CS257 Optimisation Report

u1405424

15th March 2016

CS257 - Optimisation Report
15th March 2016

1. Introduction

Scientific code must sometimes sacrifice accuracy of results in order for them to be of value. For example, weather simulations which take over 24 hours will produce useless results if wanting to simulate and predict tomorrows conditions. N-body simulations, the subject of this coursework, are another example of scientific code which may need optimisation to achieve functional usefulness. For example, if 10000 stars take 3 hours to simulate, it may be impractical to simulate millions with the same code. This report will document the optimisation of an N-body simulation such that it achieves functional usefulness, in both running time and the accuracy of output. This will be done through code refactoring to achieve better utilisation of hardware resources, along with consideration of the algorithmic complexity of the code itself.

2. Research and Design

There are numerous code optimisations which can be made to improve the runtime of the program. The main bottleneck of the program is the $O(n^2)$ runtime, in loop1. Possible improvements to the algorithmic runtime are discussed fully in section 3. After possible improvements to the algorithm, a number of code level optimisations can be considered. The first of these is loop fission. Loop fission will increase the spatial locality of code. This is done by processing all data elements of one array, before processing another array (Leeke et al, 2016a). This increases spatial locality as the compiler can infer that more cache lines can be brought from memory at once. It is important to keep memory high in the memory hierarchy, so that it can be moved into CPU registers as fast as possible, reducing latency in the fetch-decode cycle. This reduces cache misses, increasing the throughput of the code.

Loop Unrolling is also used to increase performance. The increase is gained by the fact that the loop guard will be checked fewer times than in a regular loop, by applying the loop body to multiple data elements at once. While no overly complex loop guards exist in the code, this will still reduce the number of instructions carried out for a loop, by the factor of unrolling. This factor is how many iterations are performed in a single loop body.

Further to this, vectorisation through Streaming SIMD Extensions (SSE) will improve performance greatly. Vectorisation improves performance by applying the same instruction to multiple data elements at once. An SSE unit on 64 bit architectures has 16 special purpose registers, each 128bits wide (Blair-Chappell, 2016). These registers can load multiple elements and apply a single instruction to all at once. As a float in C is 32 bits, 4 can be processed at once. This technique is highly applicable to an N-body simulation, as the same operations are being applied in batch to multiple data elements. Accuracy is an issue however with SSE when performing certain operations, for example reciprocal square root(Intel, 2016). Another issue is branching, which requires each data element to be considered. However, masking can overcome this.

Masking can be used to allow branching inside vectorised code (Leeke et al, 2016b). This is needed because when elements are vectorised, decisions cannot be applied to individual values, as they can when handling individual data values. In masking, for each part of the conditional statement, a mask is created. Then all masks are combined using logical OR. The mask has bits set to 1 for an element, if the condition is met by the variable, and 0 if not. The results for both the branches being taken and not being taken are computed. Finally, the mask is combined with these results so that if the mask is 0 for a variable, the results of first branch is taken, and if 1, then second is taken. This will mean that 4 floats can branch at once.

While SSE exploits the Data Level Parallelism of the code(Blair-Chappell, 2016), threading can be used to further increase performance. As the system has multiple cores, each with an SSE unit, thread level parallelism can be applied to the code. This needs specific tailoring to the code itself, and must consider data dependencies, synchronisation and locks. Threads have a startup cost (Lindberg, 2009), hence it must also be considered if the use of threads will actually result in increased performance at all.

3. Algorithmic Optimisation

Loop1 has a runtime of $O(n^2)$, making it the main bottleneck of the program. This is due to the nature of an N-Body simulation: all points must compute the gravitational force between itself and all other points. The Barnes-Hut algorithm (1986, cited in Sharabi, 2000) however uses approximations to reduce the runtime to $O(n*log(n))$. The algorithm uses an Octree, where each non-leaf node has 8 children, 1 for each octant of 3D space the parent represents. Importantly, each node can only contain 0 or 1 particles in the area of 3D space it represents. When another particle needs to be in the same space, 8 children are created and the particles are moved from the parent to the correct octant child.

