Optimizing Serial Jacobi Code

Luke Storry (LS14172)

October 2017

Introduction

This report describes the experiments and optimization's applied, to a supplied `jacobi.c` code file, to optimize its serial runtime on the BlueCrystalPhase3 supercomputer.

Data Layout

The original data layout was Fortran-style column-first, so the first code optimisation tried was to change this to row-first accessing of the arrays. This resulted in a drastic, over two-times speed increase, from 148 to 68 seconds on a 2000x2000 array.

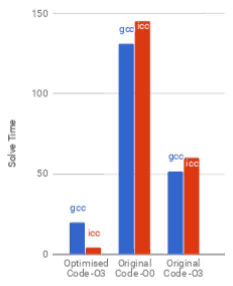
Data Type

Next was the data-type. It was found that changing the `Doubles` to `Floats` don't make the code any more erroneous, yet could result in a much faster code, especially with compiler optimization, as the much smaller overall filesize made access and memory transfer much quicker, as well as beign able to fit twice as much data in the cache.

Compiler Choice

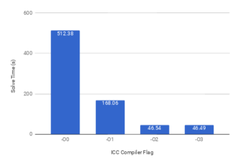
A variety of compilers were tested, but the only one that gave `gcc` a good competition was the `icc` compiler. This was also an obvious choice due to the chip architecture of the BCP3 hardware being Intel SandyBridge, the Intel compiler would have better hardware-specific optimizations available.

As you can see from this chart (running jacobi.c on a size 2000), although `icc` performed slightly worse than `gcc` with the original code, even with compiler optimizations, when the code was optimised, `icc` performed significantly better, so this was the one used in the Makefile of the final submission.



Compiler Flags

Using Intel's Compiler Optimisation Guide [1], `-O2` or `-O3` seemed like good options, but all were experimented with anyway.



As can be seen from the chart (optimized code on n=4000) , when the compiler is run with no optimization, it much slower than even the first level of `-O1` optimization. However, when the level of optimization is 2 or above, Intel's auto-vectorisation is enabled, which greatly increases the solve speed on the optimized code.

For larger input sizes, the `-O3`'s more aggressive loop-optimization made it slightly faster than the `-O2`, but only by a fraction of a millisecond.

Others

Although attempts at manual loop-fusion and vectorisation were tried, the `icc -O3` compiler's automatic optimisations resulted in much more efficient code.

Conclusion

Using a better data format and layout, and an agressive compiler optimisation, the times were brought down to:

Size	Solve Time (s)
1000	0.588
2000	4.17
4000	46.49
8000	359.60

[1] https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler