LECTURE 9: MEMORY OPTIMIZATION AND BASH SCRIPTING

Computer Systems and Networks

Dr. Pallipuram
(vpallipuramkrishnamani@pacific.edu)

Today's Class

- Recap of Memory Layout
- Memory Optimizations
- Caches and principle of locality
- Bash scripting (useful in Lab 7)

Recall: Malloc – 1D

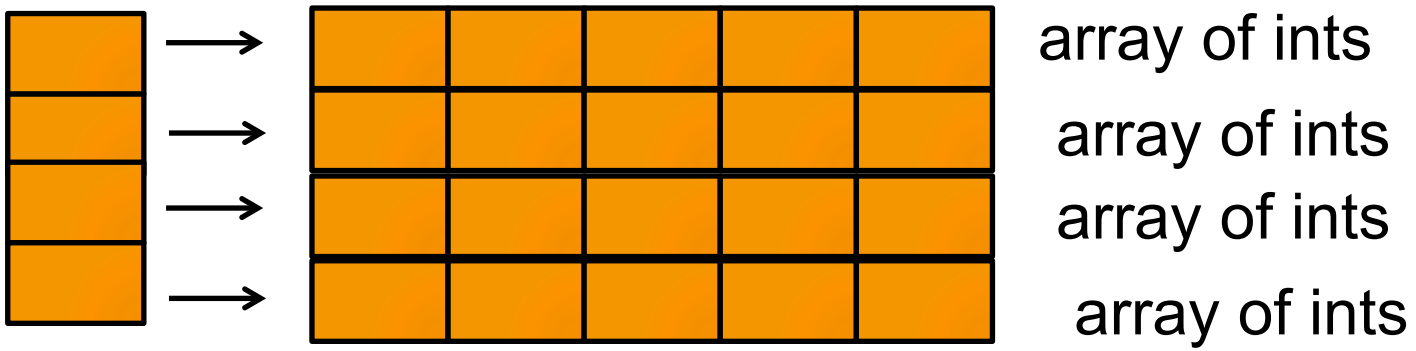
```
int *array;  //for array of integers
array = (int *)malloc(sizeof(int)*5);
```

address:	60	64	68	72	76
value:	array[0]	array[1]	array[2]	array[3]	array[4]

array (pointer variable)
value: 60
pointer's addr: 32

Recall: Malloc – 2D Allocate 4x5 integers

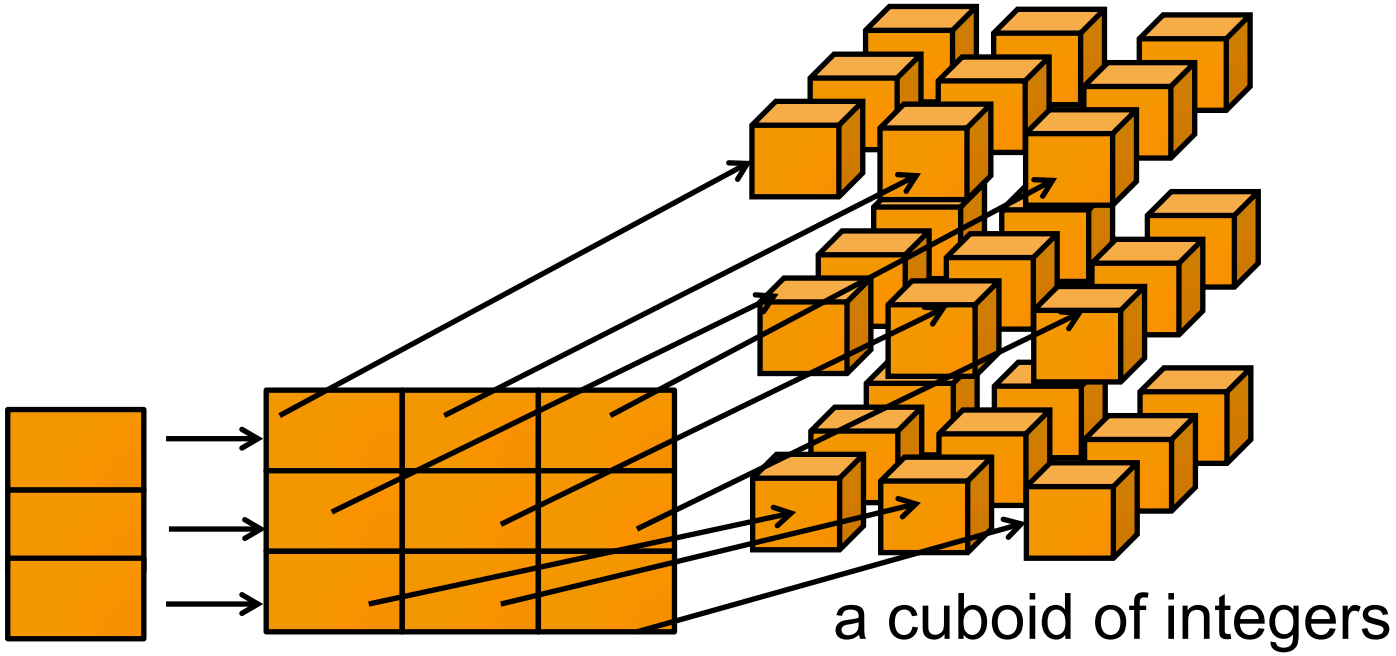
```
int **array; //a double pointer
array = (int **)malloc(sizeof(int *)*4);

 for(i=0;i<4;i++)
   array[i] = (int *)malloc(sizeof(int)*5);
```



an array of integer pointers

Recall: Malloc – 3D

```
int ***array; //a triple pointer
```



an array of double pointers

a matrix of single pointers

a cuboid of integers

Problem 1 – In Lab 6 you will answer how arrays are managed

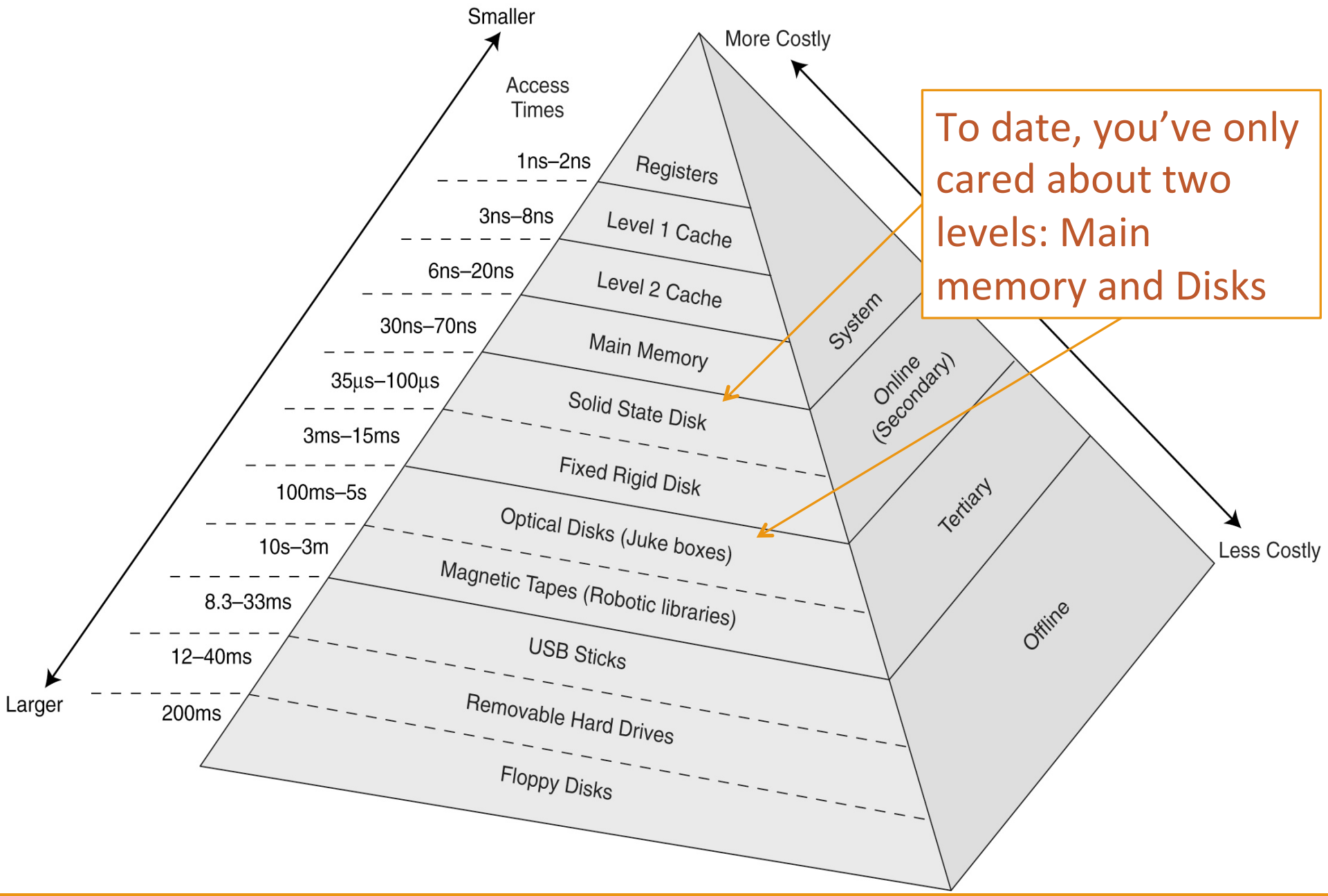
Write a C code snippet to print the addresses of elements in a 2-D array, `array[row][col]`. Visit this array in row-major format (row 0, then row 1, and so on..)

