1. Parallelization, Optimization

Before we tried to optimize the code, we parallelized it. We couldn't parallelize the outer loop, because each iteration depends on the previous one. However, we could parallelize the middle loop using a `#pragma omp parallel for`. First we parallelized the rows, dividing them among the threads, each thread computing all the columns of their assigned rows. The results were pretty satisfying, as we can see in table 1. However we can see that for 1 thread, we lose 5 seconds with the parallelization, which can be explained by the cost of the parallelization. The whole parallelization setup takes time, and it loses all benefits when no parallelization is done, namely when only 1 thread is used. So the loss for 1 thread makes sense.
We then tried to parallelize the columns instead of the rows by switching the `#pragma omp parallel for` to the most inner loop. As we can see in table 1, the results were better for 1 and 2 threads, but started to be worse for 4, 8 and 16 threads. This might come from the fact that there is more loop overhead, since we parallelize inside the middle loop, the `#pragma omp parallel for` is executed more times.

For optimization, we first thought about memory access optimization. We first tried to reason about true and false sharing. Here we have cases of false sharing. We don't modify the input square, we only access it to read the values, and we only modify each cell of the output once, so there won't be any case of true sharing. However since we modify close parts of the output, two cores might access the same cache line, inducing false sharing. One solution to reduce false sharing is to move data into different cache blocks using padding. In that case, it is not feasible because we are working on a huge structure, we can't possibly imagine using a cache line per cell.

We then thought about loop inversion. We saw in the course that sometimes, changing the order of the loops can lead to optimization, so we tried to swap the i and j indexes. Results can be found in table 1. We observe that the running times were 2 to 3x worse ! That can be explained by the locality of the caches. Since both `input` and `output` tables are one dimension, they are stocked in a continuous way in the memory. The first line is stocked, then the second, then the third etc. When parallelizing on columns, a thread will work on (1, j), (2, j), …, (L-1, j). Each time we access a new (i, j), if the value is not in the cache then it will be fetched from the memory to the cache, alongside other values that are close to (i, j), namely (i, j+1), (i, j+2) etc. As we can see, the data newly brought in the cache is not useful, the thread will not use them before the current column is complete. Since the caches of the SCITAS cluster are rather small (cf the caches description below), the (i, j+1), (i, j+2) etc that were brought in the cache will likely get replaced by the time they will be needed, which is why the number of cache misses are higher when parallelizing on columns, and thus it takes more time.

We then tried to unroll the inner-for loops. The goal of unrolling is to reduce time spent executing loop's instructions. By computing x cells in each iteration it reduces by x the number of iterations of the for loop. We tried unrolling both j and i for loops but without surprise unrolling the j one is the most effective, again by the principle of locality explained before.

Using cachegrind, we found that the L1 data cache of each SCITAS' core can store up to 32 KiB. We also found that it is 8-way associative and has blocks of 64 bytes. We also remind that a `double` needs 8 Bytes in C.

As seen in the course, accessing data using the locality principle could speed up the execution time since there would be less cache miss. We tried to implement locality by dividing the array into sub-arrays. We did a computation to choose the best sub array size. The main goal is to be able to fit both sub-arrays of `output` and `input` in the cache to avoid evicting data that would be used in future computation. `double` are encoded with 8 bytes so L1 cache stores 32 KiB/ 8B = 4096 `double` with 8 elements per block. For every computation the algorithm accesses 9x `input` and 1x `output,` but since we are working on square sub-matrices, we roughly load the same amount of `input` and `output` data, giving us 4096 / 2 = 2048 `double` per array. Square(2048) = ~45 `double` that we reduced to 40 `double` block size to have a multiple of cache block size.

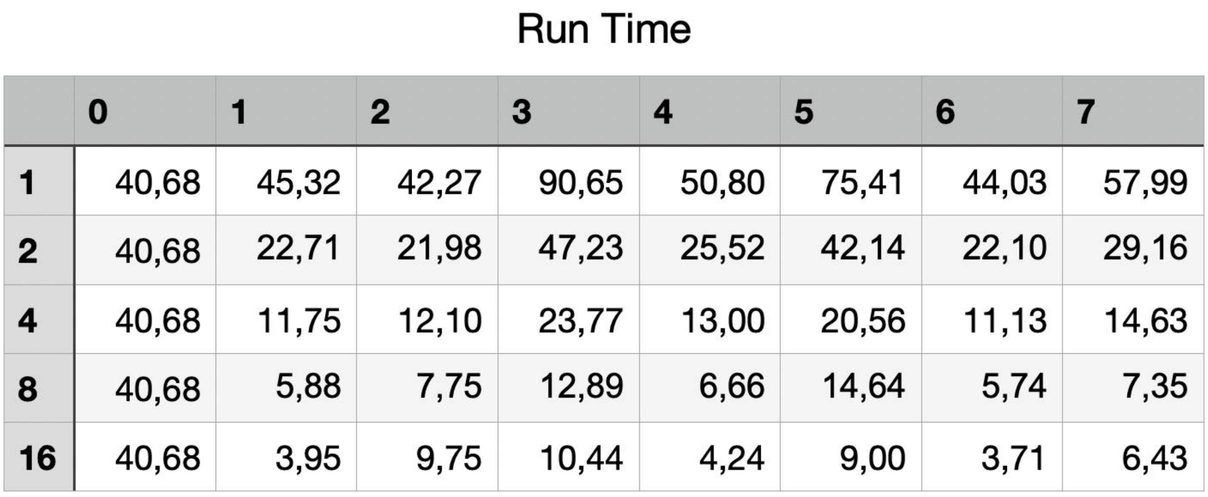
As you can see in figure 1, execution times with this implementation are quite disappointing. Our explanation for those results is the following: to go through the array block by block we added two more nested for-loops, making the program spend more time executing loop's instructions.

