CS257 Report: Optimising the simulation of N-bodies behaving under the influence of gravity in a bounded cubical area.

Alen Buhanec, Mar. 11, 2015

1 Introduction

The aim of the coursework was to optimise a simulation of N-stars behaving under the influence of gravity in a cubical

The original code of the simulation revealed a four loop structure, with each successive loop depending on the previous loop.

2 Approach and Testing Methodology

2.1 Approach

A quick inspection of the original source code reveals several inefficiencies on a structural level, such as:

- Loop 0: poor locality and largely memory-bound
- Loop 1: unnecessary memory accesses to position and acceleration arrays; expensive operations (division, accurate square root)
- Loop 2: poor locality and potentially memory-bound
- Loop 3: poor locality and branching

These observations were confirmed when the reference program was profiled, and it also revealed that the most significant loop was, as expected from the $O(n^2)$ complexity, loop 1, which ran roughly 25000x longer than the next slowest loop with 10000 bodies being simulated over 100 timesteps.

Given these observations, structural improvements were attempted first, and are described in Section 3. These improvements addressed poor locality and the memory-boundedness of loops.

Following that, vectorisation was used to further improve performance and optimisations were performed to address the expensive operations. Additionally, at this stage, the branching in loop 3 was removed. These optimisations are described in Section 4.

Finally OpenMP threading was used, as described in Section 6.

2.2 Testing System and Testing Methodology

The primary test system was a Sony VPCZ23N9E, configured with an Intel® Core™ i7 2640M Processor and 8GB of memory. The system was running a 64-bit installation of Linux Mint 17.1 (based on Ubuntu 14.04) with the stable 3.19.1 kernel. GCC version 4.8.2-19ubuntu1 was used to compile the source code.

Profiling was performed with Zoom 3.3.3, with the system using cpufreqd to force the "performance" governor to allow for maximum CPU utilisation since a userspace governor was not available to lock the CPU frequency. Despite advertising a maximum frequency of 3.5GHz [1], the CPU core speeds never exceeded 3.3GHz, which was verified with hardware calls to the CPU using cpufreq-info.

Two profiling configurations were used, one measured CPU time, sampled every 1ms, and one sampled last level cache miss CPU events, every 6000 cycles. Every tested iteration of the program was profiled 5 times in order to determine the margin of error. When profiling the simulations usually use 10000 bodies as that was experimentally determined to be past the point when the performance reaches its maximum.

Additionally the performance of each iteration was compared to the theoretical maximum performance of the system. Using the a simplified version of the GFLOP/s equation [2] (since the system only has one CPU), and knowing we can perform 8 single-precision FLOPs per clock cycle [3], we can calculate the theoretical maximum performance of the system to be 52.8GFLOP/s.

$$\begin{aligned} GFLOPS_{max} &= num\_cores \cdot \frac{frequency}{core} \cdot \frac{FLOPs}{cycle} \\ &= 2 \cdot 3.3\text{GHz} \cdot 8 = 52.8\text{GFLOP/s} \end{aligned} \tag{1}$$

3 Structural Optimisations

3.1 Fission and Unrolling of Loops 0, 2 and 3

Given how loop 1 dwarfed the other loops, loop 1 was entirely removed when profiling the other loops. The cache profile for 10000 bodies and 1000 time steps indicated (1295.5 ± 236.1)ms of the samples was attributed to cache misses in loops 0, 2 and 3. Of these samples $(40.8 \pm 2.1)\%$ originated in loop 0. Additionally, only as a reference, (11704.41 ± 64.34)GFLOP/s were reported.

In order to improve the locality, loop fission was used to separate operations on the arrays into three loops instead of one. This resulted in a reduction to only (785.0 ± 284.9)ms of the samples being attributed to cache misses in loops 0, 2 and 3. Furthermore the reported GFLOP/s increased to (12408.65 ± 102.21)GFLOP/s.

Unrolling was performed primarily to allow for the use of vectorisation, as loop fission already greatly improved locality. However after unrolling only (695.3 ± 234)ms of the samples were attributed to cache misses

in loops 0, 2 and 3 and the reported GFLOP/s slightly increased to (12510.09 ± 69.51)GFLOP/s. Overall this makes for an approximate 6.9% increase over the original code.

Merging loops 2 and 3 resulted in a performance decrease at this point due to additional operations being introduced due to the if-statement and was thus reverted.

