Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **2022**

Student: Fabio Marchesi Discussed with: Raffaele Morganti, Gioele De Pianto, Nicholas Carlotti

Solution for Project 1

Due date: 12.10.2022 (midnight)

1. Explaining Memory Hierarchies *(25 Points)*

Memory hierarchies are a core factor in execution performance, the main idea is that the smallest is the memory and the quickest you can access it (smallest memory are also the nearest to the CPU). The CPU is able to work only on values that are stored in its registers, which are always in a small number. So in all modern computer there is this hierarchy of memory levels in a way that each level (starting from the processor and going towards the hard disk) is always bigger and slower.

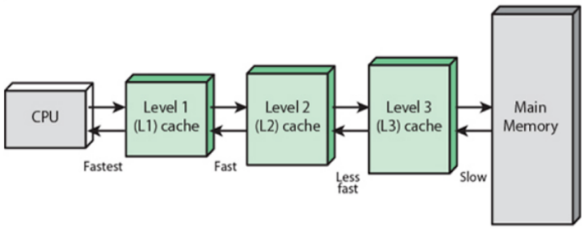
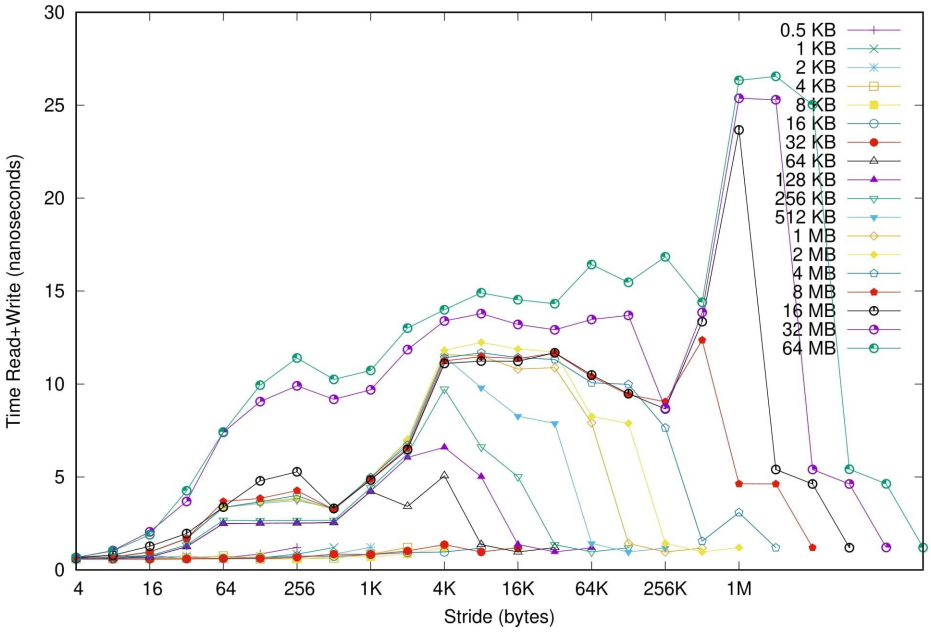


Figure 1: Visualization of cache levels, source: [includehelp(November 06, 2019)]

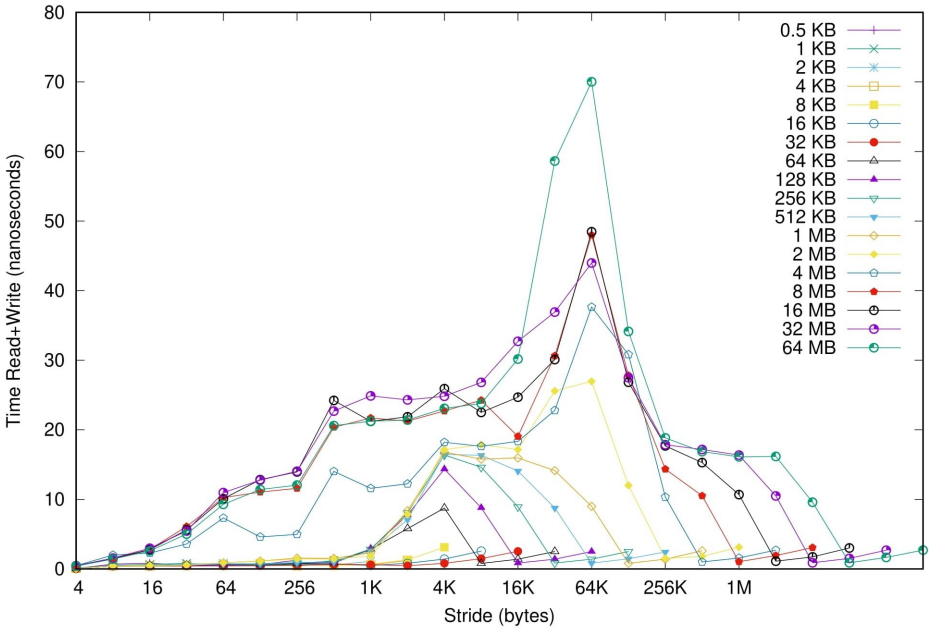
When the CPU needs to work on some data, it checks if it is in its registers, if this is not the case than that specific piece of data is searched among the other levels of the hierarchy (from the fastest to the slowest) and it's fetched to the cache. So, it's easy to get why the closest the data is to the CPU and the fastest the loading time is. When a level of the cache is full and new information need to be stored, usually the cache removes the values that haven't been accessed for the longest time. Knowing this policy, we can see that having a high temporal locality will reduce loading times, increasing the overall performance.
With the new hardware components, the difference between the time used by the CPU to perform instructions and the loading time from the outer memories is becoming huge, so, a use of the memory that exploits caching will lead to much faster performances.
From a memory analysis of a node of the cluster, these are the dimensions of each level of the hierarchy.

Memory	Size
Level 1 cache	32kb
Level 2 cache	256kb
Level 3 cache	25MB
Main memory	64GB

We also run a benchmark to see the memory performances of different memory access patterns. The test here is really simple: there are two parameters: $Size$ (the size of the array) and s (the stride). For each configuration of the parameters, an array $A[]$ of size $Size$ is generated, and its values are loaded with a stride s (we load values in positions: 0, s, $2s$, ...). For each pair of parameters, the test is run multiple times, and we compute the average time to load each value. The time per load is then plotted for various values of $Size$ and s. The graph below is the result of the test run on the cluster.



Let's look at the results that we've got in order to get some information about the importance of each level of cache. The first thing we notice is that for each value of $Size$ (for each value we have a line on the graph) that is less or equal than 32kb the average time per load is very low (indeed all such lines are always near to the X-axis), the main reason behind such result is that the whole array fits on the level 1 cache, therefore each access is a hit in the level-1 cache. Instead, when $Size$ becomes very high, the time per load becomes very hard to predict. The reason behind that is that the memory hierarchy is quite complex, and many factors play a role on the average time per load. On the other hand, what is easy to see, is that we obtained the worst performances when the $Size$ is very high, that's because we have a terrible temporal locality.
Let's look at the graph obtained on my personal machine:



We can observe that, in general, the time per load are much higher (the Y-axis shows higher values) but the shapes of the different lines remain similar to the ones on the cluster. The difference in terms of performances is mainly due to the fact that the node on the cluster has a more powerful hardware than my local machine. Let's dive into those hardware differences using the packet likwid. Using the command `likwid-topology` we get this result for the cluster:

```
CPU name:       Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
CPU type:       Intel Xeon Haswell EN/EP/EX processor
CPU stepping:   2
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                2
Cores per socket:       10
Threads per core:       1
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
4               0               4               0               *
5               0               5               0               *
6               0               6               0               *
7               0               7               0               *
8               0               8               0               *
9               0               9               0               *
10              0               10              1               *
11              0               11              1               *
12              0               12              1               *
13              0               13              1               *
14              0               14              1               *
15              0               15              1               *
16              0               16              1               *
17              0               17              1               *
18              0               18              1               *
19              0               19              1               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 1 2 3 4 5 6 7 8 9 )
Socket 1:               ( 10 11 12 13 14 15 16 17 18 19 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ( 6 ) ( 7 ) ( 8 ) ( 9 ) ( 10 ) ( 11 ) ( 12 ) ( 13 ) ( 14 ) ( 15 ) ( 16 ) ( 17 ) ( 18 ) ( 19 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   256 kB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ( 6 ) ( 7 ) ( 8 ) ( 9 ) ( 10 ) ( 11 ) ( 12 ) ( 13 ) ( 14 ) ( 15 ) ( 16 ) ( 17 ) ( 18 ) ( 19 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   25 MB
Cache groups:           ( 0 1 2 3 4 5 6 7 8 9 ) ( 10 11 12 13 14 15 16 17 18 19 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           2
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 4 5 6 7 8 9 )
Distances:              10 21
Free memory:            1701.62 MB
Total memory:           31792.3 MB
--------------------------------------------------------------------------------
Domain:                 1
Processors:             ( 10 11 12 13 14 15 16 17 18 19 )
Distances:              21 10
Free memory:            28386.1 MB
```

