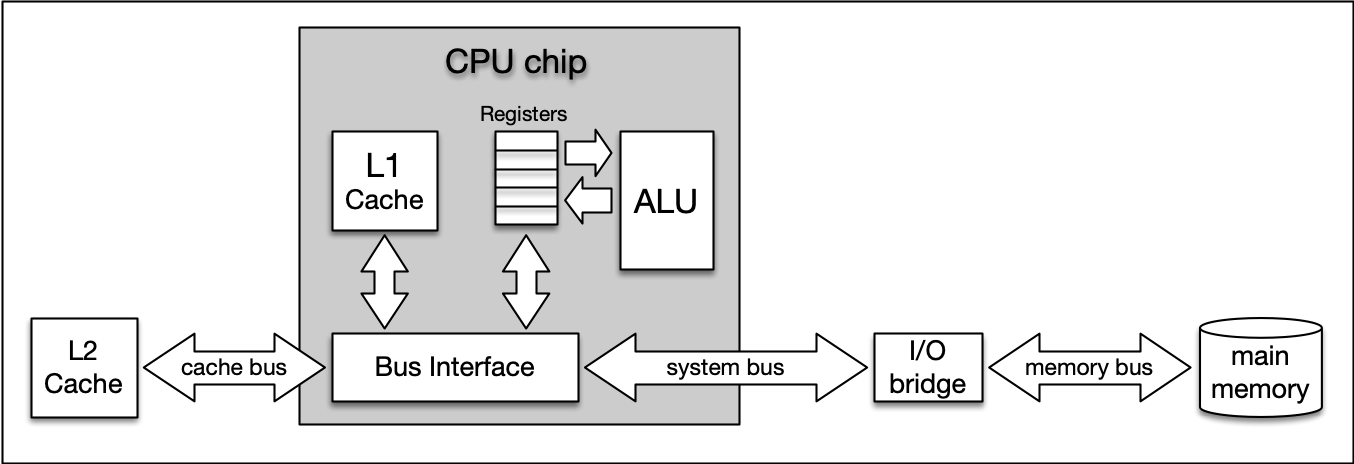
**Cache Levels**

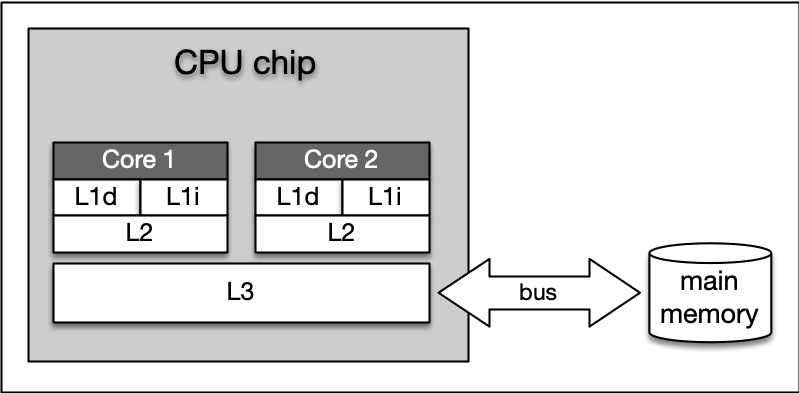
Cache memory is much faster but also significantly smaller than standard RAM. It holds the data that will (or might) be used by the CPU more often. In the memory hierarchy we have seen in the last section, the cache plays an intermediary role between fast CPU and slow RAM and hard disk. The figure below gives a rough overview of a typical system architecture:



**[System architecture diagram showing caches, ALU (arithmetic logic unit), main memory, and the buses connected each component.](https://classroom.udacity.com/nanodegrees/nd213/parts/789a1625-9b09-4615-9210-ddbc12e9247b/modules/b2145e6c-f349-4071-b1a5-682cda25eba8/lessons/ec63b3b7-590d-43ef-9492-66f6f23d9988/concepts/1239107a-5d3f-4d47-acb3-8c0ea9b0c94f)**

The central CPU chip is connected to the outside world by a number of buses. There is a cache bus, which leads to a block denoted as L2 cache, and there is a system bus as well as a memory bus that leads to the computer main memory. The latter holds the comparatively large RAM while the L2 cache as well as the L1 cache are very small with the latter also being a part of the CPU itself.

The concept of L1 and L2 (and even L3) cache is further illustrated by the following figure, which shows a multi-core CPU and its interplay with L1, L2 and L3 caches:



**Level 1 cache** is the fastest and smallest memory type in the cache hierarchy. In most systems, the L1 cache is not very large. Mostly it is in the range of 16 to 64 kBytes, where the memory areas for instructions and data are separated from each other (L1i and L1d, where "i" stands for "instruction" and "d" stands for "data". Also see "[**Harvard architecture**](https://en.wikipedia.org/wiki/Harvard_architecture)" for further reference). The importance of the L1 cache grows with increasing speed of the CPU. In the L1 cache, the most frequently required instructions and data are buffered so that as few accesses as possible to the slow main memory are required. This cache avoids delays in data transmission and helps to make optimum use of the CPU's capacity.

**Level 2 cache** is located close to the CPU and has a direct connection to it. The information exchange between L2 cache and CPU is managed by the L2 controller on the computer main board. The size of the L2 cache is usually at or below 2 megabytes. On modern multi-core processors, the L2 cache is often located within the CPU itself. The choice between a processor with more clock speed or a larger L2 cache can be answered as follows: With a higher clock speed, individual programs run faster, especially those with high computing requirements. As soon as several programs run simultaneously, a larger cache is advantageous. Usually normal desktop computers with a processor that has a large cache are better served than with a processor that has a high clock rate.

**Level 3 cache** is shared among all cores of a multicore processor. With the L3 cache, the [**cache coherence**](https://en.wikipedia.org/wiki/Cache_coherence) protocol of multicore processors can work much faster. This protocol compares the caches of all cores to maintain data consistency so that all processors have access to the same data at the same time. The L3 cache therefore has less the function of a cache, but is intended to simplify and accelerate the cache coherence protocol and the data exchange between the cores.

On Mac, information about the system cache can be obtained by executing the command sysctl -a hw in a terminal. On Debian Linux linux, this information can be found with lscpu | grep cache. On my iMac Pro (2017), this command yielded (among others) the following output:

hw.memsize: 34359738368

hw.l1icachesize: 32768

hw.l1dcachesize: 32768

hw.l2cachesize: 1048576

hw.l3cachesize: 14417920

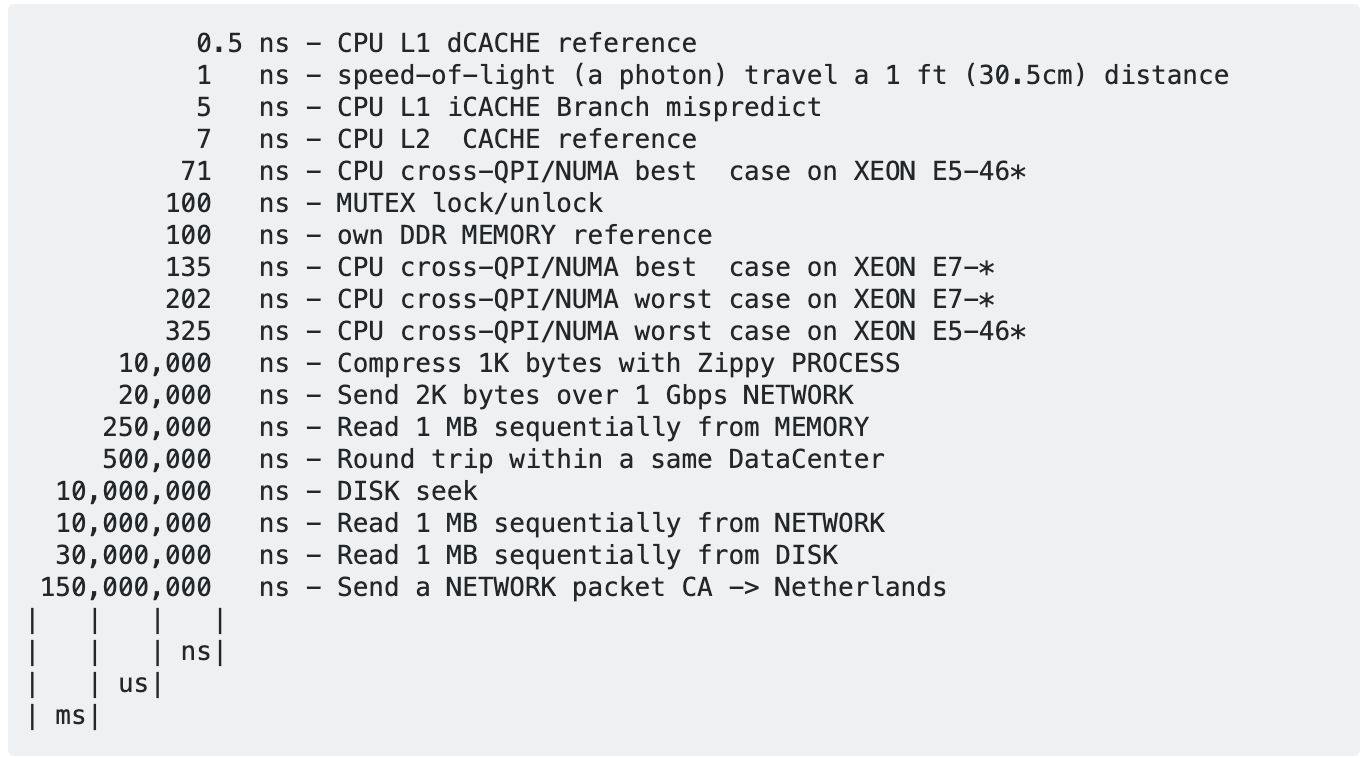
* *hw.l1icachesize* is the size of the L1 instruction cache, wich is at 32kB. This cache is strictly reserved for storing CPU instructions only.
* *hw.l1dcachesize* is also 32 KB and is dedicated for data as opposed to instructions.
* *hw.l2cachesize* and *hw.l3cachesize* show the size of the L2 and L3 cache, which are at 1MB and 14MB respectively.

It should be noted that the size of all caches combined is very small when compared to the size of the main memory (the RAM), which is at 32GB on my system.

Ideally, data needed by the CPU should be read from the various caches for more than 90% of all memory access operations. This way, the high latency of RAM and hard disk can be efficiently compensated.

### Temporal and Spatial Locality

The following table presents a rough overview of the latency of various memory access operations. Even though these numbers will differ significantly between systems, the order of magnitude between the different memory types is noteworthy. While L1 access operations are close to the speed of a photon traveling at light speed for a distance of 1 foot, the latency of L2 access is roughly one order of magnitude slower already while access to main memory is two orders of magnitude slower.



In algorithm design, programmers can exploit two principles to increase runtime performance:

**Temporal locality** means that address ranges that are accessed are likely to be used again in the near future. In the course of time, the same memory address is accessed relatively frequently (e.g. in a loop). This property can be used at all levels of the memory hierarchy to keep memory areas accessible as quickly as possible.

**Spatial locality** means that after an access to an address range, the next access to an address in the immediate vicinity is highly probable (e.g. in arrays). In the course of time, memory addresses that are very close to each other are accessed again multiple times. This can be exploited by moving the adjacent address areas upwards into the next hierarchy level during a memory access.

Let us consider the following code example:

**#include <chrono>**

**#include <iostream>**

**int** **main**()

{

*// create array*

**const** **int** size = 4;

**static** **int** x[size][size];

**auto** t1 = std::chrono::high\_resolution\_clock::now();

**for** (**int** i = 0; i < size; i++)

{

**for** (**int** j = 0; j < size; j++)

{

x[j][i] = i + j;

std::cout << &x[j][i] << ": i=" << i << ", j=" << j << std::endl;

}

}

*// print execution time to console*

**auto** t2 = std::chrono::high\_resolution\_clock::now(); *// stop time measurement*

**auto** duration = std::chrono::duration\_cast<std::chrono::microseconds>(t2 - t1).count();

std::cout << "Execution time: " << duration << " microseconds" << std::endl;

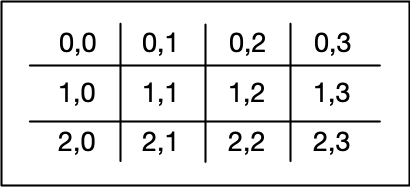
**return** 0;

}

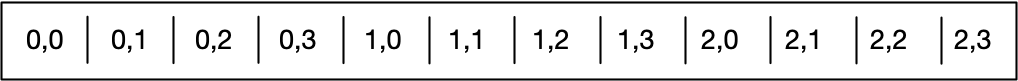
# Cache-friendly coding

In the code sample to the right, run the code and note the results. Then please modify the code slightly by interchanging the index i and j when accessing the variable x and take a close look at the resulting runtime performance compared to the original version.