3.2 Restructuring and Unrolling Loop 1

Due to the nature of loop 1, loop fission was not possible as interim results are dependent on arrays relating to x, y and z simultaneously. In preparation for vectorisation, x[i], y[i] and z[i] were referenced in the outer loop with new variables xi, yi and zi that were then used in the inner loop of loop 1. Likewise temporary sums sx, sy and sz (initialised to 0 in the outer loop) were introduced to store the sum in place of ax[i], ay[i] and az[i]. The appropriate value in the latter arrays is set after the inner loop completed its run for a given value of i.

Without the foresight of introducing vectorisation, an entire iteration of the of the inner loop could be peeled out and the value of ax[i], ay[i] and az[i] set to 0. However, using the new variables sx, sy and sz, the peeling becomes unnecessary. An additional benefit of this change was the elimination of loop 0, which became redundant as the temporary sums are initialised to 0. Given that the profiling from subsection 3.1 showed that $(40.8 \pm 2.1)\%$ of the samples without loop 1 were originating from loop 0, this turns out to be an added benefit.

Finally the inner loop was also unrolled, both to improve locality and prepare for vectorisation.

When cache profiling 10000 bodies over 10 timesteps two quantities were specifically observed - samples relating to calculating "s" in loop 1 and samples relating to other operations in the loop, including calculating r2, y2 and z2. For an additional comparison, the total samples and the reported GFLOP/s are also shown.

	Before	After
total	(11.0 ± 1.2)s	(8.1 ± 0.6)s
calc. s	$(58.4 \pm 7.0)\%$	$(72.5 \pm 7.5)\%$
other	$(39.7 \pm 7.7)\%$	$(28.5 \pm 3.5)\%$
GFLOP/s	1.41375 ± 0.00679	1.46362 ± 0.00225

Further attempts were made to reduce the cache misses by introducing loop blocking, which confirmed suspicions that loop blocking would not improve the cache miss ratio and that the added overhead would slowed down the program.

With high latency operations such as division and the square root, pipelining was delayed until vectorisation and other optimisations were completed.

4 Vectorisation Intrinsics - SSE2

The next step to introduce vectorisation by using intrinsics. During this step most of the program was rewritten with a 1-to-1 mapping using vectorisation to preserve accuracy in order to test for correct results. Immediately following this step, several additional optimisations were made, some of which compromised accuracy in favour of performance. These optimisations are discussed in this section.

4.1 Optimisations

4.1.1 Merging Loops 3 and 4

In order to reduce memory loads and stores, and to improve locality, loops 3 and 4 were fused, resulting in improved performance; overall time spent in loops 3 and 4 went down from (0.003627 ± 0.000120)s to (0.003036 ± 0.000013)s when simulating 10000 bodies over 100 timesteps.

4.1.2 Reciprocal of the Square Root

Calculating the reciprocal of the square root can be done using the __mm_rsqrt_ps intrinsic, which introduces a maximum relative error of $1.5 \cdot 2^{-12}$. While this error is quite significant in the context of single precision floating point numbers, the performance increase is equally as significant. While the exact purpose of the simulation is not given, the error does not impact the behaviour of bodies to a degree that would make a visual inspection of the simulation reveal underlying errors. Depending on the importance of precision, one or more Newton-Raphson steps could be added to increase precision at the cost of performance [8].

4.1.3 Horizontal Addition of Vectors

In order to go from the vectorised inner loop of loop 0/1 to the outer loop of loop 0/1, the sum vectors need to be horizontally added. This changes the order of floating point addition operations and introduces a deviation from the original source code. Addition of floating point numbers performs an alignment step [5], which, when adding numbers of larger and smaller magnitude, will cause the digits of the smaller magnitude to be lost. This could potentially lead to the final horizontal addition of vectors for each iteration of the outer loop introducing large relative errors, which would carry over to the calculations performed in loop 3/4. However by choosing to perform this step, the number of load and store operations on the acceleration array are practically quartered, resulting in much better performance. Additionally, while the relative errors of this optimisation might be large, the absolute errors are small, and as all the acceleration values, large and small, are multiplied by the same factor, lessening the impact of these errors.

4.1.4 Pre-calculating Factor

In order to retuce the number of operations in loop 3/4, a new constant can be introduced to replace the multiplication of dt and dmp. This changes the order of floating point multiplication operations and introduces a deviation from the original source code. While this

change introduces an error, it should be taken into consideration that the acceleration value is already subject to a much larger relative error from the reciprocal of the square root. The relative error is small because multiplication is done before rounding [6] and the errors arising in multiplication stem just from this rounding and not any alignment steps [5].

