LECTURE 16

PERFORMANCE

We are here

- Part 1: How computers works
  - Boolean logic, integers
  - Instructions
  - Memory
- Part 2: Software development
  - Compiling, make
  - ABIs & APIs
  - git
- Part 3: Correctness
  - Specifications
  - Documentation, testing
  - Static & dynamic analysis, debugging
- Part 4: Performance ⟵ TODAY
  - CPU pipelines, caches
  - Data structures
  - Parallel computation

What is performance?

We want computers to perform required actions while minimizing their use of resources:

- Time
- Power use (mobile, servers)
- Network use (mobile)
- Memory use (peak RAM usage)
- Storage (solid state drive / hard disk drive)

We care about those resources in proportion to their cost (financial, emotional, etc.)

How do we achieve good performance?

High level

↓

Low level

1. Pick a good algorithm
   - specifically, an algorithm with low computational complexity:
     $O(\log(n))$ better than $O(n^2)$ better than $O(n^6)$ better than $O(2^n)$
   - we can hope for great improvements, $100\times$ faster, $1000\times$, etc.
   - $\rightarrow$ first thing to try!

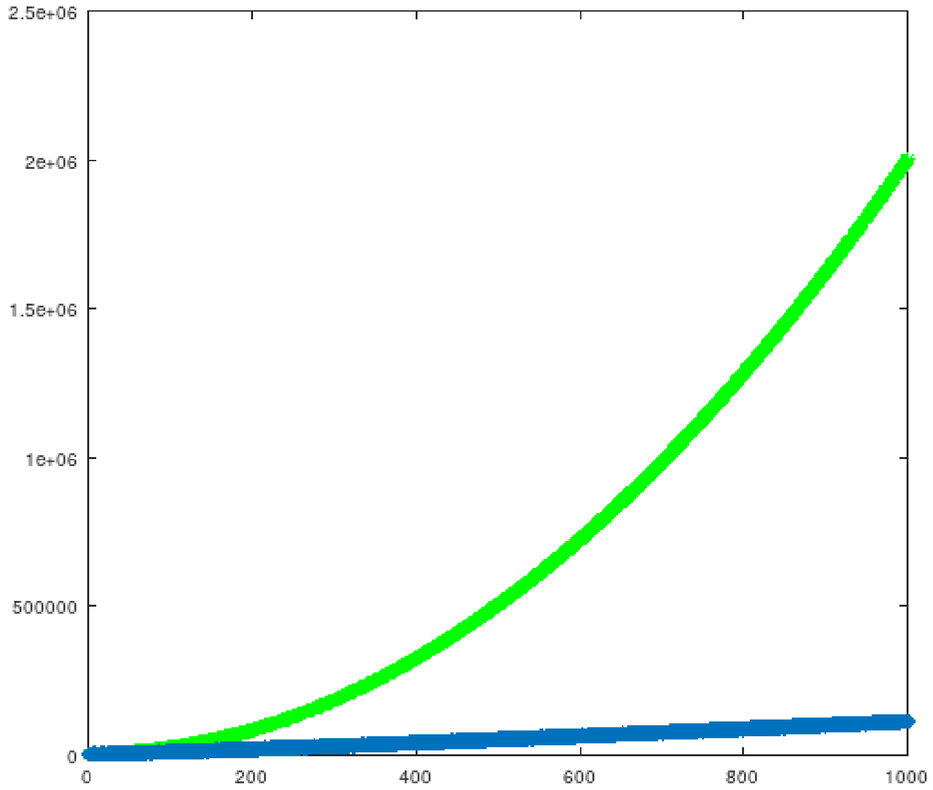
2. Pick an algorithm that is fast on the computers you use and implement it well
   - this course!
   - smaller improvements: from a few percent to $50\times$ faster

3. Translate the implementation into efficient instructions (compilers do that well)

What hides inside the big-O?

Let us compare two algorithms:

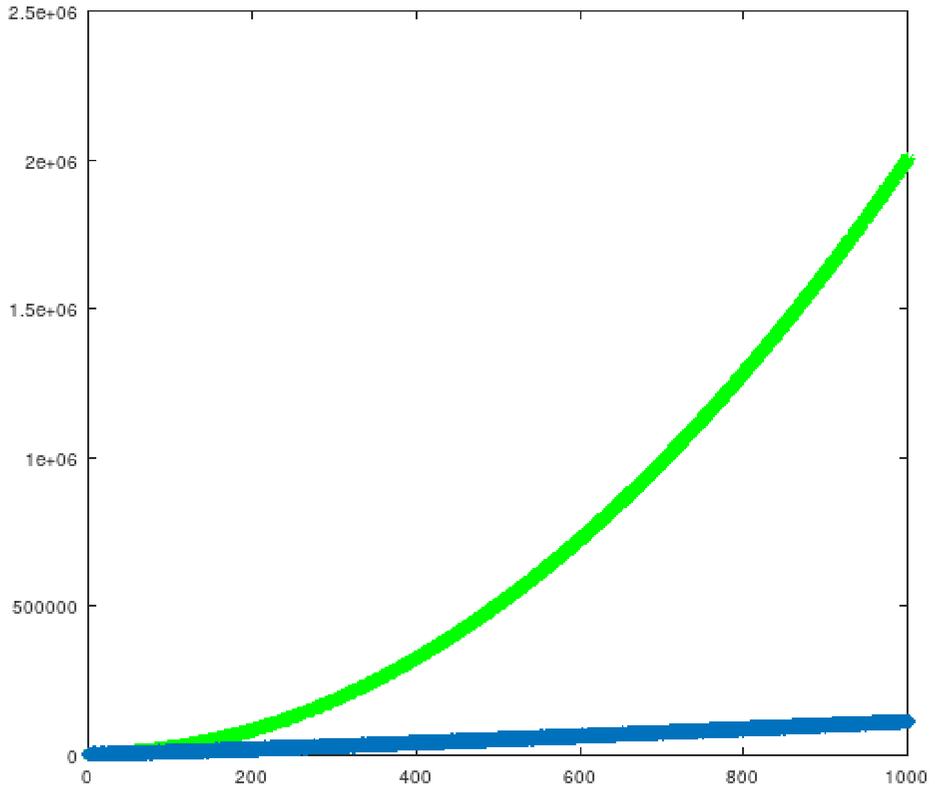
- Algorithm A has complexity $O(n^2)$
- Algorithm B has complexity $O(n \log(n))$



