Remark: The results in this paper are obtained by a remote connection to the computer of the lab, and the gcc compiler, version 8.2.0, set up with the -O3 or -Ofast optimization option, as specified for each experiment. Moreover the number of bodies is fixed to $n = 20000$ and the number of time steps is fixed to $m = 10$. We have also assumed that the output of the Baseline version is correct, referring to it in checking the code behaviour for the other optimizations.

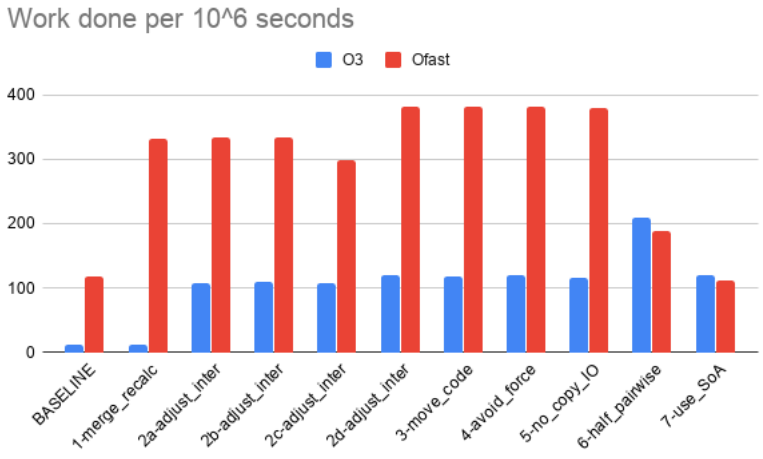
1 Baseline version, algorithmic and memory complexity, metrics

Algorithmic complexity: $O(m \times n^2)$ **explain calculations**
Memory complexity: $O(n)$ **explain calculations**
Metrics choosen: work/second, Op. rate(instructions per cycle), IPC, cache-misses /intruction, so we avoid the dependance on the problem size.
Numeric values obtained:
Total work: $m \cdot n^2 = 4$ G operations
Memory complexity: $14 \cdot 4 \cdot n = 1,12$ Mb

2 Performance statistics of the experiments

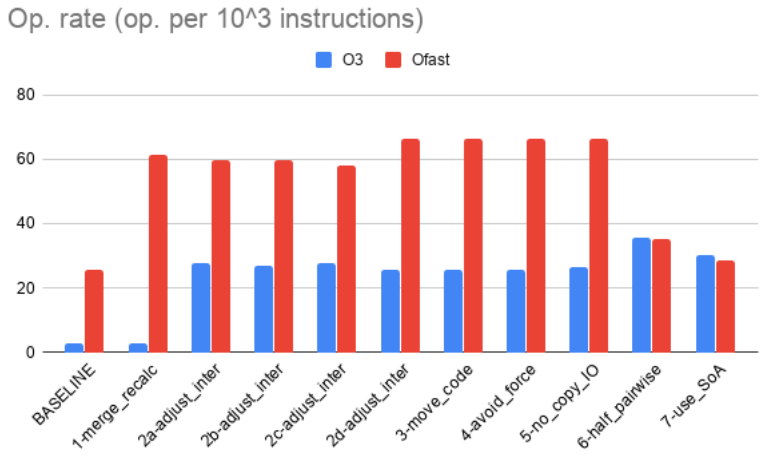
In this section we briefly describe the different incremental optimizations we applied, the results we expected from each experiment, and provide four figures which contain the quantitative results of their performances in terms of the metrics choosen before, properly rescaled to avoid both size-dependendy and too small or too high numerical values. We also analize the obtained quantitative results, focusing mainly on the optimizations which seem to highlight a bottle-neck in the performance.

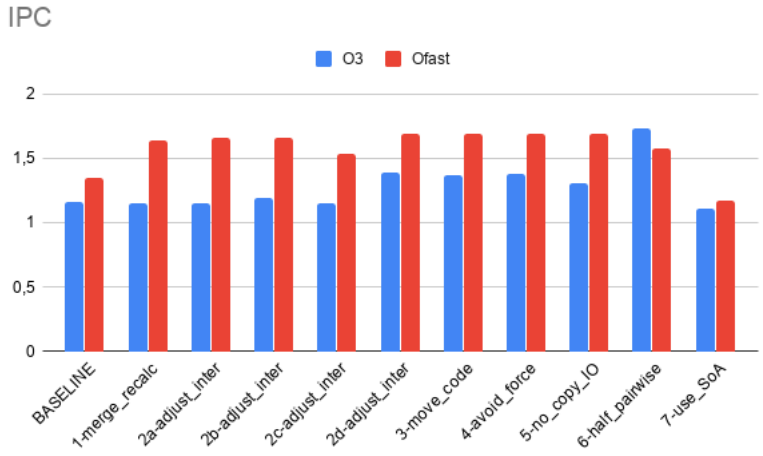
In the figures, the blue-colored data are for the -O3 flag, and the red-colored ones for the -Ofast flag. We display the data for both the types of flag since in the -Ofast case we observed a little loss of accuracy in the results, in the fourth or five decimal. Indeed it could be important in an application to choose the best option according to the desired level of accuracy in the results.
insert description of the general time, prviding some numbers like ”increased 3.5 times” etc



