Performance Analysis of N-Body Simulations

Gülçin Gedik
M1 CHPS

Contents:

We are presenting a performance analysis of a 3-Body simulation which has an overall quadratic complexity and challenging arithmetic intensity.

We will start by presenting our kernels, then we will present our target architecture and the methodology used during our benchmarks and optimizations made for each version with further possible optimizations and the numerical stability of our results.

Kernels:

Our first kernel is the innermost loop where we calculate distances between particles and the net force. This kernel has a complexity of O(N).In the first kernel we calculate the necessary arguments to calculate Newton's law, then it calculates the net force therefore this kernel has a complexity of O(N) but we can say that it's still arithmetically demanding since it contains a total of 17 arithmetic operations, and if we also consider loads and stores overall arithmetic intensity of this kernel becomes 45 operations per loop, so this kernel alone has the most operations in our code. It makes it easier to understand why our bottleneck is coming from here. We notice that a softening operation is applied in order to minimize the effects of singularities in our equation, therefore we are altering our system to obtain numerical stability and efficiency. We have to remember that this kernel is also nested into our second kernel, which is where the complexity of O(N^2) is coming from.

```
for ( every particle ) {
      for ( every particle ) {
            Compute:
                        Newton's law
                        Net force
      }
  Compute:
  Velocities
}
```

The second kernel is where we calculate the final velocities of our particles, therefore we have 6 arithmetic operations and a total of 18 operations with load and stores, if we don't include the first kernel. If we don't include the first kernel, this one has a complexity of O(N) but since they are nested together, our overall complexity increases to O(N^2). Here we also notice the data dependency in our dot product operations between the first and the second kernel, the importance of which will be explained later.

```
for ( every particle ) {
    Compute:
    Positions
}
```

And in the last kernel where we calculate the final positions of our particles according to their velocities. This one also consists of dot products which can give us the freedom to take advantage of FMA ports under suitable computer architectures to reduce execution time, since multiply and add instructions if calculated separately they cost us 8 cycles which doubles the number of cycles for the same operation made using FMA ports*.

Target Architecture and Methodology Used:

Our performance analysis takes into account various aspects that can play on the efficiency of our code, such as: compilation environments, optimization flags, size of our inputs, in which cache line they adapt, what kind of optimizations are made etc.

To obtain our benchmark of floating point operations per second we calculated the total number of arithmetic operations done by number of interactions and total number of loads and obtained the seconds elapsed during our computation with `omp_get_wtime()` routine. For the bandwidth benchmark we paid attention to repeat our benchmark at least 35 times to have an accurate calcule of the standard deviation while benchmarking our bandwidth, but since this method requires a lot of repetitions in order to have reliable results, we implemented it only in a selected version to show the considerable difference.
Additionally, we also clued our compiler through flags to take advantage of CPU specific features such as sse or avx, such as -mavx, -march, -msse, -mfma, only for the O3 and Ofast optimisation flags.

Our target architecture consists of:

=x86_64= CometLake:

Model	Cores	GHz	L1 size Kib	L2 size Kib	L3 size Mib	SIMD
i7-10750H	6	2.6	32	256	12	SSE, AVX2

Optimizations:

The code provided for our baseline application is already explained in the Kernels section. Since this application requires multiple data accesses and serious number of operations we can optimize our code:
On Memory accesses:

AoS vs SoA : this optimization aims to have a regular access pattern to our data with regular strides by switching it from array of Structs to struct of arrays.

Alignment: With aligned_alloc we give a signal to our compiler to align our data so that we can have regular access to data by loading it aligned with our caches to increase our efficiency.

On number of operations:

Frequency reduction: Unnecessary elements that can be calculated outside of the loop are calculated this way. This method aims to decrease the number of calculations, it doesn't play on complexity but it plays on the overall arithmetic intensity of our programme.

Unroll: Compiler is under exploited if we give it scalar calculations. Unrolling our loops helps us to take advantage of our cpu by giving it more work to do at the same time while reducing the number of branches. We have to remember that it is possible only under some conditions, such as independence of the data, that's why we were unable to unroll the second kernel. And we have to highlight that where we can unroll we can also vectorize our code.