And for my personal computer:

```
--------------------------------------------------------------------------------
CPU name:       AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
CPU type:       nil
CPU stepping:   0
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread          Core        Die         Socket          Available
0               0               0           0           0               *
1               1               0           0           0               *
2               0               1           0           0               *
3               1               1           0           0               *
4               0               2           0           0               *
5               1               2           0           0               *
6               0               3           0           0               *
7               1               3           0           0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 1 2 3 4 5 6 7 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   512 kB
Cache groups:           ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   4 MB
Cache groups:           ( 0 1 2 3 4 5 6 7 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           1
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 4 5 6 7 )
Distances:              10
Free memory:            5106.06 MB
Total memory:           5456.49 MB
--------------------------------------------------------------------------------
```

The first thing we notice is that even though my Level 2 cache is double the size of the Level 2 cache on the cluster, my level 3 cache is much smaller than the one on the cluster. Another factor is that the clock frequency of my CPU is 2.0Ghz versus the 2.3Ghz of the cluster's CPU.

Before ending this section relative to memory hierarchies, let's look at 2 combination of the parameters $Size$ and s on the cluster (as required from the assignment):

- $Size = 128, s = 1$: (In the graph $Size = 128$ isn't included, but the behavior is easy to predict) we can see that, when the stride is this little, no matter what is the size of the array, the time per load is always really low. In the specific case, the explanation is simple. Values

aren't loaded one by one into the cache, instead they are loaded in blocks called cache lines. Cache lines are simply consecutive bytes transferred to the cache. So, when the stride is equal to one we are loading only consecutive values on the memory, so when we have a cache miss a new cache line is fetched on the cache, this line will also contain the next values to load, so after each miss we have a series of hit (the length of the series depends on the size of the lines).

- $Size = 2^{20}, s = 2^{19}$: Also in this case, the average time per load is really low, but for a different reason. In this case, being the stride so big (half of the size of the array), we will end up reading only the same few values out of the all 2^{20}. Because of the reason, all the values can be kept on the cache. In this case, we say that there is an excellent spacial locality.

2. Optimize Square Matrix-Matrix Multiplication *(60 Points)*

Matrix multiplication is a really common procedure in a large variety of fields. Sadly, given two matrices A and B is really difficult to compute $C = A\dot{B}$ in a fast way. In particular, assuming that A and B are square matrices of size n, the classic (and naive) method to compute C requires $O(N^3)$ time. And the code looks like this.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

We want to improve this algorithm, optimizing the use of the cache. Like we said before, to exploit the cache, we need to have a high temporal and spatial locality. With this basic approach, if the size n isn't too small, the whole matrices (each having n^2 entries) don't fit into the level 1 cache. And if we look at which values we need to load during the different iterations, we see that the same values are accessed at very different times, when they are already been erased from the cache. To improve the temporal locality, we decide to logically divide A, B and C in blocks of size $s \times s$. This algorithm is called blocked matrix multiplication and in this way we can, for each block (i, j) of size $s \times s$ in C, and for each k from 0 to $n/s - 1$, update the block $C[i][j]$ using the blocks $A[i][k]$ and $B[k][j]$. In this way, the block $C[i][j]$ remains in the cache during multiple iterations and also $A[i][k]$ and $B[k][j]$ fits in the cache (choosing a feasible value of s).
The main idea can be explained with this simple pseudocode,

```
for i = 0 to n / s - 1
    for j = 0 to n / s - 1
        C[i][j] is loaded into the cache
        for k = 0 to n / s - 1
            A[i][k] is loaded into the cache
            B[k][j] is loaded into the cache
            Compute C[i][j] += A[i][k] * B[k][j]
```

