

# Modern C++ Programming

## 19. PERFORMANCE OPTIMIZATION I BASIC CONCEPTS

---

*Federico Busato*

2023-09-11

## 1 Overview

## 2 Basic Concepts

- Asymptotic Complexity
- Time-Memory Trade-off
- Developing Cycle
- Ahmdal's Law
- Throughput, Bandwidth, Latency
- Performance Bounds
- Arithmetic Intensity

## **3 Basic Architecture Concepts**

- Instruction-Level Parallelism
- Little's Law
- Data-Level Parallelism (DLP)
- Thread-Level Parallelism (TLP)
- Single Instruction Multiple Threads (SIMT)
- RISC, CISC Instruction Sets

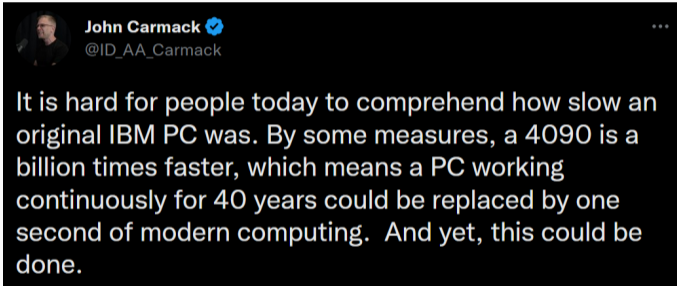
## **4 Memory Hierarchy**

- Memory Hierarchy Concepts
- Memory Locality

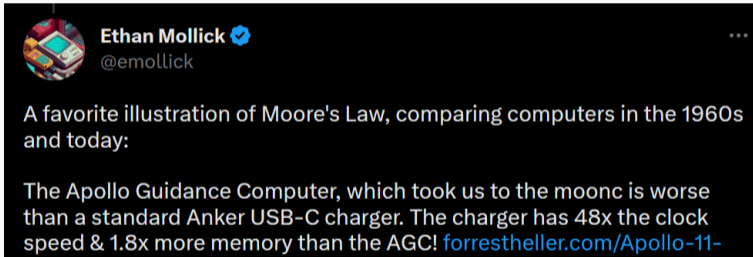
# Overview

---

# Performance and Technological Progress

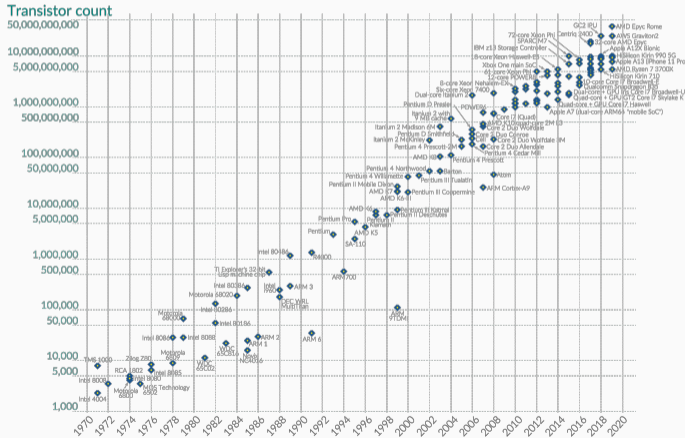


# Performance and Technological Progress

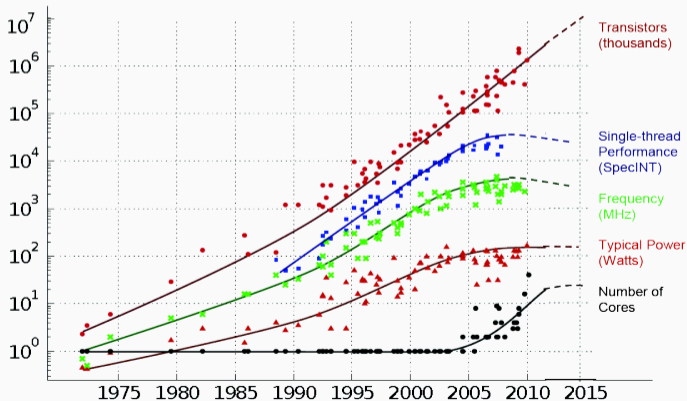


*"The number of transistors incorporated in a chip will approximately double every 24 months." (40% per year)*

