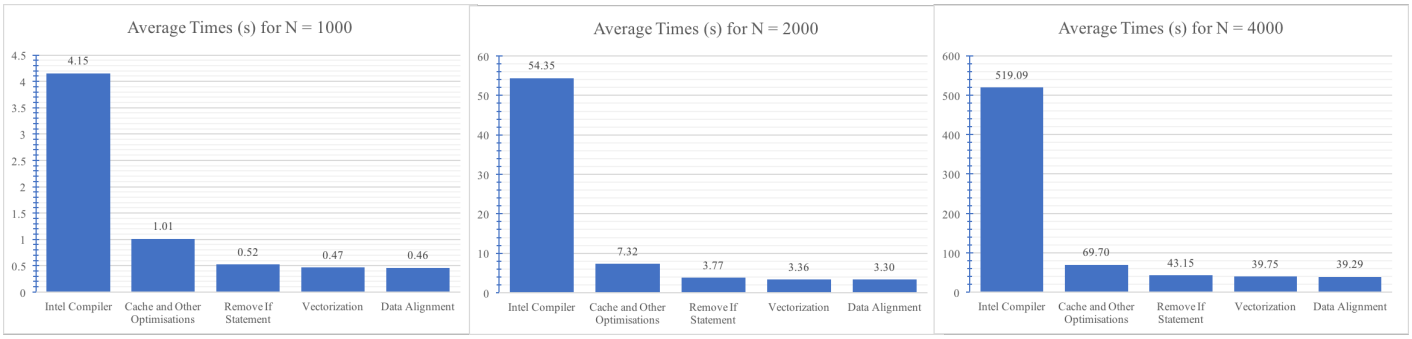
Serial Code Optimisations for the Jacobi Iterative Method

Ahmer Butt [ab15015]

1 Introduction

This report will step through a series of serial code optimisations for the given baseline implementation of the jacobi iterative method. The optimisations will target the BlueCrystal Phase 3 supercomputer and involve compiler choice and compiler flags, improving cache utilisation, logical improvements and vectorization tuning. For each, a description of the optimisations, a comparison of the unoptimised and optimised runtimes and finally an analysis of the effectiveness of the optimisation will be presented. A summary of optimisations are given below. Baseline times are 10.9s, 130s and 1180s.



2 Utilising the Intel Compiler

The initial Makefile given uses the GNU C Compiler (GCC). The target processor is Xeon E5-2670. BlueCrystalp3 offers access to the Intel C/C++ compiler (version 16u2), which can be used to apply more aggressive optimisations for an Intel target device. One can also generate more detailed compiler optimisation reports than with GCC. As well as the optimisation flag `-O3`, the flags `-no-prec-div, -xHOST, -ansi-alias` were used. The standard math library is also replaced with the Intel Math Library. The Intel Compiler also offers Profile Guided Optimisation. Applying these changes resulted in a speedup from 1180s to 519s for N=4000. `-no-prec-div` disables fully precise IEEE division, an expensive operation. `-xHOST` applies more aggressive processor specific optimisations. `-ansi-alias` lets the compiler assume that ISO C standard aliasing rules are adhered too. The remaining flags are used for generating optimisation reports. In the PGO process, the code is first instrumented such that it's execution generates a dynamic information file. Then, when next compiling, that file is used to tune further optimisations based on the run-time properties of the commonly traversed paths in the program.

3 Improving Cache Hits and Other Optimisations

Retrieving data from memory to processor is a large bottleneck. Accessing main memory is many magnitudes slower than accessing the L2 cache, which is in turn magnitudes faster than L1 cache. Therefore, reducing cache misses has a high performance benefit. There are many ways this has been improved. Overall, this resulted in a significant speedup from 519s to 70s for N=4000.

Firstly, using smaller data types so more variables can fit into the cache. Any `double`'s (64-bit) are changed to `float`'s (32-bit) as such higher precision is not required. Any and `int`'s (32-bit) which are guaranteed not to overflow have been converted to `short`'s (16-bit), which are still large enough. This will also be significant to vectorisation, which is discussed later.

Most importantly, the order of accessing array `A` has been changed from column to row major. In C, array elements are stored contiguously in row major order. However, the line `dot += A[row + col*N] * x[col]` results in accessing in column major order. Therefore, every access causes a cache miss. However, changing the code to access in row major order, using `dot += A[row*N + col] * x[col]`, results in the next element to be accessed having already been loaded in the last cache line, so there is a cache hit.

Other small optimisations have also been implemented. For example, in the convergence check loop, rather than storing the value `diff` and then calculating `sqdiff=diff*diff`, the entire value of `sqdiff` is calculated inline. This removes the need to access a register, or worse, the cache, which can take a few clock cycles. The condition for the outer do-while loop has also been changed. Instead of computing the square root on every iteration with `sqrt(sqdiff)`, the value `sqdiff` is compared to the square of the convergence threshold. Squaring is a far less expensive operation than square root.

4 Avoiding Pipeline Flushes

The Intel Xeon processors and Intel Compiler use branch prediction to improve the flow in the instruction pipeline. Without this, the pipeline would have to stall and wait for the result of the condition to pass the execution stage before the next instruction can enter the

fetch stage. The predictor tries to avoid this by guessing whether the conditional branch is taken or not and then speculatively executing the corresponding code. The prediction is based on previous results of the branch. Therefore, in the nested loop, the compiler will predict that `if (row != col)` will be `true`, as it evaluates as such most times. However, when branch prediction misses (when on a diagonal element of `A`), a costly pipeline flush must occur so the other branch can be loaded instead.