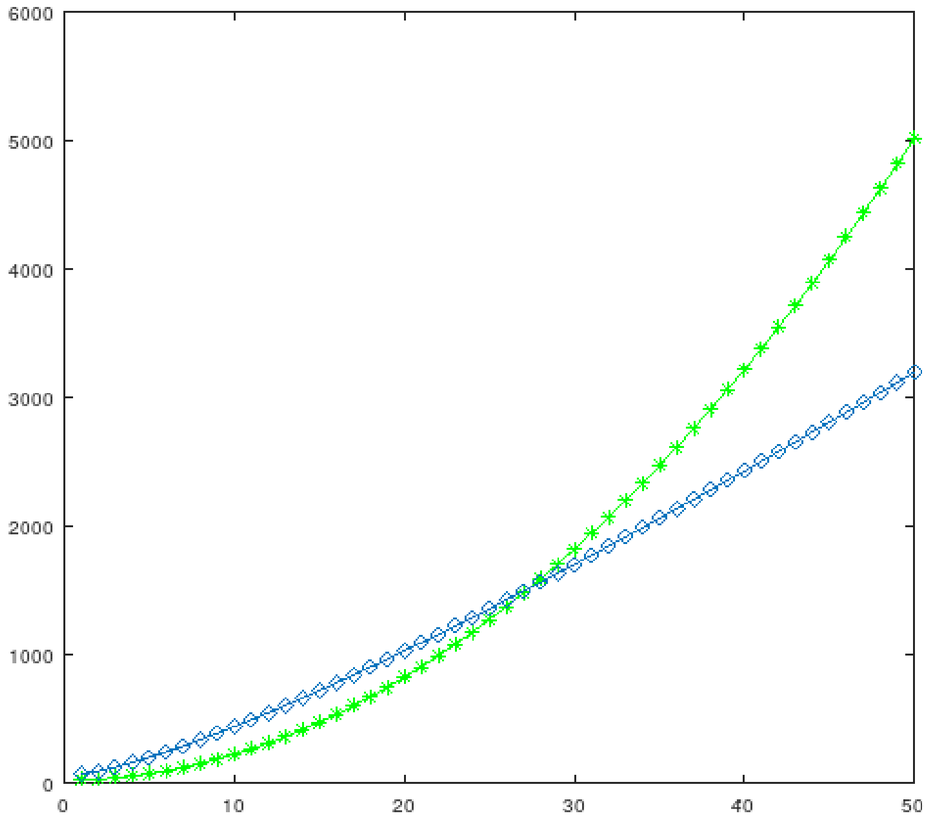
Algorithm A	Algorithm B
$O(n^2)$	$O(n \log(n))$

Specifically,

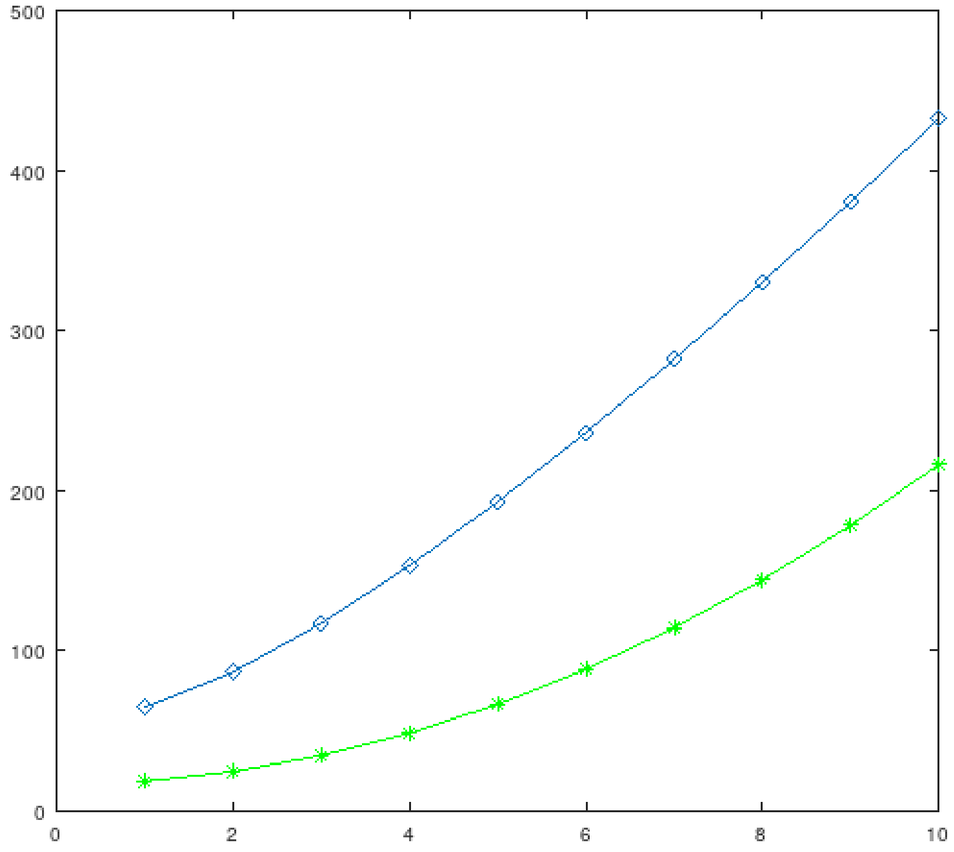
- Algorithm A performs $2n^2 + 16$ operations
- Algorithm B performs $16n \log(n) + 64$ operations



