CS257 Report

Brandon Hong u2101112

March 21, 2023

1 Introduction

This document entails design implementations and choices to the optimisation of CS257 coursework. Computation time and speed-ups mentioned in this document refers to running the program in a 100x100x100 dimension as speed-ups are easier to detect with larger dimensions. Number of floating point operations (FLOP) did not change after any described optimisation as no algorithmic optimisation or changes were made to the code. (provided that the dimensions tested on were the same, FLOP varies for different dimensions)

2 Floats instead of Doubles

Instances of doubles in the code to initialising them as floats. The main difference between floats and doubles are their respective byte sizes and precision level [1]. Doubles have twice the byte size of floats and a higher accuracy holding up to 15 decimal values. Floats being smaller can result to fewer cache misses, lower memory usage and with SIMD (Single Instruction Multiple Data), twice as many elements per vector for loops to vectorise and perform operations on [2]. The initial change from doubles to floats in the coursework code led to reduction in code execution time for waxpby.c ($1.1s \rightarrow 1.08s$) and sparsemv.c ($10.7s \rightarrow 10.4s$) while leading to a great increase in time instead for ddot.c ($0.8s \rightarrow 2s$), but in addition to multi-threading and vectorising will decrease these value by a huge margin. However, this optimisation will affect floating-point correctness. The difference between computed and exact value now holds less decimal places (up to 7 instead of 15) while final residual will hold 1-2 decimal places less than the case of using doubles.

3 SIMD

Vectorisation is a SIMD operation that handles operations on entire arrays/loops instead of dealing with individual elements as in a for loop. This means that one instruction carries out the same operation on a number of operands in parallel, allowing for faster computation, execution time and cleaner code. [3] Vectorisation was applied to all the files as there were no inter-loop dependencies, indirect memory access nor code branching within the for loops. While the original code for sparsemv.c had indirect memory access (x[cur_inds[j]]), the gather function was used to load a new vector using x as the base address and cur_inds as the index to allow vectorisation. [4]

3.1 AVX

Using AVX's 256-bit registers instead of the 128-bit SSE registers allow for faster code execution time as the amount of data which the computer can work is greater, allowing quicker processing of data. [5] With purely scalar operations, a loop would have to carry out N operations in N clock cycles. However, the number of instructions executed using vectorisation is approximately N/M; M being the number of operands of the size that it can operate on simultaneously. [6] Hence, using a larger bit register would allow for more operations to be performed in a single clock cycle than a smaller register. In return, this reinforces the idea of converting the initialisation of doubles in the code to floats as a 256-bit register can hold and operate on a set of 8 float operands (32-bits each) instead of 4 double operands (64-bits each), which reduces computation time with the trade-off of a small decrease in

floating-point correctness. Vectorisation was measured to be effective, decreasing the computation time for all 3 files. For waxpby.c ($1.08s \rightarrow 0.7s$), ddot.c ($2s \rightarrow 1.5s$) and sparsemv.c ($10.4s \rightarrow 7s$). This chosen optimisation did not affect floating-point correctness of the program.

3.2 Alignment

Memory alignment helps the CPU fetch data from memory in an efficient manner leading to less cache misses and avoid loading of cache lines more times than it is actually needed. [7] As the program uses floats, the memory alignment is set to 32 bits. By attempting to improve spatial locality of the cache, the program saw very minimal difference within code speed-up showing that data was already being stored rather efficiently but did result in more consistent timings.

3.3 Register

The register keyword was assigned to frequently used variables to suggest to the compiler that a variable should be stored in a register instead of memory. This can potentially make the variable access faster, since accessing a register is typically faster than accessing memory [8], observing a decrease of 0.1s in total time.

4 Loop Optimisation

The list of loop optimisation operations that were unused include loop peeling, loop blocking, loop fusion, loop fission and loop interchange. These loop optimisations were not applicable in an appropriate way to the coursework, only leaving loop unrolling as an option.

4.1 Loop Unrolling

Loop unrolling was applied to all three functions in the coursework file that are timed as there were no inter-loop dependencies and a speed-up was observed, enabling the program to perform multiple operations before updating the loop counter. Another reason for using loop unrolling is due to the large number of times the loops are being iterated (especially in large dimensions), leading to a significant increase to the speed of the program. [9] With unrolling the loops in each file with a loop factor of 8 and vectorisation applied, the computation time of each file has approximately **halved** from it's original computed time under a 100x100x100 dimension.

Loop factors up to 64 were attempted with each program, loading multiple 256-bit registers at a time but no further speed-up was observed for waxpby.c and ddot.c. For sparsemv.c, a small decrease of computation time of 0.15s was observed from increasing the loop factor from 8 to 24 while a loop factor greater than 24 did not result in any further decrease. No affects to floating-point correctness was observed with this chosen optimisation.

Another good way of measuring speed-ups for the program would be the number of MFLOP/s, a greater number would represent that more floating-point operations per second are being performed and demonstrates better usage of the CPU. [10] After applying vectorisation and loop unrolling, the value increased from 700 to 1300 MFLOP/s showing higher efficient use of CPU's floating point units.

5 Multi-Threading

Multi-Threading is the most important optimisation technique applied to the system as the biggest decrease in code execution time was expected as well as observed. Theoretically, splitting the loop and distributing the workload to multiple worker threads instead of running the program serially should obviously decrease code execution time. The main thing to note while multi-threading is to avoid race-conditions and overwriting of data between threads. Hence, the OpenMP API [11] which provides a large and robust library to handle such conditions was used to multi-thread the system instead of POSIX (Pthreads). This is also due to the nature of the code where multi-threading is only needed in loops that are computed on all the cores without the need of starting a separate process, while using

Pthreads which is harder to maintain would over-complicate the code base.

