CS257: Advanced Computer Architecture

Code Optimisation Coursework - Report

u1617935

March 14, 2018

Introduction

The purpose of this report is to document, justify and explain changes made to scientific code designed to simulate the gravitational forces produced and experienced by a collection of N given stars. The provided code is separated into four separate loops, each of which varies in purpose, complexity and therefore run-time; each loop will require different optimisations based on these factors. The optimisations are designed to increase performance of the program by making use of extensive established approaches from utilising hardware in a more effective manner such as multi-threading in order to increase CPU utilisation, to algorithmic approaches which encompass changing how the code is structured and calculates answers in order to reduce it's complexity.

Testing equipment

The performance of the code can be measured in either time to complete or number of instructions executed per unit time (Gflop/s) as they are proportional to each other. As such, it is important to specify the architecture used to test the scientific code. The system used for testing and developing this code was a DCS machine with the following specification:

- Model name : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
- Cache size : 6144 KB
- Cpu cores : 4

Peak GFLOPs = Clock Speed (GHz) * Number of Cores * (FLOPs/Cycle) 435 GFLOPs = 3.4GHz*4*32

Document structure & Terminology

The report approaches the optimisation of the provided code by first establishing and explaining the relevant optimisations that were considered, and then moves to an implementation section whereby the effectiveness of these optimisations is discussed, in conjunction to explanations for the change in performance. Finally, the final version of the optimised code is tested and evaluated.

The following terminology will be used throughout the document:

- **Speed-up** - Total optimised runtime/Total unoptimised run time
- **Percentage error** - $(1 - (Unoptimised\ answer\ /\ Optimised\ answer)) * 100$
- **Performance** - GFLOPs of code

Planning and Design

Before any changes were made to the code, it was important to consider a wide range of optimisation techniques applicable to the given code. This section contains the optimisations considered for implementation, detailing how the optimisations aim to increase performance, as well as any necessary prerequisites for implementation.

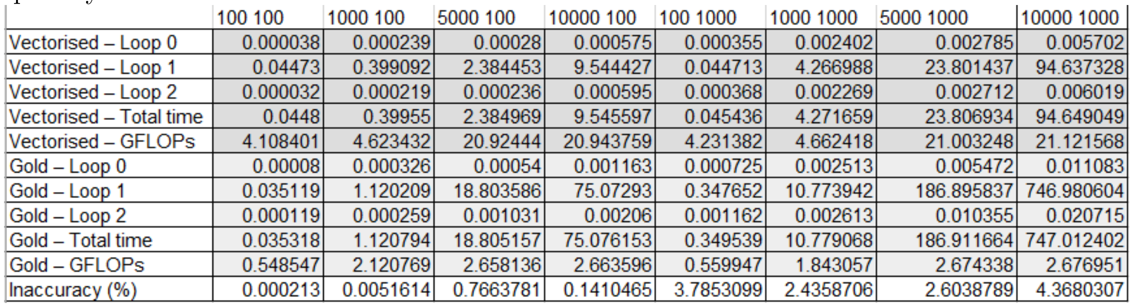
- **Vectorisation** - This optimisation makes use of code that falls under Single Instruction Multiple Data, as defined by Flynn's taxonomy. Code that satisfies this SIMD classification is able to exploit specific hardware dedicated to executing a single operation over multiple data simultaneously. There exists a wide range of SIMD implementations; for this project SSE was the implementation of choice. In practise, vectorisation is used to optimise loops, allowing multiple iterations to be executed simultaneously, opposed to sequentially. For vectorisation to be applied, the loop must be unrolled.
- **Multithreading** - This enables parallel execution of designated code across multiple threads of execution. The aim of this is to increase CPU utilisation by generating, ideally, a thread per core such that there are no idle cores, thereby increasing speed of execution. Two main implementations are available for this optimisation, pThreads and openMP. It was decided to use openMP, for the ease of use, despite it's larger overhead when compared to pThreads.
- **Loop fission** - A trivially simple optimisation which involves separating a loop performing unrelated operations into multiple smaller loops dedicated to operations on specific related data. This increases locality, as it prevents unrelated data from being loaded into and swapped out of cache, thereby increasing cache hit rate and performance.
- **Loop interchange** - A simple process which aims to increase cache utilisation by interchanging the loops in order to change the memory access pattern to improve locality and thereby cache hit rates.
- **Loop blocking** - A more complex optimisation which aims to alter the access pattern of nested for loops. The loops are restricted from iterating across their entire range in one go, and instead do so in 'blocks'. This aims to increase cache utilisation from reducing the number of cache swaps needed within each block.
- **Algorithmic optimisation** - For this problem there exists an efficient algorithm known as Barnes-Hut [1] which is able to reduce the complexity of the problem from $O(n^2)$ to $O(nlog(n))$, such an optimisation would require the use of an octree to split the simulation volume into cubic cells.

