1 Performance Overview

The following table shows the impact of some optimisations on the performance of the program, tested on an input of `100 100 100`, using average timings.

Component	waxpby.c	ddot.c	sparsemv.c	Total
Unoptimised	1.018e+0s	7.033e-1s	1.096e+1s	12.7e+0s
Other opt.s	1.11e-1s (9.17x)	1.39e-1s (5.05x)	1.29e+0s (8.49x)	
OpenMP Threads	5.56e-2s (18.3x)	2.56e-2s (27.5x)	5.55e-1s (19.7x)	
AVX Intrinsics	6.54e-2s (15.6x)	1.81e-1s (3.89x)	1.08e+0s (10.2x)	
Both (Manual)	5.51e-2s (18.5x)	6.38e-2s (11.0x)	5.46e-1s (20.1x)	
OMP for SIMD	5.27e-2s (19.3x)	1.89e-2s (37.2x)	5.55e-1s (19.7x)	0.618e+0s (20.6x)

2 Optimisations

2.1 Structural optimisations

- `waxpby.c`: The individual if clauses were removed, and only the final (else) clause was reimplemented. Whilst this does mean having an unnecessary multiplication when `alpha==1.0` or `beta==1.0`, when coupled with other optimisations (eg. intrinsic vectorisation) it was found that the overhead of implementing the if statement was larger than the performance hit of the unnecessary multiplication.

- `ddot.c`: The if statement was similarly removed here, in favour of the else clause. Whilst this could mean an unnecessary memory fetch, testing shows that when further optimised with intrinsics, ddot is bound more by the loading of data into the vector registers, therefore the impact of this extra fetch is negligable. Testing showed that the impact of removing the if statement is within expected time variance, and therefore negligable.

2.2 Floating point optimisation

One of the larger optimisations undertaken was the change from using double-precision floating point numbers to single-precision. This is an ideal optimisation to make for this project, because the limited capacity of vector registers means that they can only store 4 doubles - or 8 floats. This means that theoretically, making this change provides a 2x speedup to all vectorised parts of the code, since 2x less vector operations will have to take place. For the code as a whole, in practice we get less than a 2x speedup. This is because not all of the code can be vectorised, as it does not compute based on large arrays of data. For example, `generate_matrix` operates at about the same speed whether or not the values in the matrix are floats or doubles. Additionally, whilst using floats may reduce the amount of data that needs to be fetched, it does not reduce the latency of memory fetches. Therefore, it is not as effective at speeding up memory-bound components, such as `sparsemv`.

For most parts of the code, making this change was as simple as replacing occurrences of double with float. However, in some places additional changes needed to be made.

- All intrinsic instructions using doubles had to be replaced with their float-counterparts. For the most part this was easy, since AVX provides drop-in replacements for most instructions.

- Any component requiring the summation of a vector had to be rewritten to sum 8 elements instead of 4. This added some complexity to the code and resulted in a slight slowdown, but this slowdown was insignificant compared to the speedup gained from using floats.

- Some parts of the code are reliant on the precision of doubles, such as `mytimer.c`. These parts weren't changed.

2.3 Tolerance tuning

Due to the floating point optimisation above, some precision was lost. Given the complexity of the residuals calculated, this means the program is no longer able to represent values below about 1e-6 1e-7. For almost all uses this level of precision is completely sufficient, hence why we were able to justify this level of precision.
This also allows us to improve the performance of the program by tweaking the tolerance defined in `main.c`. If we don't set a tolerance, the program will always continue to try and get closer and closer to the exact values, but only ever stop when it reaches the threshold defined in `max_iter` of 150 iterations, even if it is unable to represent a more precise value. This means it will always reach the 150 iteration cap before halting. By setting a tolerance of 1e-6, we can guarantee that the program will progress until it reaches a sensible, usable accuracy, but will halt once it's progress is significantly slowed due to reaching the limits of float representation.

2.4 Compiler flag optimisation

A number of compiler flags are used, or were considered for use with the code.

- `-Wall`, `-Wextra` and `Werror` were kept to keep the code error free during development, and have no bearing on the performance of the code.

- `-march=skylake` enables target-specific features when compiling. This allows the compiler to use optimisations that are supported on the target CPU (Intel Core i5-8500) but may not be supported on other CPUs.

- `-O3` was used instead of `O0`. This enables many standard optimisations - including some which reduce code size and execution time, as well as some that involve a space-speed tradeoff in favour of speed. This is favourable for our purposes.
  Some significant flags enabled by this involve automatic loop peeling, which unrolls a loop. This eliminates unnecessary loop control instructions, reducing execution time.

- `-funroll-loops` unrolls more loops, and in some cases completely peels a loop, removing all loop control instructions.

- `-ftree-parallelize-loops=6` is used to automatically parallelise some loops that were not parallelised by hand, and gives a small speedup. It may not be as effective as parallelising by hand, but is significant. This compiler flag is only available on DCS systems after loading the `gcc9` module.

Other compiler flags affecting maths, such as `-fno-math-errno`, `-fno-signaling-nans`, `-ffinite-math-only`, and `-fcx-limited-range` were considered. They provide a noticeable speedup in the program, and do not affect the outcome of the program, since the math features they disable are not used by the program. However, unfortunately the coursework specification is not very specific as to what it counts as "relaxed math options" - hence they were not used.