Compiler optimizations:

Compiler optimisations can vectorize our code to get a better performance out of SIMD instructions to have a better bandwidth. Since ymm are wider than xmm registers, we can have a more efficient code by manipulating more data with single instruction, hence the name. They can also in-line some functions to reach them faster, or unroll loops as explained earlier, or make optimizations in such a way that it executes math related instructions faster at the expense of numerical accuracy.
O1 level optimization doesn't do any kind of optimization listed above. O2 enables to generate SIMD instructions. O3 includes also O2, so it also vectorizes our code,to help the compiler to generate a better suitable code for our architecture we passed

architecture specific flags to favor vectorized instructions. Lastly, Ofast includes O3 while activating the -ffast-math flag.

Versions:

V0: It's our baseline application. This version applies an array of structs, without any alignment.

V1: This version applies struct of array, without any alignment. Our purpose is to show the influence of different methods to access the data. By comparing it with V0.

V2: This version applies Array of Struct, with alignment.

V3: This version applies an array of structs, without any alignment, and with unrolling.

V4: This version applies struct of arrays, with alignment and unrolling. Here we have to point out that if there's a dependency between the loops we can hardly unroll them. Here we can see the dependency between the first and the second kernel.

V5: This version implements struct of arrays, with alignment, and unrolling, but this time it also implements frequency reduction.

V8: This version is V0 but instead of making a call to pow() function it calculates the power and sqrt within the code. Sqrt is implemented with Lomont reverse square root then reversed. Although it is susceptible to have less numerical precision.

V13: This applies SIMD AVX2 instructions with intel intrinsics, with array of structs, alignment and unrolling 2 times in the third kernel.

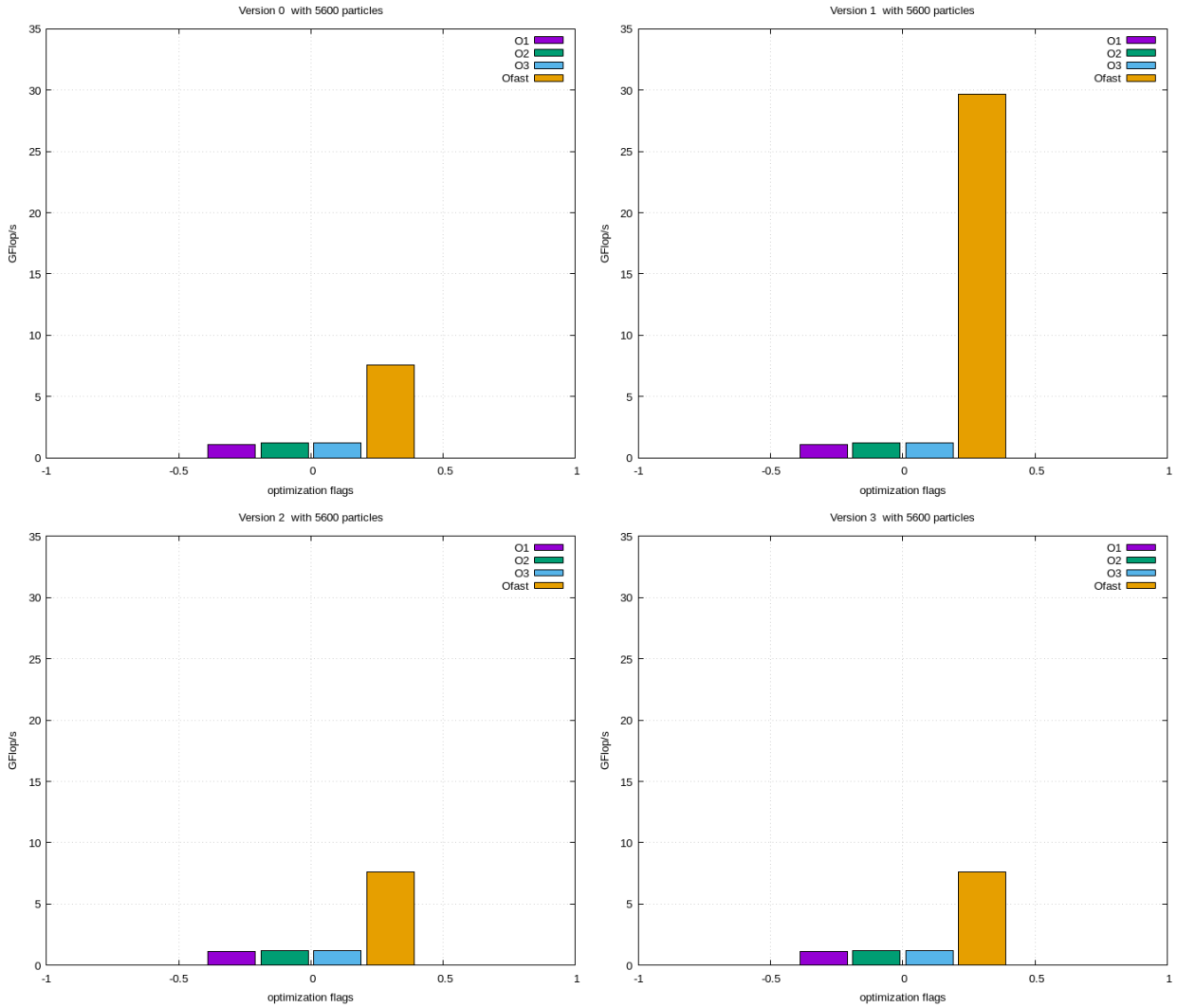
V14: In this version, we implemented struct of arrays with alignment, frequency reduction and unrolling two times in the third kernel while using FMA instructions with intel intrinsics.