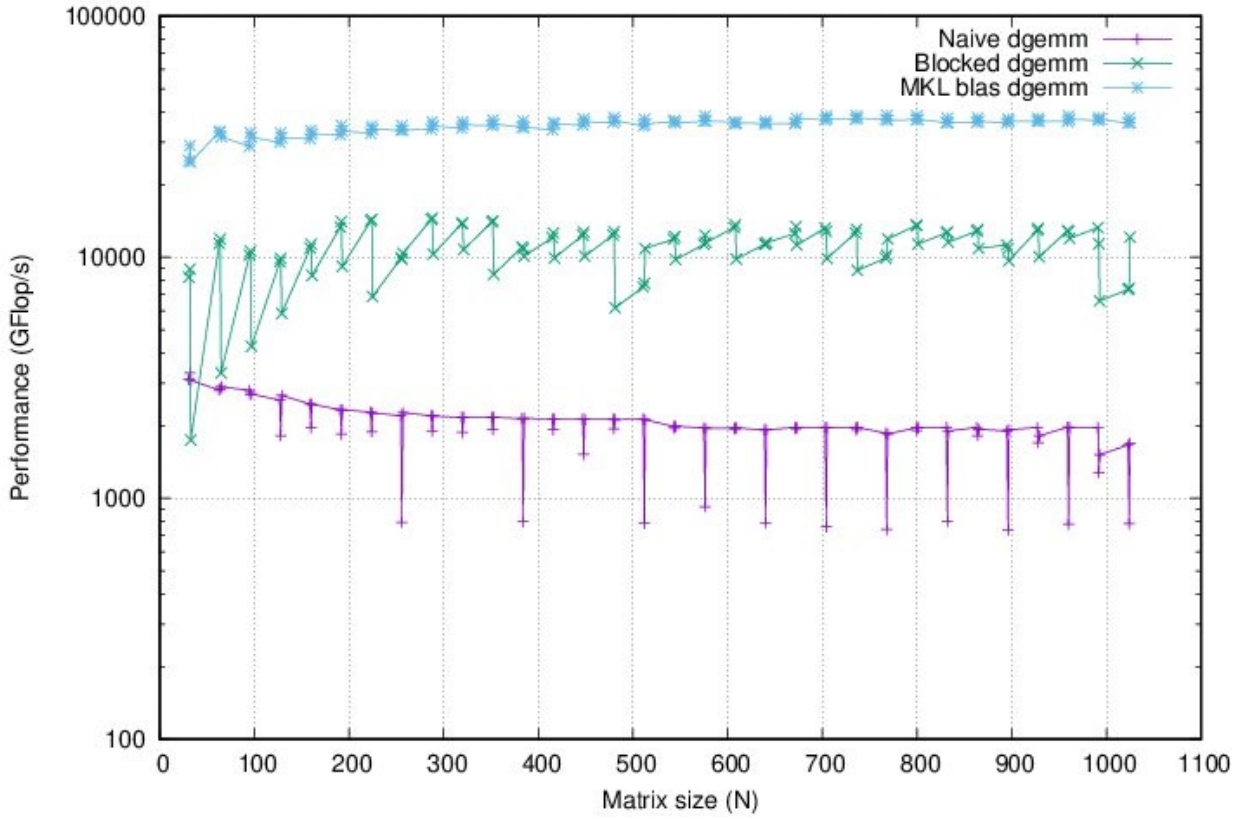
The product of the blocks $A[i][k]$ and $B[k][j]$ can be done with the naive method because all the required values are loaded into the cache.
The code for implementing the blocked matrix multiplication looks like this (we assume that n is a multiple of s, otherwise the matrices are padded before the division in blocks), in the code A is stored in the row-major format, while B is stored in the column-major format, the reason for this is explained after.

```
for (int i_block = 0; i_block < n; i_block += s) {
  for (int j_block = 0; j_block < n; j_block += s) {
    for (int k_block = 0; k_block < n; k_block += s) {
      for (int i = i_block; i < i_block + s; i++) {
        for (int j = j_block; j < j_block + s; j++) {
          for (int k = k_block; k < k_block + s; k++) {
            padded_C[n*j+i] += padded_A[i*n+k] * padded_B[j*n+k];
          }
        }
      }
    }
  }
}
```