Memory Hierarchy

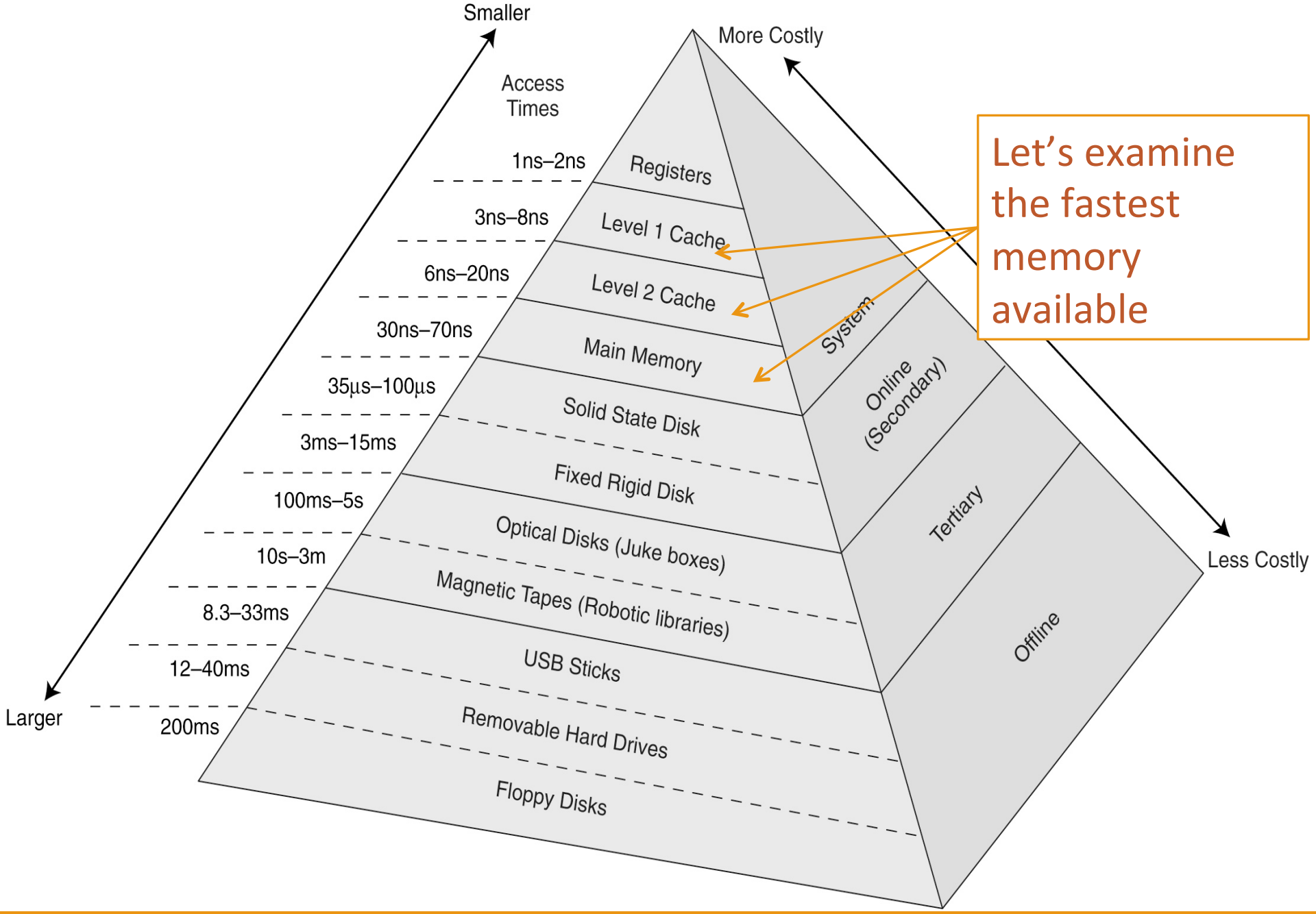
Goal as system designers:

Fast Performance **and** Low Cost

Tradeoff: Faster memory is
more expensive than slower memory



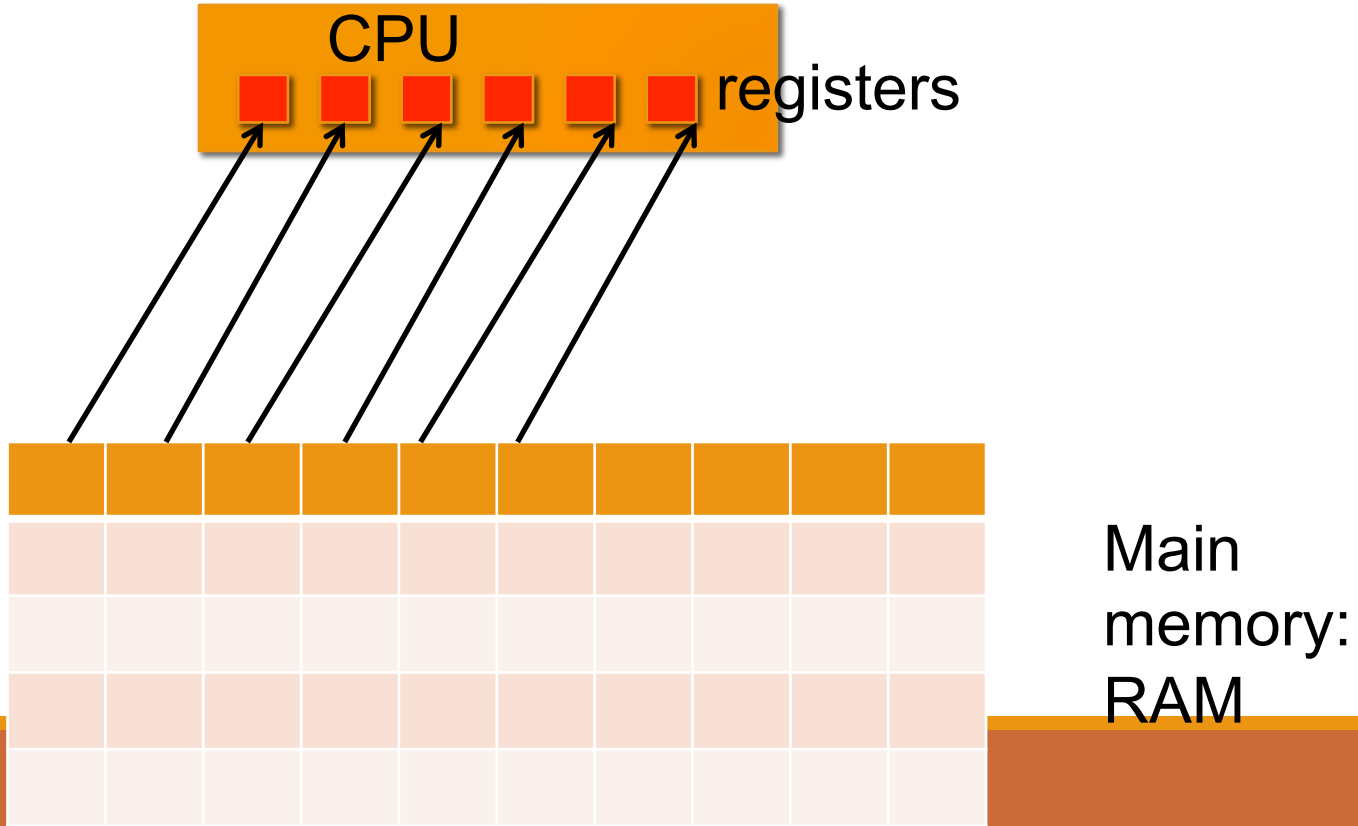
To date, you've only cared about two levels: Main memory and Disks


Smaller
More Costly
Access Times
1ns–2ns
Registers
3ns–8ns
Level 1 Cache
6ns–20ns
Level 2 Cache
30ns–70ns
Main Memory
35µs–100µs
Solid State Disk
3ms–15ms
Fixed Rigid Disk
100ms–5s
Optical Disks (Juke boxes)
10s–3m
Magnetic Tapes (Robotic libraries)
8.3–33ms
USB Sticks
12–40ms
Removable Hard Drives
200ms
Floppy Disks
Larger
Less Costly
System
Online (Secondary)
Tertiary
Offline
Let's examine the fastest memory available

Registers

The compiler places function local variables in the processor registers for fast access.