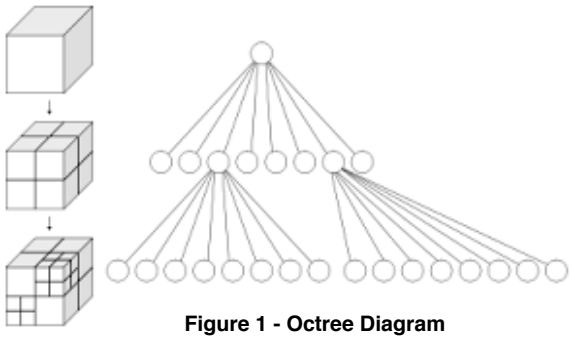


Figure 1 - Octree Diagram

The algorithm is as follows: Group the input particles by subdividing the 3D space, using an Octree. As children are added to the tree, update their parent node so that the represented centre of mass is the centre of of mass of all children, and the total mass is the sum of all child masses. Then for each body P_i, traverse the Octree starting at the root. Using an evaluation function, determine if the current particle P_j has a centre of mass far enough from P_i. If not far enough, and P_j is not in a leaf node, then recursively check all child nodes of P_j. If P_j is far enough from P_i, or is a leaf, then use the centre of mass and total mass of P_j, to approximate the force of all of the children of P_j on P_i. This means that not all particles are compared using the algorithm.

While the runtime would improve performance, the relative gain may be achievable through other optimisations to the code such as vectorisation. The algorithm is also an approximation, and may yield results which are too inaccurate to be useful. Considering the technical complexity of implementing the algorithm, it was decided to spend time implementing other optimisations. However, given a longer time, the Barnes-Hut algorithm would have been used to optimise the program.

4. Implementation

4.1. Loop 0

The operations in this loop have no dependencies. This means it is possible to fission the loop into 3, so each array is processed fully before the next. To increase performance further, SSE is applied. An SSE register `ini` is loaded with 4 floats valued at 0.0. The loop is then passed through in strides of 4, storing the value of `ini` at the current position. The loop initially ran in a few milliseconds. For this reason, parallelisation was not applied as the time to create threads would outweigh the processing time of

the loop. It would be possible to fuse loop0 and loop1, initialising each position as it was needed in loop1. However, this would decrease spatial locality of loop1 itself, so was not applied.

4.2. Loop 1

The loop consists of SIMD instructions, hence is a candidate for vectorisation, through SSE. Due to the nested nature, the outer loop has strides of 4. This means then the inner loop is applied to 4 elements of the outer loop at once. This approach works well with parallelisation, discussed below. For each inner loop pass, 4 copies of the position array at location j are moved into an SSE register, so then each instruction can be applied to copies in the inner loop and values of the outer loop. An inverse square root is performed using the `_mm_rsqrt` function. This is only accurate to 11 bits (Intel, 2016), but has a throughput of 1cc (Clock Cycle). `_mm_sqrt` is accurate to more bits, however has a throughput of 7cc on the architecture in use. The trade of accuracy for precision is important here, discussed further in section 6. The result of the inner loop is stored in an accumulator, and then stored once in the acceleration arrays, when the inner loop has terminated. This will remove N load and N store operations compared to storing on each inner loop pass. Striding by 4 on the inner loop would have resulted in *N*N/4* memory writes, incurring more delay due to memory than the solution being used.

Due to the runtime of the loop, for even relatively small inputs, parallelising the code improves performance. Initially, the inner loop was parallelised. However, this resulted in worse performance, as threads had to synchronise to write. A better solution was found, paralleling the outer loop. As the inner loop only reads from the location arrays, threads may reads elements in the array concurrently. As each thread is given part of the outer loop to parallelise, it can write to the outer loop acceleration array without synchronisation issues. `#pragma omp parallel for schedule(static) shared(N)` is used to parallelise. The `for` keyword means that each pass can be in a different thread. `Schedule static` indicates that each thread is given a range of values for the loop to compute, e.g. thread 1 executes loops 0-99, thread 2 executes 100-199 etc. Dynamic scheduling would mean that when a thread finishes, it takes the next free thread from a shared queue (Lindberg, 2009). Overhead is incurred with this method, as threads must compete for the next element if they finish concurrently. Static removes this competition.