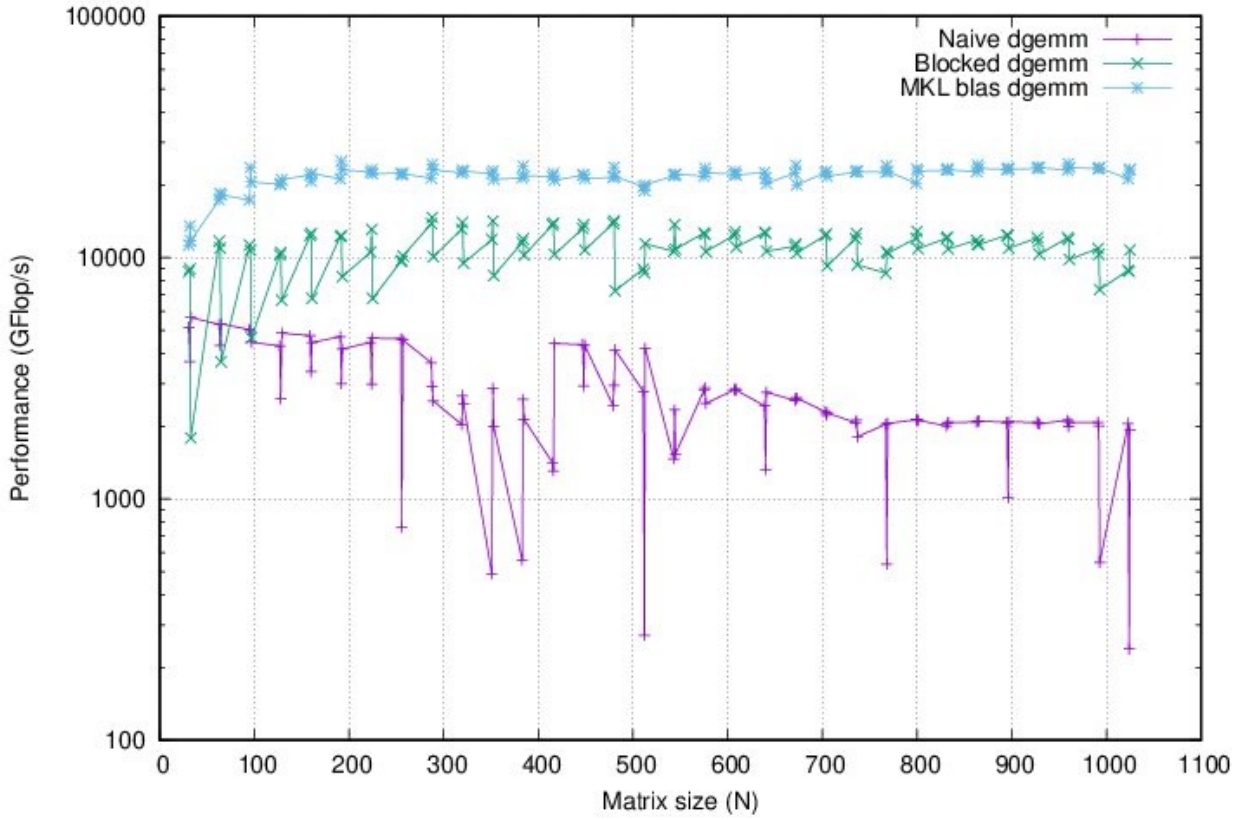
A really important factor of this algorithm is the value of s, the length of the size of the block. After various tests, we established that $s = 32$ gives the best results in terms of performance ($s = 16$ was a close runner-up). Increasing the size of the block would lead to a worse performance, that is because if we look at the blocked version of the algorithm we see that we would like to fetch on the level 1 cache three $s \times s$ blocks ($C[i][j]$, $A[i][k]$, $B[k][j]$). These blocks use $3 \cdot s \cdot s \cdot 8$ bytes (three $s \times s$ blocks of doubles, each double uses 8 bytes). Now, with $s = 16$ we obtain $6kb$, with $s = 32$ we obtain $24kb$ and the level 1 cache size is $32kb$. But we also need to remember that values are fetched onto the cache in lines so, in order to store the three matrices, we would need a little more memory than the values computed before. Indeed, when we tried to increase s with values bigger than 32 the performances dropped.
To improve the implementation, we stored A in a row-major order (indeed we access its values from left to right on the same row), while B is stored in a column-major order, in this way the lines loaded into the cache will contain also values that are about to get used, improving the spatial locality.
In order to evaluate the performance of the blocked version of matrix multiplication, we run a benchmark where we compared the performances of three algorithms.