Algorithm A	Algorithm B
$2n^2 + 16$	$16n\log(n) + 64$



Algorithm A	Algorithm B
$2n^2 + 16$	$16n\log(n) + 64$



Algorithm A	Algorithm B
$2n^2 + 16$	$16n\log(n) + 64$

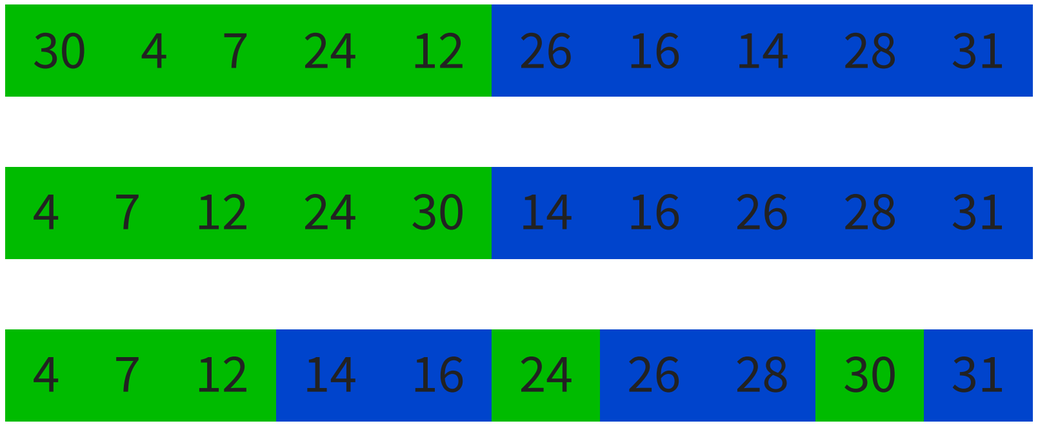
Which algorithm do we choose?

Assume that

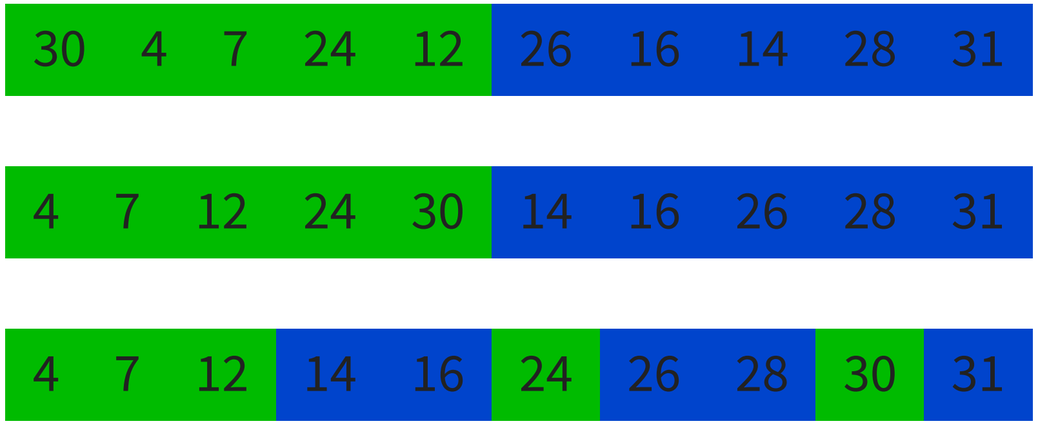
- Algorithm A is insertion sort
- Algorithm B is merge sort