4.3. Loop 3

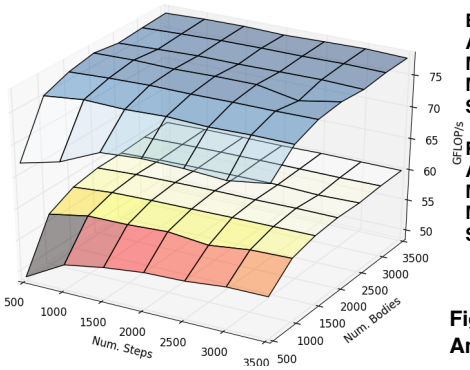
The loop has conditional branches, which hinder vectorisation. To resolve this, bit masking is applied. In the code, the first branch does not change the value of the velocity, and branch 2 does change the velocity of a particle. With this concept, SSE can be applied to the branches. Masks are created for 4 values at once from the conditions of the if statement, and then `_mm_or_ps(_mm_and_ps(mask, mul), _mm_andnot_ps(mask, vx_v));` is applied. `_mm_and_ps(mask, mul)` states to keep the bits of `mul` (velocity * -1) where the mask is 1. `_mm_andnot_ps(mask, vx_v))` will keep the values of the initial velocity, `vx_v`, where the mask is 0. OR is applied to combine the results which will be disjoint, creating a single vector. This means the entire loop can now be vectorised, which incurs a large speedup. Another optimisation applied to this loop was loop fission to increase spatial locality.

4.4. Loop 2

This was similar to loop 0. No dependencies existed within the loop, hence loop fission can be applied to increase temporal locality. SSE is also applied to allow the operations to be performed four at a time.

An attempt to remove loops 0 and 2, was made. This was possible due to accumulators within the inner loop of loop1 being used, removing the need for the acceleration arrays. Then, loop0 was unneeded, and fusing loop2 with loop1 meant velocity was obtained by integrating the accumulators. This however resulted in similar performance to what was already achieved in testing, but reduced spatial locality as the position vectors and velocity vectors were accessed in an interleaved manner, instead of in blocks.

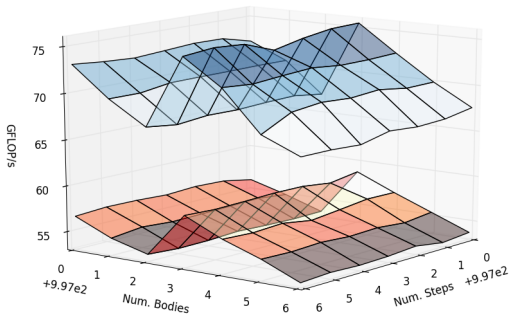
5. Performance



Blue (rsqrt)
Average: 76.25 GFLOP/s
Min: 67.24 GFLOP/s
Max: 77.91 GFLOP/s
Standard Deviation: 2.40

Red (sqrt)
Average: 58.52 GFLOP/s
Min: 49.01 GFLOP/s
Max: 60.01 GFLOP/s
Standard Deviation: 2.29

Figure 2 - Macro Analysis of GFLOP/s



Blue (rsqrt)
Average: 70.70 GFLOP/s
Min: 66.36 GFLOP/s
Max: 75.62 GFLOP/s
Standard Deviation: 2.84

Red (sqrt)
Average: 55.68 GFLOP/s
Min: 53.47 GFLOP/s
Max: 58.79 GFLOP/s
Standard Deviation: 1.66

Figure 3 - Micro Analysis of GFLOP/s

The machine used to test performance and accuracy used an Intel i5-4590@3.30Ghz, running Linux RedHat (DCS 0.06). With 6584MIPS (~2 SSE instructions per cycle), the machine was calculated to have a theoretical peak of 105.6 GFLOP/s. This figure is used as a benchmark for the report. 490 total tests were carried out with a test harness. To speed up testing, `usleep(usecs)` on line 170, cs257.c, was commented out. This removed the 60fps cap.