**Gordon Moore, Intel co-founder**

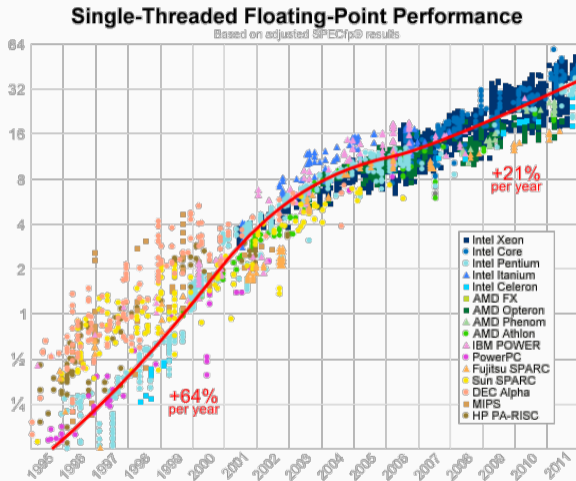


The Moore's Law is not (yet) dead but the same concept is not true for *clock frequency, single-thread performance, power consumption, and cost*. How we can provide value?





# Single-Thread Performance Trend



A Look Back at Single-Threaded CPU Performance

Herb Sutter, The Free Lunch Is Over



## Reasons for Optimizing

- In the first decades, the *computer performance was extremely limited*. Low-level optimizations were essential to fully exploit the hardware
- Modern systems provide much higher performance, but *we cannot more rely on hardware improvement* on short-period
- Performance and efficiency add market value (fast program for a given task), e.g. search, page loading, etc.
- Optimized code uses less resources, e.g. in a program that runs on a server for months or years, a small reduction in the execution time/power consumption translates in a big saving of power consumption

# The Role of Software Engineering and Algorithms

## *Forget Moore's Law. Algorithms drive technology forward*

*"Algorithmic improvements make more efficient use of existing resources and allow computers to do a task faster, cheaper, or both. Think of how easy the smaller MP3 format made music storage and transfer. That compression was because of an algorithm."*

Technology	<pre>01010011 01100011 01101001 01100101 01101110 01100011 01100101 00000000</pre>		
	<b>Software</b>	<b>Algorithms</b>	<b>Hardware architecture</b>
Opportunity	Software performance engineering	New algorithms	Hardware streamlining
Examples	Removing software bloat Tailoring software to hardware features	New problem domains New machine models	Processor simplification Domain specialization

- Forget Moore's Law
- What will drive computer performance after Moore's law?

## Going the Other Way

- Computing systems are unfathomably complex
- Optimization is complicated and surprising
- Doing something sensible had opposite effect
- We often try clever things that don't work
  
- How about trying something silly then?

25 / 72

from *"Speed is Found in the Minds of People"*,  
**Andrei Alexandrescu**, CppCon 2019

## References

- `Optimized C++`, *Kurt Guntheroth*
- `Awesome C/C++ performance optimization resources`, *Bartłomiej Filipek*
- `Optimizing C++`, *wikibook*
- `Optimizing software in C++`, *Agner Fog*
- `Hacker Delight (2nd)`, *Henry S. Warren*
- `Algorithmica: Algorithms for Modern Hardware`
- `What scientists must know about hardware to write fast code`

# Basic Concepts

---

The **asymptotic analysis** refers to estimate the execution time or memory usage as function of the input size (the *order of growing*)

The *asymptotic behavior* is opposed to a *low-level analysis* of the code (instruction/loop counting/weighting, cache accesses, etc.)