Implementing optimisations

The test results for threading and fission are supplied in the report folder, and for conservation of space have not been included in this document. The reader is encouraged to use this as an aid when reading the following section.

- **Vectorisation** - This was applied to loop 0, 1 and 2. It was attempted to be implemented in loop 3, however with a lack of knowledge regarding branch instructions in the SSE instruction set, it was unable to be implemented and as such isn't considered in the testing for this section. From figure 2, it is clear that the vectorised approach speeds up all three loops for all presented combinations of stars and time-steps except one, loop 0 and 1 for 100 100, which is approximately five and 1.25 times slower respectively. an explanation for this is due to the fact that for such a small problem size, the overhead associated with initialising the vector registers is slower than simply iterating through the

array sequentially. However, this performance hit is so minimal it accounts for 0.000007% of the total run time of these three loops.
Looking at the data, a pattern emerges whereby the speed-up gained grows as the number of stars does, plateauing as the number of stars grows larger than 5000. From 100 to 1000 stars at 100 timesteps the speed-up is as follows: 0.714, 2.805, 7.885, 7,86. One explanation for this is that in the initial stages, the overhead of the vector registers plays a more significant role, but as the workload grows, the overhead becomes less and less of a factor, meaning the full performance boost from vectorisation can be achieved. One noticeable difference between data from tests of this optimisation and the data of the others is that vectorisation has introduced inaccuracy, albeit relatively minimal. The reason for this is due to the use of the rqrt instruction, which is the quicker, but less accurate alternative to two separate, inverse and square root instructions. This minimal accuracy penalty for an increase in performance is justifiable, as the accuracy penalty is minimal for all values of N.



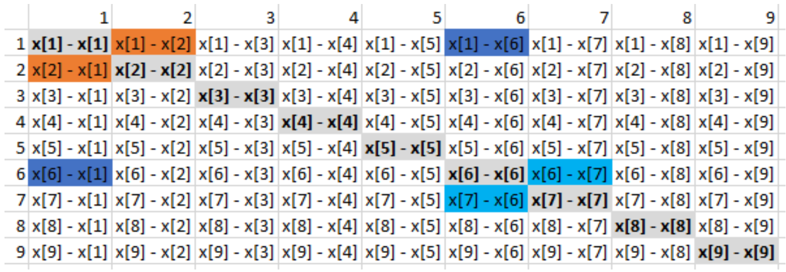
	100 100	1000 100	5000 100	10000 100	100 1000	1000 1000	5000 1000	10000 1000
Vectorised – Loop 0	0.000038	0.000239	0.00028	0.000575	0.000355	0.002402	0.002785	0.005702
Vectorised – Loop 1	0.04473	0.399092	2.384453	9.544427	0.044713	4.266988	23.801437	94.637328
Vectorised – Loop 2	0.000032	0.000219	0.000236	0.000595	0.000368	0.002269	0.002712	0.006019
Vectorised – Total time	0.0448	0.39955	2.384969	9.545597	0.045436	4.271659	23.806934	94.649049
Vectorised – GFLOPs	4.108401	4.623432	20.92444	20.943759	4.231382	4.662418	21.003248	21.121568
Gold – Loop 0	0.00008	0.000326	0.00054	0.001163	0.000725	0.002513	0.005472	0.011083
Gold – Loop 1	0.035119	1.120209	18.803586	75.07293	0.347652	10.773942	186.895837	746.980604
Gold – Loop 2	0.000119	0.000259	0.001031	0.00206	0.001162	0.002613	0.010355	0.020715
Gold – Total time	0.035318	1.120794	18.805157	75.076153	0.349539	10.779068	186.911664	747.012402
Gold – GFLOPs	0.548547	2.120769	2.658136	2.663596	0.559947	1.843057	2.674338	2.676951
Inaccuracy (%)	0.000213	0.0051614	0.7663781	0.1410465	3.7853099	2.4358706	2.6038789	4.3680307

