CS257 Advanced Architecture Coursework

Omkar Satish Gadhave

1806161

Introduction

The main purpose of this coursework was to carry out optimisations on the code provided using a variety of optimisation techniques such as loop fusion, loop unrolling, multi-threading etc. Majority of the optimisations were carried out in the simulation.c file. While doing the coursework a lot of optimisation techniques were used and then removed either because rather than improving on the time it slowed it down further or the optimisation made very little to no improvement on the runtime. The optimisations which actually did help in improving the runtime will be talked about as we go along the report.

Optimisation

A bunch of optimisation techniques were used to improve on the runtime. The optimization techniques are explained below function specifically.

- computeTentativeVelocity() - Loop peeling was used to merge the two loops together by modifying the loop range. The main reason for using this technique was to remove dependencies created by the earlier iterations on the remaining iterations in turn enabling parallelization. This made it possible to implement threading which further improved on the runtime. The chosen optimisations had no affect on the floating-point correctness.

- computeRhs() – Loop unrolling was used as there were no broken inter-loop dependencies. This technique reduced the number of iterations required thereby increasing the program’s speed by eliminating loop control instruction and loop test instructions. It also reduced the loop overhead which increased the program’s efficiency. Threading was also used with the intention of further improving the program’s runtime, but it made little to no improvement to its runtime. The chosen optimisations had no affect on the floating-point correctness.

- poissonSolver() – The optimization technique used to optimize this function were loop fission and threading. The code in this function uses a red-black successive over relaxation scheme to simulate the fluid flow. Red-Black SOR divides the grid into a chessboard of red and black cells as shown in Figure 1. As we can clearly see in the figure that all the black cells have the red cells as neighbor on all four sides. This red-black strategy allows us to divide the iteration into a red phase and a black phase. The strategy used in this method is such, that first the red cells are updated and then the black cells are updated using the values previously computed for the red cells. Using such a strategy allows for parallelization. (Mittal, 2014)

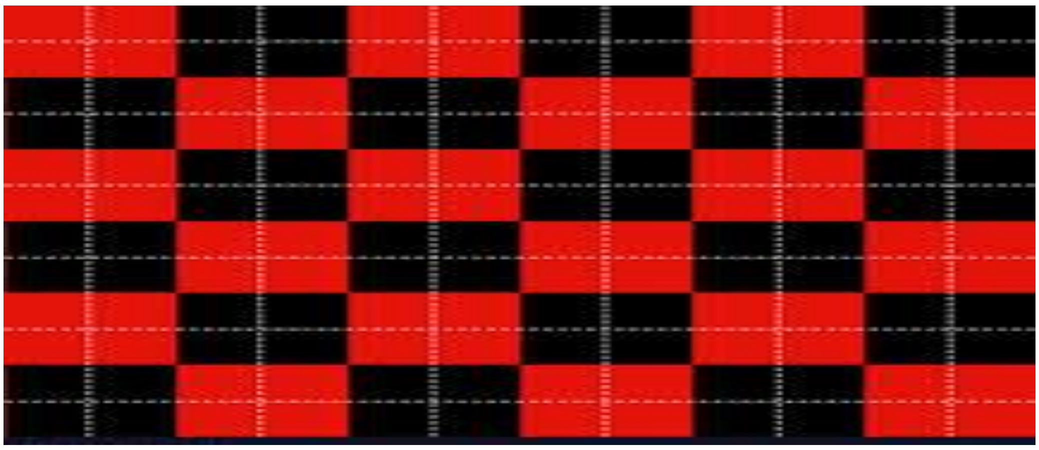

Figure 1 The chessboard look of the SOR

In poissonSolver() the for loops are structured in such a way that if-condition checking is minimised which in turn reduces branch misprediction. And since the branch conditions are essentially removed, the runtime is significantly improved. Also as there are no inter-loop dependencies multiple iterations of the loop can be executed concurrently using SIMD instructions. After carefully analysing the code already provided in this method, it was determined that a good amount of the code was useless and was removed as it was slowing down the runtime. All of this provided a substantial improvement to the runtime. Analysing the code also made it apparent that (1.-omega) was a constant value which was being calculated every time in all the for loops of this function. Taking it out of the loops and replacing it with a constant brought the runtime down from previously around 9.0s to about 8.5s. But doing this generated an incorrect output. Running diffbin on it showed that the floating point difference was actually miniscule. Even then this wasn't implemented in the end as accuracy and correctness was given a higher priority.

- updateVelocity() – The optimisations in this function were the same as the ones used in computeTentativeVelocity(). Even then it didn't really provide much of an improvement to the runtime.

- setTimesstepInterval() – Once again loop peeling and threading were used to optimize on the time. But as compared to computeTentativeVelocity() and updateVelocity() the threading used was different. For the purposes of threading the omp simd directive was applied to the loop so that multiple iterations of the loop can be executed concurrently using SIMD instructions.

- applyBoundaryConditions() – There were no optimisations implemented as couldn’t think of anything. But one way to optimize this method was to optimize the switch case statement using inline assembly. Tried to implement this but didn’t in the end as this proved to be quite difficult.

Conclusion

Overall optimisations were performed in every function apart from applyBoundaryConditions(). The main reason for not being able to implement inline assembly was the limited timeframe left. The most improvement brought to the runtime was due to the use of multithreading as compared to the loop optimisations. Although loop-optimisations didn’t really improve the runtime by much, they paved the way for parallelisation because of which multithreading proved to be this helpful. The runtime of around 8.98 seconds achieved could have been further improved by the use of vectorisation techniques. The reason for not implementing vectorisation was due to the limited understanding of the topic. In all, the limited timeframe proved to be a bottleneck for not being able to optimise the code even further which would have helped in achieving the modal runtime.

References

Mittal, S. (2014). A Study of Successive Over-relaxation Method Parallelization Over Modern HPC Languages. 6.