- The naive version.
- The blocked version.
- The subroutine for matrix multiplication included in the BLAS (Basic Linear Algebra Subprograms). This version is highly optimized.

The performances are analyzed on matrices of various sizes, and for each of such matrices, we computed the $GFlop/s$, the number of floating point operations per second. We used this metric because it allows us to understand how much time we waste loading values into the CPU's registers. These are the results obtained on the cluster:



While these are the results obtained on my local machine.



It's really easy to see how the blocked-dgemm algorithm performs better than the naive approach

while the BLAS version outperforms the other 2 solutions, especially on the cluster node. Taking into consideration that the plots use a logarithmic scale for the Y-axis, we can say that there is a notable difference in terms of performance between the naive and the blocked version, for such a simple (but smart) optimization.
From this graph, we can also say that if there is the need to perform matrix multiplication, then the most efficient way to do it's to use the highly optimized blas version. We also analyzed the percentage of theoretical peak, that is, how much are we optimizing the loading phase (remember that loading data into the CPU's registers is much slower than the time used by the CPU to perform basic operations, especially if the data is located onto the outer memories).
Computing the average of the percentage of theoretical peak (APTP) for each approach on matrices of different sizes, we obtained the following results:

Approach	APTP on the cluster	APTP on local machine
Naive	6%	5%
BLAS	95%	96%
Blocked	29.8%	29%

This table confirms how the BLAS approach is much more efficient than the other 2 algorithms. The blocked version of the matrix multiplication shows a much better temporal and spatial locality than the naive implementation.
To get to these results, I tried different strategies and did some testing, in the end the following decisions were taken:

- The optimization flags have been changed. Using the flag `O3` instead of `O2` we get slightly better results in terms of performance. Using `Ofast` the performances improve even more, but we need to be careful because `Ofast` enables `-ffast-math` which speeds up operations (especially on doubles) but could lead to unpleasant behaviors, for example the same code could lead to different results depending on the environment. But in our case the result of the multiplication was correct also with `Ofast`. The flag `march='native'` was also added, also this flag is a risky one because it makes the binary file (possibly) not compatible on a various range of architectures different from than one that produced it. The `march='native'` tells the compiler to tune generated code for the micro-architecture and ISA extensions of the host CPU.
- The matrices are padded in order to make their size a multiple of the block size.
- For the reasons explained before, the block size is 32×32.
- A is transposed in the row major format, improving spatial locality.
- After some experiments, I noted that using aligned_alloc and aligning the allocation of the padded matrices to the size of the cache line instead of using `malloc` we get a much better performance. After some thinking, I concluded that this improvement in performance is due to the fact that aligning the allocation to the cache line size, we decrease the probability of loading any unwanted value when we load a new line on the cache.

During the experiments, I also tested the following ideas that didn't improve the performances:

- Avoid to transpose A, keeping it column-major and swapping the order of the loops for the multiplication in order to improve the spatial locality.
- Using the register keyword for the variables i, j, k, in this way this values which are used many times remain loaded in the CPU's registers.
- Avoid to pad the matrices and using some if statements to manage the last row and column of blocks. If we don't pad the matrices, these blocks could be of a size smaller than s.

References

[includehelp(November 06, 2019)] includehelp. Cache memory and its different levels. *includehelp*, November 06, 2019.