

Modern C++ Programming

16. PERFORMANCE OPTIMIZATION I

BASIC CONCEPTS

Federico Busato

2022, v3.15

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
- Thread-Level Parallelism
- RISC, CISC Instruction Sets

4 Memory Hierarchy

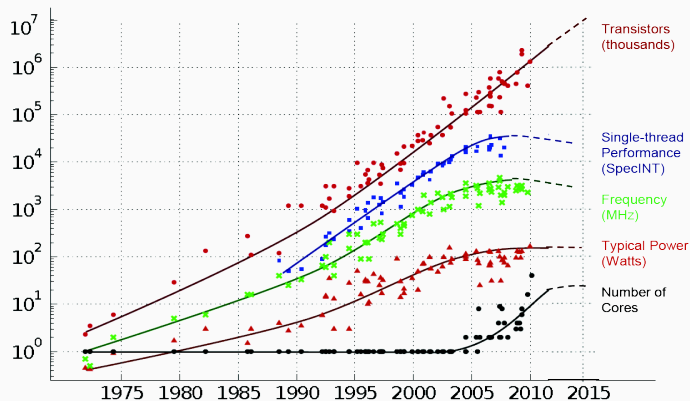
- Memory Hierarchy Concepts
- Memory Locality
- Internal Structure Alignment
- External Structure Alignment

Overview

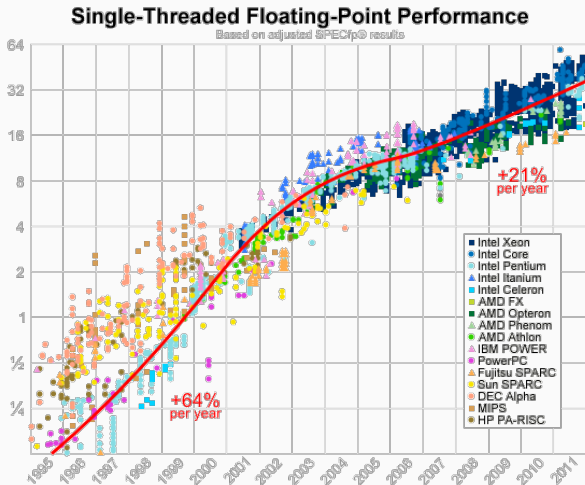
Gordon Moore, Intel co-founder



The Moore's Law is not (yet) dead but the same concept is not true for clock frequency, single-thread performance, and power consumption. 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 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	01010011 01100011 01101001 01100101 01101110 01100011 01100101 00000000		
	Software	Algorithms	Hardware architecture
	Software performance engineering	New algorithms	Hardware streamlining
	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, Bartlomiej Filipek`
- `Optimizing C++, wikibook`
- `Optimizing software in C++, Agner Fog`
- `Hacker Delight (2nd), Henry S. Warren`
- `Algorithms for Modern Hardware`

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 one billion elements (4GB)
- *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"

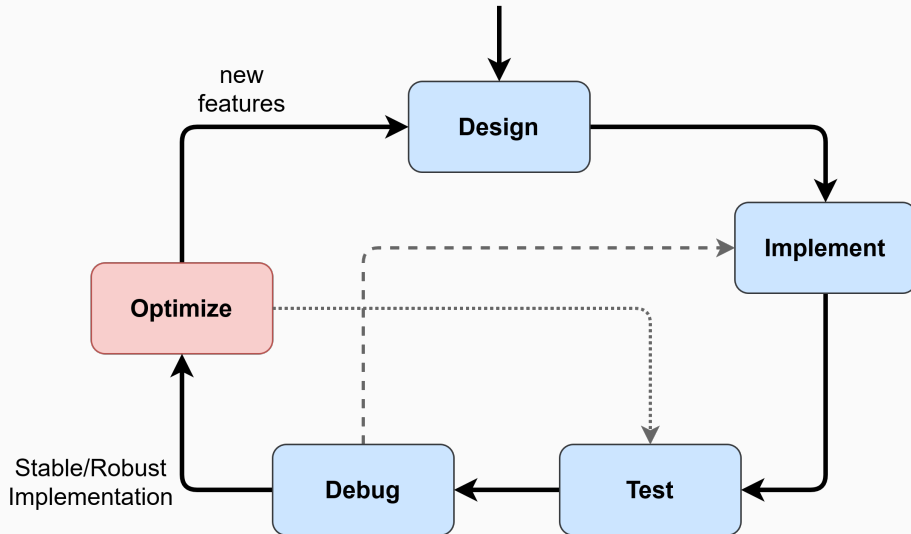
"Inside every large program is an algorithm trying to get out"

Tony Hoare, Turing Award

"Premature optimization is the root of all evil"

Donald Knuth, Turing Award

"Code for correctness first, then optimize!"



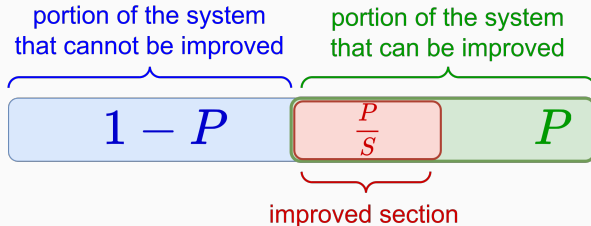
- One of the most important phase of the optimization cycle is the **application profiling** for finding regions of code that are *critical for performance* (**hotspot**)
 - Expensive code region (absolute)
 - Code regions executed many times (cumulative)
- Most of the times, **there is no the perfect 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

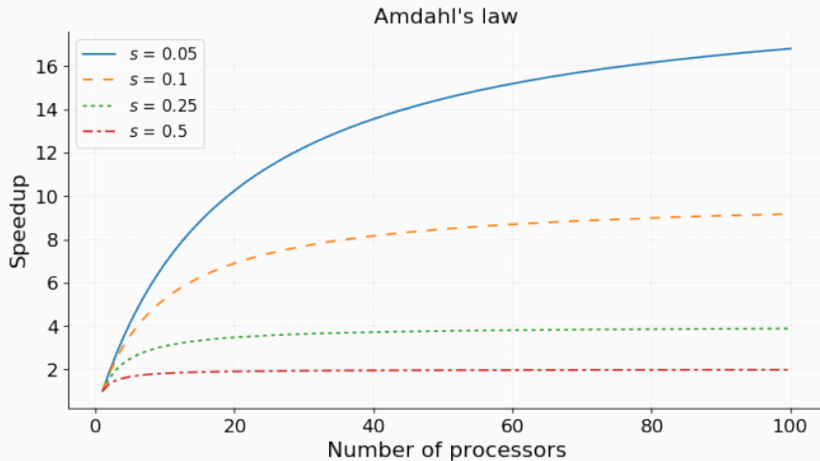
Observation: The performance of any system is constrained by the speed of the slowest point

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:

$(\text{CPU speed in Hz}) \times (\text{CPU instructions per cycle}) \times$
 $(\text{number of CPU cores}) \times (\text{number of CPUs per node})$

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:

$(\text{Frequency in Hz}) \times (\text{Bus width in bit} / 8) \times (\text{Pump rate, memory type multiplier})$

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

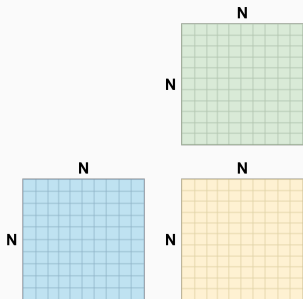
- **Memory-bound.** The program spends its time primarily in performing *memory accesses*. The performance is limited by the *memory bandwidth* (rarely memory-bound also refers to the amount of memory available)
- **Compute-bound.** The program spends its time primarily in computing *arithmetic instructions*.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



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

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

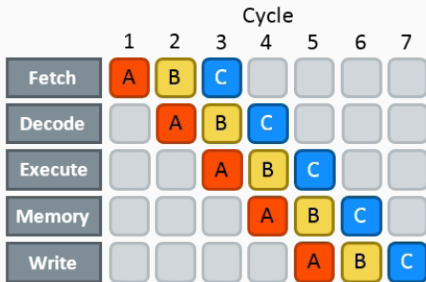
A modern CPU performs 100 GFlops, and has about 50 GB/s memory bandwidth

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:

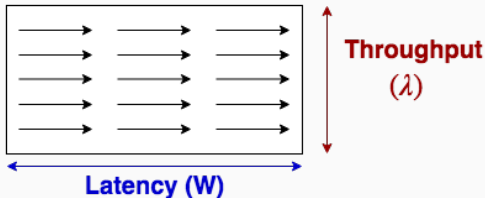
```
for (int i = 0; i < N; i += 4) { // four independent multiplications  
    C[i]      = A[i]      * B[i]; // per iteration  
    C[i + 1] = A[i + 1] * B[i + 1]; // A, B, C are not alias  
    C[i + 2] = A[i + 2] * B[i + 2];  
    C[i + 3] = A[i + 3] * B[i + 3];  
}
```

ILP and Little's Law

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

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

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



Data-Level Parallelism

Data-Level Parallelism 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

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 refers to the execution of separate computation “*thread*” on different processing units

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** (μ 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:**

