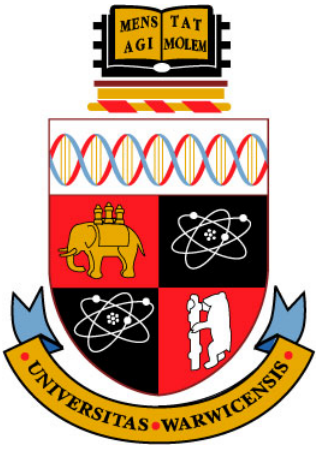

Optimisation Project Report

CS257 Advanced Computer Architecture

Rohan Tantepudi

Module Lead: Rossella Suma

Department of Computer Science

University of Warwick

Contents

1 Introduction

The problem given was to optimize a set of programs to simulate the application of a momentary force to a three-dimensional mesh. This force is dissipated in the mesh until either the residual moves below a threshold or enough time steps pass from initialisation.

The code will be tested and run on a DCS machine with an Intel i5-7500[5], containing 4 cores with 1 thread each. These are accessed by SSHing to the `kudu` machines provided by DCS. When optimisations are tested, 4 problem sizes are used: `50x50x50`, `100x100x100`, `200x200x200` and `300x300x300`. For each problem size, an optimisation is run 5 times, and the time spent on each of the `ddot.c`, `waxpby.c` and `sparsemv.c` kernels is recorded, according to the output file. An average of the 5 runs is taken, and each optimisation is compared - using these averages - against the non-optimised runs to denote the speedup rates of it.

1.1 Optimisations

1.1.1 Loop Unrolling

Loop unrolling consists of denoting a specific `loopFactor`. This value denotes how many consecutive instructions are run in a single loop iteration. From this loopFactor, a loopN value is calculated that determines the maximum iteration that the unrolled loop can iterate to. This reduces the overall number of iterations the compiler has to perform, with the trade-off of using more memory space to store the program, as well as any temporary variables[4]. For most loops in the given program, loop unrolling seemed to be effective, improving operating times by a factor of 1.32x for smaller dataset sizes and a factor of 1.16x for larger sizes. This may be a result of the loops that unrolling was implemented in having simple instructions that did not involve added complexities, such as continuous branching statements.

1.1.2 SIMD Intrinsics

Vectorisation is a form of executing SIMD (Single Instruction Multiple Data) behaviour through registers that hold multiple data elements at any one time. These registers can apply the same instruction across all elements simultaneously, saving time that would have been spent individually altering each element. This can be implemented on top of loop unrolling through intrinsics[3], lines of code that implement a map to underlying assembly instructions. Code used to repeat similar instructions on loops are replaced by single instructions on specified vectors, with the width of the vectors depending on the given vector type (for example, `m256d` vectors are 4 elements wide[1]), and usually correlating to the `loopFactor` specified when loop unrolling.

However, in the case of this program, intrinsics did not improve the processing speeds of the code, and in fact harmed them, resulting in an overall optimisation factor of only 1.16x for smaller dataset sizes and 1.14x for larger sizes, compared to the respective 1.32x and 1.16x speedups that loop unrolling by itself gave. This may be as a result of the data to be vectorised itself not being aligned, which may lead to ineffective memory access[6] - something that loop unrolling is flexible enough to avoid. While there is a way to allocate memory in an aligned way, attempting to do so resulted the program constantly exiting with a segmentation fault.

1.1.3 OpenMP Statements

OpenMP[7] is a library within C that allows for multi-processing within a program. This is an API used to explicitly direct multi-threaded, shared memory parallelism. This is done primarily by specifying the number of threads that will be used in each file in which multithreading is to be used, through the `omp_set_num_threads()`[2] function (in the optimised files, the number of

threads to be used has been set to 4, to fully utilise all of the hardware provided from the `kudu` machines used for testing. From there, `pragma omp`[2] statements can be used to signal the start of a threaded region. Every one of these statements used in this project denoted a `parallel`[2] section of code to be executed, as having parts of these sections run serially using `critical`[2] would end up being too expensive performance-wise to implement efficiently. Instead, specific variables outside parallelised loops can be set their own scopes for when they are used by threads within these loops.

For example, the `reduction`[2] scope can be used on variables such as `local_result`, which allows them to be added to by different threads while also having OpenMP manage any potential race conditions that may occur, all without the performance impact a `critical`[2] declaration may have. An interesting scope used in this project is `lastprivate` - unlike `firstprivate`, which gives each thread its own private form of the mentioned variable with the initial value similar to what it was outside the loop, this sets the variable's value after the loop has ended to the value held by the thread that finishes the loop. This is useful to keep track of incrementing variables that track how far in loops the program is.