For testing, a macro and micro approach was taken. The micro test used a small range of values of bodies and time steps in a square, 997x997 to 1003x1003. Each test was ran 3 times and the mean value taken. The focus on a small area will reveal how performance varies over irregular inputs, such as an non multiples of 4. As seen in Figure 3, the GFLOP/s was effected highly by the number of bodies that the simulation computed, but remained moderately consistent for the number of time steps being simulated for (while the number of bodies are kept constant). The speedup of the micro tests was on average 34.96x, with a peak of 37.36x. This was computed by comparing the GFLOP/s of the original program against the optimised version. This performance took the time of loop1 from an average of 9.87s to 0.28s. An average of 70.70 GFLOP/s and a peak of 75.62 was achieved, reaching 67.08% and 71.75% of the theoretical maximum respectively. From this, it is seen that performance varies within a small range of values, peaking on multiples of 4 bodies. This corresponds to the SSE vector size, which can compute 4 bodies concurrently, so is somewhat expected behaviour.

The Macro analysis was performed over a much larger range of values, from 500x500 to 3500x3500, with a step of 500 for bodies and time period. The optimised code was ran 3 times for each point, and the unoptimised 2 times for each point, due to the

duration of some loops. The macro analysis revealed that as both the number of bodies and time points increased from about 68 GFLOP/s rapidly, before plateauing around 76 GFLOP/s, seen in Figure 2. One reason for this apparent increase is that the code initially ran in too little time to be recorded correctly, for example running 4 bodies in 4 time points registers 0 GFLOP/s as a result. The macro analysis is encouraging, as it suggests that the code will have similar performance for most inputs beyond a certain input size. The average performance was 76.25 GFLOP/s, with a peak of 77.91 GFLOP/s, achieving 72.21% and 73.78% of peak performance respectively. This is a respective 37.70x and 38.41x speedup. It can be predicted from the results that the performance will follow the zig-zag pattern from Figure 3 for the range of possible inputs, but averaging to around 76 GFLOP/s for each block of 4 bodies. From this, it can be predicted that the average performance for all inputs above a certain number of bodies will be high, with an approximate 38x speedup compared to the original program's performance.

During testing on alternate machines, a peak performance of 82.61 GFLOP/s was achieved for 10000x1000, a utilisation of 80.67%, on an i5-3470@3.2Ghz (DCS, 003). However, the machine performed worse than other test machines for smaller input sizes, showing that the code performance will vary depending on the architecture in use.

6. Accuracy

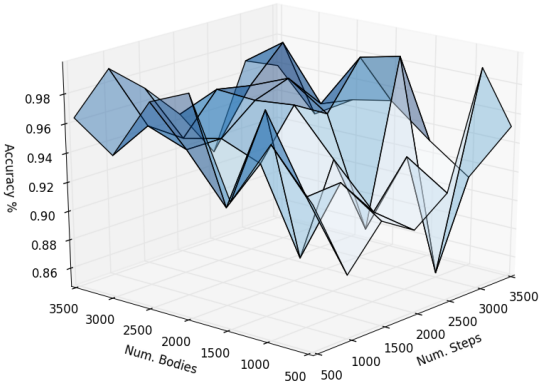


Figure 4 - Macro Analysis of Accuracy (%)

Average: 94.63%
Min: 85.01%
Max: 99.83%
Standard Deviation: 3.26%

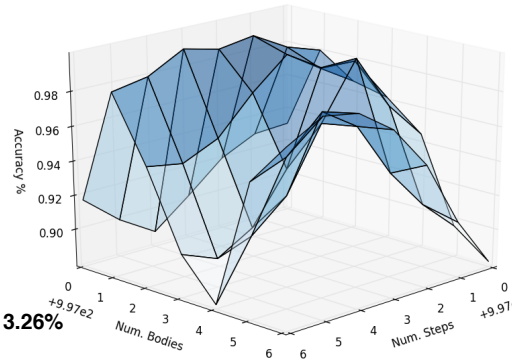


Figure 5 - Micro Analysis of Accuracy (%)