We then wanted to try another way of implementing the sub-arrays. Instead of having a fixed size for the arrays, we wanted to try a dynamic version in terms of the number of threads. We implemented it and got a better time than with the fixed size, but a worse time than the basic parallelized version. The fact that it took more time than the basic parallelized version surprised us at first, but thinking about it we realised that to implement the sub-array version, we introduced more loop iteration, more overhead and basically more work that took over the benefits of the locality.

In any of our versions changing scheduling would not have been interesting since all the workloads are well balanced. Default (static) scheduling is the one with least overhead and allocates contiguous for-loop's indexes thus contiguous chunk in memory as well, so we chose not to change the scheduling and to leave the default one.

2. Optimization measures

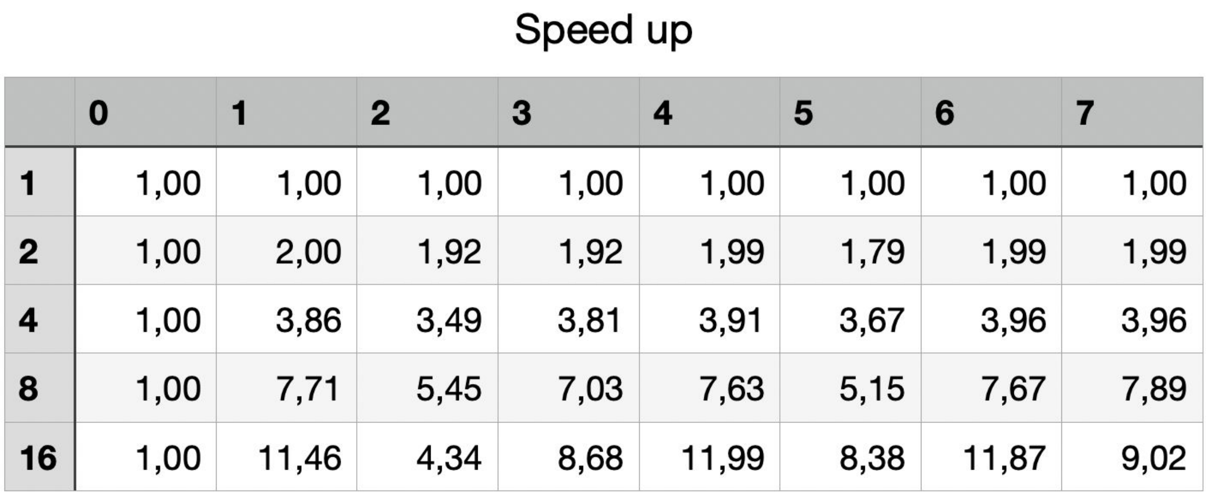
Versions:
0 - Given algorithm
1 - Simple parallelization on i
2 - Simple parallelization on j
3 - I, j for loops inverted, parallelization on j
4 - 40x40 block matrix
5 - Length/thread x Length/thread block matrix
6 - Unrolling over j
7 - Unrolling over i



Run Time

	0	1	2	3	4	5	6	7
1	40,68	45,32	42,27	90,65	50,80	75,41	44,03	57,99
2	40,68	22,71	21,98	47,23	25,52	42,14	22,10	29,16
4	40,68	11,75	12,10	23,77	13,00	20,56	11,13	14,63
8	40,68	5,88	7,75	12,89	6,66	14,64	5,74	7,35
16	40,68	3,95	9,75	10,44	4,24	9,00	3,71	6,43

Table 1



Speed up

	0	1	2	3	4	5	6	7
1	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
2	1,00	2,00	1,92	1,92	1,99	1,79	1,99	1,99
4	1,00	3,86	3,49	3,81	3,91	3,67	3,96	3,96
8	1,00	7,71	5,45	7,03	7,63	5,15	7,67	7,89
16	1,00	11,46	4,34	8,68	11,99	8,38	11,87	9,02

Table 2

3. Graph, Conclusion

Overall, we are quite disappointed by our results, especially after reaching incredible time by changing and optimising the algorithm (before learning it was not allowed). The horizontal unrolling optimization is the only one that improves the execution time thus it's also our best version.