Merge sort example

30 4 7 24 12 26 16 14 28 31

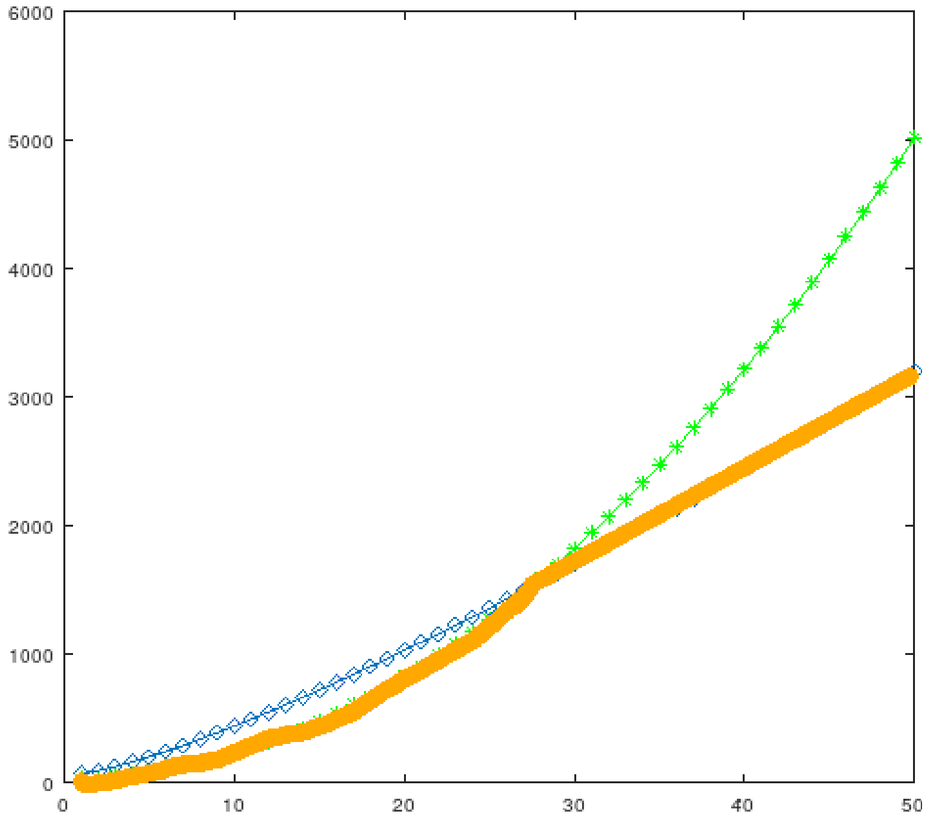


```
def merge_sort(a):

    # single-element list
    if (len(a) <= 1):
        return [ a ]

    # two-elements list
    if (len(a) == 2):
        if a[0] <= a[1]:
            return [ a[0], a[1] ]
        else:
            return [ a[1], a[0] ]

    # split list, sort each part
    pivot = len(a) // 2

    part1 = merge_sort(a[:pivot])
    part2 = merge_sort(a[pivot:])

    # merge parts
    r = []
    i1 = 0
    i2 = 0

    while i1 < len(part1) or i2 < len(part2):
        if (not i1 == len(part1)) and (i2 == len(part2) or part1[i1] < part2[i2]):
            r.append(part1[i1])
            i1 = i1 + 1
        else:
            r.append(part2[i2])
            i2 = i2 + 1

    return r
```

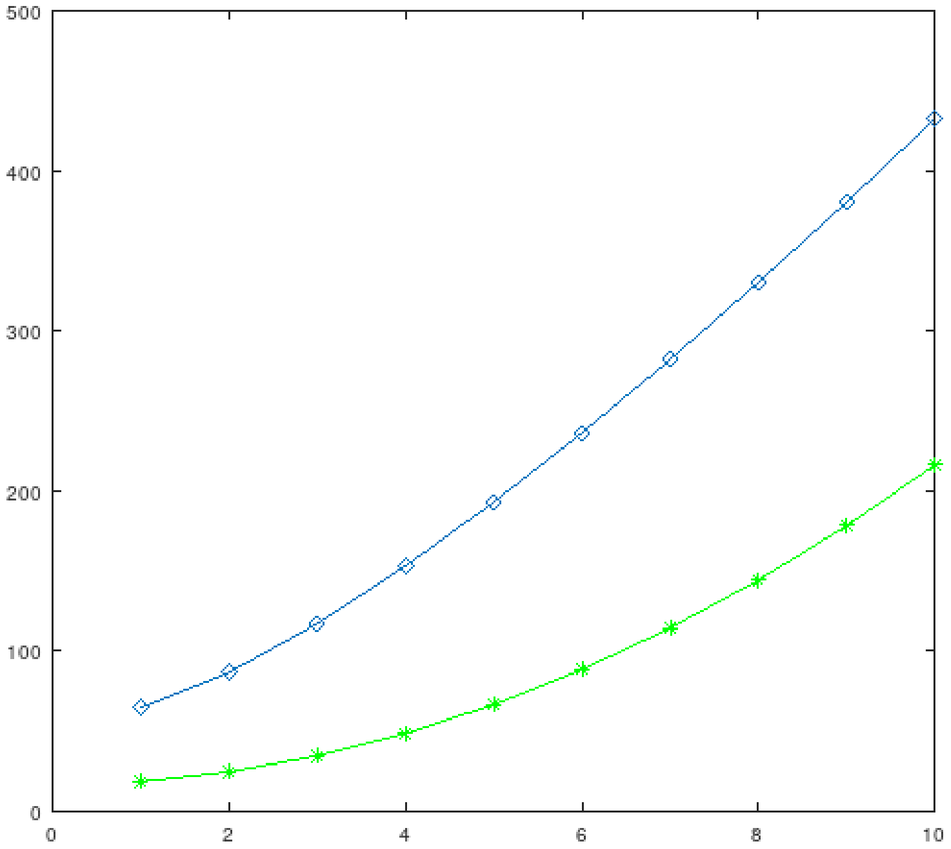

30 4 7 24 12 26 16 14 28 31
4 7 12 24 30 14 16 26 28 31
4 7 12 14 16 24 26 28 30 31