Recall *reduced memory access optimization*



Cache – Holy grail of memory optimizations

Caches

A space larger than registers and on the processor itself

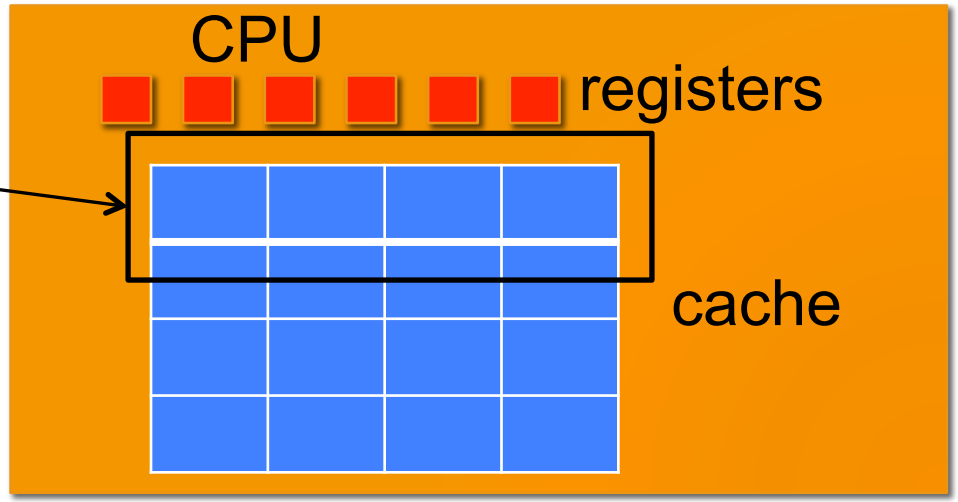
Although cache is much smaller than main memory, its access time is much faster!

Cache is **automatically** managed by the hardware memory system

Clever programmers can help the hardware use the cache more effectively

Consider a hypothetical cache memory architecture

cache line is 16 bytes. Space for 4 ints per line.