## Drawbacks:

- The *worst-case* is not the *average-case*
- Asymptotic complexity does not consider small inputs (think to *insertion sort*)
- The hidden constant can be relevant in practice
- Asymptotic complexity does not consider instructions cost and hardware details

Be aware that only **real-world problems** with a small asymptotic complexity or small size can be solved in a “*user*” *acceptable time*

Three examples:

- *Sorting*:  $\mathcal{O}(n \log n)$ , try to sort an array of some billion elements
- *Diameter of a (sparse) graph*:  $\mathcal{O}(V^2)$ , just for graphs with a few hundred thousand vertices it becomes impractical without advanced techniques
- *Matrix multiplication*:  $\mathcal{O}(N^3)$ , even for small sizes  $N$  (e.g. 8K, 16K), it requires special accelerators (e.g. GPU, TPU, etc.) for achieving acceptable performance



# Time-Memory Trade-off

The **time-memory trade-off** is a way of solving a problem or calculation in less time by using more storage space (less often the opposite direction)

Examples:

- *Memoization* (e.g. used in dynamic programming): returning the cached result when the same inputs occur again
- *Hash table*: number of entries vs. efficiency
- *Lookup tables*: precomputed data instead branches
- *Uncompressed data*: bitmap image vs. jpeg

*“If you’re not writing a program, don’t use a programming language”*

**Leslie Lamport**, Turing Award

*“First solve the problem, then write the code”*

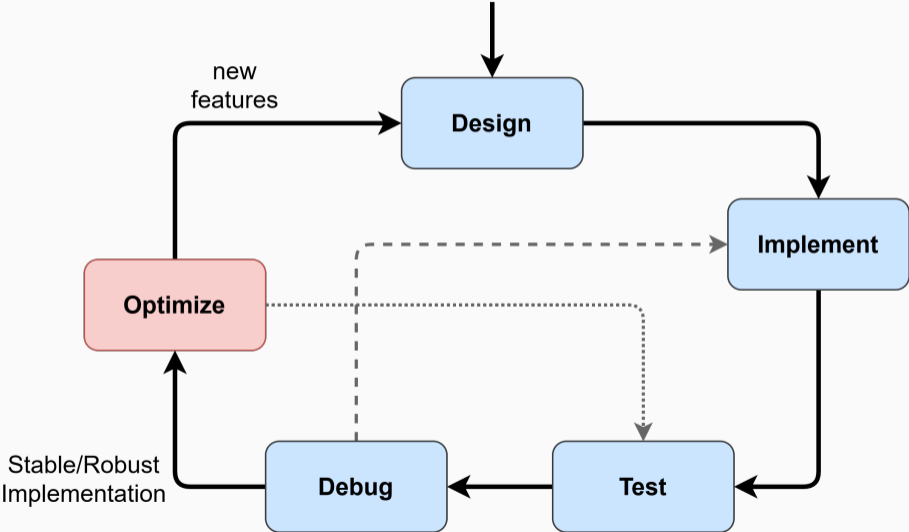
*“Inside every large program is an algorithm trying to get out”*

**Tony Hoare**, Turing Award

*“Premature optimization is the root of all evil”*

**Donald Knuth**, Turing Award

*“Code for correctness first, then optimize!”*



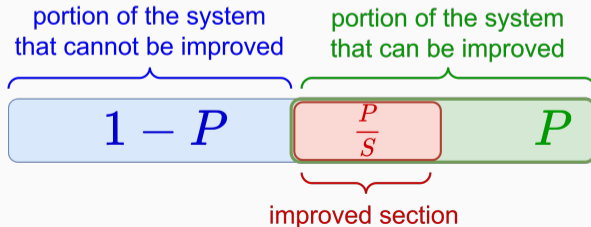
- One of the most important phase of the optimization cycle is the **application profiling** for finding regions of code that are *critical for performance* (**hotspot**)
  - Expensive code region (absolute)
  - Code regions executed many times (cumulative)
- Most of the times, **there is no the perfect algorithm for all cases** (e.g. insertion, merge, radix sort). Optimizing also refers in finding the correct heuristics for different program inputs/platforms instead of modifying the existing code

**Ahmdal's Law**

The **Ahmdal's law** expresses the maximum improvement possible by improving a particular part of a system

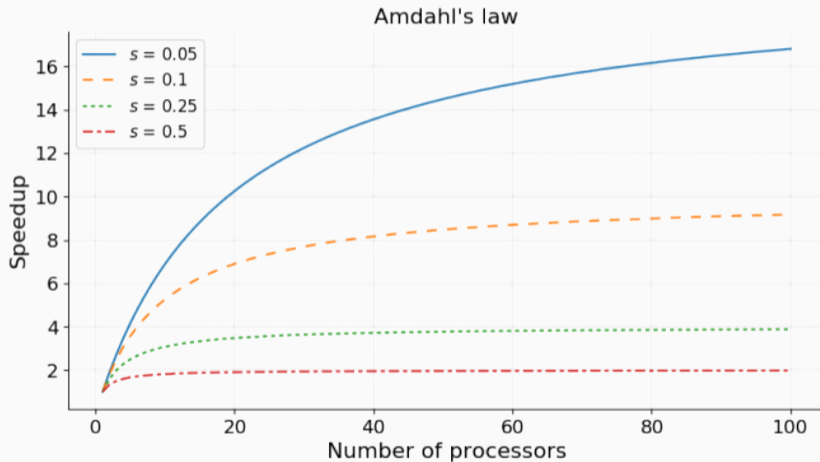
*Observation:* The performance of any system is constrained by the speed of the slowest point

$S$  : improvement factor expressed as a factor of  $P$



$$\text{Overall Improvement} = \frac{1}{(1 - P) + \frac{P}{S}}$$

P \ S	25%	50%	75%	2x	3x	4x	5x	10x	∞
10%	1.02x	1.03x	1.04x	1.05x	1.07x	1.08x	1.09x	1.10x	1.11x
20%	1.04x	1.07x	1.09x	1.11x	1.15x	1.18x	1.19x	1.22x	1.25x
30%	1.06x	1.11x	1.15x	1.18x	1.25x	1.29x	1.31x	1.37x	1.49x
40%	1.09x	1.15x	1.20x	1.25x	1.36x	1.43x	1.47x	1.56x	1.67x
50%	1.11x	1.20x	1.27x	1.33x	1.50x	1.60x	1.66x	1.82x	2.00x
60%	1.37x	1.25x	1.35x	1.43x	1.67x	1.82x	1.92x	2.17x	2.50x
70%	1.16x	1.30x	1.43x	1.54x	1.88x	2.10x	2.27x	2.70x	3.33x
80%	1.19x	1.36x	1.52x	1.67x	2.14x	2.50x	2.78x	3.57x	5.00x
90%	1.22x	1.43x	1.63x	1.82x	2.50x	3.08x	3.57x	5.26x	10.00x



note:  $s$  is the portion of the system that cannot be improved

# Throughput, Bandwidth, Latency

The **throughput** is the rate at which operations are performed

*Peak throughput:*

(CPU speed in Hz) x (CPU instructions per cycle) x  
(number of CPU cores) x (number of CPUs per node)

NOTE: modern processors have more than one computation unit

The **memory bandwidth** is the amount of data that can be loaded from or stored into a particular memory space

*Peak bandwidth:*

(Frequency in Hz) x (Bus width in bit / 8) x (Pump rate, memory type multiplier)

The **latency** is the amount of time needed for an operation to complete



The performance of a program is *bounded* by one or more aspects of its computation. This is also strictly related to the underlying hardware

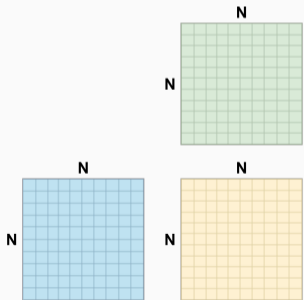
- **Memory-bound.** The program spends its time primarily in performing *memory accesses*. The performance is limited by the *memory bandwidth* (rarely memory-bound also refers to the amount of memory available)
- **Compute-bound** (Math-bound). The program spends its time primarily in computing *arithmetic instructions*. The performance is limited by the *speed of the CPU*

- **Latency-bound.** The program spends its time primarily in waiting *the data are ready* (instruction/memory dependencies). The performance is limited by the *latency of the CPU/memory*
- **I/O Bound.** The program spends its time primarily in performing *I/O operations* (network, user input, storage, etc.). The performance is limited by the *speed of the I/O subsystem*

