

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
<th>Value during execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>a = 2</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>b = 4</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>tmp2 = a + b</td>
<td>// 6</td>
</tr>
<tr>
<td>03</td>
<td>tmp3 = tmp2 + a</td>
<td>// 8</td>
</tr>
<tr>
<td>04</td>
<td>tmp4 = b + b</td>
<td>// 8</td>
</tr>
<tr>
<td>05</td>
<td>tmp5 = b * b</td>
<td>// 16</td>
</tr>
<tr>
<td>06</td>
<td>tmp6 = tmp2 + tmp4</td>
<td>// 14</td>
</tr>
<tr>
<td>07</td>
<td>tmp7 = tmp5 + tmp6</td>
<td>// 30</td>
</tr>
</tbody>
</table>

```python
if (tmp3 > 7):
    print tmp3
else:
    print tmp7
```
Diminishing returns of superscalar execution

Most available ILP is exploited by a processor capable of issuing four instructions per clock (Little performance benefit from building a processor that can issue more)

Source: Culler & Singh (data from Johnson 1991)
Moore’s Law: The number of transistors on microchips doubles every two years

Moore’s law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.
ILP tapped out + end of frequency scaling

Processor clock rate stops increasing

No further benefit from ILP

Transistor density
Clock frequency
Power
Instruction-level parallelism (ILP)

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

Image credit: “The free Lunch is Over” by Herb Sutter, Dr. Dobbs 2005
The “power wall”

Power consumed by a transistor:

Dynamic power $\propto$ capacitive load $\times$ voltage$^2$ $\times$ frequency

Static power: transistors burn power even when inactive due to leakage

High power = high heat

Power is a critical design constraint in modern processors

<table>
<thead>
<tr>
<th>Device</th>
<th>TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple M1 laptop:</td>
<td>13W</td>
</tr>
<tr>
<td>Intel Core i9 10900K (in desktop CPU):</td>
<td>95W</td>
</tr>
<tr>
<td>NVIDIA RTX 4090 GPU</td>
<td>450W</td>
</tr>
<tr>
<td>Mobile phone processor</td>
<td>$1/2$ - 2W</td>
</tr>
<tr>
<td>World’s fastest supercomputer</td>
<td>megawatts</td>
</tr>
<tr>
<td>Standard microwave oven</td>
<td>900W</td>
</tr>
</tbody>
</table>

Source: Intel, NVIDIA, Wikipedia, Top500.org
Power draw as a function of clock frequency

Dynamic power \propto \text{capacitive load} \times \text{voltage}^2 \times \text{frequency}

Static power: transistors burn power even when inactive due to leakage

Maximum allowed frequency determined by processor’s core voltage

Single-core performance scaling

The rate of single-instruction stream performance scaling has decreased (almost to zero)

1. Frequency scaling limited by power
2. ILP scaling tapped out

Architects are now building faster processors by adding more execution units that run in parallel
(Or units that are specialized for a specific task: like graphics, or audio/video playback)

Software must be written to be parallel to see performance gains. No more free lunch for software developers!
Example: multi-core CPU

Intel “Comet Lake” 10th Generation Core i9 10-core CPU (2020)
One thing you will learn in this course

- How to write code that efficiently uses the resources in a modern multi-core CPU

- Example: assignment 1 (coming up!)
  - Running on a quad-core Intel CPU
  - Four CPU cores
  - AVX SIMD vector instructions + hyper-threading
  - Baseline: single-threaded C program compiled with -O3
  - Parallelized program that uses all parallel execution resources on this CPU...

~32-40x faster!