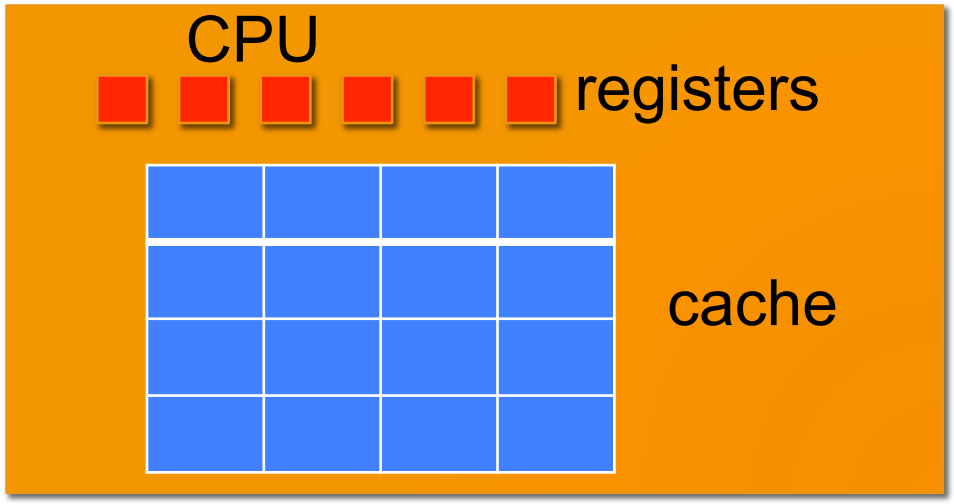


How does CPU get array elements a[0], a[1], a[2], etc.?

```
for(i=0;i<30;i++)
    a[i];
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

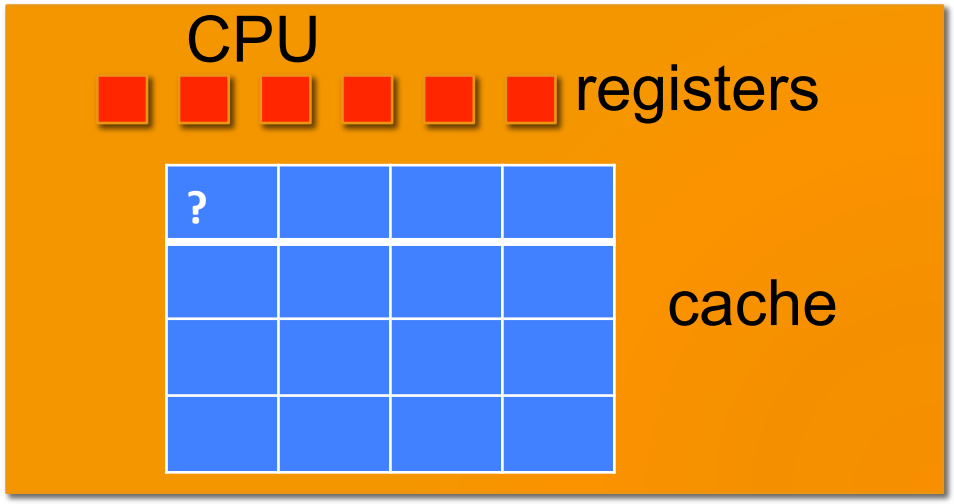
Main memory: RAM



Access a[0]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

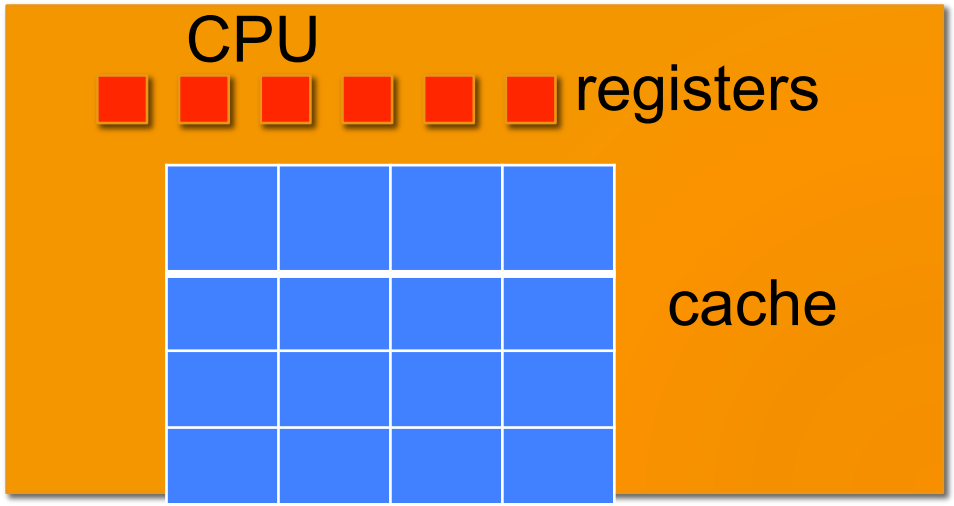
Main memory RAM



Step 1: query the cache.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

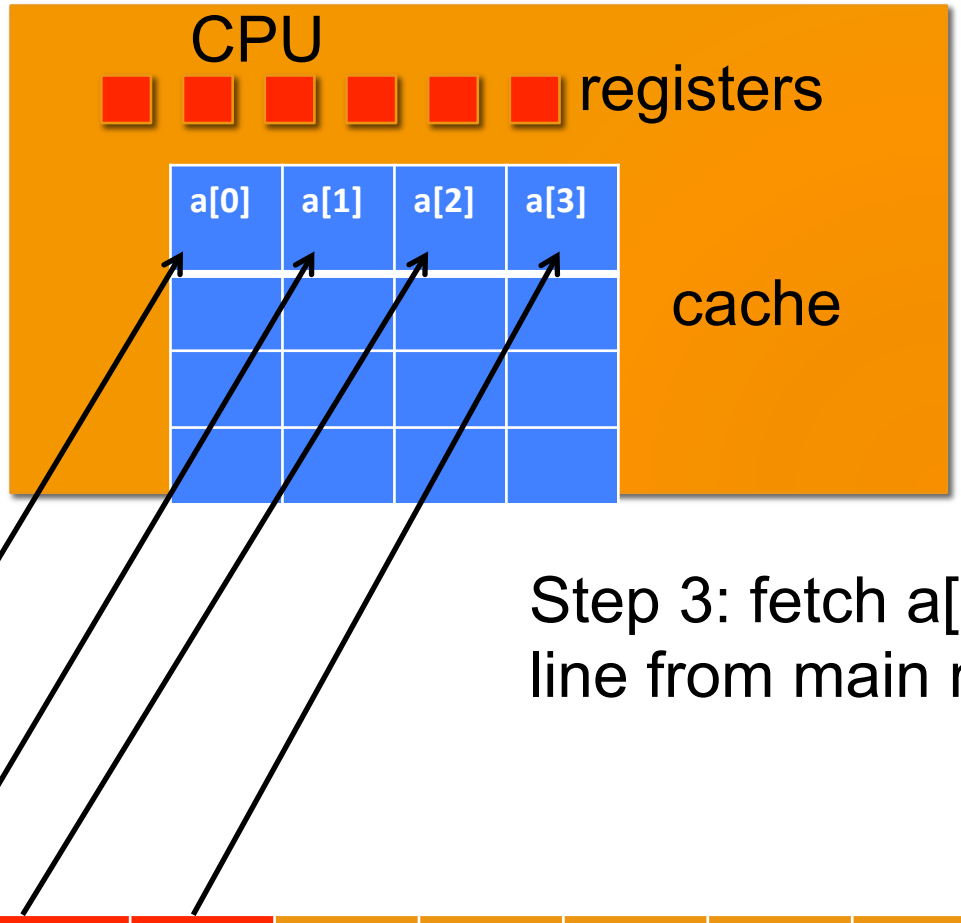
Main memory RAM



Step 2: a[0] not there (a miss).

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM



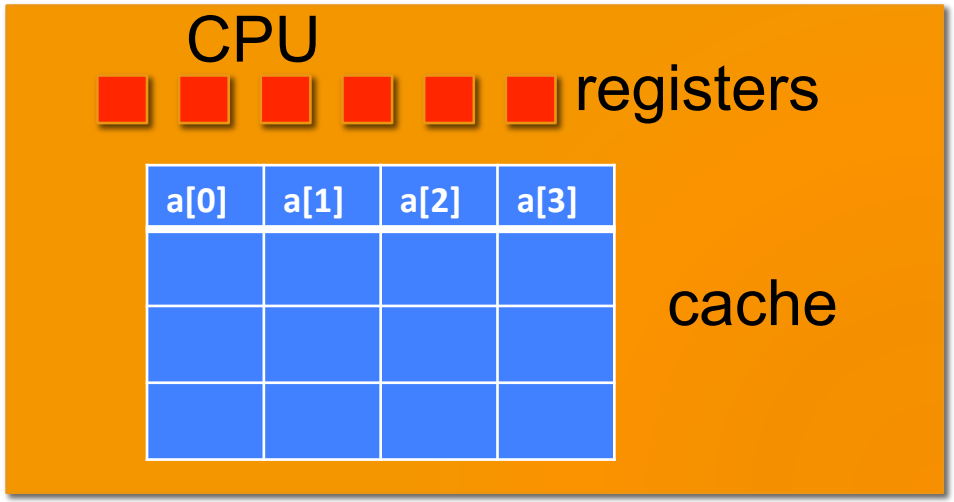
Step 3: fetch a[0] and a cache line from main memory

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

Spatial Locality

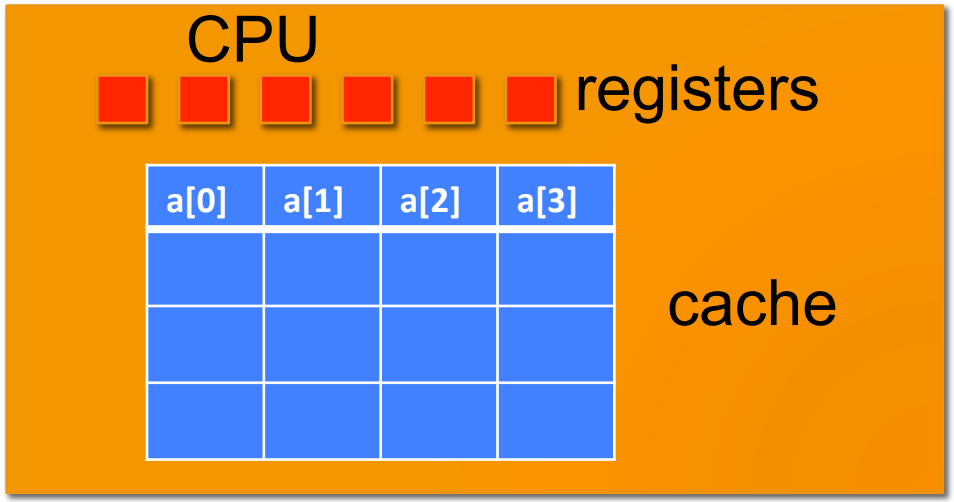
Once a data element is accessed, it is likely that a nearby data element (or even the same element) will be needed soon



Access a[1]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

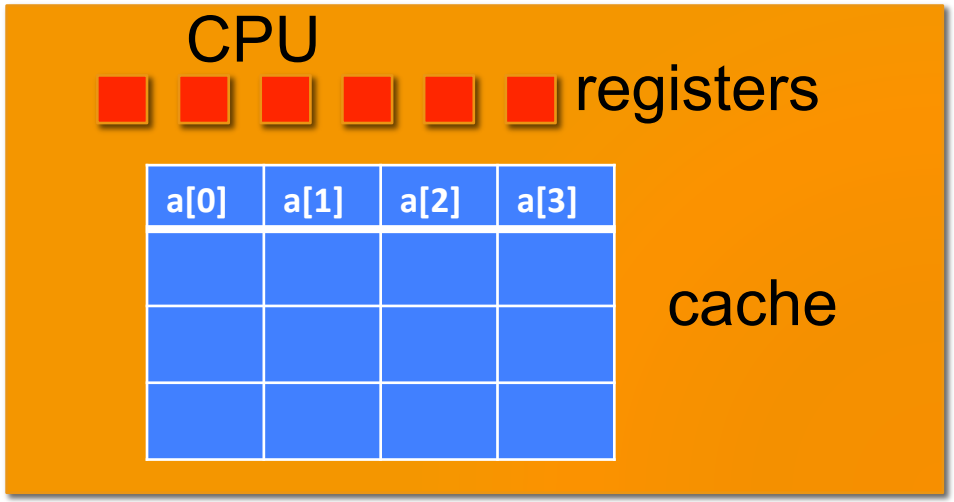


Step 1: query the cache. There! a hit!

Access a[1]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

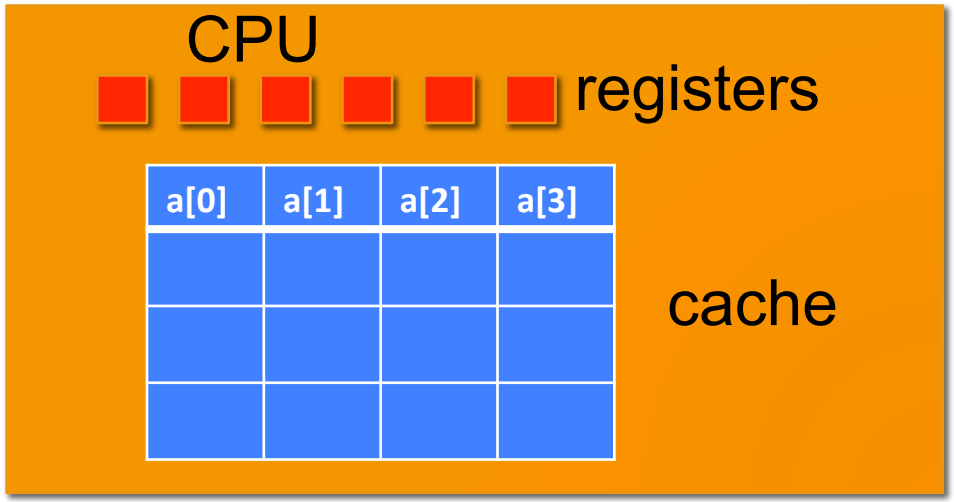
Main memory RAM



Access a[2]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

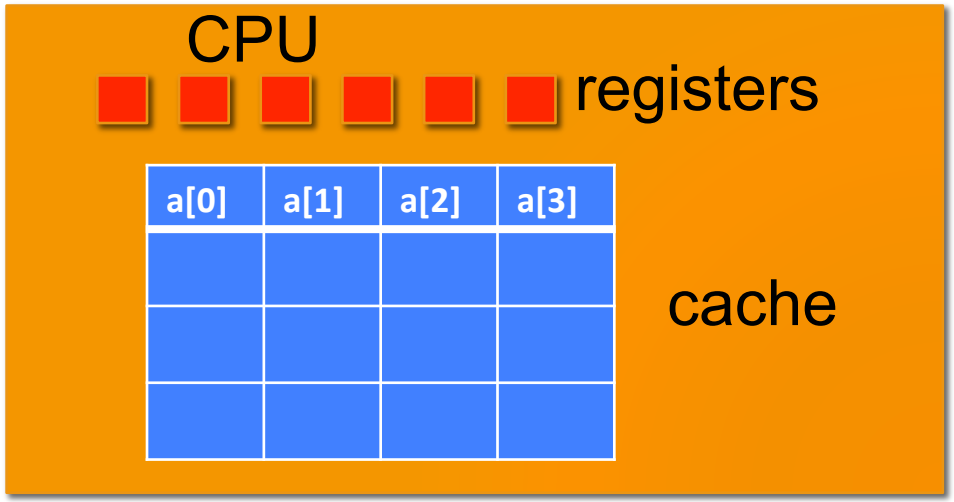


Step 1: query the cache. There! a hit!

Access a[2]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]	

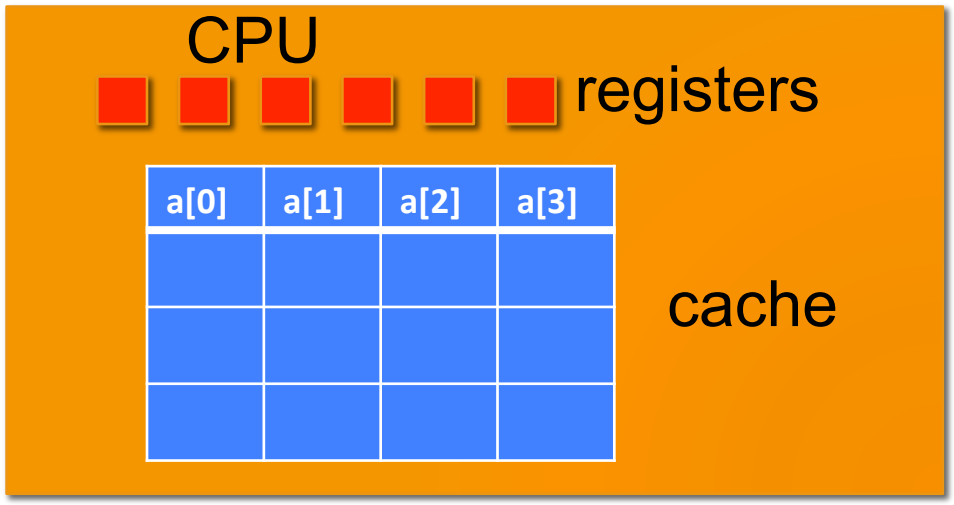
Main memory RAM



Access a[3]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

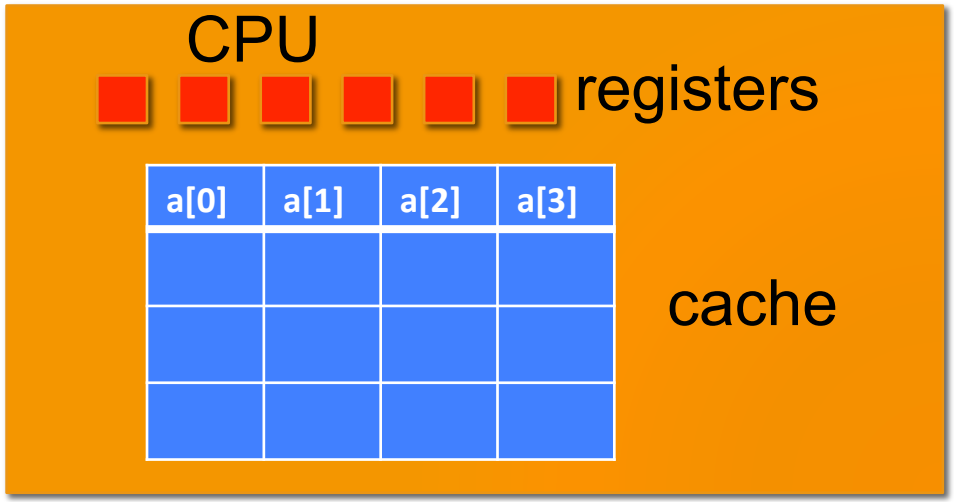


Step 1: query the cache. There! a hit!