4.1.5 Removing Branching

In the now merged loop 3/4 vectorisation intrinsics are used to substitute branching with logical operations. This does not impact precision, but results in significantly faster code as the removal of branching is considered useful when branching mispredictions happen [7]. Additionally removal of branching allowed for vectorisation and resulted in a significant overall the performance gain.

4.2 Comparison with Original Code

Vectorisation and the optimisations included with it produced the most significant step in the overall increase in performance, as the performance reached (20.074729 ± 0.083412)GFLOP/s simulating 10000 bodies over 100 timesteps. At this stage the code reached roughly 38% CPU utilisation and achieved a 13x improvement over the original code, which ran at (1.525433 ± 0.025433)GFLOP/s.

5 Pipelining

Due to the tightly packed multiplications and the reciprocal of the square root inside the inner loop of the loop 0/1 the source code suggested that a lot of the operations were stalling for 4 cycles, which was confirmed when the assembly code was inspected in Zoom. A four and two stage pipeline were implemented. Compared to the (20.074729 ± 0.083412)GFLOP/s version with no pipelining, the four stage pipeline had significantly less stalls, but the overall performance topped at $(17.056500 \pm 0.0616870)$GFLOP/s simulating 10000 bodies over 100 timesteps. Under the same conditions the two stage pipeline achieved performance almost comparable to the version with no pipeline, reaching $(20.020777 \pm 0.0584386)$GFLOP/s.

At this point, any gains from pipelining were probably being offset by caching issues. The version with no pipelining was kept and the other two discarded. Furthermore, at this point, the decision to not merge loop 1 with loop 3/4 was made based on the fact that caching was becoming an issue in loop 1, and that loop 3/4 already ran approximately 1700 times faster when simulating 10000 bodies over 100 timesteps. Any potential gain from discarding three loads per outer iteration would be offset by having worse locality and no vectorisation.

6 Threading - OpenMP

The final step to optimising the code was introducing threading. Using OpenMP to parallelise loop 3/4 unfortunately caused more negative overhead than positive speed-up, resulting in a significant slowdown when simulating 10000 bodies over 100 timesteps. On the other hand using OpenMP to parallelise the inner loop 1 resulted in a significant performance increase, bringing the final peak GFLOP/s to (38.976402 ± 0.011282)GFLOP/s. This brings the final version to a 25x improvement over the original code, and to 73.8% CPU utilisation.

7 Conclusion

While the initial structural changes to the code did not result in a significant performance increase, they set the groundwork for later optimisations. Despite being unable to address the caching issues within loop 1, vectorisation and faster mathematical calculations resulted in the most significant performance increase. Pipelining was unsuccessful on a Sandy Bridge CPU, but might be more feasible on a different architecture with better latencies. Finally threading resulted in a considerable increase in performance, bringing the final solution to a 25x improvement.

References

[1] Intel Corporation. "Intel® Core™ i7 2640M Processor." Internet: http://ark.intel.com/products/53464/Intel-Core-i7-2640M-Processor-4M-Cache-up-to-3_50-GHz, Unknown. [Mar. 9, 2015].

[2] M.R. Fernandez, "High Performance Computing." Internet: http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2329, Nov. 10, 2011. [Mar. 10, 2015].

[3] Intel Corporation. "Optimizing Performance with Intel® Advanced Vector Extensions." Internet: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf, Sep., 2014. [Mar. 10, 2015].

[4] Intel Corporation. "Intel Intrinsics Guide." Internet: https://software.intel.com/sites/landingpage/IntrinsicsGuide/, Feb. 20, 2015. [Feb. 24, 2015].

[5] M. Shaaban. "Floating Point Arithmetic Using The IEEE 754 Standard Revisited." Internet: http://meseec.ce.rit.edu/eecc250-winter99/250-1-27-2000.pdf, Jan. 27, 2000. [Mar. 9, 2015].

[6] IEEE Computer Society. "IEEE Standard for Floating-Point Arithmetic." Internet: http://www.math.fsu.edu/~gallivan/courses/FCM1/IEEE-fpstandard-2008.pdf.gz, Aug. 29, 2008. [Mar. 9, 2015].

[7] Intel Corporation. "Avoiding the Cost of Branch Misprediction." Internet:

https://software.intel.com/en-us/articles/avoiding-the-cost-of-branch-misprediction, Feb. 20, 2009. [Mar. 9, 2015].

[8] T. Berger-Perrin. "sse_approx.h." Internet: https://nume.googlecode.com/svn/trunk/fosh/src/sse_approx.h, May 16, 2009. [Feb. 26, 2015].