CS257 Advanced Computer Architecture Coursework Report

U1504815

November 21, 2017

Introduction

The purpose of this document is to present the optimisations performed on the provided star simulation program, and to analyse their effects on performance and result accuracy. The coursework was completed in its entirety on a Dell Inspiron 15R-5520 which has an Intel i7-3612QM processor, is running Ubuntu v16.04, and has a 67.2 GFLOPs theoretical peak performance.

General Approach

The coursework task was completed by performing three types of optimisations: loop optimisations, vectorisation (using SSE3), and threading (using OpenMP).

The first stage consisted of loop optimisations performed in an exhaustive manner: for every loop, all compatible techniques were tried, and those that improved performance were kept. Vectorisation was then applied on the resulting code and adjusted so that it would perform optimally, i.e. without redundant operations. Finally, threading was applied as well, but this step required changes in the previously-obtained code, as previous optimisations made the code more difficult to parallelise. It should be noted that the final two methods led to a considerable performance boost, but also led to a small precision loss, which will be discussed in detail later on.

Apart from the last test, the input values for the tests used throughout the optimisation process have been chosen based on two criteria: (1) the program would not run smoothly on them (so that the effects of optimisations would be more visible), and (2) the program would finish execution in a reasonable amount of time, i.e. approximately 30 seconds (so that waiting time would remain low). Here is a list of the tests used, ordered by input size (all with visualisation enabled and 300 timesteps):

- **Test 1: 3000 stars.** This test was demanding enough for the Gold version, so the effects of loop optimisations would be clearly visible.
- **Test 2: 5000 stars.** This test was an ideal candidate for vectorisation testing, since Test 1 was too easy for the vectorised code, but this one was never made to run smoothly (60fps) even with vectors.
- **Test 3: 7000 stars.** This test was used to analyse the effects of threading, for reasons similar to those from Test 2.
- **Test 4: 10000 stars.** This test was used to analyse the effects of smaller, fine-tuning optimisations which appeared to have no effect on easier tests.
- **Test 5: 20000 stars.** This test was used to compare the final version of the program with the Gold version, in order to compare how much it would take both versions to run very large tests, and how much the results would differ, in order to check whether the precision loss in the final version was justified by the faster runtime.

Stage 1: Loop Optimisations

For each loop, all possible optimisation techniques were attempted, and those that maximised performance were kept.

In Loop 0, only fission proved to be useful, since it exploits spatial locality by dividing the loop into three loops and allowing the program to focus on one array at a time. Unrolling was also attempted, but it did not have any effect.

Loop 1 had the most optimisation options, as it was the most complex. Unrolling and interchanging didn't have any noticeable effect on performance. Pipelining, although more difficult to implement, gave the best results for this loop, namely a 1.1x speedup. It was possible to pipeline this loop because the inner loop operations can be grouped in six steps, each of which is dependent on the previous. Blocking was also implemented, since the nested for loops and the way the rx, ry, and rz values are computed make Loop 1 similar to a matrix traversal, and thus a candidate for blocking, but it did not offer any performance boost, and so it was removed.

Loop 2 was fissioned in a similar style to Loop 0, but this did not offer any performance boost. Neither did unrolling it, and pipelining this loop proved to be even worse for performance due to storage overheads. Similarly, Loop 3 was fissioned and unrolled, but with no improvement. Eventually, the resulting loops from Loop 2 and Loop 3 were fused to further exploit spatial locality, as both Loop 2 and Loop 3 were split in three loops, each performing operations on variables related to a single coordinate axis, so it was natural to fuse corresponding loops. While at this stage the fusion and fissions did not provide any speedup, they would prove useful in the next optimisation step.

Stage 2: Vectorisation

For most of the code, vectorisation was applied in the same way: loops would be unrolled by a factor of 4, variables inside those loops would be replaced with vectors, and operations would be vectorised. Some constant vectors had to be initialised in the beginning of the function due to frequent use, such as a vector filled with 1's or a vector filled with the *eps* constant. The fusion of Loop 2 and 3 proved to be the most difficult to vectorise, since an *if* statement had to be vectorised, giving rise to overheads from bit operations that would have seemingly hindered performance. However, this was not the case.

Vectorisation greatly increased the performance of each loop, leading to a speedup of approximately 5.0x. The most important factor in this performance boost was the usage of the reverse square root vector function in Loop 1, which, although slightly inaccurate (results were usually 3% larger than they should have been), allowed much larger tests to be run in a reasonable amount of time, which previously would have taken too much.

Lastly, small vectorisation adjustments were made in order to limit the usage of load/store vector operations, leading to a total speedup of approximately 6.0x. One such example was the case of loading Loop 1 vectors with values dependent on the outer loop counter inside the inner loop. These values would be loaded for each iteration of the inner loop, even though they were always the same and they were not modified. Thus, these load operations were moved outside the inner loop, leading to a significant performance boost.