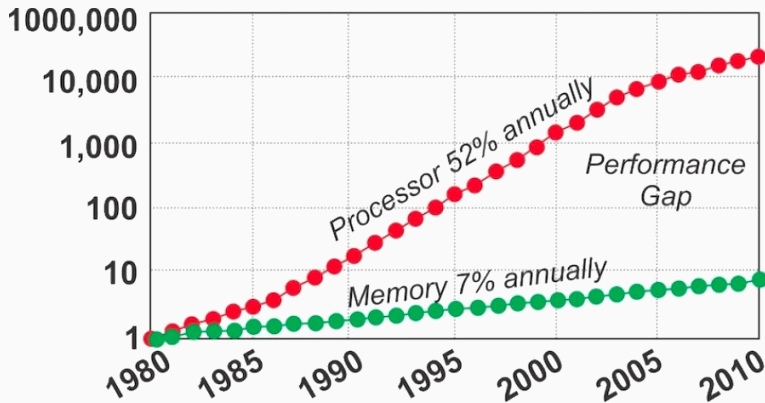
- *CISC*: Intel i5 10th Generation: 64W
- *RISC*: Arm-based smartphone < 5W

“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

**For these reasons, computer architectures need
a sophisticated memory hierarchy**

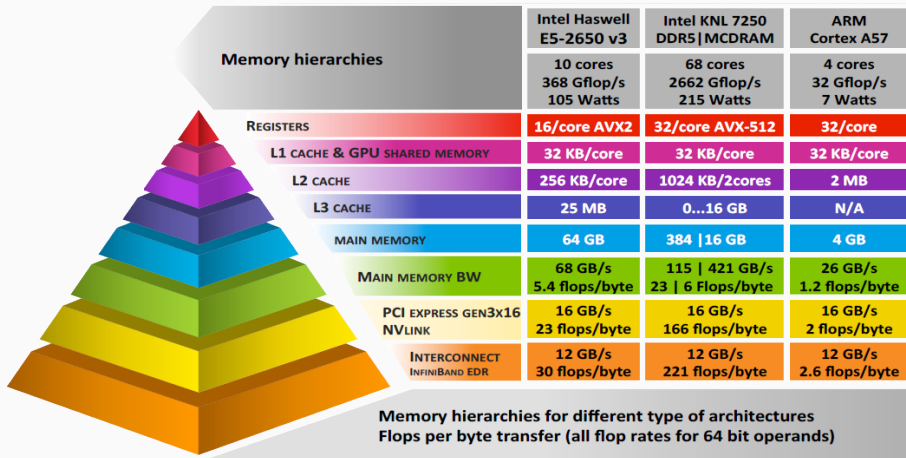
Modern architectures rely on complex memory hierarchy (primary memory, caches, registers, scratchpad memory, etc.). Each level has different characteristics and constrains (size, latency, bandwidth, concurrent accesses, etc.)



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 Coffee Lake Core-i7-8700 example:

Hierarchy level	Size	Latency	Latency Ratio	Bandwidth	Bandwidth Ratio
L1 cache	192 KB	1.5 ns	1.0x	1,600 GB/s	1.0x
L2 cache	1.5 MB	4 ns	2.6x	570 GB/s	2.8x
L3 cache	12 MB	12 - 40 ns	8-27x	320 GB/s	5x
DRAM	/	60 ns	40x	40 GB/s	40x
SDD Disk (swap)	/	70 μ s	4.7 * 10 ⁴ x	2 GB/s	800x
HDD Disk (swap)	/	10 ms	6.6 * 10 ⁶ x	2 GB/s	800x

- 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

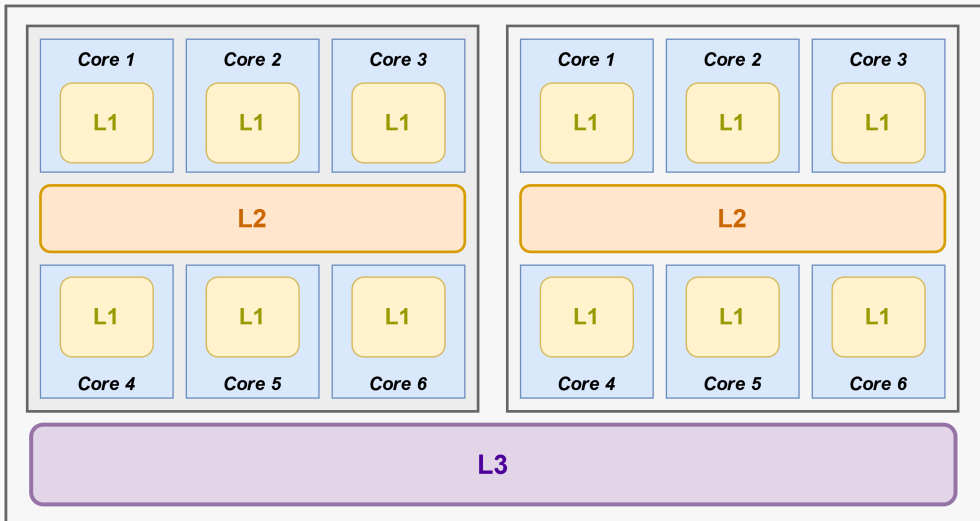
Characteristics and content:

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



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)

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

A, B, C matrices of size $N \times N$

$$C = A * B$$

```
for (int i = 0; i < N; i++) {  
    for (int j = 0; j < N; j++) {  
        int sum = 0;  
        for (int k = 0; k < N; k++)  
            sum += A[i][k] * B[k][j]; // row × column  
        C[i][j] = sum;  
    }  
}
```

$$C = A * B^T$$

```
for (int i = 0; i < N; i++) {  
    for (int j = 0; j < N; j++) {  
        int sum = 0;  
        for (int k = 0; k < N; k++)  
            sum += A[i][k] * B[j][k]; // row × row  
        C[i][j] = sum;  
    }  
}
```

Benchmark:

N	64	128	256	512	1024
A * B	< 1 ms	5 ms	29 ms	141 ms	1,030 ms
A * B ^T	< 1 ms	2 ms	6 ms	48 ms	385 ms
Speedup	/	2.5x	4.8x	2.9x	2.7x

Temporal-Locality Example

Speeding up a random-access function

```
for (int i = 0; i < N; i++)      // V1
    out_array[i] = in_array[hash(i)];
```

```
for (int K = 0; K < N; K += CACHE) { // V2
    for (int i = 0; i < N; i++) {
        auto x = hash(i);
        if (x >= K && x < K + CACHE)
            out_array[i] = in_array[x];
    }
}
```

V1 : 436 ms, V2 : 336 ms \rightarrow 1.3x speedup (temporal locality improvement)

.. but it needs a careful evaluation of `CACHE` and it can even decrease the performance for other sizes

pre-sorted `hash(i)` : 135 ms \rightarrow 3.2x speedup (spatial locality improvement)

Internal Structure Alignment

```
struct A1 {  
    char    x1; // offset 0  
    double  y1; // offset 8!! (not 1)  
    char    x2; // offset 16  
    double  y2; // offset 24  
    char    x3; // offset 32  
    double  y3; // offset 40  
    char    x4; // offset 48  
    double  y4; // offset 56  
    char    x5; // offset 64 (65 bytes)  
}
```

```
struct A2 { // internal alignment  
    char    x1; // offset 0  
    char    x2; // offset 1  
    char    x3; // offset 2  
    char    x4; // offset 3  
    char    x5; // offset 4  
    double  y1; // offset 8  
    double  y2; // offset 16  
    double  y3; // offset 24  
    double  y4; // offset 32 (40 bytes)  
}
```

Considering an *array of structures* (AoS), there are two problems:

- We are wasting 40% of memory in the first case (A1)
- In common x64 processors the cache line is 64 bytes. For the first structure A1 , every access involves two cache line operations (2x slower)

External Structure Alignment and Padding

Considering the previous example for the structure `A2`, random loads from an array of structures `A2` leads to one or two cache line operations depending on the alignment at a specific index, e.g.

index 0 → one cache line load

index 1 → two cache line loads

It is possible to fix the structure alignment in two ways:

- The **memory padding** refers to introduce extra bytes at the end of the data structure to enforce the memory alignment
e.g. add a `char` array of size 24 to the structure `A2`
- **Align keyword or attribute** allows specifying the alignment requirement of a type or an object (next slide)

C++ allows specifying the alignment requirement in different ways:

- C++11 `alignas(N)` only for variable / struct declaration
- C++17 `aligned new` (e.g. `new int[2, N]`)
- Compiler Intrinsic only for variables / struct declaration
 - GCC/Clang: `__attribute__((aligned(N)))`
 - MSVC: `__declspec(align(N))`
- Compiler Intrinsic for dynamic pointer
 - GCC/Clang: `__builtin_assume_aligned(x)`
 - Intel: `__assume_aligned(x)`

Data alignment is also essential to exploit hardware vector instructions (SIMD) like SSE, AVX, etc.


```
struct alignas(16) A1 { // C++11
    int x, y;
};

struct __attribute__((aligned(16))) A2 { // compiler-specific attribute
    int x, y;
};

auto ptr1 = new int[100, 16]; // 16B alignment, C++17
auto ptr2 = new int[100];      // 4B alignment guarantee
auto ptr3 = __builtin_assume_aligned(ptr2, 16); // compiler-specific attribute
auto ptr4 = new A1[10];        // no alignment guarantee
```