Figure 1: Test results for vectorised approach

- **Multithreading -** This was attempted for all loops, however was only kept for loop 1. Loop 0, 2 and 3 all saw significant performance decreases ranging up to 300 times slower as a result. This is due to the overhead associated with initialising the threads. For these smaller loops, this cost wasn't mitigated by the parallel execution due to the small problem size, meaning multithreading was a signifciant handicap for these sections. Loop 1 also suffers for n = 100 when compared to the vectorised approach, bringing up an interesting choice of whether to multithread loop1 for small n. However, for larger problem sizes where the overhead is justified, parallel execution can lead to an impressive performance increase. After multi-threading loop 1, a large performance boost following a similar pattern to that of the vectorised test results is apparent. For N above 5000 the performance gain begins to plateau, indicating that a cap in terms of CPU utilisation is being hit.
- **Loop fission -** This was applied to loop 0, 2 and 3. The results from testing are inconclusive, showing marginal if any speed-up. This is mainly due to the fact that the loops this optimisation was applied to has very little impact in terms of runtime in respect to loop 1. Regardless, the principle stands that temporal locality should improve, and as a result this optimisation was left in.

- **Loop blocking, Loop Interchange -** these optimisations were applied for loop1, however it was found that both of these had no effect on runtime in a best case scenario, and a large performance hit in the majority of test cases and as such weren't included in the final optimisation.

- **Algorithmic Optimisation -** The loop operates in $O(N^2)$ complexity, however the runtime can be cut down to $(N^2)/2$. The reason for this is due to the fact that the nested for loop calculates all values for an N by N array of values between for i and j for rx, ry, rz. Figure 2 illustrates this process.



	1	2	3	4	5	6	7	8	9
1	**x[1] - x[1]**	x[1] - x[2]	x[1] - x[3]	x[1] - x[4]	x[1] - x[5]	x[1] - x[6]	x[1] - x[7]	x[1] - x[8]	x[1] - x[9]
2	x[2] - x[1]	**x[2] - x[2]**	x[2] - x[3]	x[2] - x[4]	x[2] - x[5]	x[2] - x[6]	x[2] - x[7]	x[2] - x[8]	x[2] - x[9]
3	x[3] - x[1]	x[3] - x[2]	**x[3] - x[3]**	x[3] - x[4]	x[3] - x[5]	x[3] - x[6]	x[3] - x[7]	x[3] - x[8]	x[3] - x[9]
4	x[4] - x[1]	x[4] - x[2]	x[4] - x[3]	**x[4] - x[4]**	x[4] - x[5]	x[4] - x[6]	x[4] - x[7]	x[4] - x[8]	x[4] - x[9]
5	x[5] - x[1]	x[5] - x[2]	x[5] - x[3]	x[5] - x[4]	**x[5] - x[5]**	x[5] - x[6]	x[5] - x[7]	x[5] - x[8]	x[5] - x[9]
6	x[6] - x[1]	x[6] - x[2]	x[6] - x[3]	x[6] - x[4]	x[6] - x[5]	**x[6] - x[6]**	x[6] - x[7]	x[6] - x[8]	x[6] - x[9]
7	x[7] - x[1]	x[7] - x[2]	x[7] - x[3]	x[7] - x[4]	x[7] - x[5]	x[7] - x[6]	**x[7] - x[7]**	x[7] - x[8]	x[7] - x[9]
8	x[8] - x[1]	x[8] - x[2]	x[8] - x[3]	x[8] - x[4]	x[8] - x[5]	x[8] - x[6]	x[8] - x[7]	**x[8] - x[8]**	x[8] - x[9]
9	x[9] - x[1]	x[9] - x[2]	x[9] - x[3]	x[9] - x[4]	x[9] - x[5]	x[9] - x[6]	x[9] - x[7]	x[9] - x[8]	**x[9] - x[9]**