Access a[3]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

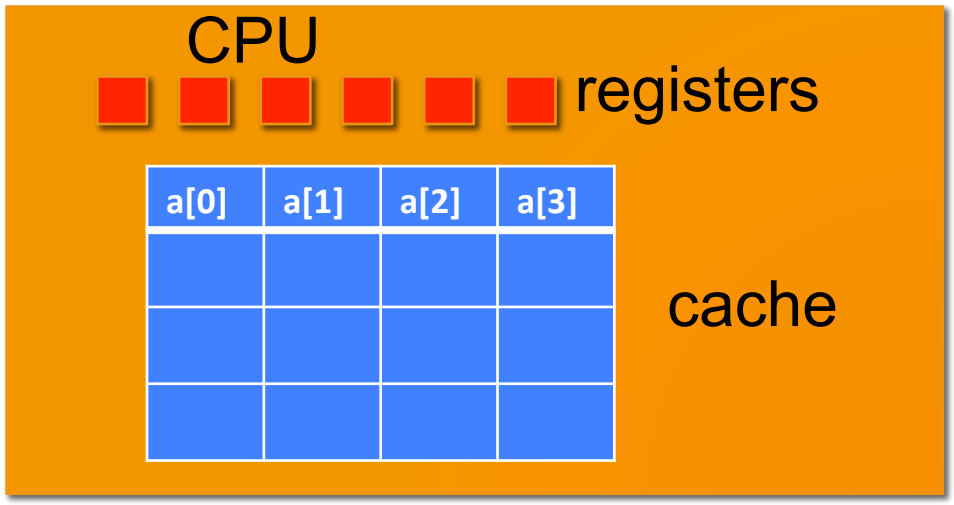
Main memory RAM



Access a[4]

✗ a[0]	✓ a[1]	✓ a[2]	✓ a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

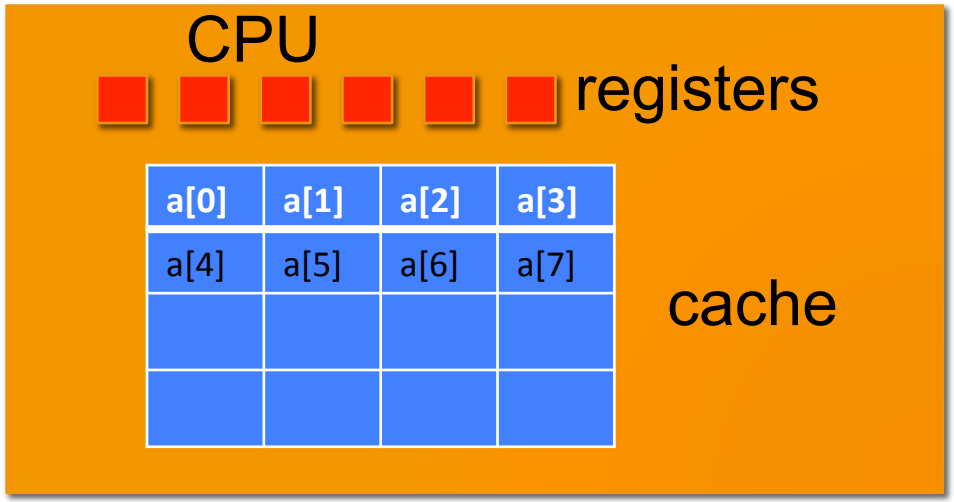


Step 1: query the cache. a miss ☹

Access a[4]

✖ a[0]	✔ a[1]	✔ a[2]	✔ a[3]	✖ a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

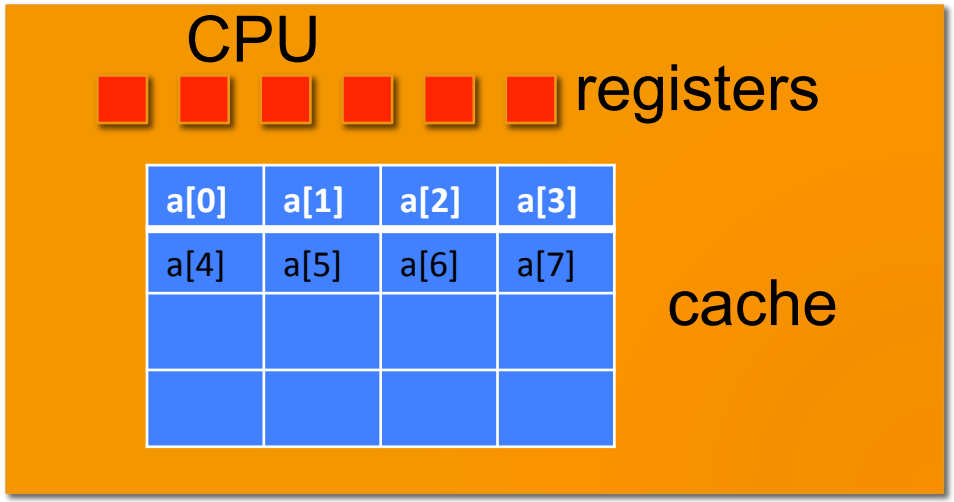


Access a[4]

Step 2: get a[4] and a line from main memory

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

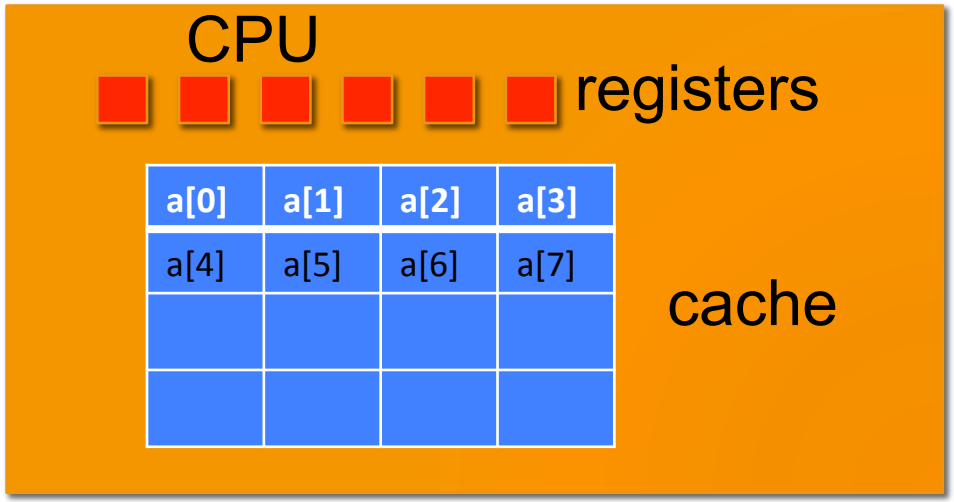


Step 1: query the cache. There! a hit!

Access a[5]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]	

Main memory RAM

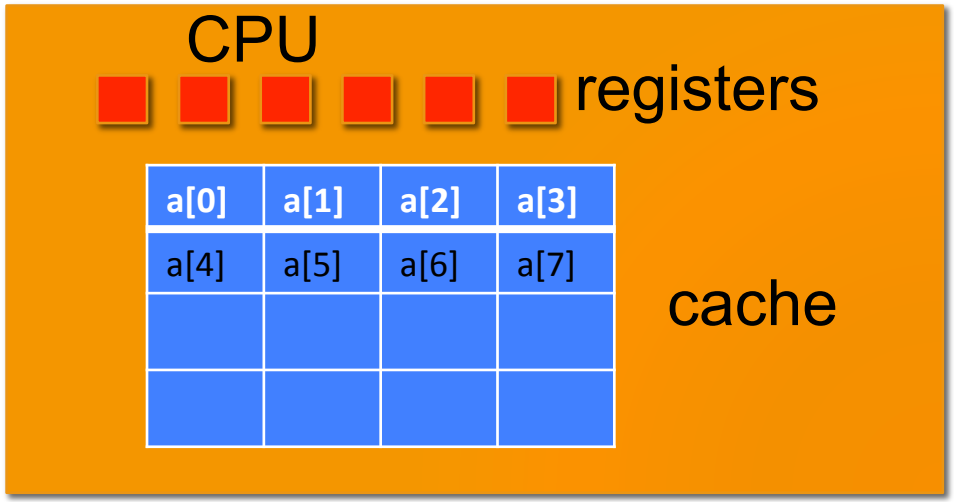


Step 1: query the cache. There! a hit!

Access a[6]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]	

Main memory RAM



Step 1: query the cache. There! a hit!

Access a[7]

✗ a[0]	✓ a[1]	✓ a[2]	✓ a[3]	✗ a[4]	✓ a[5]	✓ a[6]	✓ a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

Cache Analysis

In our case, cache line width is 16 bytes (space for 4 integers). we always load data into the cache **1 "line" at a time**

3 hits for every 4 integer accesses

Why there were cache hits?

values were stored contiguously, which is easy for hardware to fetch the data

If cache width is for `m` integers and the data access is contiguous, then only 1 miss for every `m` integer accesses

Temporal Locality

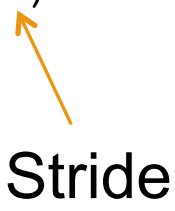
Once a data element is accessed, it is likely that it will be needed soon

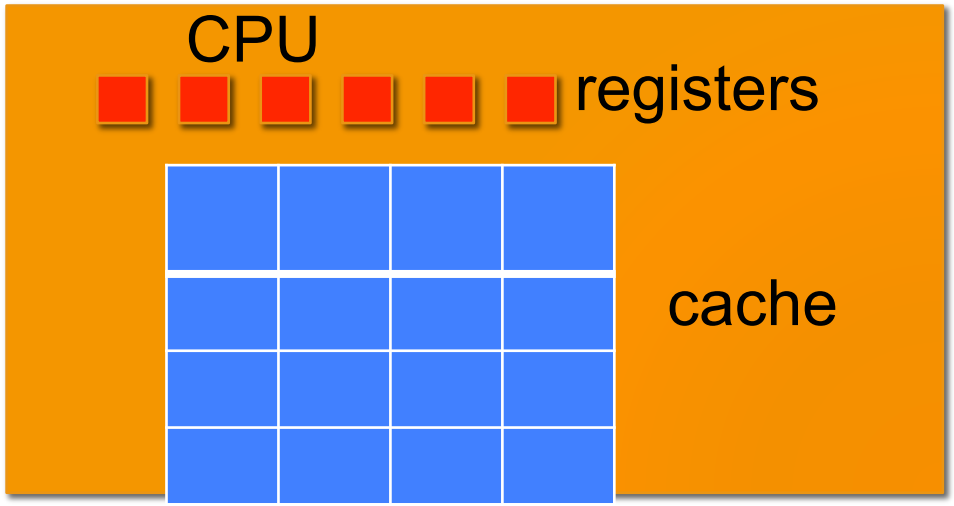
```
for(i=0;i<sum;i++)
{