Stage 3: Parallelisation

In order to parallelise the program, Loop 1 had to be replaced with a vectorised version of the original, unpipelined loop, as the pipeline proved to be too difficult to be parallelised without too many overheads. Even if the unpipelined version of Loop 1 was slower, it allowed for excellent parallelisation of both the inner and the outer loop, by sharing loop iterations between threads and avoiding false sharing and race conditions using the *reduction* keyword. This led to a further speedup, the program being almost 20 times faster than the Gold version.

To decide how to parallelise Loop 1, the *pragma omp* keywords *for* and *schedule(dynamic)* were used. This was chosen following a trial & error approach which checked how the performance would change according to the schedule type and chunk size used.

Loop 0 and the Loop 2 & 3 fusion have been parallelised as well, either by distributing loop chunks between threads or distributing the loops themselves between threads (e.g. each loop resulting from fission would be handled by a single thread), but due to their fast execution time, the overheads from parallelisation proved to outweigh the threading speedup, making the loops between 10 and 100 times slower, depending on *schedule*. As such, parallelisation was reserved only for Loop 1.

The final version of the program, which includes parallelisation and vectorisation, has a better precision than the Stage 2 version, although the reason for this is unclear, as is the case with many floating-point operations (Muller et al.). This is likely the case that operations are performed in a different order due to parallelisation, even though results are always consistent for the same test (meaning that operations are always performed in the same order, which is unlikely for threaded code).

Evaluation

The table below presents the results obtained for each version of the program on each test mentioned above. The final version is approximately 20 times faster than the Gold version, but its answer is roughly 1.80% off. This is, however, a small trade-off, given that the threaded version is able to compute in a matter of minutes very good approximations of results that would take hours or days for the Gold or Stage 1 versions to compute, given the fast growth of execution time based on input size.

Test #	Metrics	Gold	Stage 1	Stage 2	Stage 3
Test 1	Execution Time (s)	34.002680	30.783817	5.942637	2.113488
	GFLOP/s	1.587581	1.753584	9.083846	25.541667
	GB/s	0.001059	0.001169	0.006058	0.017033
	Answer	715139072.000000	715139072.000000	741190976.000000	713050880.000000
Test 2	Execution Time (s)	94.675804	85.159986	15.805872	4.911194
	GFLOP/s	1.584037	1.761038	9.488246	30.536362
	GB/s	0.000634	0.000705	0.003796	0.012217
	Answer	1849400064.000000	1849400064.000000	1832903680.000000	1867559424.000000
Test 3	Execution Time (s)	186.220973	168.263077	31.314197	9.136557
	GFLOP/s	1.578544	1.747014	9.387372	32.173827
	GB/s	0.000451	0.000499	0.002682	0.009194
	Answer	3341301760.000000	3341301760.000000	3174920448.000000	3262717696.000000
Test 4	Execution Time (s)	377.038181	344.156612	63.984903	17.970899
	GFLOP/s	1.591192	1.743218	9.376274	33.383991
	GB/s	0.000318	0.000349	0.001875	0.006677
	Answer	1387696896.000000	1387696896.000000	1314193792.000000	1320038656.000000
Test 5	Execution Time (s)	1515.013603	1374.262534	266.872310	70.510959
	GFLOP/s	1.584065	1.746304	8.992615	34.035560
	GB/s	0.000158	0.000175	0.000899	0.003404
	Answer	787536960.000000	787536960.000000	797845888.000000	790289664.000000
Average Speedup (compared to Gold)		1.00x	1.10x	5.84x	19.64x
Average Error (compared to Gold)		0%	0%	3.22%	1.77%

Conclusion

The final version of the program successfully made use of all three optimisation techniques to improve performance and maximise CPU usage. While the approach of applying one type of optimisation at a time and building on the resulting code proved unsuitable when parallelisation was implemented, it nevertheless offered a clear way of analysing performance changes and keeping track of the remaining optimisation options.

The program could have benefited from a finer-tuned multithreading, and a successfully parallelised pipeline for Loop 1 would have further improved performance, but due to time constraints and the difficulty of these tasks, these optimisations were not attempted. Still, a level of 50.65% of peak performance and a 20x speedup have been achieved without the usage of more powerful vectorisation extensions, such as AVX, which could have potentially doubled the effects of the SSE vectorisation (Lomont).

References

1. Muller, Jean-Michel et al. Handbook Of Floating-Point Arithmetic. Boston: Birkhauser Boston, 2010, chapter 1.3.

2. Lomont, Chris. "Introduction To Intel Advanced Vector Extensions". obpm.org/download/Intro_to_Intel_AVX.pdf Web. 12 Mar. 2017.