We’ll talk about these terms next time!
AMD Ryzen Threadripper 3990X
64 cores, 4.3 GHz

Four 8-core chiplets
NVIDIA AD102 GPU

GeForce RTX 4090 (2022)
76 billion transistors

18,432 fp32 multipliers organized in 144 processing blocks (called SMs)
GPU-accelerated supercomputing

Frontier (at Oak Ridge National Lab)
(world’s #1 in Fall 2022)
9472 x 64 core AMD CPUs (606,208 CPU cores)
37,888 Radeon GPUs
21 Megawatts
Mobile parallel processing

Power constraints also heavily influence the design of mobile systems

Apple A15 Bionic (in iPhone 13, 14)
15 billion transistors
6-core CPU
Multi-core GPU

5 GPU blocks
2 “big” CPU cores
4 “small” CPU cores
Mobile parallel processing

Raspberry Pi 3
Quad-core ARM A53 CPU
But in modern computing, software must be more than just parallel... IT MUST ALSO BE EFFICIENT
Parallel + specialized HW

- Achieving high efficiency will be a key theme in this class

- We will discuss how modern systems not only use many processing units, but also utilize specialized processing units to achieve high levels of power efficiency
Specialized processing is ubiquitous in mobile systems

Apple A15 Bionic (in iPhone 13, 14)

15 billion transistors

6-core GPU
  2 “big” CPU cores
  4 “small” CPU cores

Apple-designed multi-core GPU
Neural Engine (NPU) for DNN acceleration +
Image/video encode/decode processor +
Motion (sensor) processor
Specialization for datacenter-scale applications

Google TPU pods
TPU = Tensor Processing Unit: specialized processor for ML computations

Image Credit: TechInsights Inc.
Specialized hardware to accelerate DNN inference/training

- Google TPU3
- Graphcore IPU
- Apple Neural Engine
- Intel Deep Learning Inference Accelerator
- SambaNova Cardinal SN10
- Cerebras Wafer Scale Engine
- Ampere GPU with Tensor Cores
Achieving efficient processing almost always comes down to accessing data efficiently.
What is memory?
A program’s memory address space

- A computer’s memory is organized as an array of bytes

- Each byte is identified by its “address” in memory (its position in this array)
  (We’ll assume memory is byte-addressable)

  “The byte stored at address 0x8 has the value 32.”

  “The byte stored at address 0x10 (16) has the value 128.”

In the illustration on the right, the program’s memory address space is 32 bytes in size
(so valid addresses range from 0x0 to 0xF)
Load: an instruction for accessing the contents of memory

Professor Kayvon’s Very Simple Processor

Fetch/Decode

ALU (Execution Unit)

Execution Context

R0: 96
R1: 64
R2: 0xff681080
R3: 0x80486412

ld R0 ← mem[R2]

“Please load the four-byte value in memory starting from the address stored by register R2 and put this value into register R0.”

Memory

... 0xff68107c: 1024
     0xff681080: 42
     0xff681084: 32
     0xff681088: 0
     ...
Terminology

- **Memory access latency**
  - The amount of time it takes the memory system to provide data to the processor
  - Example: 100 clock cycles, 100 nsec

![Diagram showing data request and memory latency](image)
Stalls

- A processor “stalls” (can’t make progress) when it cannot run the next instruction in an instruction stream because future instructions depend on a previous instruction that is not yet complete.

- Accessing memory is a major source of stalls
  
  ```
  ld r0 mem[r2]
  ld r1 mem[r3]
  add r0, r0, r1
  ```

  Dependency: cannot execute ‘add’ instruction until data from mem[r2] and mem[r3] have been loaded from memory

- Memory access times ~ 100’s of cycles
  - Memory “access time” is a measure of latency
What are caches?

