CS257: Advanced Computer Architecture
Code optimisation coursework assignment

U1721304

Introduction
This report will document the planning, implementation and analysis of an attempt at optimising a simulation for N number of stars lasting T timesteps. The planning section will explain how each optimisation technique that will be attempted works at a high level. The implementation section will explain the specifics of how each optimisation technique works in this specific project. It will also detail any difficulties faced and changes from the original plan. Finally, there is an analysis section that will take data provided by bash scripts to determine the performance of the optimisation techniques detailed in the implementation section.

Equipment description
Specs for the machine used:

- Dual code 3.3 GHz Intel Core i5 3.3 GHz
- 2 instructions per cycle
- 4 SSE Units

Theoretical peak = 3.3 * 2 * 4 * 2 = 52.8 GFLOP/s

Optimization plan
Loop interchange
Normal nested loop access has poor spatial locality as multi-dimensional data structures are stored sequentially in memory row after row. Doing column first memory access will lead to much better spatial locality as the data structure will now be accessed in the sequential order its stored in memory. This improved spatial locality will lead to more cache hits, increasing the cache hit to miss ratio, and reducing the number of cache reads and writes. This reduced overhead leads to more optimized performance in nested loops.

Loop unrolling
Loop unrolling performs multiple iterations of a loop, which doesn't have inter-loop dependencies, in one iteration. This reduces the loop overhead and improves program speed leading to more optimized loop execution. This does come at a sacrifice of larger program binary size, but the main goal is performance optimisation, so this isn't an issue.

Vectorisation
Vectorisation uses registers called vectors which use special hardware that can perform SIMD operations on the data stored in them. These SIMD operations perform the same behaviours which are often implemented as loops, but in a more efficient manner. An example of vectorizable behaviour is adding elements of two arrays and placing them in a third array. Replacing traditional loop implementations of behaviours that can be vectorised leads to more efficient code, thus leading to optimisation.

Threading (to implement parallelisation)
Threading is an optimization technique in multicore CPUs that enables blocks of code to be run simultaneously, over the multiple cores. This increases the number of instructions run per cycle and reduces the time to complete parallelised blocks of code, leading to optimisation.

Loop fission
Loop fission breaks down a loop with multiple unrelated operations into separate loops. This improves the spatial locality of memory access within each loop which increases the cache hit to miss ratio and leads to optimisation.

Optimisation implementation

Loop Interchange

Initially loop interchange was used for loop 1 even though it only gave a 1.07 times speedup. However, when the other optimisations were implemented, loop interchange actually slowed down performance, hence it was discarded from the final code.

Loop fission

Loop fission was only implemented for loop 2, even though loop 1 and loop 3 met its requirements. Because of the infrequent usage of loop 3, the effects of using fission were negligible during testing. The fission implementation for Loop 0 was replaced using memset. This is because memset is heavily optimised and it can set multiple bytes at a time unlike the fission implementation, which only sets one byte at a time. As a result, it wasn’t surprising that during testing memset produced higher GFLOP/s on average.

Loop unrolling

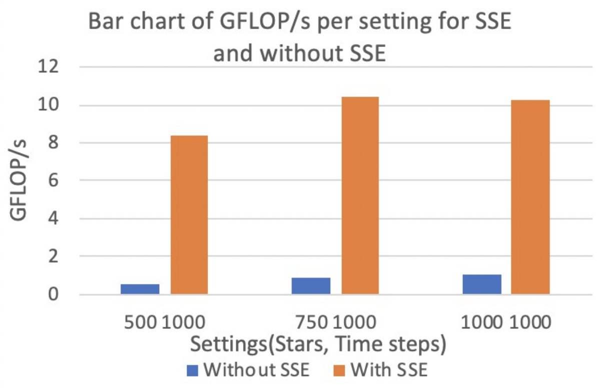
Loop unrolling was used in loop 1 and loop 2. In both loops it was used to split up data that was initially accessed cell by cell into blocks suitable for vectorisation. The value for increment is 4 because a vector in SSE is 16 bytes, and sizeof(float) is 4 bytes hence the number of floats per iteration is 16/4 = 4, meaning the increment must also be 4. Loop 3 met the requirements for loop unrolling however it had minimal impact on performance, so it was discarded.