A complete solution to this problem is possible if one can remove the condition from the loop itself yet still achieve the same semantics. This has been done by splitting the matrix `A` in the matrix `R` and the matrix `D`. Matrix `R` contains the same values as `A` but the value `0.0` across the diagonal. `D` contains the diagonal of `A` and is represented using a one dimensional array. The body of the inner loop is changed to contain no condition, just `dot += R[row*N + col] * x[col]`. The remaining instruction of the outer loop is changed to `xtmp[row] = (b[row] - dot) / D[row]`. The loops to initialise the matrix and check solution error have also been updated to use `R` and `D`. This optimisation resulted in a speedup from 69.7s to 43.2s.

5 Vectorization and Data Alignment

Vectorization refers to taking advantage of SIMD (Single Instruction Multiple Data) hardware so that multiple operations can run from a single instruction. In the `run` function, this means that the array operations within the inner nested `for` loop and convergence check loop can be executed on multiple array elements at once. As the target hardware is Intel Xeon E5-2670, the compiler can produce Intel AVX (Advanced Vector Extensions) assembly to perform SIMD instructions. In previous sections, using smaller datatypes for each element in the array was discussed. Doing this means that upto twice the data can fit into a register. As a result, the number of elements that can be operated on in one vectorized loop has the potential to double. To aid the compiler in performing auto-vectorization, the programmer can tell the compiler that there is only one pointer to each array `R, D, b, x, xtmp`. The type qualifier `restrict` tells the compiler that there are no aliasing pointers. One also writes `#pragma ivdep` before the loop body, which tells the compiler that there are no write-read or other dependencies across loop iterations in the operation to vectorize. The flags `-restrict` and `-vec-threshold0` are also added. The opt-report states that there is an estimated potential speedup from vectorization of 6.900 for the nested inner loop and 5.310 for the convergence loop. However, we can improve this by utilising data alignment.

When a loop is vectorized it is split into remainder, peel and kernel loops. The remainder loop executes the remaining iterations when the trip (loop) count `N` is not a multiple of the vector length. The peel loop 'peels' some loops from kernel loop so that more arrays in the kernel loop are aligned. The kernel loop is the remaining vectorized loop. The opt-reports show that the peel and remainder loops are not as well vectorized. For example, the inner loop's speedup in remainder drops to 4.846.

The peel loop may be removed by aligning the arrays to a 64-byte boundary. It is done by replacing the call `malloc(size)` with `_mm_malloc(size, 64)` and, then, before each loop to be vectorized, informing the compiler that the data is aligned with `#pragma vector aligned nontemporal`. The `nontemporal` argument explicitly tells compiler to use streaming register stores, which can have a significant performance improvement versus nonstreaming. The run-time now drops from 0.47 to 0.46 for N=1000. This is because aligning the data results in more efficient memory movement and hence a greater potential vectorization speedup. The potential speedup of the kernel loops increase to 13.800 and 11.210. The opt-reports show that the references are always aligned and the vector lengths have also increased from eight to sixteen. Peel loops are no longer generated. Also, unmasked unit stride loads go from 2 to 4 and are always aligned, as opposed to using costly gathers and scatters. These explain the speedup.

6 Unsuccessful Optimisation Attempts

One unsuccessful optimisation attempt was to move the statement `xtmp[row] = (b[row] - dot) / D[row]` in the outer loop to its own loop. This was done by making `dot` an array in which we store the intermediate result of multiplying each row with `x`, and then use another loop to calculate `xtmp[row]`. The motivation of this was to allow the operation in the outer loop to be vectorized too. The resulting opt-report shows that the compiler moved two load/stores out of the inner loop. The scalar and vector loop costs halved, with a similar potential speedup of 13.330 maintained. However, the solver run-time increases to 0.50s for N=1000, 3.62s for N=2000 and 42.7s for N=4000. The probable reason for this is the overhead introduced by storing and accessing the new `dot` array. Even though each access of each array element should be a cache hit, previously, the float `dot` may have been stored in a register. Accessing the L1 cache still takes multiple clock cycles, whereas accessing a register may be non blocking.

Loop fusion was also attempted. The new loop can be fused with the convergence check loop by the compiler. However, the slower times persist. Loop fusion was not beneficial because of the way it interacts with vectorization. Fusing loops causes the cost of the loop body to increase. A more complex body is also harder to vectorize. As a result, the scalar loop cost increases to 55. The vector loop cost increases at a rate of even greater magnitude to 6.250. Hence, the estimated potential speedup from vectorization drops to 8.660. As the trip count N is large, it appears that the benefit gained from fusing loops to increase cache hits is outweighed by the loss in vectorization.

7 Conclusion

In this report, optimisations for a serial implementation of the Jacobi iterative method, targeted for the BlueCrystal Phase 3 supercomputer, were detailed. Some key areas were covered. Firstly, using the Intel Compiler to take advantage of more aggressive optimisations, better processor specific optimisations and more vectorization tuning power. Secondly, improving the cache hit rate significantly by using row major order. The significance of using floats instead of array elements to increase vectorization speedup was also discussed. Thirdly, avoiding repeated pipeline flushes by removing the conditional in loops. Fourthly, a more in-depth look into tuning vectorization using data alignment was presented. Lastly, further analysis of vectorization and the potentially negative effects of it's interaction with other optimisations was discussed.