}
```

both `i` and `sum` are repeatedly accessed, hence cached

Problem 2: For cache line of width 16 bytes, how many cache hits will this code get? Assume sizeof(int)=4

```
int a [24];
for (i=0;i<24;i=i+4)
     sum+=a[i];
```





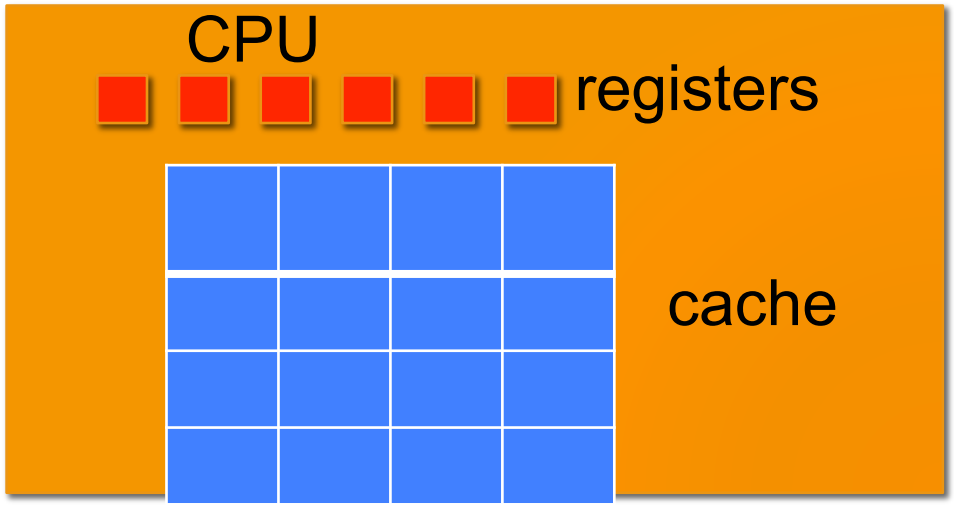
Access a[0].