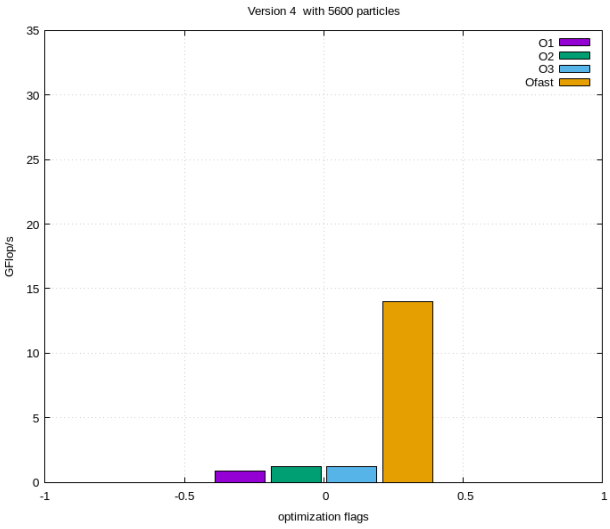
V12: it's the same as V11 but here all the kernels are transformed into assembly codes. A numerical error that we were unable to traceback is committed therefore this version is not included in our benchmarks.

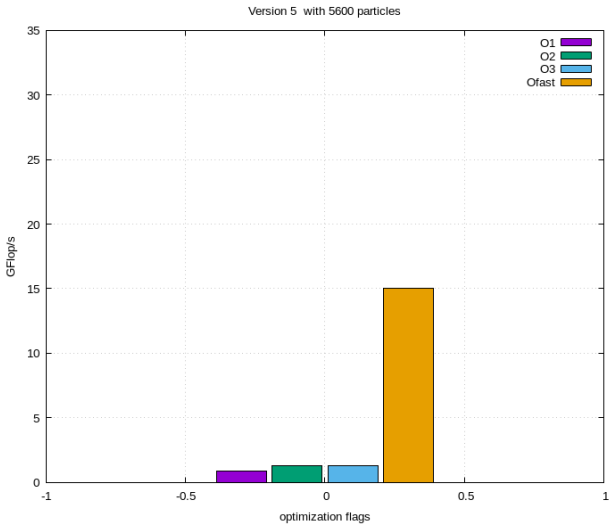
Benchmark Results:

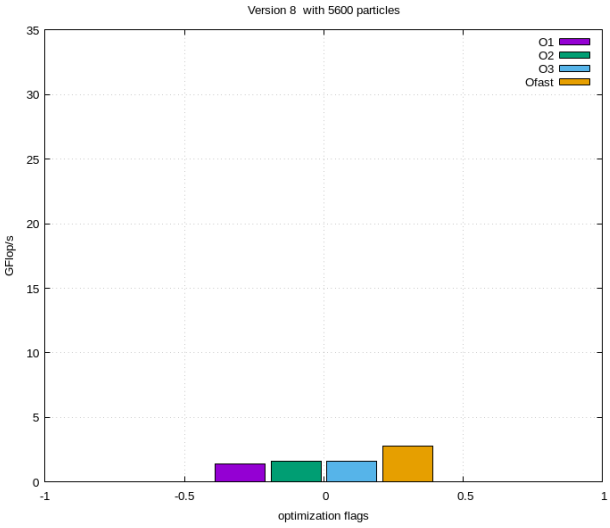
GFlop/s:

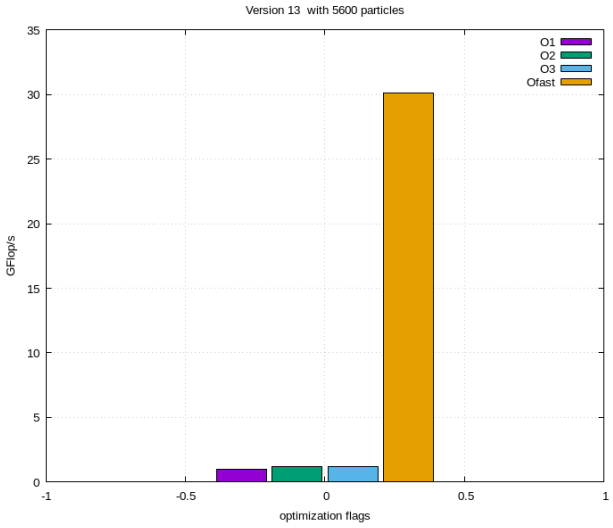
We aimed to see the effect of different optimizations on the number of floating operations made in various versions. With compiler flags to indicate CPU architecture specific features we aimed to clue the compiler to apply better optimisations/ vectorisations for our code.

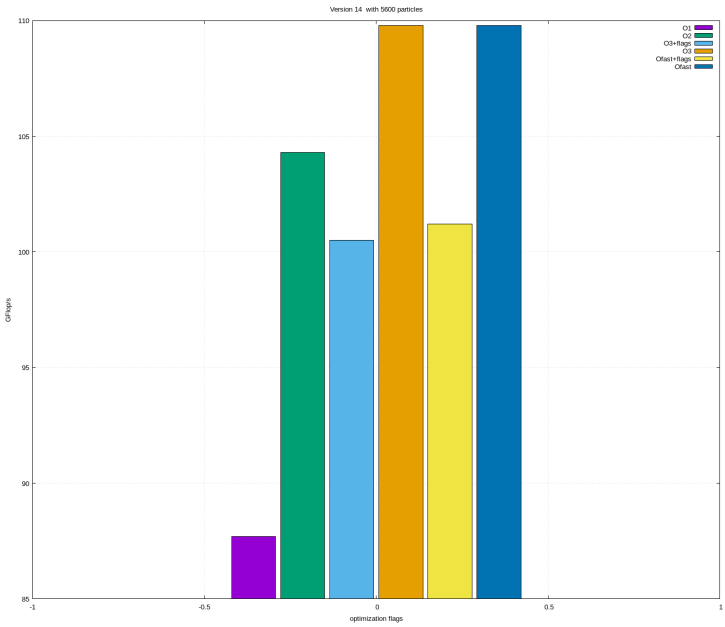








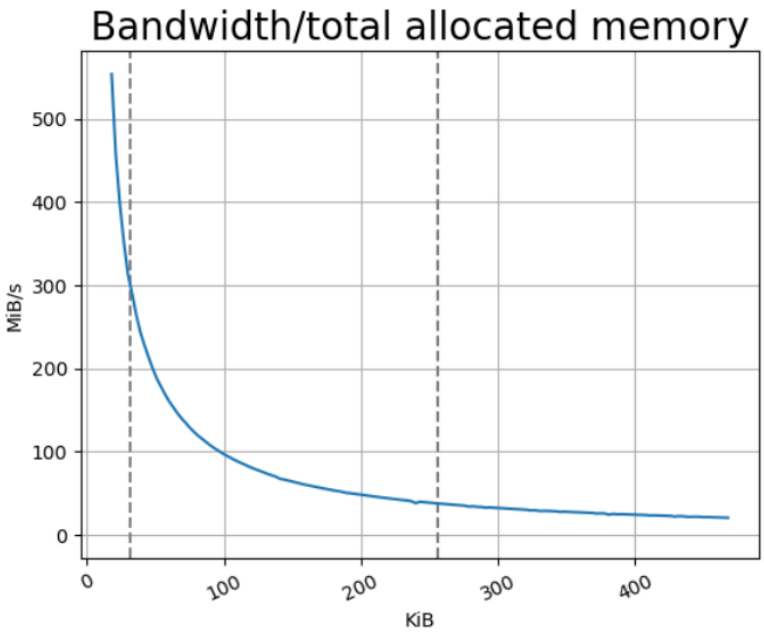




Here we see our intrinsics code compiled with gcc and optimisation flags. While constructing this benchmark our initial aim was to see the behavior of our intrinsics code with different flags and we also added CPU architecture specific flags for O3 and Ofast hoping that it would benefit our code. Surprising to us O3 and Ofast with this version did not succeed O2, therefore we did another benchmark without these flags and obtained a better result. If we can examine optrpt files generated we can

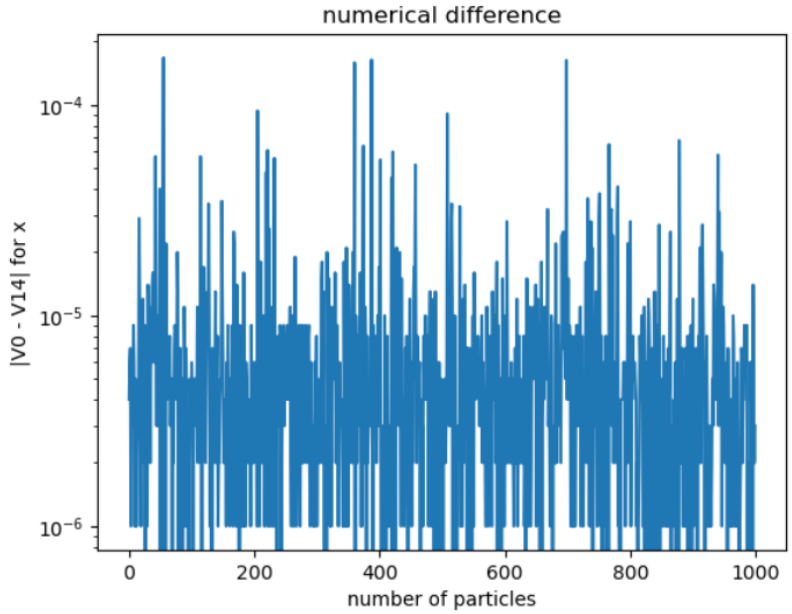
understand the reason behind this, or we can make an objdump and examine the binaries generated.

Bandwidth through caches:



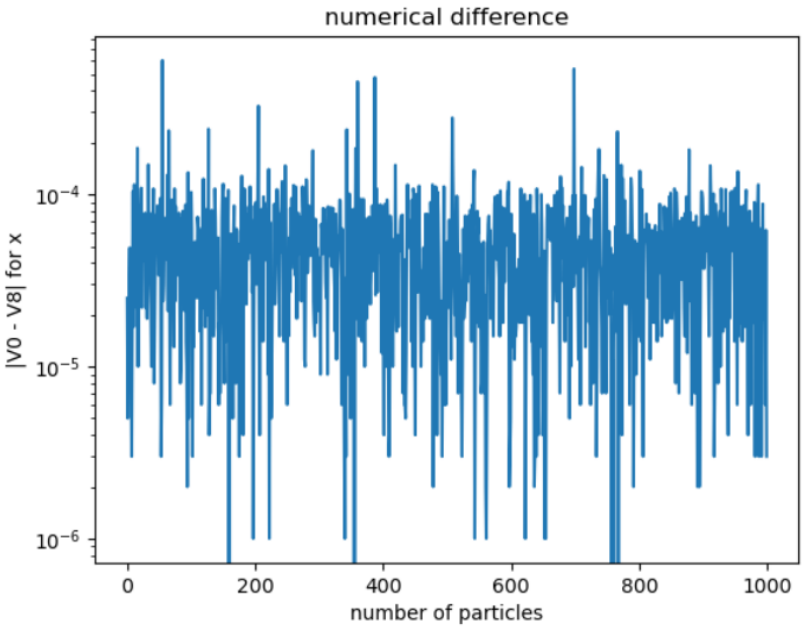
Here we compiled our intrinsics code with gcc O3 optimization flag. We increased the number of particles regularly and calculated MiB/s. This graph shows the total allocated memory size on the x axis, and MiB/s on the y axis, and the horizontal lines are to show the capacities of L1 and L2.
We aimed to show the difference of speed between different cache levels. We can clearly see that as we change caches the slope of the graph gets smaller, therefore even though we don't see the change of speed directly we can say that L1 is faster than L2 and L2 is faster than L3.

Numerical precision:



Here we aim to show the numerical difference between our intrinsics and the base code over the number of particles with gcc O3 optimization. We know that floating points are not the exact representations of our data and the way compiler generated code and the intrinsics instruction handle them can be different, therefore any change made on the code or the compilation makes our

results susceptible to numerical errors which could be accumulated through the loops and influence our results. Here we see that our overall numerical difference compared to our base code is below 10e-4, which can be considered as acceptable for floating points.



Here we aim to show the numerical difference between our base version and version8 where we calculated the square root with Lomont Method. Again we can see that our overall difference stays in between 10e-4 and 10e-5 which could be considered as acceptable for floating points.

Conclusion:
We have implemented different versions, each covering a specific optimisation. Through our benchmarks we can see that applying good coding practices such as structure of arrays, unrolling and frequency reduction can benefit our code. If we want to take it further we can also benefit from compiler level optimisations and vectorized instructions if they are available for our architecture. We also covered possible numerical differences between optimisations. We notice that with these practices we can make our code 100 times faster.

Future Work:

Since we have used different SIMD instructions such as instructions for FMA or simple multiply add, or different methods to calculate square root etc our calculations are susceptible to numerical differences/errors. Therefore examining numerical stability in our calculations is a must for our future work.

We had written two versions of the same code, one with FMA instructions the other one with multiply-add. But we were unable to trace back the numerical error we had committed during our calculations. We aim to correct those mistakes and make a comparison on the number of calculations per second between these two methods.

After perfecting our assembly, we can compare our vectorized version and the compiler's vectorized versions and their efficiencies, we might also get help from a debugger to resolve the bottlenecks that we might have.

We can also compare the intrinsics code we implemented with the compiler assembly code generated by the same intrinsics code, so that we can have a better understanding of compiler optimizations. Additionally, we can also make comparisons with and without compiler flags such as -msse, mavx etc.

To reduce timing biases, we could apply different timing measurements such as elapsed cycled time with rdtsc etc.

Since some of our loops are independent from each other we could paralyze them with openMp directives.

We could also apply other algorithms developed for this particular program that we have seen in the article, such as tree method, which seems pretty interesting since it has a complexity of O(NlogN).

Resources:
https://github.com/srinathv/ImproveHpc/blob/master/intel/2015-compilerSamples/C%2B%2B/intrinsic_samples/intrin_dot_sample.c
http://kayaogz.github.io/teaching/programmation-parallele-distribuee-2019/cours/simd.pdf
https://academic.oup.com/mnras/article/324/2/273/1020633
fmadd_ss, mul_ss, add_ss:
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html