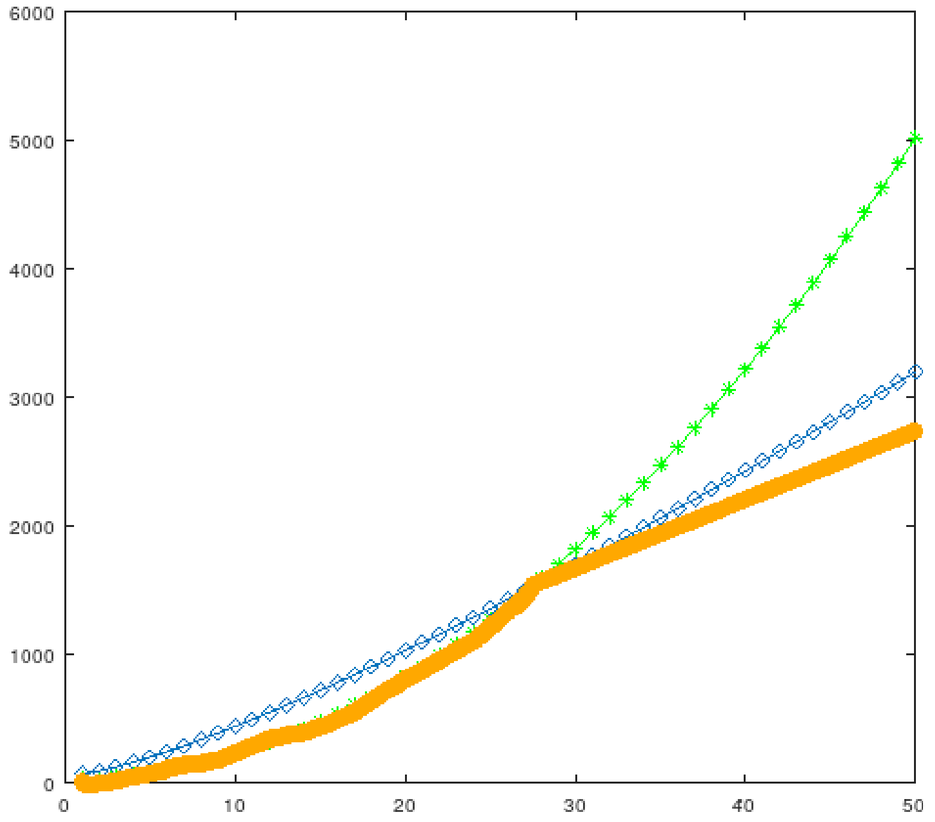
```
def merge_sort(a):

    # single-element list
    if (len(a) <= 28):
        return insertion_sort(a)

    # split list, sort each part
    pivot = len(a) // 2

    part1 = merge_sort(a[:pivot])
    part2 = merge_sort(a[pivot:])

    # merge parts
    r = []
    i1 = 0
    i2 = 0

    while i1 < len(part1) or i2 < len(part2):
        if (not i1 == len(part1)) and (i2 == len(part2) or part1[i1] < part2[i2]):
            r.append(part1[i1])
            i1 = i1 + 1
        else:
            r.append(part2[i2])
            i2 = i2 + 1

    return r
```

$\longrightarrow$ we should pick the right algorithm for each value of n:



But here, we can do even better: we can combine algorithms





→

This is how sort algorithms are implemented in practice

CPU PIPELINES

Back to instructions

Instruction decoding:

```
48 2b 06              sub    rax, QWORD PTR [rsi]
48 0f af 06           imul   rax, QWORD PTR [rsi]
48 0f af 02           imul   rax, QWORD PTR [rdx]
```

From `48 0f af 06`, the CPU needs to understand:

- that it must perform a multiplication (as opposed to, say, a subtraction)
- that one term is the value of a 64-bit register, `rax`
- that the other term comes from memory: the 64-bit value pointer to by `rsi`

What happens after instruction decoding?

```
48 0f af 06          imul   rax, QWORD PTR [rsi]
```

- the CPU has Boolean circuitry to compute multiplications
- it must ensure that one of the two inputs of the multiplier is `rax`
- the CPU has Boolean circuitry to access memory
- it must ensure that the input of the memory circuitry is `rsi`
- it sets the second input of the multiplier to the output of the memory circuitry
- it stores the output of the multiplier back to `rax`

It is no longer possible to do all this in a single cycle

(e.g. at 4 GHz, i.e. 4 billion cycles per second, so in 0.25 ns)

Imagine that it takes:

- one cycle to decode an instruction
- one cycle to fetch data from memory
- one cycle to perform arithmetic

	`sub rax, [rdi]`	
	`imul rbx, [rsi]`	
decoder	-	
memory	-	
arithmetic	-	
	-	
	-	

	-	
	sub rax, [rdi]	
decoder	imul rbx, [rsi]	decode "imul" _____
memory	-	_____
arithmetic	-	_____
	-	
	-	

	-	
	sub rax, [rdi]	
decoder	-	____________________
memory	imul rbx, [rsi]	fetch [rsi] _______
arithmetic	-	____________________
	-	
	-	

	-	
	sub rax, [rdi]	
decoder	-	____________________
memory	-	____________________
arithmetic	imul rbx, [rsi]	compute rbx * [rsi]
	-	
	-	

	-	
	-	
decoder	`sub rax, [rdi]`	`decode "sub" ______`
memory	-	`_____________________`
arithmetic	-	`_____________________`
	`imul rbx, [rsi]`	
	-	

	-	
	-	
decoder	-	____________________
memory	sub rax, [rdi]	fetch [rdi] _______
arithmetic	-	____________________
	imul rbx, [rsi]	
	-	

	-	
	-	
decoder	-	____________________
memory	-	____________________
arithmetic	`sub rax, [rdi]`	`compute rax - [rdi]`
	`imul rbx, [rsi]`	
	-	

	-	
	-	
decoder	-	
memory	-	
arithmetic	-	
	`sub rax, [rdi]`	
	`imul rbx, [rsi]`	

Each instruction takes 3 cycles

However, in this model,

- while the memory circuitry is busy fetching `QWORD PTR [rsi]`, the multiplier is idle
- while the multiplier computes the result, the memory is idle
- during instruction decoding, everything else is idle

We can exploit this!