- Recall memory is just an array of values
- And a processor has instructions for moving data from memory into registers (load) and storing data from registers into memory (store)

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>16</td>
</tr>
<tr>
<td>0x1</td>
<td>255</td>
</tr>
<tr>
<td>0x2</td>
<td>14</td>
</tr>
<tr>
<td>0x3</td>
<td>0</td>
</tr>
<tr>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>0x6</td>
<td>6</td>
</tr>
<tr>
<td>0x7</td>
<td>0</td>
</tr>
<tr>
<td>0x8</td>
<td>32</td>
</tr>
<tr>
<td>0x9</td>
<td>48</td>
</tr>
<tr>
<td>0xA</td>
<td>255</td>
</tr>
<tr>
<td>0xB</td>
<td>255</td>
</tr>
<tr>
<td>0xC</td>
<td>255</td>
</tr>
<tr>
<td>0xD</td>
<td>0</td>
</tr>
<tr>
<td>0xE</td>
<td>0</td>
</tr>
<tr>
<td>0xF</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>128</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x1F</td>
<td>0</td>
</tr>
</tbody>
</table>
What are caches?

- A cache is a hardware implementation detail that does not impact the output of a program, only its performance.
- Cache is on-chip storage that maintains a copy of a subset of the values in memory.
- If an address is stored “in the cache” the processor can load/store to this address more quickly than if the data resides only in DRAM.
- Caches operate at the granularity of “cache lines”.

In the figure, the cache:
- Has a capacity of 2 lines.
- Each line holds 4 bytes of data.

Implementation of memory abstraction:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>16</td>
</tr>
<tr>
<td>0x1</td>
<td>255</td>
</tr>
<tr>
<td>0x2</td>
<td>14</td>
</tr>
<tr>
<td>0x3</td>
<td>0</td>
</tr>
<tr>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>0x6</td>
<td>6</td>
</tr>
<tr>
<td>0x7</td>
<td>0</td>
</tr>
<tr>
<td>0x8</td>
<td>32</td>
</tr>
<tr>
<td>0x9</td>
<td>48</td>
</tr>
<tr>
<td>0xA</td>
<td>255</td>
</tr>
<tr>
<td>0xB</td>
<td>255</td>
</tr>
<tr>
<td>0xC</td>
<td>255</td>
</tr>
<tr>
<td>0xD</td>
<td>0</td>
</tr>
<tr>
<td>0xE</td>
<td>0</td>
</tr>
<tr>
<td>0xF</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>128</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x1F</td>
<td>0</td>
</tr>
</tbody>
</table>
Cache example 1

Array of 16 bytes in memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>16</td>
</tr>
<tr>
<td>0x1</td>
<td>255</td>
</tr>
<tr>
<td>0x2</td>
<td>14</td>
</tr>
<tr>
<td>0x3</td>
<td>0</td>
</tr>
<tr>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>0x6</td>
<td>6</td>
</tr>
<tr>
<td>0x7</td>
<td>0</td>
</tr>
<tr>
<td>0x8</td>
<td>32</td>
</tr>
<tr>
<td>0x9</td>
<td>48</td>
</tr>
<tr>
<td>0xA</td>
<td>255</td>
</tr>
<tr>
<td>0xB</td>
<td>255</td>
</tr>
<tr>
<td>0xC</td>
<td>255</td>
</tr>
<tr>
<td>0xD</td>
<td>0</td>
</tr>
<tr>
<td>0xE</td>
<td>0</td>
</tr>
<tr>
<td>0xF</td>
<td>0</td>
</tr>
</tbody>
</table>

Assume:
- Total cache capacity of 8 bytes
- Cache with 4-byte cache lines (So 2 lines fit in cache)
- Least recently used (LRU) replacement policy

There are two forms of “data locality” in this sequence:

Spatial locality: loading data in a cache line “preloads” the data needed for subsequent accesses to different addresses in the same line, leading to cache hits.

