Remark: All the result in this assignment are obtained by a remote connection to the computer of the lab, and the gcc compiler, version 8.2.0, always set up with the -O3 optimization option. Moreover n and m are both fixed to 4096, while the max number of iterations is set up to 100.

The table below shows the performance results of the Baseline code compared with the ones of the 6 optimizations we performed. The optimization steps are consecutive, in the sense that a certain code optimization incorporates also the optimizations that have been done before it.
The number of *stencil operations* is always of the order of $n \times m \times$ *iter-max* $\sim 1,677$G operations.
As a general remark we can observe that in the various steps there is a constant decreasing in the cache-misses value. This implies that the main memory of the computer is accessed less, and leads to a faster execution.

	BASELINE	LOOP FUSION	LOOP INTERCHANGE	STRENGTH REDUCTION	DOUBLE BUFFER	CODE MOTION	BIG MATRIX
cycles	456,630,187,309	352,055,974,458	31,761,921,859	31,955,399,767	25,975,165,024	22,574,044,343	21,286,091,610
instructions	61,131,258,235	55,774,249,212	44,886,458,554	44,827,367,532	49,815,275,781	40,276,887,562	31,897,757,996
cache-misses	10,784,618,622	8,288,176,850	46,702,290	45,149,626	15,171,065	11,457,130	10,458,403
task-clock [msec]	135,997.80	104,800.99	9,491.75	9,503.76	7,733.82	6,749.42	6,350.98
clock frequency [GHz]	3.358	3.359	3.346	3.362	3.359	3.345	3.352
insn per cycle	0.13	0.16	1.41	1.4	1.92	1.78	1.5
CPUs utilized	1.000	1.000	1.000	1.000	1.000	1.000	1.000
seconds user	135.78948300	104.7205230	9.405289	9.446708	7.579452	6.716931	6.334514
work per second [Mstenc/s]	12.3	15.99	178.3	177.5	221.26	249.67	264.74
codif efficiency [stenc/Kinstr]	27.44	30	37.36	37.4	33.76	41.64	52.57

1 Loop fusion

In this first attempt to optimize the code the instructions for computing the new cell values and the ones for computing the error are put together into the same function. The goal is to save the number of instructions and then make the program to execute faster, and indeed the performance results confirm that this reduction has been achieved. To be more precise we can observe that the work done per second is of 16 M *stencil/second*, and so there is a **1,3x** improvement in the performance.
As seen in the table this can be explained as a consequence of the reduction of the operation rate (operations per instruction) due to a slighty better codification efficiency, and a little improvement of the IPC, since the clock-frequency is almost the same for both the implementations.

2 Loop interchange

Here the order in which the data in the input and output matrices are accessed is modified. Instead of accessing the data in the matrix *by columns*, they are accessed *by rows*, as they are stored in the computer memory.
As we can see in the performance results this change leads to a very consistent improvement in the performance: this is infact **14.49x** better then the Baseline code, and **11,15x** better then the Loop fusion code.
This can be explained by a consistent improvement in the IPC and by a moderate improvement in the codification efficiency.
Our conjecture is that this huge improvement in the IPC is due to the fact that the processor has to access close memory cells to execute the instructions and does not have to jump in the memory to find the needed data. This should lead to less time consumption in the instruction execution, and then more instructions can be executed in one clock-cycle.

3 Strength reduction

Since multiply operations are faster than division operations, instead of dividing by 4 when computing the stencil operation, we can multiply by 0.25.

The performance results are almost the same as for the Loop interchange step, so we can say that this kind of optimization does not affect much the efficiency of this particular code.

4 Double buffer

Here the stencil computation is performed alternatively in both directions: one iteration of the convergence loop will be done from A to Anew, and the next iteration will be done from Anew to A.
This, avoiding the operations needed to copy the matrix Anew into A at each iteration, should lead to a decreasing in the execution time. Indeed we obtain a **1,25x** better performance than in the previous case and a **17,98x** better one than in the Baseline case.
This can be explained mainly by the improvement in the IPC (almost two instructions per cycle) and by the better exploited access to the fast cache memory of the processor (very strong decreasing in the cache-misses value).

5 Code motion

Since the square root function is monotonically increasing, the square root computation can be moved outside the inner loop of the code computing the error, while maintaining the exact functionality of the program. In terms of speedup we obtain **1,13x** better than Double buffer and **20,29x** better than the Baseline version.
While the IPC is less than the previous version, the code efficiency is greater, and also the total number of instructions decrease by 20%.

6 Big matrix

The idea of this optimization step is to store a big matrix in which the data of A and Anew can be contained simultaneusly. This, accordingly also to our conjecture in the Loop interchange analysis, should improve the execution time since the data are stored in close memory positions, and so this should reduce the amount of time needed by the processor to find data.
Actually we obtain a faster execution, and we observe also that the IPC is almost the same as in the Loop interchange case.
Despite the IPC is less than in the previous case, we have a very strong reduction in the number of instructions, which can fairly explain the faster execution time. As a difference between the previous cases we need in this case to allocate a contiguos memory block of size $2 \times n \times m$.

7 Ideas and errors

The optimizations we have thought about on our own before reading the hints, are the ones concerning the Loop fusion and the Double buffer.
After the reading of the Loop interchange we thought to the Big matrix optimization.
We also thought to another possible optimization: allocate a matrix A with malloc function, and then expand it dinamically with realloc to store the values of Anew at every iteration. In our mind this should have allow to the machine to retain less memory cells every time, and so to reduce the amount of long-time memory operations. But after the analysis of the Double buffer case, we realized that this would not have lead to a faster execution, since we couldn't find a way to avoid the copy of the values from the expanded part of the matrix to the initial one.
An error we made has been to not initialize the border of Anew in the Bigmatrix optimization step. It was difficult to debug because when we printed to debug we showed only the first component of each couple of values in the matrix, instead of showing also the second one in which the Anew matrix was stored.