**Arithmetic Intensity**

**Arithmetic/Operational Intensity** is the ratio of total operations to total data movement (bytes or words)

The naive matrix multiplication algorithm requires  $N^3 \cdot 2$  floating-point operations\* (multiplication + addition), while it involves  $(N^2 \cdot 4B) \cdot 3$  data movement



---

\* What Is a Flop?

$$R = \frac{\text{ops}}{\text{bytes}} = \frac{2n^3}{12n^2} = \frac{n}{6}$$

which means that for every byte accessed, the algorithm performs  $\frac{n}{6}$  operations → **compute-bound**

N	Operations	Data Movement	Ratio	Exec. Time
512	$268 \cdot 10^6$	3 MB	85	2 ms
1024	$2 \cdot 10^9$	12 MB	170	21 ms
2048	$17 \cdot 10^9$	50 MB	341	170 ms
4096	$137 \cdot 10^9$	201 MB	682	1.3 s
8192	$1 \cdot 10^{12}$	806 MB	1365	11 s
16384	$9 \cdot 10^{12}$	3 GB	2730	90 s

A modern CPU performs 100 GFlops, and has about 50 GB/s memory bandwidth

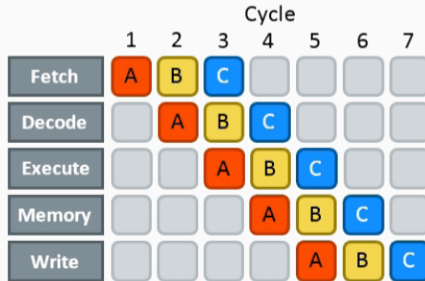
# Basic Architecture Concepts

---

Modern processor architectures are deeply pipelined → superscalar processor

**Instruction-Level Parallelism (ILP)** is a measure of how many instructions in a computer program can be executed simultaneously by issuing *independent* instructions in sequence (*out-of-order*)

**Instruction pipelining** is a technique for implementing ILP within a single processor



Microarchitecture	Pipeline stages
Core	14
Bonnell	16
Sandy Bridge	14
Silvermont	14 to 17
Haswell	14
Skylake	14
Kabylake	14

The pipeline efficiency is affected by

- **Instruction stalls**, e.g. cache miss, an execution unit not available, etc.
- **Bad speculation**, branch misprediction

```
for (int i = 0; i < N; i++) // with no optimizations, the loop
    C[i] = A[i] * B[i];     // is executed in sequence
```

can be rewritten as:

```
for (int i = 0; i < N; i += 4) { // four independent multiplications
    C[i]      = A[i]      * B[i]; // per iteration
    C[i + 1] = A[i + 1] * B[i + 1]; // A, B, C are not alias
    C[i + 2] = A[i + 2] * B[i + 2];
    C[i + 3] = A[i + 3] * B[i + 3];
}
```

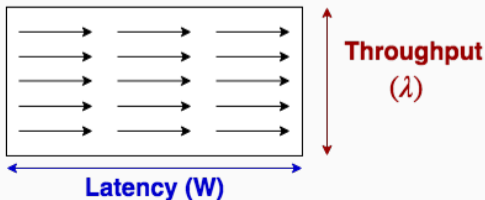


## ILP and Little's Law

The **Little's Law** expresses the relation between *latency* and *throughput*. The *throughput* of a system  $\lambda$  is equal to the number of elements in the system divided by the average time spent (*latency*)  $W$  for each element in the system:

$$L = \lambda W \quad \rightarrow \quad \lambda = \frac{L}{W}$$

- $L$ : average number of customers in a store
- $\lambda$ : arrival rate (*throughput*)
- $W$ : average time spent (*latency*)



## Data-Level Parallelism (DLP)

**Data-Level Parallelism (DLP)** refers to the execution of the same operation on multiple data in parallel

*Vector processors or array processors* provide SIMD (*Single Instruction-Multiple Data*) or vector instructions for exploiting data-level parallelism

The popular vector instruction sets are:

**MMX** *MultiMedia eXtension*. 80-bit width (Intel, AMD)

**SSE** (SSE2, SSE3, SSE4) *Streaming SIMD Extensions*. 128-bit width (Intel, AMD)

**AVX** (AVX, AVX2, AVX-512) *Advanced Vector Extensions*. 512-bit width (Intel, AMD)

**NEON** *Media Processing Engine*. 128-bit width (ARM)

**SVE** (SVE, SVE2) *Scalable Vector Extension*. 128-2048 bit width (ARM)

# Thread-Level Parallelism (TLP)

A **thread** is a single sequential execution flow within a program with its state (instructions, data, PC, register state, and so on)

**Thread-level parallelism (TLP)** refers to the execution of separate computation “*thread*” on different processing units (e.g. CPU cores)

## Single Instruction Multiple Threads (SIMT)

An alternative approach to the classical data-level parallelism is **Single Instruction Multiple Threads (SIMT)**, where multiple threads execute the same instruction simultaneously, with each thread operating on different data.

GPUs are successful examples of SIMT architectures.

**SIMT** can be thought of as an evolution of *SIMD* (Single Instruction Multiple Data). *SIMD* requires that all data processed by the instruction be of the same type and requires no dependencies or inter-thread communication. On the other hand, **SIMT** is more flexible and does not have these restrictions. Each thread has access to its own memory and can operate independently.

The **Instruction Set Architecture** (ISA) is an abstract model of the CPU to represent its behavior. It consists of addressing modes, instructions, data types, registers, memory architecture, interrupt, etc.

It does not defined how an instruction is processed

The **microarchitecture** ( $\mu$ arch) is the implementation of an **ISA** which includes pipelines, caches, etc.

## Complex Instruction Set Computer (CISC)

- Complex instructions for special tasks even if used infrequently
- Assembly instructions follow software. Little compiler effort for translating high-level language into assembly
- Initially designed for saving cost of computer memory and disk storage (1960)
- High number of instructions with different size
- Instructions require complex micro-ops decoding (translation) for exploiting ILP
- Multiple low-level instructions per clock but with high latency

### *Hardware implications*

- High number of transistors
- Extra logic for decoding. Heat dissipation
- Hard to scale

## Reduced Instruction Set Computer (RISC)

- Simple instructions
- Small number of instructions with fixed size
- 1 clock per instruction
- Assembly instructions does not follow software
- No instruction decoding

### *Hardware implications*

- High ILP, easy to schedule
- Small number of transistors
- Little power consumption
- Easy to scale

# Instruction Set Comparison

## x86 Instruction set

```
MOV AX, 15; AH = 00, AL = 0Fh  
AAA; AH = 01, AL = 05  
RET
```

## ARM Instruction set

```
MOV R3, # 10  
AND R2, R0, # 0xF  
CMP R2, R3  
IT LT  
BLT elsebranch  
ADD R2, # 6  
ADD R1, #1  
elsebranch:  
END
```



# CISC vs. RISC

- **Hardware market:**

- *RISC* (ARM, IBM): Qualcomm Snapdragon, Amazon Graviton, Nvidia Grace, Nintendo Switch, Fujitsu Fukaku, Apple M1, Apple Iphone/Ipod/Mac, Tesla Full Self-Driving Chip, PowerPC
- *CISC* (Intel, AMD): all x86-64 processors

- **Software market:**

- *RISC*: Android, Linux, Apple OS, Windows
- *CISC*: Windows, Linux

- **Power consumption:**

- *CISC*: Intel i5 10th Generation: 64W
- *RISC*: Arm-based smartphone < 5W

*“Incidentally, the first ARM1 chips required so little power, when the first one from the factory was plugged into the development system to test it, the microprocessor immediately sprung to life by drawing current from the IO interface – before its own power supply could be properly connected.”*