```
int a [24];
for (i=0;i<24;i=i+4)
    sum+=a[i];
```

stride = 4

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

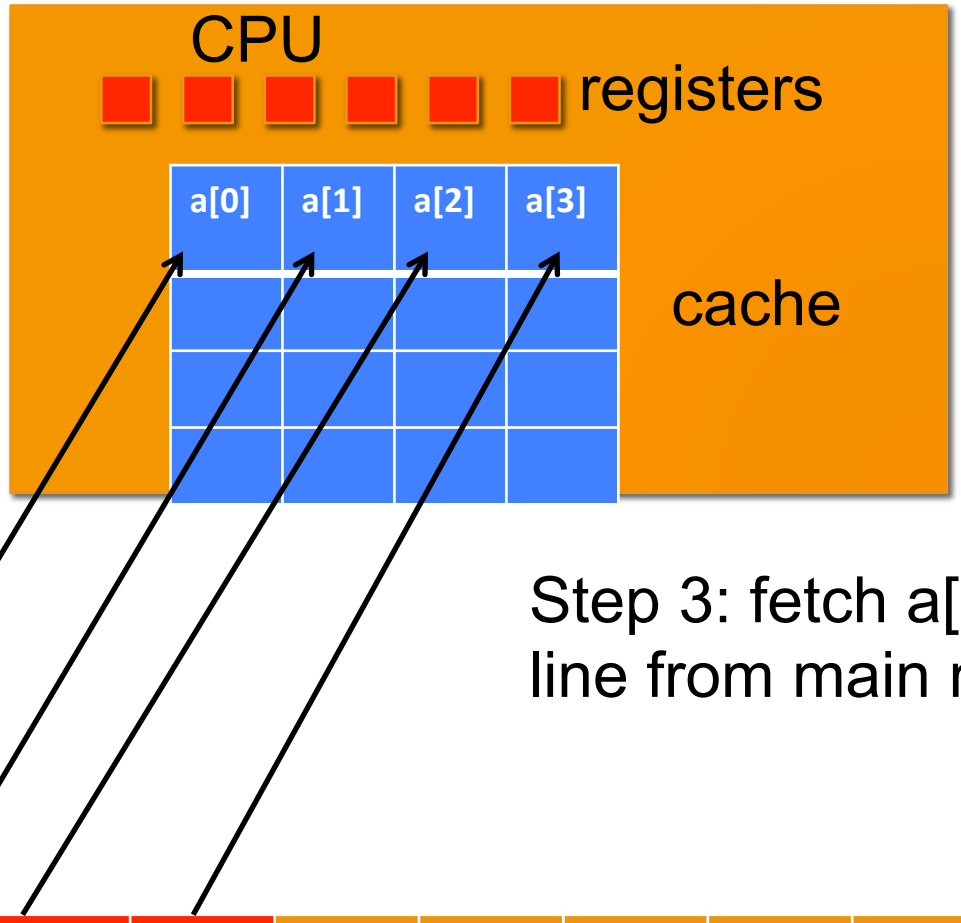
Main memory RAM



Step 2: a[0] not there in the cache (a miss).

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

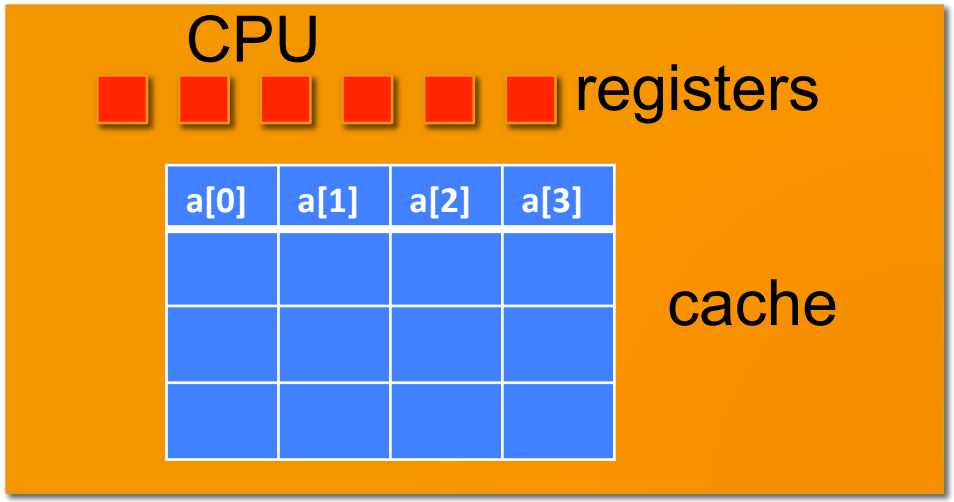
Main memory RAM



Step 3: fetch a[0] and a cache line from main memory

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM



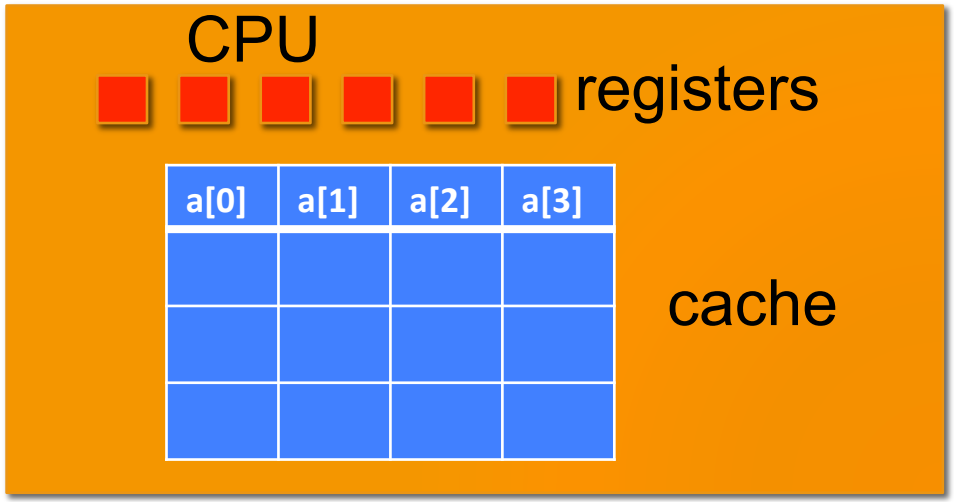
Access a[4].

```
int a [24];
for (i=0;i<24;i=i+4)
    sum+=a[i];
```

stride = 4

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

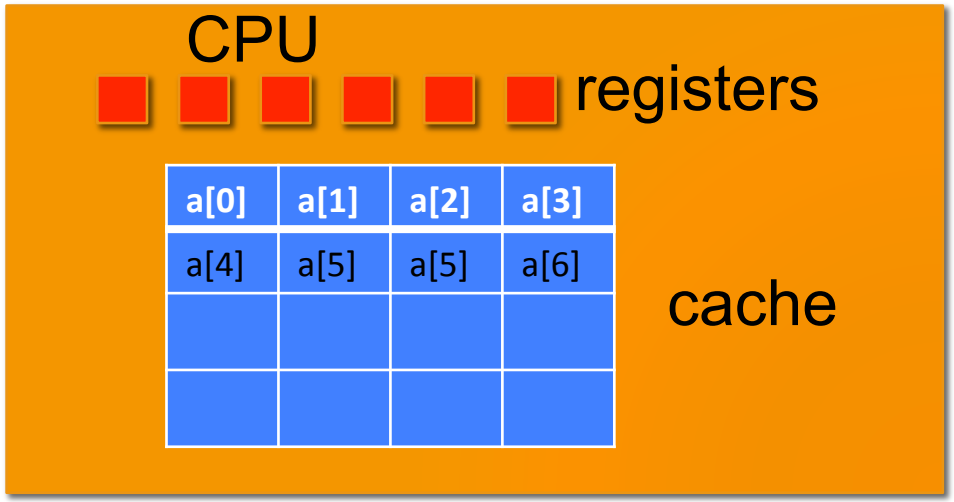


Step 2: a[4] not there (a miss).

```
stride = 4
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

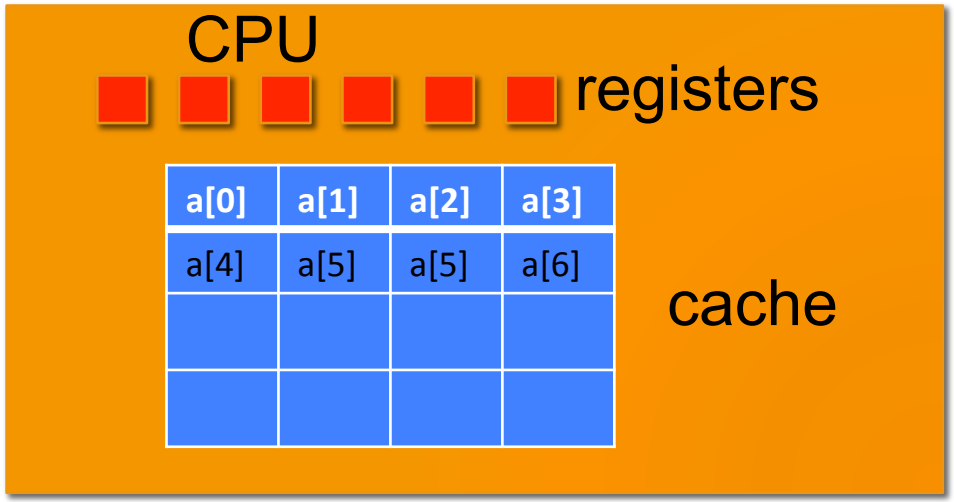


Step 3: fetch a[4] and a cache line from main memory

```
stride = 4
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM



a[8] not there (a miss).