Figure 2: Displaying the inefficiency of loop 1

The colour-matched fields in figure 2 represent examples of wasted calculation. As Loop 1 executes, it will recalculate values it doesn't need to, x[1] - x[2] is equal to x[2] - x[1] * -1. Therefore, if the values were to be calculated beforehand in a lower triangular matrix, assigning rx, ry and rz a value becomes picking an index in the two dimensional array; if j >= i then the value in array[j][i] would be returned, otherwise the value lies in the upper triangle of the matrix and therefore array[i][j] *-1 should be returned.
The Barnes-Hut algorithm [1] mentioned in the research section could be applied here. **Prerequisite -** Would require the stars to be divided up into cubic cells, making use of an octree to do so.
Unfortunately both of these possible approaches weren't implemented due to the complexity and time consumption associated with doing so.

In summary the following optimisations were applied in the final version:
- Loop 0 - Vectorisation (**prerequisite** - loop unrolling) + loop fission
- Loop 1 - Vectorisation (**prerequisite** - loop unrolling) + threading (for larger N)
- Loop 2 - Vectorisation (**prerequisite** - loop unrolling) + loop fission
- Loop 3 - Loop fission

The Implementation has both a threaded and un-threaded version of loop 1, due the fact that in testing for small n such as 100, threading was considerably slower approx. 0.44GFlOPs to 4.1 GFLOPs due to the overhead of creating the necessary threads. After extensive testing the point whereby the threaded and vectorised version surpassed the vectorised approach occurred around 250 stars and was thereby used as a simple check to determine which version of loop 1 would run.

