



---

# LINMA2710 Scientific Computing

---

SIMON DESMIDT

Academic year 2024-2025 - Q2



UCLouvain

# Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Single Instruction Multiple Data (SIMD)</b> | <b>2</b> |
| <b>2</b> | <b>Shared-Memory Multiprocessing</b>           | <b>3</b> |
| 2.1      | How memory works . . . . .                     | 3        |

# **Single Instruction Multiple Data (SIMD)**

# Shared-Memory Multiprocessing

## 2.1 How memory works

### 2.1.1 Memory hierarchy

To store the data that it will use, the CPU uses memory. Memory is hierarchical like a pyramid. The higher it is, the faster it goes, but the less space there is.

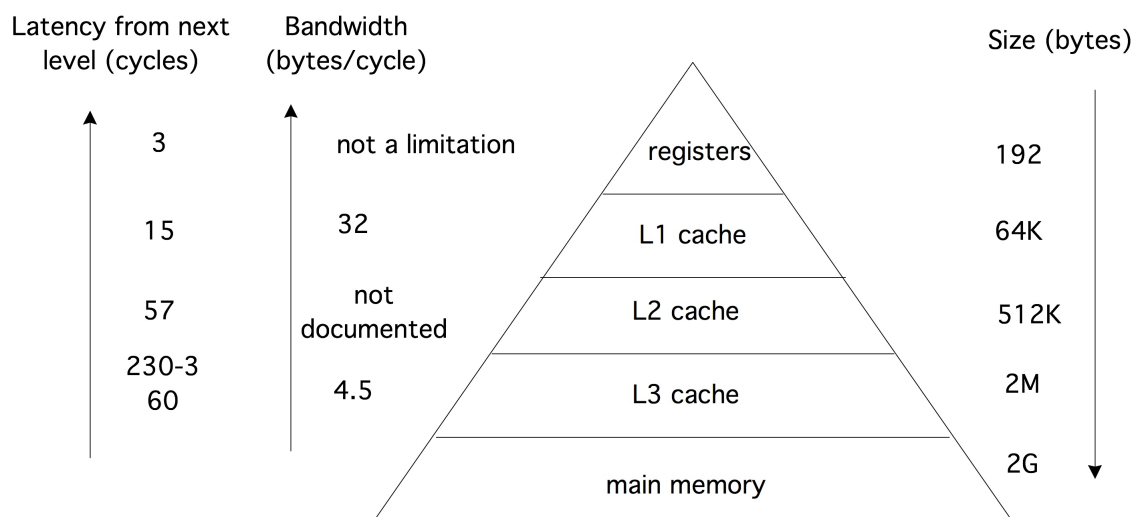


Figure 2.1: Memory hierarchy

First we need to define a cycle. It's an unit of time that is defined like this:  $cycle = \frac{1}{CPU_{freq}}$ . For example, for the CPU *AMD Ryzen 5 5600X*, the maximum frequency is  $4.6GHz$ , so the cycle is  $cycle = \frac{1}{4.6GHz} \approx 0.217ns$ . The **bandwidth** is the number of bytes that can be transferred in one cycle. And the **latency** is the number of cycle required to access a level of memory. There's also the **latency** but for a number of bytes, it's formula is  $\alpha + \beta n$  where  $\alpha$  is the level latency,  $\beta$  is the inverse of the bandwidth and  $n$  is the number of bytes.

| Level    | Level Latency<br>[cycle] | Bandwidth<br>[bytes/cycle] | Size        | What is stored                    | example                             |
|----------|--------------------------|----------------------------|-------------|-----------------------------------|-------------------------------------|
| Register | 3                        | No limit                   | $\pm 192B$  | "Immediate" data for the CPU      | Results of addition, memory address |
| Cache L1 | 15                       | 32 – 64                    | $\pm 64KB$  | Instructions and "Immediate" data | Local variable                      |
| Cache L2 | 57                       | 16 – 32                    | $\pm 512KB$ | Data used recently                | Data struct, code part              |
| Cache L3 | 230 – 360                | 4 – 10                     | $\pm 64MB$  | Data shared between core          | Global variable                     |
| RAM      | 300 – 500                | 9 – 50GB/s                 | +4GB        | Running programs                  | Running software, open document     |
| Disks    | $+10^6$                  | Usually $\leq 3GB/s$       | +128GB      | Persistent data                   | Document, OS, etc                   |

### 2.1.2 Caches lines and prefetching

A **cache line** is a small fixed-size contiguous block of memory, usually 64 or 128 bytes. It's not necessarily stored in the cache. We use them to organize the memory, because it is easier to deal with fixed size block. When the CPU need to access a memory location, it loads the entire cache line into the cache of the CPU. If the wanted data is not in the cache, there will be a **cache miss**. After that the CPU will load the entire cache line into the cache.

The **prefetching** is the fact that the CPU will load the cache line that is next to the one that is needed. It's because of the spacial locality. Spacial locality is the reason why we use cache lines. For example, if we store an array of data, it may use some space greater than one cache line so for precaution, the CPU will load the next cache line too. And so we save time, by anticipating.

For example of the importance of data locality we have:

- **Temporal locality:** If a data is used frequently, we will keep it in the cache.
- **Spacial locality:** If a data is used, the data next to it could be usefull too.

### 2.1.3 Arithmetic intensity

The **arithmetic intensity** is a concept in performance analysis for memory-bound and compute-bound programs. Let's consider a programs that do  $o$  arithmetic operations and  $m$  memory operations, we define:

- **Arithmetic intensity:**

$$a = \frac{o}{m} \quad (2.1)$$

It helps to find if the program is limited by a compute-bound or memory-bound.

- **Arithmetic time:**

$$t_{arith} = \frac{o}{\text{CPU freq}} \quad (2.2)$$

It's the time needed to performs  $o$  operations.

- **Memory transfer time:**

$$t_{mem} = \frac{m}{\text{bandwidth}} = \frac{o}{a \times \text{bandwidth}} \quad (2.3)$$

It's the time needed to do  $m$  memory operations.

The overall performance of a program is thus defined by the worst component of the PC, and so we get the **time per iteration**:

$$\min \left( \frac{t_{arith}}{o}, \frac{t_{mem}}{o} \right) \quad (2.4)$$

With some algebra, we can find the **number of operations per second**:

$$\max (\text{CPU freq}, a \times \text{bandwidth}) \quad (2.5)$$

#### 2.1.4 The roofline model

#### 2.1.5 cache hierarchy for a multi-core CPU