```
int a [24];
for (i=0;i<24;i=i+4)
    sum+=a[i];
```

stride = 4

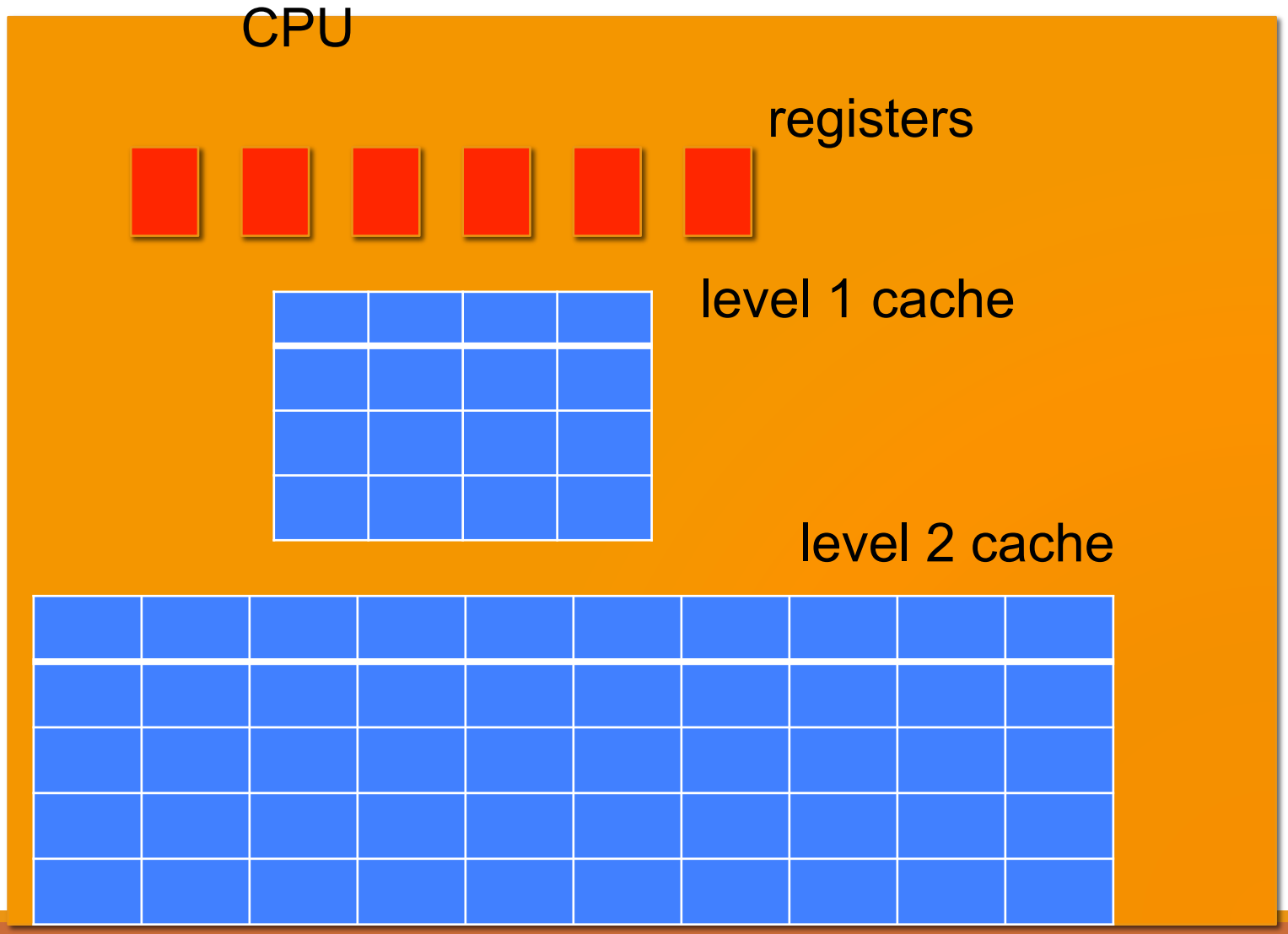
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Main memory RAM

Programs with good locality **run faster** than programs with poor locality

A program that randomly accesses memory addresses (but never repeats) will gain no benefit from a cache

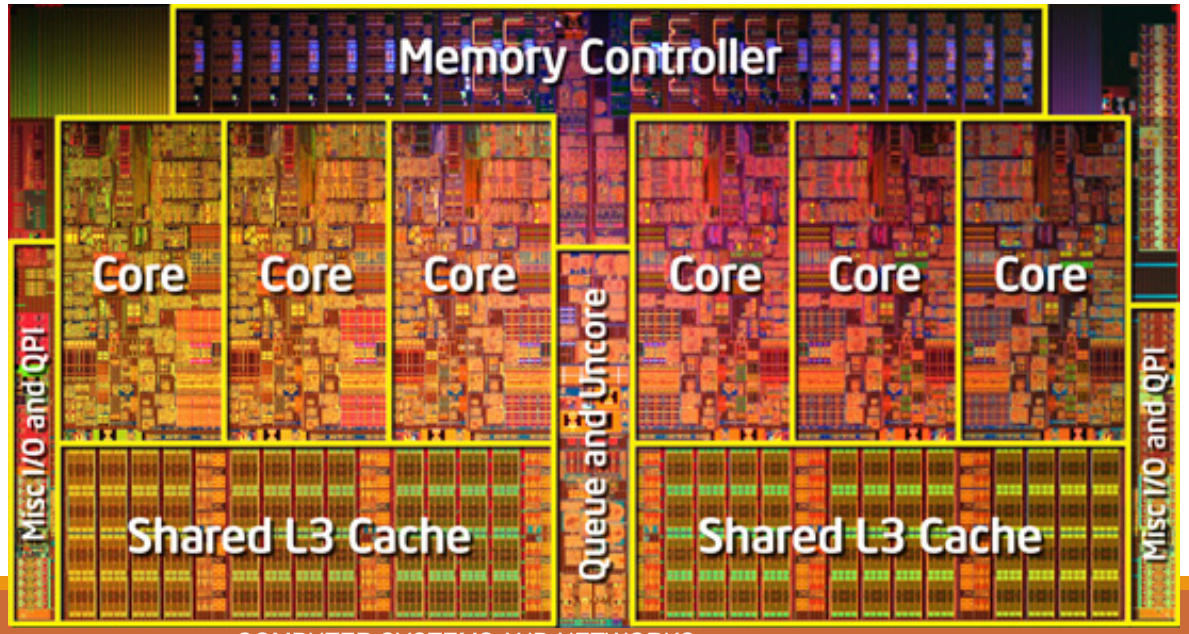
Multi-level cache hierarchy



Cache Example – Intel Core i7 980x

6 core processor with a sophisticated multi-level cache hierarchy

3.5GHz, 1.17 billion transistors



Cache Example – Intel Core i7 980x

Access time? (Measured in 3.5GHz clock cycles)

- 4 cycles to access L1 cache
- 9-10 cycles to access L2 cache
- 30-40 cycles to access L3 cache

Smaller caches are faster to search

- And can also fit closer to the processor core

Larger caches are slower to search

- Plus we have to place them further away

Problem 3

Imagine a computer system only has main memory (no cache was present). Is *temporal* or *spatial locality* important for performance when repeatedly accessing an array with 8-byte elements?

- **No.** Locality is not important in a system without caching, because every memory access will take the same length of time.

Problem 4

Imagine a memory system has main memory and a 1-level cache, but each cache line size is only 8 bytes in size. Assume the cache is much smaller than main memory. Is *temporal* or *spatial locality* important for performance here when repeatedly accessing an array with 8-byte elements?

- Only 1 array element is loaded at a time in this cache
- Temporal locality is important (access will be faster if the same element is accessed again)
- Spatial locality is not important (neighboring elements are not loaded into the cache when an earlier element is accessed)

Problem 5

Imagine your program accesses a 100,000 element array (of 8 byte elements) once from beginning to end with stride 1. The memory system has a 1-level cache with a line size of 64 bytes. How many cache misses would be expected in this system?

- **12500** cache misses. The array has 100,000 elements. Upon a cache miss, 8 adjacent and aligned elements (one of which is the miss) is moved into the cache. Future accesses to those remaining elements should hit in the cache. Thus, only 1/8 of the 100,000 element accesses result in a miss

Lab 7 Preview: Which code will have more cache hits? Assume array size larger than cache

```
for (i=0;i<row;i++)
 for(j=0;j<col;j++)
    array[i][j];
```

```
for (j=0;j<col;j++)
 for(i=0;i<row;i++)
    array[i][j];
```

Bash Script Automation

FOR LAB 7. IF TIME PERMITS, WE'LL SEE SOME ADVANCED CONCEPTS

Exercise 1 -- Bash Scripting

Every bash script usually begins with a **Shebang (#!)** – It is used to specify the absolute path of the bash

Example: `#!/bin/bash`

Create a folder in your home folder called BASH. cd to BASH.

Gedit a file: `mybash1.sh`. Add the above shebang and save. Change the mode of `mybash1.sh` to an executable. Recall our Linux exercise to grant execute privilege

Exercise 2: For Loops in Bash – Version 1

Add this code in `mybash1.sh`

```
#!/bin/bash
for i in 1 2 3 4 5
do
        echo "Welcome number: $i"
done
```

execute: ./mybash1.sh

Exercise 3 – A better for loop

```
#!/bin/bash
for ((i=0;i<12;i++))
do
        echo "Welcome number: $i"
done
```

$ replaces the variable with its value

Create a new file called mybash2.sh. Can you modify the above code to create folders: lab2 to lab12?

Exercise 4 -- Conditionals in bash

```
if ((<some C-like conditional>))
then
<commands>
fi
```

Create a new file called mybash3.sh. Can you modify the above code to create folders: lab02, lab03,..,lab09,.. lab12?

Concept of Arrays in Bash

Syntax:

```
declare -a arrayname=(element1 element2 element3)
```

```
declare -a Unix=('Debian' 'Red hat' 'Red hat' 'Suse' 'Fedora');
```

length of an array: `${#ArrayName[@]}`

example of accessing an element at i^{th} position:

`${ArrayName[i]}`

Putting it all together

I want to run the amplification program on Lenna_org_1024.pgm, gaussian width = 11, and for sigma values: 0.3, 0.4, …1.1 (totaling 9 executions). Automate these lines:

```
./amplify Lenna_org_1024.pgm 11 0.3 2
./amplify Lenna_org_1024.pgm 11 0.4 2
./amplify Lenna_org_1024.pgm 11 0.5 2
:
./amplify Lenna_org_1024.pgm 11 1.1 2
```

Exercise 5

I want to run the amplification program on Lenna_org_1024.pgm for sigma values: 0.3, 0.4, …1.1 AND

gaussian widths: 5 7 11 15 19

Next Class

Recommended -- TA will come in to assist you