2.1 merge_recalc

This experiment merge the calculation of the new positions of the bodies with the cycle which calculates the forces. It consists in a classical loop fusion. As seen in the figures, it seem to be a bottle-neck optimization when we compile with the -Ofast flag. In this case the improvement is both in the operation rate and in the microarchitecture throughout (see the IPC).
analize assembly code to see if in this case the compiler is able to generate SIMD





2.2 adjust_interaction

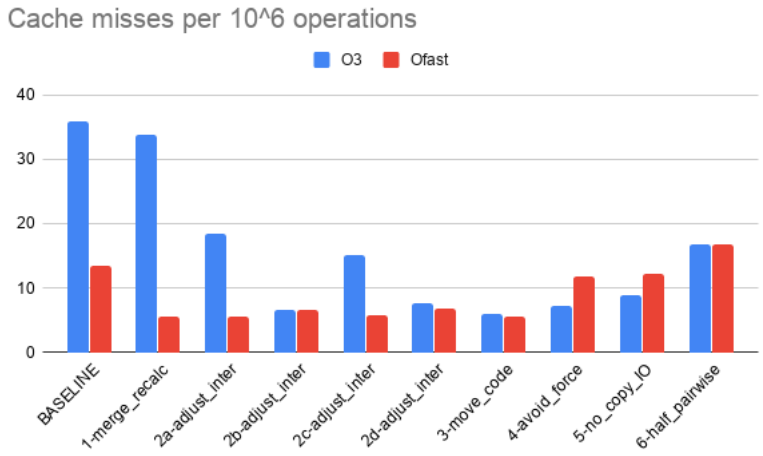
These four different experiments consist in different implementations of the expression that performs the computation of the force between body i and body j. In particular they play with the order of multiplication and square root calculations in the calculation of the forces. The aim of these improvements is to achieve a better coding efficiency (represented in the figures as the operation rate): indeed the square root calculation is known to be demanding, so trying to act on that could lead to good results in coding efficiency. Actually, as proved in figures, these optimizations are a bottle-neck for the coding efficiency, especially in the -O3 case.
The case 2d seems to be of particular interest, both in the -O3 case and in the -Ofast one.
In the -O3 case it increases the IPC, while in the -Ofast one it increases the coding efficiency up to the best value reached by all the optimizations.
analize assempbly code to find SIMDs if present

2.3 move_code

Here we perform a function inlining by moving the code of the function bodyBodyInteraction() into the integrate function: we expect an increased speed-up of the program, since avoid calling a function means also to avoid passing values by copy to the function. As we see in the figures this optimization does not lead to the expected improvements: to explain this we conjecture that the values needed to the function were already be copied in the GPU cache memory, so pass them to the function was not time-expensive. However we won't investigate the real reason of this result, since it doesn't seem to be a bottle-neck for the performances.

2.4 avoid_force

This optimzation consists in rewriting the code inside the integrate() function to avoid using the force[] vector. We expected a decreasing in the cache misses operations, indeed we conjectured that the processor does not need to read and write data from a fixed, perhaps slow, memory position. As in the previous case, data prove that this doesn't change almost anything in the performance profile. As before we are not going to analyze this



case further.

2.5 no_copy_IO

In this improvement, the useless data copy from vector **pout** to vector **pin** is avoided. We didn't expect much improvement from this adjustment, since by our choice in data, this copy is repeated just $m = 10$ times. In general we conjecture that this avoided data-copy should lead to a better IPC value, since the more data are written or read from the memory, the more time the processor is waiting to get the values.

2.6 half_pairwise

Here we use the symmetry of the problem to calculate only half of the forces exerted between the bodies.
need to see the code to try to explain the decreasing of the performances, and maybe also the assembly code to try to find out the reason

2.7 use_SoA

Here we modify the data structures representing the bodies: we represent the positions of the n bodies using 4 arrays of n elements pinx[n], piny[n], pinz[n] and pinw[n].
This experiment shows a decreasing in the performance outcome: in particular, looking at the data, we observe a significant increasing in the cache misses, around 4000 times the cache misses of the previous versions of the program. Indeed we conjecture that with these new data storage the memory accesses become a very important bottle neck for the performances.

3 Summary

Experiments show that the most succesful optimizations are the ones which plays with the order of the operations in computing the force beetween the bodies. Indeed the main bottle neck for the performnces in this program seems to be this one.
be more precise, giving some precise referrement to the assembly code analized before.
Instead, reducing the memory storage, operating with code motion or avoid copying data doesn't affect the general performance significantly.