2.5 Intrinsic vectorisation

All 3 main components require the vectorisation of a for loop iterating through arrays of data. Therefore, generally, the approach taken to achieve this was to first unroll the loop by the defined `LOOPFACTOR` (4 times for doubles, 8 times for floats - the target CPU supports AVX2, which uses 256-bit vector registers, which is enough to store 4 doubles, or 8 single-precision floats - floats were eventually used throughout).
OpenMP also supports use of the `for simd` pragma, which uses compiler-driven optimisation to vectorise a for loop. In some cases, this was found to be more effective than our hand-coded implementation of vectorisation. By the nature of the optimisation (not affecting outcome of calculations), vectorisation does not affect floating point accuracy by itself.

2.5.1 Implemented

- `waxpby.c`: The `simd` omp pragma is used along with OpenMP threading to vectorise the arrays. This approach is slightly faster than the hand-implemented approach included in the code - see §1 for the timings used to make this decision.

- `ddot.c`: The `simd` pragma is also used here. This approach was significantly faster than the hand-optimised approach for ddot - this stems from a race condition which was occurring in the hand-optimised approach, and fixing this race condition significantly slowed the vectorisation.

- `sparsemv.c`: Here, the hand-optimised approach was found to be slightly faster than OpenMP's vectorisation. This is likely due to the complexity stemming from double indexing in the original linear code (`x[cur_inds[j]]`).
  Firstly, 8 floats and 8 integers are loaded into vector registers from `cur_vals` and `cur_inds`.
  `_mm256_i32gather_ps` is then used to fetch 8 float values from `x`, using the indices fetched from `cur_inds`.
  After multiplying our float values together, the resulting vector is summed, and the result is added to `sum`.
  This vector summation is fairly complex, and is implemented in `hsum_float_avx`. It works by extracting the lower and upper halves of the input vector into SIMD 128-bit vectors, adding together pairs of numbers until the sum is stored in the final index of the vector, and then returning that sum.

2.5.2 Considered

- `waxpby.c`: A hand-optimised solution was created for waxpby, providing only slightly slower execution times (see §1). It works by broadcasting alpha and beta to every value in vectors `alphavec` and `betavec` respectively, using `_mm256_set1_ps`. Values are then loaded into `xvec` and `yvec`, and multiplied by `alphavec` and `betavec`, and then added in order to produce the correct values for `w`.
  In hindsight, this solution could probably have provided the same or better performance by passing in a pre-loaded `alphavec` and `betavec`, rather than generating these vectors each time waxpby is called. Nonetheless, it was deprecated in favour of the OpenMP compiler-optimised parallelisation approach.

- `ddot.c`: Likewise, a hand-optimised version was built and considered. However, it performed significantly worse than the OpenMP vectorisation. This stems from a race condition which was occurring in the hand-optimised approach, and fixing this race condition significantly slowed the vectorisation.
  This approach used a similar approach to the considered waxpby approach, loading values into vectors, multiplying them, summing the resultant vector using `hsum_float_avx`, and adding it to the result.

2.6 OpenMP parallelisation

For all 3 main components, `#pragma omp parallel for schedule(static)` was used to parallelise for loops. The static schedule was chosen since for each loop, it was found through testing that using a static schedule is more effective than a dynamic one. This is due to the workload of each thread being roughly equal, hence the overhead of using eg. `schedule(dynamic)` goes mostly to waste.

- `waxpby.c`: The sole for loop is parallelised using the above pragma. Additionally, as described earlier, `for simd` is used to vectorise the loop, since it gives marginally better performance.

- `ddot.c`: Similarly, `simd` is used to parallelise this loop, giving a significant speedup. Here, `reduction(+:local_result)` is used to set `local_result` as a reduction variable, allowing the final result to be accurate.

- `sparsemv.c` has only it's outer loop parallelised. This is the best way to parallelise; putting the pragma inside the loop would create a lot of overhead from continually creating and destroying threads in a loop.

Pthreads were considered instead of OpenMP, however their use is much more involved, and adds overhead through management of eg. mutex locks. Therefore, OpenMP was used instead.

3 References

What are Race Conditions? Some issues and formalisations, Robert Netzer, Barton Miller (1991)
https://minds.wisconsin.edu/bitstream/handle/1793/59458/TR1014.pdf

Linux man pages, Various (Accessed 2022-03-22)
https://linux.die.net/man/

Intel Intrinsics Guide, Intel (Accessed 2022-03-22)
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Coffeelake compiler support, WikiChip (Accessed 2022-03-22)
https://en.wikichip.org/wiki/intel/microarchitectures/coffee$_l$*akeCompiler*$_s$*upport*

Intel(R) Core(TM) i5-8500 CPU specification, Intel (Accessed 2022-03-22)
https://www.intel.co.uk/content/www/uk/en/products/sku/129939/intel-core-i58500-processor-9m-cache-up-to-4-10-ghz/specifications.html?wapkw=i5-8500

GCC Optimize Options, The GNU Project (Accessed 2022-03-22)
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

OpenMP documentation, IBM (Accessed 2022-03-22)
https://www.ibm.com/docs/en/xl-c-and-cpp-linux/16.1.0

Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors; Liu, Chow, Smelyanskiy, Dubey (Accessed 2022-03-22)
https://faculty.cc.gatech.edu/ echow/pubs/ics26-liuPS.pdf

How to sum __m256 horizontally? (Accessed 2022-03-22)
https://stackoverflow.com/questions/13219146/how-to-sum-m256-horizontally

CS257 Advanced Computer Architecture Assignment, DCS (Accessed 2022-03-22)
https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs257/coursework-21/coursework.pdf

Computer Organization and Architecture: Designing for Performance (10th Edition), W. Stallings; Pearson, 2016

The C Programming Language (2nd Edition), B. W. Kernighan, D. Ritchie; Prentice Hall, March 1988.