Depending on the machine used for executing the two code versions, there will be a huge difference in execution time. In order to understand why this happens, let us revisit the memory layout we investigated with the gdb debugger at the beginning of this lesson: Even though we have created a two-dimensional array, it is stored in a one-dimensional succession of memory cells. In our minds, the array will (probably) look like this:



In memory however, it is stored as a single line as follows:



As can be seen, the rows of the two-dimensional matrix are copied one after the other. This format is called "row major" and is the default for both C and C++. Some other languages such as Fortran are "column major" and a memory-aware programmer should always know the memory layout of the language he or she is using.

Note that even though the row major memory layout is used in C++, this doesn't mean that all C++ libraries have the same default; for example, the popular Eigen library used for matrix operations in C++ [defaults to column major](https://eigen.tuxfamily.org/dox/group__TopicStorageOrders.html).

As we have created an array of integers, the difference between two adjacent memory cells will be sizeof(int), which is 4 bytes. Let us verify this by changing the size of the array to 4x4 and by plotting both the address and the index numbers to the console. Be sure to revert the array access back to x[i][j] = i + j. You can plot by uncommenting the printout line in the inner for loop:

0x6021e0: i=0, j=0 0x6021e4: i=0, j=1 0x6021e8: i=0, j=2 0x6021ec: i=0, j=3 0x6021f0: i=1, j=0 0x6021f4: i=1, j=1 0x6021f8: i=1, j=2 0x6021fc: i=1, j=3 0x602200: i=2, j=0 0x602204: i=2, j=1 0x602208: i=2, j=2 0x60220c: i=2, j=3 0x602210: i=3, j=0 0x602214: i=3, j=1 0x602218: i=3, j=2 0x60221c: i=3, j=3 Execution time: 83 microseconds

Clearly, the difference between two inner loop cycles is at 4 as predicted.

When we interchange the indices i and j when accessing the array as

When we interchange the indices i and j when accessing the array as

x[j][i] **=** i **+** j;

std::cout **<<** **&**x[j][i] **<<** ": i=" **<<** j **<<** ", j=" **<<** i **<<** std::endl;

we get the following output:

0x6021e0: i=0, j=0

0x6021f0: i=1, j=0

0x602200: i=2, j=0

0x602210: i=3, j=0

0x6021e4: i=0, j=1

0x6021f4: i=1, j=1

0x602204: i=2, j=1

0x602214: i=3, j=1

0x6021e8: i=0, j=2

0x6021f8: i=1, j=2

0x602208: i=2, j=2

0x602218: i=3, j=2

0x6021ec: i=0, j=3

0x6021fc: i=1, j=3

0x60220c: i=2, j=3

0x60221c: i=3, j=3

Execution time: 115 microseconds

As can be see, the difference between two rows is now 0x10, which is 16 in the decimal system. This means that with each access to the matrix, four memory cells are skipped and the principle of spatial locality is violated. As a result, the wrong data is loaded into the L1 cache, leading to cache misses and costly reload operations - hence the significantly increased execution time between the two code samples. The difference in execution time of both code samples shows that cache-aware programming can increase code performance significantly.

#include <chrono>

#include <iostream>

int main()

{

// create array

const int size = 4000;

static int x[size][size];

auto t1 = std::chrono::high\_resolution\_clock::now();

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

x[i][j] = i + j;

//std::cout << &x[i][j] << ": i=" << i << ", j=" << j << std::endl;

}

}

// print execution time to console

auto t2 = std::chrono::high\_resolution\_clock::now(); // stop time measurement

auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(t2 - t1).count();

std::cout << "Execution time: " << duration << " microseconds" << std::endl;

return 0;

}