As a result of threading, the time for waxpby.c is now ($0.4s \rightarrow 0.16s$), ddot.c ($1.1s \rightarrow 0.14s$) and sparsemv.c ($5.5s \rightarrow 2.08s$). However, as a result of using threads on ddot.c there are now small fluctuations within the value of final residual and difference between computed and exact, but still within the appropriate order of magnitude. The reason to this may be due to loop iterations being distributed throughout threads, resulting in a different order of value being performed each time. Finally, the number of MFLOP/s result to about 4000 and a total time of 2.4s for all three programs.

5.1 Scheduling

All scheduling types (dynamic,guided and static) were attempted. Scheduling workload to worker threads can improve performance as it balances overhead and data distribution, preventing situations where a thread finishes it's computation earlier than other threads, wasting CPU usage as a thread then sits idle. [12] There were no noticeable difference in scheduling ddot.c but scheduling waxpby.c with static resulted in a more consistent timing, showing each iteration taking roughly the same time. [13] For sparsemv.c, guided was measured to have the best outcome reducing computation time by 0.1s. Overall, scheduling the program did not result in significant changes but if ran on a machine with a greater number of CPUs, this will prove to be effective as the described idle condition above is more likely to occur with more threads.

6 Restrict

The restrict keyword was used to inform the compiler that the assigned pointer is the only means of accessing a particular object during lifetime. [14] This can help the compiler to generate more efficient code especially in sparsemv.c as there are multiple pointer-based operations being called frequently. This tells the compiler that compiler is not aliased with another pointer, hence it is safe to vectorise the loop because there are no overlapping memory references. [15]

7 Dead-Code Elimination

Dead-code removal is a compiler optimisation to remove code which will never be ran in the program or does not affect the program result in any scenario. [16] ddot.c, it was noticed that the if-else statement is redundant as if (y == x) then x * x is equivalent to x * y leading to a case where both clauses execute the same code operation. By stripping dead-code, this shrinks the program size and complexity allowing the program to avoid executing irrelevant operations (such as evaluating the conditions for the if statement), which reduces program running time. [17] The same was observed for waxpby.c where the if statements are redundant as alpha is set to 1 when calling waxpby.c in conjugateGradient.c. While the speed-up of the program is not of a noticeable difference due to both files already requiring a small run time ($< 1s$), this is done as a good coding practice and might prove to be beneficial with the scaling of the program in the future. This chosen optimisation did not affect floating-point correctness of the program.

8 Conclusion

The overall time taken for the program to run ended up with 2.4s for 100x100x100 dimension. With the original compute time of 13s, this is a 5.4x speed-up. Various input sizes were later tested on the program with all of them providing an appropriate solution in the end. At this stage, it was hard to detect any weaknesses in the optimisation strategies with different dimensions as the unoptimised results for such dimensions were not recorded ahead of time. Lastly, the tolerance of the system was not changed leading the program to fully run it's maximum number of iterations for large dimensions.

References

[1] D. Goldberg. What every computer scientist should know about floating-point arithmetic. https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html. (accessed: 15.03.2023).

[2] S. Qadir. What is the difference between float and double? https://www.educative.io/answers/what-is-the-difference-between-float-and-double. (accessed: 15.03.2023).

[3] Konstantin. Improving performance with simd intrinsics in three use cases. https://stackoverflow.blog/2020/07/08/improving-performance-with-simd-intrinsics-in-three-use-cases/. (accessed: 18.03.2023).

[4] J.-L. D. Ph. Preux. Vector addressing processor for direct and indirect accesses. https://www.sciencedirect.com/science/article/abs/pii/016560749090314Y. (accessed: 18.03.2023).

[5] What is sse and avx? https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx. (accessed: 18.03.2023).

[6] How does register size affect processor performance? https://stackoverflow.com/questions/29524614/how-does-register-size-affect-processor-performance. (accessed: 18.03.2023).

[7] Purpose of memory alignment. https://stackoverflow.com/questions/381244/purpose-of-memory-alignment. (accessed: 21.03.2023).

[8] A. Trivedi. Understanding "register" keyword in c. https://www.geeksforgeeks.org/understanding-register-keyword/. (accessed: 21.03.2023).

[9] D. Lemire. Why are unrolled loops faster? https://lemire.me/blog/2019/04/12/why-are-unrolled-loops-faster/. (accessed: 16.03.2023).

[10] What is flop/s and is it a good measure of performance? https://stackoverflow.com/questions/329174/what-is-flop-s-and-is-it-a-good-measure-of-performance. (accessed: 19.03.2023).

[11] M. J. Quinn, *Parallel programming in C with MPI and OpenMP*. McGraw-Hill Higher Education, 2004.

[12] V. Kale. Loop scheduling in openmp. https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP_BoothTalk.pdf. (accessed: 20.03.2023).

[13] Y. Liu. Openmp-scheduling. https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP/. (accessed: 20.03.2023).

[14] Mike Acton. Demystifying the restrict keyword. https://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html/. (accessed: 21.03.2023).

[15] W.-H.Chung. Restrict keyword. https://www.sciencedirect.com/topics/computer-science/restrict-keyword. (accessed: 21.03.2023).

[16] P. D. Thain, *Introduction to Compilers and Language Design*. University of Notre Dame, 2019.

[17] Dead-code elimination. https://en.wikipedia.org/wiki/Dead-code_elimination. (accessed: 21.03.2023).