COMS30005 An Introduction to High Performance Computing - Assignment 1

1. Introduction

For this assignment we were asked to optimise an already written C code to run as fast as possible on just one core. The program being optimised is the iterative Jacobi method.

For all runtime comparisons please look at the Appendix in section 5. Runtime in the tables is measured in seconds.

2. Serial code optimisation

2.1. Compilers

The first thing I did was test the program using both the GCC and the Intel Compiler (ICC) and see which one performs better. Without any flags or changes to the given code I compiled and ran the program using both compilers. By looking at the results I concluded that the ICC compiler was faster, but this was only because ICC uses the "O2" flag that performs some basic loop optimisations by default.

2.2. Starter flags

Now I switched to the GCC compiler and tried the same flag in order to check if ICC is still faster, but this wasn't the case because GCC turned out to be faster.

Afterwards I tried the more aggressive speed optimisation (-O3). This uses all the optimisations specified in "O2" and some additional ones. Performance stayed the same for both levels and GCC remained faster.

I also tried various other flags like fast, m64, unroll, qopt-prefetch, ffast-math, ipo, Ofast, no-prec-div, ansi-alias, static, but since there was no improvement I decided to use the "-O" optimisations for now and once I optimise the code try and see if any other flags help.

2.3. Profiling

In order to find the code that slows down your program I tried several profiling tools, but the one that I found most useful was VTune, because it gave me line by line profiling. After I determined where the code needs optimisation I moved on to actually optimising.

2.4. Code optimisation

The first thing I noticed in the code was that the 15 decimal places in the data type double were unnecessary for this program so I switched it to float and by lowering the decimal places to 6. I halved the storage size needed from 8 to 4 bytes. Changing this didn't give me any speed for matrices of lower size, but for bigger matrices it was quite an improvement.

The next change was what made the Jacobi program really fast. I changed the matrix from column-major order to row-major order. The programming language C unwraps 2D arrays in memory as a 1D array in row major order, but in the nested loops in the program it was done in column major order and because of this we had a non-contiguous access pattern that ng around a lot and since elements loaded are not adjacent, they will be loaded separately using multiple instructions, which is less efficient. On the other hand, in row major order it will always access the elements sequentially and it will hit the caches. In fact, hit the fastest L1 cache because the CPU correctly pre-fetches all data ahead.

After this change and probably because ICC is better optimised for Intel CPUs in using its caches it outperformed the GCC compiler by far. At this point I started using only ICC.

Afterwards I saw that the loop checking for convergence and the one performing the Jacobi iterations are doing different operations but use the same data so I figured I could fuse them together. This helps by enabling reuse. Fusing the loops only improved my code for larger matrices and only by using the "-Ofast" flag which adds floating-point optimisations to the "-O3" flag.

Lastly since we are avoiding the multiplication of the diagonal matrix with the vector using an if statement and because if statements are quite expensive so I figured I should remove it. So, I saved the values of the diagonal matrix in a new vector so they can be returned after the multiplication finishes because I need them for the solution error. Then I made the diagonal matrix equal to 0 so when multiplying with the vector and adding it to dot it would be as skipping because that value because adding 0 does not do anything. This improved performance quite a lot.

3. Vectorisation

3.1. Using restrict

The first step I took in vectorising my code is using the restrict keyword. This informs the compiler that each pointer with restrict provides exclusive access to a certain memory region meaning that the memory referenced by a pointer is not accessed in any other way.

3.2. Using pragmas

Next, I started experimenting with pragmas which are directives that provide instructions to the compiler to use pragmatic or implementation-dependent features.

The first one I tried was the unroll pragma. The compiler had already set an unroll factor of 2, but I wanted to try and see what happens when I use 4, 6, 8. By comparing the results I arrived at a conclusion that 4 was the best unroll factor.

Afterwards I tried the omp simd align pragma that transforms the loop to be executed concurrently using Single Instruction Multiple Data (SIMD) instructions and then aligns the data, but aligning the data using SIMD instructions was slower so I removed it and added the vector aligned pragma which performed better.

3.3. Data alignment

Using only pragmas for data alignment is not enough. I also needed to add in front of malloc and free "_mm_" and to add the "_assume_aligned" and "_assume_" clauses to tell the compiler that the property holds at the particular point in the program where the clause appears and that the pointer is aligned at 64 bytes and that N is a multiple of 8.

3.4. Extra flags for vectorisation

Since I used restrict in my code the flag "-restrict" also was to be added, then I added "-qopenmp-simd" to make the compiler support vectorisation and accept SIMD instructions, but this made the runtime slower so removed it. I then added "-xavx" to use the supported instructions and optimise the code for my CPU. In my case increase the vector length to 8 which improved performance.

4. Conclusion

To conclude I would like to say that all the changes I made to the code optimised the Jacobi algorithm quite a lot and now the program for a matrix of 4000x4000 runs for 39 seconds against the starting 1196. I tried a lot of optimisations from which some helped, some didn't work or some made the program slower. I am sure if there was more time I could have tried a lot more optimisations and written more stuff in the report, but given the limited time I am quite happy with the runtime of my program and the report.

5. Appendix

Code changes	unoptimised code					double to float		row-major order	
compiler	gcc no flags	icc-O2	gcc-O2	icc-O3	gcc-O3	gcc-O3	icc-O3	gcc-O3	icc-O3
1000x1000	11.051	3.248	3.201	3.549	3.205	3.312	3.412	2.690	0.588
2000x2000	131.560	52.828	51.443	59.374	51.217	21.564	21.672	19.943	4.162
4000x4000	1196.471	575.508	500.102	549.135	527.398	362.12	361.566	156.476	46.437

Code changes	loop fusion		removed If statement	vectorised code omp simd	vectorised final
Compiler	icc-O3	icc-Ofast	icc-Ofast	icc-Ofast	icc-Ofast
1000x1000	0.592	0.587	0.454	0.510	0.429
2000x2000	4.166	4.041	3.237	3.636	3.115
4000x4000	46.497	44.980	40.163	41.487	38.993

Note: Number of iterations and the solution error did not change during these tests.

References: https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays/ ,
https://software.intel.com/en-us/articles/common-vectorization-tips
https://sites.google.com/a/case.edu/hpc-upgraded-cluster/home/important-notes-for-new-users/helpful-references/code-optimization