Performance Analysis of N-Body Simulations

Gülçin Gedik
M1 CHPS

Contents:

In this report, we are presenting a performance analysis of a 3-Body simulation which has a complexity of O(N^2) since the force applied on each particle depends on other particles, therefore this simulation has a challenging arithmetic intensity since every dotproduct requires at least 2 loads, 2 operations and 1 store.
We will start by presenting our kernels, then we will present our target architecture and the methodology used during our benchmark and optimizations made for each version. Detailed target information can be found in target_info.txt.

This simulation is composed of three kernels:
In the first kernel we calculate the necessary arguments to calculate Newton's law, then it calculates the net force, this kernel has a complexity of O(N).
In this kernel we have multiple multiplications, divisions and a square root. With all the arithmetic intensity and the complexity combined, this is where the bottleneck of our problem is originating from. Therefore our optimizations should have been more focused on here.
This first kernel is nested into our second kernel, which is where the complexity of O(N^2) 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.
The second kernel is where we calculate the final velocities of our particles, therefore it is composed of a dot prod which is an operation heavily used in real life HPC applications.
This is the reason we have chosen to benchmark this simulation since it includes multiple real life operations we come across during our applications.
And the last kernel is also a dot prod where we calculate thé final positions of our particles according to théorie vélocities. Here we should highlight some specialities of dotprod operation,it plays on the operations of load, store, multiplication and addition, which also gives the freedom to do a multiplication and an addition in a single cycle thanks to FMA.
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.

Optimizations made:
AoS vs SoA : this optimization aims to have a regular access 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 can have regular access to data to increase our efficiency.

Frequency reduction: unnecessary elements that can be calculated outside of the loop are calculated this way. This method aims to decrease number of calculations, it doesn't play on complexity but it plays on the 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 some time. We have to remind that it is possible only under some conditions, such as independency of the data. 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. They can also in-line some functions to reach them faster.

V0: it's our baseline application. This version applies 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 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.
V7: this version is V0 compiled with the vectorize compiler option.
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 squareroot then reversed. Although it is susceptible to have less numerical precision.
V11: This version applies SIMD AVX2 instructions with fma instructions and structs of arrays with alignment. Only the third kernel is transformed to assembly, to keep the numerical accuracy. The rest is same as V1.
Additional versions that we have prepared:
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.
V13: although this version should be imitating V11 with the only difference being the usage of multiply + add instructions instead of fma instructions, it doesn't give the same results, a numerical error has been committed that we were unable to traceback during our experiments.

The fact that we were unable generate graphs doesn't stop us still making comments on our data generated with the second version of our script(explained later in the challenges):
Here we are comparing all the data obtained by various versions (form 0 to 11) with -O3 optimization flag (which activates vectorization compiler options as well as inlining, switching loops, options with O2 options), compiled with gcc and ten thousand particles: It is obvious that Version 8 gives us the best efficiency, which is kind of surprising, knowing that the only optimization we made on this version was inlining the math functions to resolve the bottleneck around calculating the forces, rest of the optimizations were handled by the compiler itself. Although we tried to help the compiler with our vectorized instruction and fma dotproducts in version 11, it seems that they were not that helpful.

Challenges faced:
Unfortunately our script to generate data from all the variations that we have created takes more than an hour to finish. At this moment it's still running and it's been two hours and it's not even half way done. Although I am unable to understand why it takes that long apparently number of particles for different cache sizes plays a lot on our programme. And the fact that we are so forgetful that we had forgotten to put BW measurements, it's a painful lesson to learn and an example for future students.
From now on I'll re run the script with less particles so that I can have some plots to make examinations from.
The script that takes too long to complete takes the caches into account: for that we run out programme with 4000 for L1, 10000 for L2 and 50000 particles for L3. Since the caches on our system are 192Kib, 1,5 Mib and 12Mib respectively. Since that script was not completed in two hours we had to switch our model to run only with 4000 and 10000 particles. At least we will be able to examine if there's a change in our measurements.
Unfortunately it took us 3 days to let go our assembly versions, we were so blocked on it that we didn't realize how much time had passed, therefore this report.

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 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.
To reduce timing biases, we could apply different timing measurements such as elapsed time with clock get time 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).