Pipelined execution

	`sub rax, [rdi]`	
	`imul rbx, [rsi]`	
	`add rcx, [rbp]`	
	-	
decoder	-	(idle)
memory	-	(idle)
arithmetic	-	(idle)
	-	
	-	
	-	

Pipelined execution

	-	
	`sub rax, [rdi]`	
	`imul rbx, [rsi]`	
	`add rcx, [rbp]`	
decoder	-	(idle)
memory	-	(idle)
arithmetic	-	(idle)
	-	
	-	
	-	

Pipelined execution

	-	
	-	
	sub rax, [rdi]	
	imul rbx, [rsi]	
decoder	add rcx, [rbp]	decode "add" _____
memory	-	(idle)_____
arithmetic	-	(idle)_____
	-	
	-	
	-	

Pipelined execution

	-	
	-	
	-	
	sub rax, [rdi]	
decoder	imul rbx, [rsi]	decode "imul" ________
memory	add rcx, [rbp]	fetch [rbp] __________
arithmetic	-	(idle)_____________
	-	
	-	
	-	

Pipelined execution

	-	
	-	
	-	
	-	
decoder	`sub rax, [rdi]`	`decode "sub" _____`
memory	`imul rbx, [rsi]`	`fetch [rsi] _____`
arithmetic	`add rcx, [rbp]`	`compute rcx + [rbp] _`
	-	
	-	
	-	

Pipelined execution

	-	
	-	
	-	
	-	
decoder	-	(idle) _____
memory	sub rax, [rdi]	fetch [rdi] _____
arithmetic	imul rbx, [rsi]	compute rbx + [rsi] _
	add rcx, [rbp]	
	-	
	-	

Pipelined execution

	-	
	-	
	-	
	-	
decoder	-	(idle) _____
memory	-	(idle) _____
arithmetic	sub rax, [rdi]	compute rax + [rdi] _
	imul rbx, [rsi]	
	add rcx, [rbp]	
	-	

Pipelined execution

	-	
	-	
	-	
	-	
decoder	-	(idle) _____
memory	-	(idle) _____
arithmetic	-	(idle) _____
	`sub rax, [rdi]`	
	`imul rbx, [rsi]`	
	`add rcx, [rbp]`	

Throughput vs. latency

- Latency:
  - Executing each instruction still takes 3 cycles!

- Throughput:
  - But on average, we execute up to 1 instruction per cycle.

Data dependencies

	`mul rcx, [rdx]`	
	`add rdx, [rsi]`	
	-	
decoder	-	`(idle)_____`
memory	-	`(idle)_____`
arithmetic	-	`(idle)_____`
	-	
	-	
	-	

Data dependencies

	-	
	`mul rcx, [rdx]`	
	`add rdx, [rsi]`	
decoder	-	`(idle)`
memory	-	`(idle)`
arithmetic	-	`(idle)`
	-	
	-	
	-	

Data dependencies

	-	
	-	
	`mul rcx, [rdx]`	
decoder	`add rax, [rsi]`	`decode "add"`
memory	-	`(idle)`
arithmetic	-	`(idle)`
	-	
	-	
	-	

Data dependencies

	-	
	-	
	-	
decoder	`mul rcx, [rdx]`	`decode "mul" _____`
memory	`add rdx, [rsi]`	`fetch [rsi] _____`
arithmetic	-	`(idle)_____`
	-	
	-	
	-	

Data dependencies

	-	
	-	
	-	
decoder	-	(idle) _____
memory	mul rcx, [rdx]	(rdx not ready) _____
arithmetic	add rdx, [rsi]	add [rsi] to rdx ____
	-	
	-	
	-	

Data dependencies

	-	
	-	
	-	
decoder	-	(idle) _____
memory	mul rcx, [rdx]	fetch [rdx] _____
arithmetic	-	(idle) _____
	add rdx, [rsi]	
	-	
	-	

Data dependencies

	-	
	-	
	-	
decoder	-	(idle) _____
memory	-	(idle) _____
arithmetic	`mul rcx, [rdx]`	`compute rcx * [rdx] _`
	-	
	`add rdx, [rsi]`	
	-	

Data dependencies

	-	
	-	
	-	
decoder	-	(idle) _____
memory	-	(idle) _____
arithmetic	-	(idle)_____
	`mul rcx, [rdx]`	
	-	
	`add rdx, [rsi]`	

Conditional branching

```
if (a < b) {
    YYY
}
ZZZ
```

```
        cmp     rdi, rsi
        jge     .L1
        YYY
.L1:
    ZZZ
```

	YYY	
	jge .L1	
	cmp rdi, rsi	
decoder	-	(idle)
memory	-	(idle)
arithmetic	-	(idle)
	-	
	-	
	-	