Performance Analysis

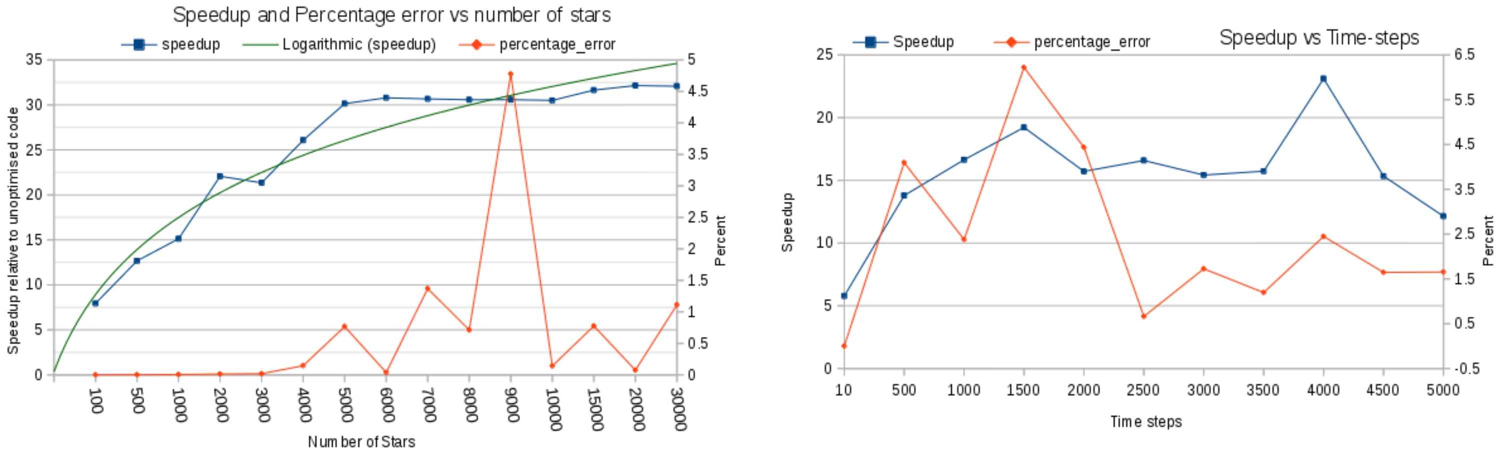


Figure 3: Displaying charts created from data in testing.ods

Testing methodology

The data gathered for the above graphs were averages of 5 runs on each given test point in order to mitigate the effect of anomalous data. The tests were run solely on one DCS machine in order keep variables which can affect the results obtained as constant as possible.
From the data displayed above, it is apparent that the optimisations have provided significant speed-up. Looking at the speed-up vs number of stars, the speed-up grows quickly then becomes asymptotic. The reason for this is as the number of stars increases, the heavier the workload on loop 1. As a result, the higher the utlilisation of the CPU and vector registers. For small N there are idle cores, which is why for N $<$ approx 4000 there is rapid growth, because more of the cores and thereby cpu is being used. However, for N $>$ approx. 4000 the cores are already near maximum load and there is little room for any extra performance benefit. This is shown more clearly by the green, logarithmic line of best fit drawn on the graph.
Speedup against time-steps is a little less clear cut. The speed-up remains relatively consistent for time steps larger than 500, indicating that the time steps have little effect on the performance of the code.

In both graphs the percentage error is also displayed, the error stayed under 1.5 percent for the number of stars vs speed-up graph, except one entry peaking near 5%. Regardless, the average error percentage of approximately 1.5% is a small price for a speed-up which sits consistently above 30x for N $>$ 5000.
Error for the speedup vs Time-steps graph is more consistent, but averages higher. Again however, the average error is approximately 3%, which is a small inaccuracy for an albeit lesser but still respectable consistent 15x speedup.

Conclusion

A lack of time to implement all possible optimisations acted as a bottleneck for this project, and as a result, although performance increase seen from the optimisations covered in this document is respectable, it leaves a lot of headroom for further improvements. Future improvements would consists of a focus on algorithmic optimisations, loop 1 still handles the vast majority of the work, making it's complexity vital to the optimisation of the code. Implementing the aforementioned Barnes-Hut algorithm [1] would be a top priority in future development, in order to change the n^2 complexity to $nlog(n)$. The program would see a significant improvement not only in speed-up, but also in the ability of the code to scale as N grows, making this optimisation a crucial part in further development.
With more time, a more detailed and in-depth approach to implementing loop blocking would be used, in an attempt to increase the cache utilisation and locality of loop 1. In addition to this, pipelining would be looked into in greater depth to attempt to increase the throughput of operations per unit time and therefore performance.
In summary, the optimisations made provide a solid and useful foundation to build off of. They provide a consistent and significant boost in performance for what is a minimal and justifiable cost in accuracy. With more time to implement the aforementioned optimisations, the program would see an even larger speed-up and better scaling capabilities, with a particular focus on algorithmic optimisation techniques such as the Barnes-Hut algorithm [1].

References

[1] TOM VENTIMIGLIA & KEVIN WAYNE, http://arborjs.org/docs/barnes-hut