2024 High Performance Computing Coursework

Amin Donald
University of Bristol
Bristol, England
sq21386@bristol.ac.uk

***Abstract*— In this paper, we attempt to complete the University of Bristol 2024 High Performance Computing coursework. This entails optimising some existing code for a Lattice Boltzmann CFD simulation on the BlueCrystal Phase 4 supercomputer.**

Index terms—*High Performance Computing*

I. Serial optimisation

A. Compiler Arguments

For gcc, the important arguments we used were `-Ofast` and `-march=native`. `-Ofast` tells the compiler to use all the optimising techniques in the `-O3` argument, but also enables some techniques that, while beneficial, can cause UB in some programs (e.g `-fallow-store-data-races`). The `-march=native` argument (and similarly the `-xHost` arg for `icc`) tells the compiler to not only tune its optimisations to the specific kind of hardware we're compiling for, but to try and use platform specific instructions, such as vector instructions.

B. Loop Fusion

To begin with, the code calls 5 functions in a given timestep: `accelerate`, `propagate`, `rebound`, `collision` and `avg_velocity`. There is also a `cells` array, the main simulation space, and a `tmp_cells` array, a scratch space for calculation. These each run their own loop, so the first optimisation we did was to make this only perform one loop. After manually inlining each function and removing each's inner loop, it becomes apparent that a cell only collides or rebounds, but never both, so we made them conditional, and that the collision step already calculates the average velocity, so we just return this value from the `timestep` function. Another optimisation is the "pointer swap". This is where if, instead of writing back to the `cells` array in the collide and rebound step, we only read from `cells`, write back to `tmp_cells`, and then just swap their pointers after each timestep, we end up only mutating one array rather than two.

II. Vectorisation

A. Compiler Arguments

We used `icc` to vectorise the code, as it was well documented. Other than `-xHost` so that the compiler knows the cpu can run vector instructions, the directives to vectorise exist within the source code. Though, `-opt-report` can tell you to what extent your code was vectorised.

B. Structuse of Arrays

Instead of the previous array of structs containing cell speeds, we switched to a struct of arrays, where the first array contained all the speed in one direction, the second contained all the speeds in another direction, and so on. This provides our compiler with memory that is contiguous for one paramter (i.e. all the speed in some direction are one after the other), and considering each timestep does the same operation, this can be vectorised easily.

C. Memory alignement

To use vector instructions, the "vector" in memory must be aligned. In our case we used `icc`'s `_mm_malloc` function to ensure the cell arrays were aligned in the physical memory, and then within the timestep loop, used the `__assume_aligned` function to tell the compiler that the data is aligned for the purposes of optimisation. We also use an `__assume` function to tell the compiler that the loop literations are divisble by 64, so that is can do several iterations in one vectorised iterations, and not need to worry about having to do a partial iteration.

III. Parallelisation

A. Compiler Arguments

We used OpenMP to parallelise our code, so we need to tell `icc` we are using it with the `-qopenmp` argument.

```
...
#pragma omp parallel for collapse(2)
reduction(+ : tot_u) reduction(+ :
tot_cells)
for (int jj = 0; jj < params.ny; jj++) {
  for (int ii = 0; ii < params.nx; ii++) {
...
```

Listing 1: Excerpt from the parallelised `timestep` loop

B. "parallel for" pragma

OpenMP contains very verstatile pragmas that can, in just one or two lines, parallelise your code. In our case, on the main loop we use the pragma shown in Listing 1. `omp`, `parallel`, and `for` read literally as "parallelise this for loop", but by way of splitting all the iterations among your different threads. `collapse(2)` tells OpenMP that the for loop is actually 2 for loops, with one nested in the other, and that the pragma should turn them into just 1 for loop, so that the allocation of iterations can be more fine grained than just the outer loop. The `reduction` arguments are to do with getting a result from something that is common among all threads. None of the threads ever modify the same cell, but every thread needs to increment the number of cells and contribute to the average velocity. So `reduction(+ : tot_cells)` says "make the `tot_cells` variable local to each thread so there are no store race-conditions, but then after the loop is done, add each thread's `tot_cells` together so that we end up with one total `tot_cells` variable", and similarly for `tot_u`.

- Compare serial vs parallel? (might just be a graph idk)

the runtime is faster lmao

- Analysis of scaling from 1 core to 28

look at the graph, write words

References