Conditional branching

```
if (a < b) {
    YYY
}
ZZZ
```

```
        cmp     rdi, rsi
        jge     .L1
        YYY
.L1:
    ZZZ
```

	-	
	YYY	
	jge .L1	
decoder	cmp rdi, rsi	decode "cmp"
memory	-	(idle)
arithmetic	-	(idle)
	-	
	-	
	-	

Conditional branching

```
if (a < b) {
    YYY
}
ZZZ
```

```
        cmp     rdi, rsi
        jge     .L1
        YYY
.L1:
    ZZZ
```

	-	
	-	
	YYY	
decoder	jge .L1	decode "jge" _____
memory	cmp rdi, rsi	(still idle) _____
arithmetic	-	(idle)_____
	-	
	-	
	-	

Conditional branching

```
if (a < b) {
    YYY
}
ZZZ
```

```
        cmp     rdi, rsi
        jge     .L1
        YYY
.L1:
    ZZZ
```

	-	
	-	
	-	
decoder	YYY or ZZZ ???	choose one or wait? _
memory	jge .L1	(still idle) _____
arithmetic	cmp rdi, rsi	compare rdi and rsi _
	-	
	-	
	-	

Branch prediction

- in practice, the CPU will try to predict which branch will be taken

  (based on past choices at that specific instruction)

- and speculatively choose that branch

	-	
	YYY 3	
	YYY 2	
decoder	YYY	decoding YYY _____
memory	jge .L1	(still idle) _____
arithmetic	cmp rdi, rsi	compare rdi and rsi _
	-	
	-	
	-	

	-	
	-	
	YYY 3	
decoder	YYY 2	decoding YYY 2 _____
memory	YYY	(still idle) _____
arithmetic	jge .L1	decide taken branch _
	cmp rdi, rsi	
	-	
	-	

	-	
	-	
	ZZZ	← going here instead
decoder	YYY 3	Misprediction!_____
memory	YYY 2	Misprediction!_____
arithmetic	YYY	Misprediction! _____
	jge .L1	
	cmp rdi, rsi	
	-	

	-	
	-	
	-	
decoder	ZZZ	decoding "ZZZ" _____
memory	YYY 3	(idle) _____
arithmetic	YYY 2	(idle) _____
	YYY	
	jge .L1	
	cmp rdi, rsi	

In practice

- When all goes perfect, processors can actually execute more than one instruction per cycle
- Modern processor pipelines have between 5 and 40 stages
- At each stage, there are multiple circuitry blocks

  (decoders, arithmetic and logic unit (ALU) “ports”, etc.)
- Branch mispredict penalty is typically $\geq$ 10 cycles
- Main memory latency is 50–200 cycles

How do we write good code?

- These parameters vary widely from CPU to CPU
- Specific characteristics are often not public
- It is almost impossible to predict the number of cycles a given set of instructions will take

  (in the presence of branches and memory accesses)
- $\Rightarrow$ Qualitatively: we try to understand the phenomena at play
- $\Rightarrow$ Quantitatively: We measure at runtime

Out-of-order execution

	`add rdx, rdi`		
	`add rcx, rbx`		
	`mov rax, [rsi]`		
memory	`____________________`		
ALU 1	`____________________`	`____________________`	ALU 2
	-		
	-		
	-		

Out-of-order execution

	-	
	add rdx, rdi	
	add rcx, rbx	
memory	mov rax, [rsi] _	
ALU 1	_____	_____ ALU 2
	-	
	-	
	-	

Out-of-order execution

	-	
	-	
	-	
memory	`mov rax, [rsi] _`	
ALU 1	`add rdx, rdi ___` `add rcx, rbx ___`	ALU 2
	-	
	-	
	-	

Out-of-order execution

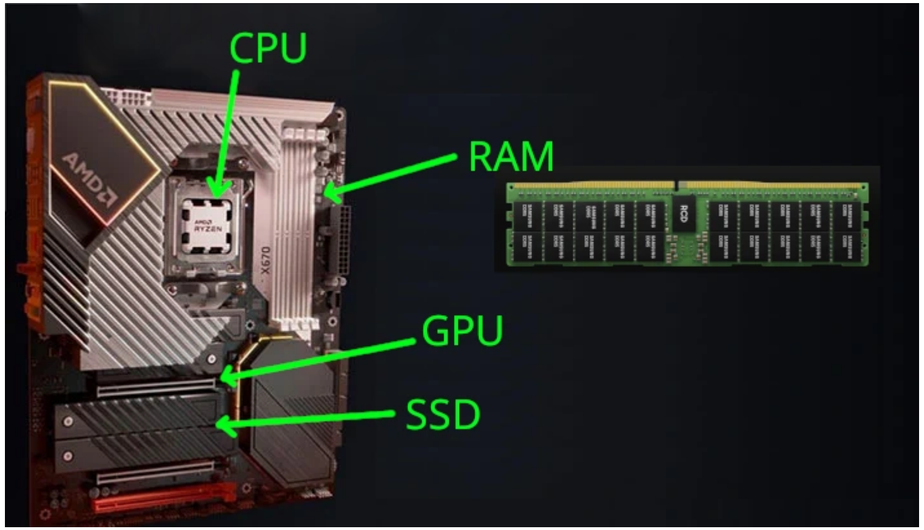
	-	
	-	
	-	
memory	`mov rax, [rsi] _`	
ALU 1	_____ _____	ALU 2
	`add rcx, rbx`	
	`add rdx, rdi`	
	-	

Out-of-order execution

	-	
	-	
	-	
memory	_____	
ALU 1	_____	_____ ALU 2
	`mov rax, [rsi]`	
	`add rcx, rbx`	
	`add rdx, rdi`	

MEMORY

Access to memory (“random access memory” or RAM) is slow



On desktop computers, RAM is typically on distinct integrated circuit (IC) packages, physically centimeters away from the CPU.