Temporal locality: repeated accesses to the same address result in hits.
Assume:

- Total cache capacity of 8 bytes
- Cache with 4-byte cache lines (So 2 lines fit in cache)
- Least recently used (LRU) replacement policy

### Array of 16 bytes in memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>16</td>
</tr>
<tr>
<td>0x1</td>
<td>255</td>
</tr>
<tr>
<td>0x2</td>
<td>14</td>
</tr>
<tr>
<td>0x3</td>
<td>0</td>
</tr>
<tr>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>0x6</td>
<td>6</td>
</tr>
<tr>
<td>0x7</td>
<td>0</td>
</tr>
<tr>
<td>0x8</td>
<td>32</td>
</tr>
<tr>
<td>0x9</td>
<td>48</td>
</tr>
<tr>
<td>0xA</td>
<td>255</td>
</tr>
<tr>
<td>0xB</td>
<td>255</td>
</tr>
<tr>
<td>0xC</td>
<td>255</td>
</tr>
<tr>
<td>0xD</td>
<td>0</td>
</tr>
<tr>
<td>0xE</td>
<td>0</td>
</tr>
<tr>
<td>0xF</td>
<td>0</td>
</tr>
</tbody>
</table>

### Cache example 2

<table>
<thead>
<tr>
<th>Address accessed</th>
<th>Cache action</th>
<th>Cache state (after load is complete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>&quot;cold miss&quot;, load 0x0</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x1</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x2</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x3</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x4</td>
<td>&quot;cold miss&quot;, load 0x4</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x5</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x6</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x7</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x8</td>
<td>&quot;cold miss&quot;, load 0x8 (evict 0x0)</td>
<td>0x00111111, 0x00111111</td>
</tr>
<tr>
<td>0x9</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0xA</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0xB</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0xC</td>
<td>&quot;cold miss&quot;, load 0xC (evict 0x4)</td>
<td>0x00111111, 0x00111111</td>
</tr>
<tr>
<td>0xD</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0xE</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0xF</td>
<td>hit</td>
<td>0x00111111</td>
</tr>
<tr>
<td>0x0</td>
<td>&quot;capacity miss&quot;, load 0x0 (evict 0x8)</td>
<td>0x00111111, 0x00111111</td>
</tr>
</tbody>
</table>

### Time progression
- Time
- "cold miss", load 0x0
- "hit"
- "hit"
- "hit"
- "cold miss", load 0x4
- "hit"
- "hit"
- "hit"
- "hit"
- "hit"
- "cold miss", load 0x8 (evict 0x0)
- "hit"
- "hit"
- "hit"
- "hit"
- "capacity miss", load 0x0 (evict 0x8)
Caches reduce length of stalls (reduce memory access latency)

- Processors run efficiently when they access data that is resident in caches
- Caches reduce memory access latency when processors accesses data that they have recently accessed!

* Caches also provide high bandwidth data transfer
The implementation of the linear memory address space abstraction on a modern computer is complex.

The instruction “load the value stored at address X into register R0” might involve a complex sequence of operations by multiple data caches and access to DRAM.

Common organization: hierarchy of caches:
Level 1 (L1), level 2 (L2), level 3 (L3)

Smaller capacity caches near processor → lower latency
Larger capacity caches farther away → larger latency
Data access times
(Kaby Lake CPU)

Latency (number of cycles at 4 GHz)

- Data in L1 cache: 4 cycles
- Data in L2 cache: 12 cycles
- Data in L3 cache: 38 cycles
- Data in DRAM (best case): $\sim 248$ cycles
Summary

- Today, single-thread-of-control performance is improving very slowly
  - To run programs significantly faster, programs must utilize multiple processing elements or specialized processing hardware
  - Which means you need to know how to reason about and write parallel and efficient code

- Writing parallel programs can be challenging
  - Requires problem partitioning, communication, synchronization
  - Knowledge of machine characteristics is important
  - In particular, understanding data movement!

- I suspect you will find that modern computers have tremendously more processing power than you might realize, if you just use it efficiently!
Welcome to CS149!

- Get signed up on the website
- Find yourself a partner! (remember, we can help you)

Prof. Kayvon  Prof. Olukotun

James  Minfei  Yasmine  Senyang

Zhenbang  Neha  Michael

Jensen  Shiv  Tom