Average: 95.01%
Min: 88.04%
Max: 99.94%
Standard Deviation:3.60%

The code uses the `_mm_rsqrt` function to compute the reciprocal square root in loop1. This is a cause of inaccuracy in the program output. According to Intel (2016), this is only accurate to 11 bits, compared to the accuracy of a standard `_mm_sqrt` operation. In a macro analysis however, results were surprising. The average accuracy when using `_mm_sqrt` was found to be 95.89%, which was only slightly better than the average of `_mm_rsqrt`, at 94.63% in the macro analysis, Figure 5. The standard deviation of accuracy for `_mm_rsqrt` was 3.25%, close to the 3.11% standard deviation found for `_mm_sqrt`, meaning that the accuracy will not change too much for different input sizes. The micro analysis produced very similar findings. This shows that there is little resulting difference in the accuracy of the returned result when using `_mm_rsqrt` compared to `_mm_sqrt`. However, the throughput of the operation is 1cc (Intel, 2016), compared to `_mm_sqrt` which is 7cc. An analysis of GFLOP/s performance showed `_mm_sqrt` had an average of 58.52 GFLOP/s in the macro analysis, seen in Figure 2. This is far from the 76.25 GFLOP/s average found when using `_mm_rsqrt`, which reflects the throughput of the operations. For this reason, a trade of approximately 0.8% accuracy for a further 1.3x speedup is valid. This is especially true when considering a very large number of bodies, which may take a period of days to compute.

Both analysis showed that there was little consistency in the accuracy of the code. Looking at the micro test results, the accuracy quickly changes with alterations to the number of bodies and time steps. It always remains reasonably accurate, with a minimum accuracy of 88.04%. The standard deviation of the macro analysis, 3.6%, suggests that 95% of tests will be within 87.81% accuracy, compared to the original answer. This is reasonable considering the performance increases, as justified above. An important consideration is that the original code does not use double precision floating point, hence the answer being compared to may not be accurate itself. For this reasons, the given accuracies are only relative to the output answer of the original, not the true answer of the problem. For this reason, it is difficult to say if the optimised code is actually more accurate relative to the true answer or not, than the original.

The accuracy of the macro analysis revealed more variations to the accuracy of the code. A smaller standard deviation was found than in the macro analysis, suggesting that while appearing to be random, the accuracy will not fall below a certain level: 95% of inputs should be no more than 11.89% inaccurate. Unfortunately, due to the time taken for the unoptimised code to run, gathering more data was impractical. However, it can be hypothesised due to the micro analysis having a small standard deviation, that the accuracy will not be much less than those found so far. It can therefore be concluded from the accuracy analysis that while the code has a level of inaccuracy, the average value and minimums are high enough for it to still produce a meaningful and useful output.

7. Conclusion

The optimisations applied to the given code have massively improved performance, achieving an approximate 37x speedup on the machine optimised for. This speedup has met the need for a reasonable runtime, for the simulation to be useful. For example, simulations of 1000 stars for 1000 steps now take 0.28s, down from 9.79 seconds. When scaled up, simulating 10000 stars only takes a few minutes. Of course, this incurred an accuracy loss, on average 5.37%. This has been reasoned to be an acceptable trade, hence meets the need for the result to be accurate, for the code to still produce useful outputs. Considering this, the project has been successful in optimising the scientific code to perform an N-Body simulation to a good standard.

8. References

Intel. (2016). *Intel Intrinsics Guide.* [ONLINE] Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE,SSE2&expand=4463,4465,4911. [Accessed 10 March 2016].

Leeke, M and Martin, G. (2016). *Algorithmic Optimisations and Code Refactoring.* [ONLINE] Available: http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs257/6_algorithmic_optimisations_and_code_refactoring.pdf. [Accessed 10th March 2016].

Leeke, M and Martin, G. (2016). *Compiler Optimisations and Parallelism.* [ONLINE] Available: http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs257/6_compiler_optimisations_and_parallelism.pdf. [Accessed 10 March 2016].

Blair-Chappell, S. (2016). *The Significance of SIMD, SSE and AVX.* [ONLINE] Available at: http://www.polyhedron.com/web_images/intel/productbriefs/3a_SIMD.pdf. [Accessed 15 March 2016].

Lindberg, P. (2009). *Performance Obstacles for Threading: How do they affect OpenMP code?.* [ONLINE] Available:https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code. [Accessed 12 March 2016)

Sharabi, E. (2000) *Parallelization of Barnes-Hut algorithm for the N-body Problem.* [ONLINE]. Available: http://ta.twi.tudelft.nl/PA/onderwijs/week13-14/Nbody.html. [Accessed 12 March 2016].