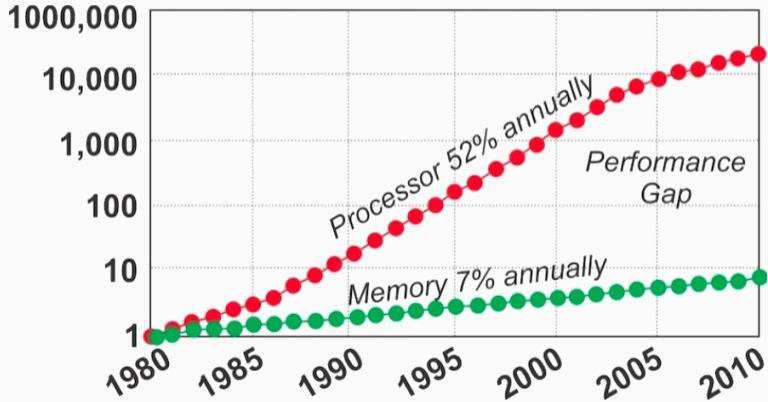
---

Happy birthday, ARM1. It is 35 years since Britain's Acorn RISC Machine chip sipped power for the first time

# Memory Hierarchy

---

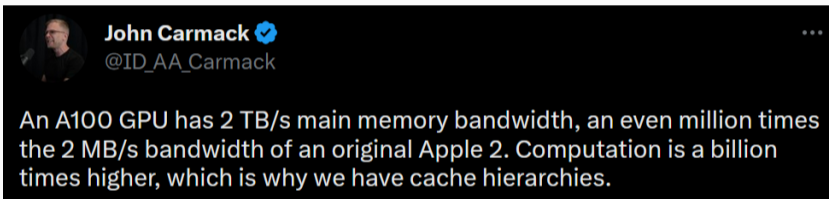
## Access to memory dominates other costs in a processor



# The Von Neumann Bottleneck

The efficiency of computer architectures is limited by the **Memory Wall** problem, namely the memory is the slowest part of the system

Moving data to and from main memory consumes the vast majority of *time* and *energy* of the system



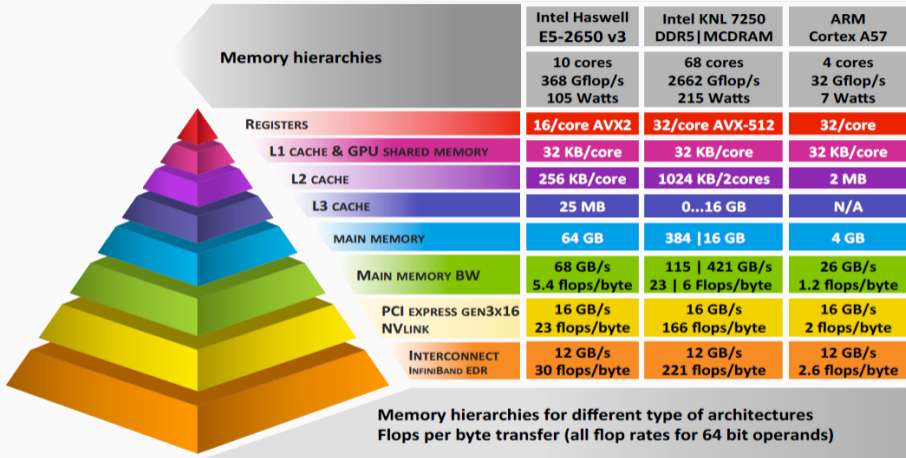
Modern architectures rely on complex memory hierarchy (primary memory, caches, registers, scratchpad memory, etc.). Each level has different characteristics and constrains (size, latency, bandwidth, concurrent accesses, etc.)



*1 byte of RAM (1946)*



*IBM 5MB hard drive (1956)*



Source:

*“Accelerating Linear Algebra on Small Matrices from Batched BLAS to Large Scale Solvers”,  
ICL, University of Tennessee*



Intel Alder Lake 12th-gen Core-i9-12900k (Q1'21) + DDR4-3733 example:

Hierarchy level	Size	Latency	Latency Ratio	Bandwidth	Bandwidth Ratio
L1 cache	192 KB	1 ns	1.0x	1,600 GB/s	1.0x
L2 cache	1.5 MB	3 ns	3x	1,200 GB/s	1.3x
L3 cache	12 MB	6 - 20 ns	6-20x	900 GB/s	1.7x
DRAM	/	50 - 90 ns	50-90x	80 GB/s	20x
SDD Disk (swap)	/	70 $\mu$ s	10 <sup>5</sup> x	2 GB/s	800x
HDD Disk (swap)	/	10 ms	10 <sup>7</sup> x	2 GB/s	800x

- [en.wikichip.org/wiki/WikiChip](https://en.wikichip.org/wiki/WikiChip)
- Memory Bandwidth Napkin Math

A **cache** is a small and fast memory located close to the processor that stores frequently used instructions and data. It is part of the processor package and takes 40 to 60 percent of the chip area

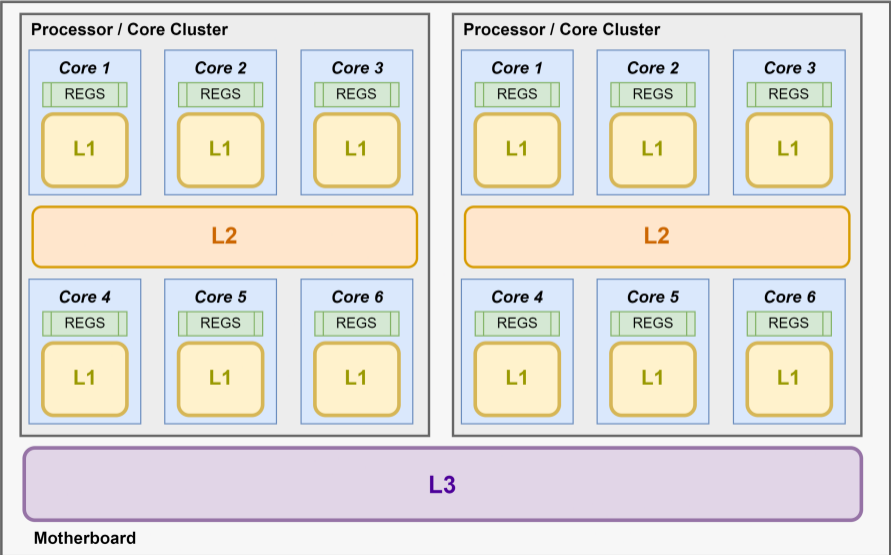
*Characteristics and content:*

**Registers** Program counter (PC), General purpose registers, Instruction Register (IR), etc.

**L1 Cache** Instruction cache and data cache, private/exclusive per CPU core, located on-chip

**L2 Cache** Private/exclusive per single CPU core or a cluster of cores, located off-chip

**L3 Cache** Shared between all cores and located off-chip (e.g. motherboard), up to 128/256MB



A **cache line** or **cache block** is the unit of data transfer between the cache and main memory, namely the memory is loaded at the *granularity* of a cache line

The typical size of the cache line is 64 bytes. A cache line can be further organized in banks or sectors

*Cache access type:*

**Hot** Closest-processor cached, L1

**Warm** L2 or L3 caches

**Cold** First load, cache empty

- A **cache hit** occurs when a requested data is *successfully found* in the cache memory
- The **cache hit rate** is the number of *cache hits divided by the number of memory requests*
- A **cache miss** occurs when a requested data is *not found* in the cache memory
- The **miss penalty** refers to the *extra time required to load the data* into cache from the main memory when a cache miss occurs
- A **page fault** occurs when a requested data is in the process address space, but *it is not currently located in the main memory* (swap/pagefile)
- Page **thrashing** occurs when page faults are frequent and the OS spends significant time to swap data in and out the physical RAM

# Memory Locality

- **Spatial Locality** refers to the use of data elements within relatively close storage locations e.g. scan arrays in increasing order, matrices by row. It involves mechanisms such as *memory prefetching* and *access granularity*  
When spatial locality is low, many words in the cache line are not used
- **Temporal Locality** refers to the reuse of the same data within a relatively small time duration, and, as consequence, exploit lower levels of the memory hierarchy (caches), e.g. multiple sparse accesses  
*Heavily used memory locations can be accessed more quickly than less heavily used locations*