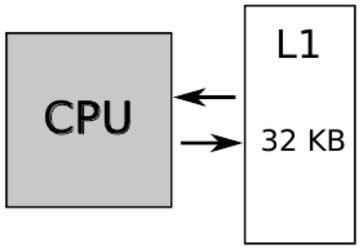
Solution: caching

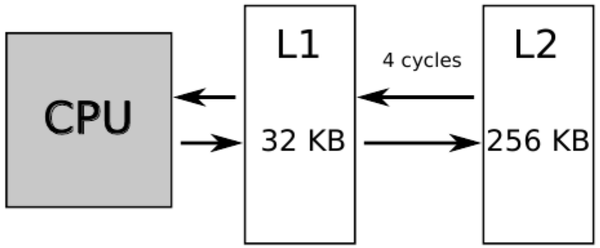
Caching

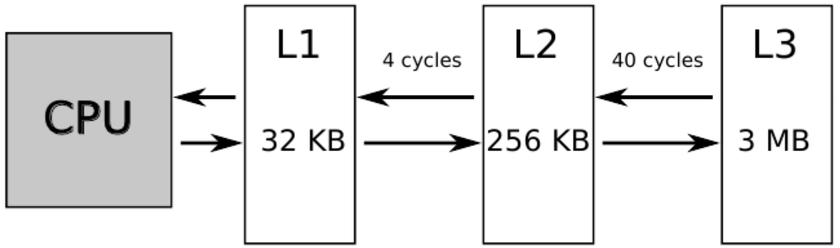
Level 1 (“L1”) cache:

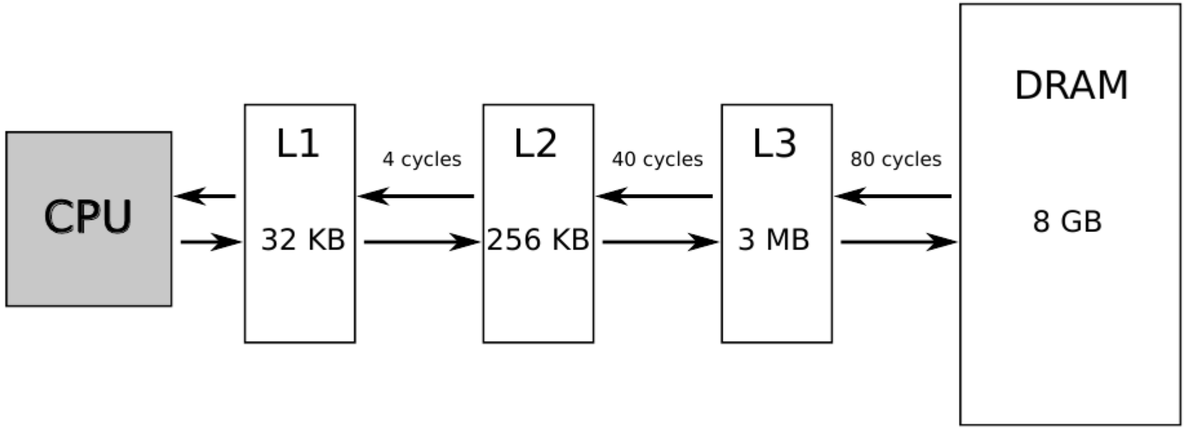
- The CPU contains a small amount of extremely fast memory
- This memory requires many of logic gates on-package
- But it is always available (no latency)
- The CPU contains logic to decide which part of the main memory gets stored in its L1 cache
- This continuously changes over time

- Level 2 (“L2”) cache:
  - slower than L1
  - but requires fewer logic gates, so we can have more
- Level 3 (“L3”) cache:
  - slower than L2
  - but requires fewer logic gates, so we can have more


CPU
L1
32 KB


CPU
L1
32 KB
4 cycles
L2
256 KB


CPU
L1
32 KB
4 cycles
L2
256 KB
40 cycles
L3
3 MB


CPU
L1
32 KB
4 cycles
L2
256 KB
40 cycles
L3
3 MB
80 cycles
DRAM
8 GB

Typical configuration

- memory transits through in units of one cache line
  - 64 bytes on x86_64
  - 128 bytes on M1 Macs
- there is no concept of locality beyond cache lines
- every memory access is performed through L1 cache
- when all cache entries are full, we need to overwrite one
  - $\longrightarrow$ cache eviction policies e.g. least-recently used (LRU)
- pipelined CPUs feature a memory prefetcher (speculatively fills caches in advance)

Zen 4 Cache

	L1I cache	L1D cache	L2 cache	L3 cache
Cache size	32kB	32kB	1MB	xxx
Associativity	8 way	8 way	8 way	xxx
Cache line size	64 b	64 b	64 b	64 b

How do we write good code?

- Again, cache operation varies widely from CPU to CPU
- It is almost impossible to predict how it will behave with complex instruction streams
- $\Rightarrow$ Qualitatively: we try to understand how caches work
- $\Rightarrow$ Quantitatively: We measure at runtime