Threading

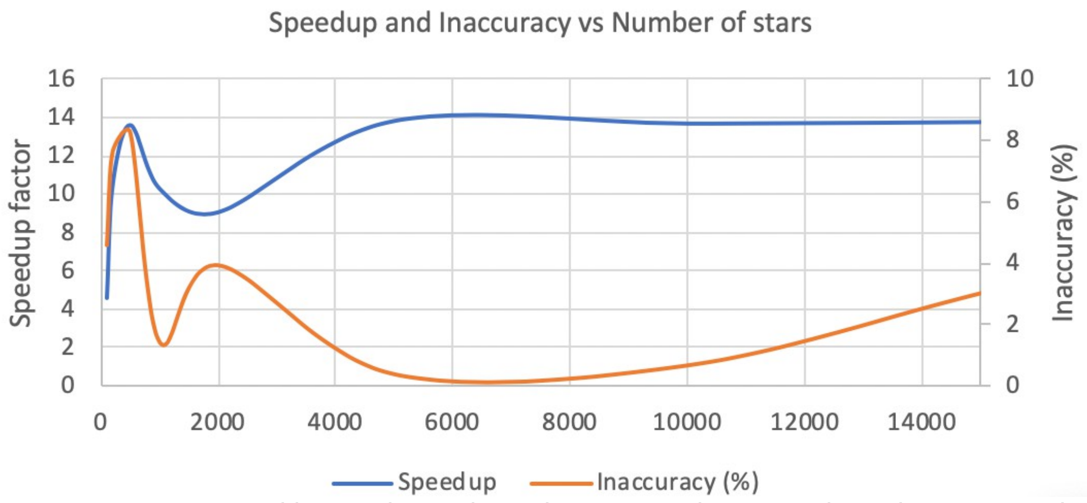
Threading was only implemented in loop 1. The overhead from creating threads combined with the less frequent use of the other loops made it inefficient to use threading elsewhere. In the choice between *pThreads* and *OpenMP*. The latter was used because it makes the compiler handle threading by using the *omp parallel* pragma. In addition, the *OpenMP* API will contain code that is written by professionals, meaning it will be far more efficient and far more thoroughly tested than code that would have been written using *pThreads*. The only downside with letting the compiler handle all of the threading was that in some cases it wouldn’t be able to generate threads properly due to incompatible user written code. However, in testing, threading worked properly in all cases, so this isn’t an issue.

Vectorisation

In the final version of the code, SSE vectorisation was used instead of the more efficient AVX. This is because time constraints prevented the conversion of SSE vectorisation into AVX vectorisation. In order to implement SSE vectorisation, loop unrolling had to be used and the operations performed on single array elements was replaced by SIMD operations across a whole block of data within the array. This was used for loop 1 and loop 2 and had a significant effect on performance as seen below.



Analysing optimisation performance



The results were collected by running 5 simulations per settings tested. Settings are a pair of numbers representing the number of stars and the number of timestamps that a simulation runs on.

Speedup analysis
The optimisations implemented have had a significant impact upon the speed up of the code as seen from the graph where a peak speed up of 14 times the original GFLOP/s is seen. The increase in speed up is significant around N < 800 but becomes constant after this point. This is most likely because with a small N, cores are idle and only a single thread is likely to be used. As more and more stars are added the effects of parallelisation become prevalent until it becomes constant when all the cores are being used, hence the plateau. The dip in performance speedup around N = 2000 is likely anomalous data. This is because only 5 runs of N = 2000 were used to take the average so it's possible a few anomalies dragged down the average to the levels seen on the graph without being caused by poor performance.

Accuracy analysis
Accuracy remains high even after optimisations as seen points of almost 0% inaccuracy. However, there is a spike in inaccuracy at around N = 500 which is also most likely due to anomalous data having a large impact as only 5 runs of each setting was used.

Conclusion
The results for optimised performance are very promising but there is still room for further performance gains in future optimisation attempts. In future attempts it's important that AVX is used instead of SSE vectorisation. This will provide further gains without much extra programming effort as AVX is extremely similar to SSE but with larger vector sizes. In addition, better test data should be collected when analysing future optimisation attempts. There are a few data points in this project that skewed data simply due to anomalies and a lack of more trial runs.