We realise now that once you have parallelized your program, there is not much you can do to optimize it without changing the algorithm. Often when trying to change the way you access the data in order to lower cache misses, you introduce complexity and overhead that overshadows the progress you make on the data locality.

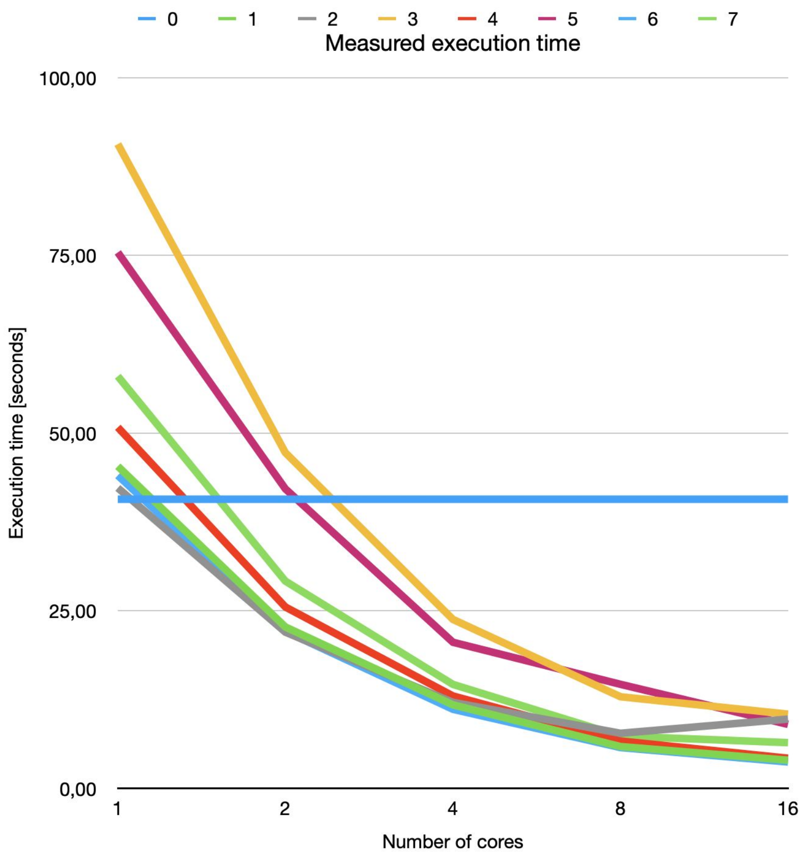


Figure 1

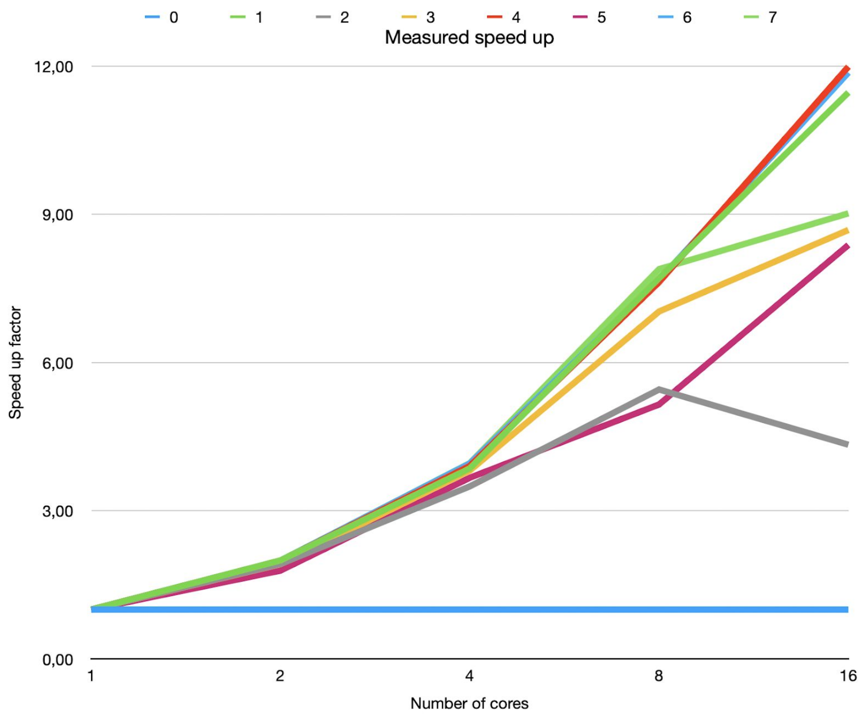


Figure 2