For all files that were optimised, parallelisation proved to give the best speedup, ranging from 3.7x for smaller data sets to 3.8x for larger data sets. The ability for code to divide a process into separate threads greatly improves the process' runtime, with the only downside being potential segmentation faults due to race conditions with accessing shared variables (although this can be fixed by using specific flow control commands/variable scopes) In addition, combining OpenMP statements with SIMD intrinsics improved processing times more than just OpenMP on its own.

1.1.4 Failed Optimisations

An attempt was made at trying to convert the doubles used in the program into single floats. As floats occupy half the space that doubles do, operations involving floats can use less memory bandwidth, leading to faster data transfer and reduced cache misses. In addition, SIMD float vectors can hold double the values that double vectors can, meaning that more vector elements can be operated on in a single instruction, increasing parallelism. However, due to the reduced accuracy between a double and a float, converting this increased the error between the calculated and actual final values from 7e^12 to 2e^5, which seemed like too much of a decrease in accuracy to warrant the change. In addition, while decreases in processing time were seen in both `waxpby.c` and `sparsemv.c`, `ddot.c` suffered from a drastic increase in processing time (an improvement factor of 0.2x), which may have come as a result of not taking into account the changes to the vectorisation code itself to account for a different data format.

1.2 Files

1.2.1 waxpby.c

Problem Size	Base Time	Loop Optimisations	SIMD Intrinsics	OpenMP + SIMD	OpenMP w/o SIMD
50	62.4	56.0	69.2	21.0	19.4
100	479	470	528	159	149
200	3920	3800	4220	1650	1640
300	14400	14000	15000	5670	5650

Table 1: Effects of optimisations on waxpby.c in terms of time (ms, 3s.f.)

This file's greatest speedup came from OpenMP parallelisation, resulting in an average of 3.2x speedup for smaller problem sizes and an average of 2.4x speedup for larger problem sizes. What is interesting is the fact that using OpenMP parallelisation without SIMD intrinsics ended up with faster runtimes than with them (With intrinsics, improvements only ranged from 2.5-2.9x rates). This could be due to the simplicity of the loops themselves, having no need for nested loops or summations to a shared variable, and thus lending themselves to loop unrolling's flexibility. Loop unrolling itself provided a further 1.2x performance improvement to processing speeds, thus also being used for the final optimisation of `waxpby.c`.

1.2.2 sparsemv.c

Problem Size	Base Time	Loop Optimisations	SIMD Intrinsics	OpenMP + SIMD	OpenMP w/o SIMD
50	900	655	757	209	219
100	7110	5400	6280	1680	1780
200	58200	49200	50200	13800	14600
300	220000	184000	186000	48000	48600

Table 2: Effects of optimisations on sparsemv.c in terms of time (ms, 3s.f.)

As this file contains the most complex code and thus takes the longest time out of the 3 optimised files to run, optimisations to `sparsemv.c` are vital. The greatest improvement to processing speeds for this file, once again, came from the implementation of OpenMP parallelisation, resulting in speedups of 3.0x for smaller problem sizes and 3.8x for larger problem sizes. This may be because at larger problem sizes, OpenMP's usage of multiple threads to execute similar instructions can greatly reduce the number of clock cycles to execute programs. Unlike `waxpby.c`, however, using OpenMP was better utilised with SIMD intrinsics, providing an extra 1.05x speed-up compared to not using intrinsics. This may also be a result of parallel processing providing greater impact on larger problem sets and more nested programs. For this reason, for the final optimisation of `sparsemv.c`, a combination of SIMD intrinsics and OpenMP parallelisation would be used.

1.2.3 ddot.c

Problem Size	Base Time	Loop Optimisations	SIMD Intrinsics	OpenMP + SIMD	OpenMp w/o SIMD
50	68.0	67.3	62.0	12.2	19.6
100	533	540	373	94.1	153
200	4310	4330	4080	783	1240
300	16300	16300	10800	2710	4110

Table 3: Effects of optimisations on ddot.c in terms of time (ms, 3s.f.)

The greatest optimisation for this file comes from utilising OpenMP parallelisation, bringing a speedup of 5.1x for smaller problem sets and 4.0x for larger sets. Unlike the other 2 files, SIMD intrinsics give a better speedup compared to just loop unrolling, decreasing processing times by a further 1.2x on average. This could be due to how these intrinsics are better suited for more complex calculations, such as adding to a shared variable, compared to simple loop unrolling. This could be the same reason as to why combining OpenMP with SIMD intrinsics proves to be

a better optimisation (giving a speedup of 4.0x from SIMD intrinsics) than running OpenMP on its own (giving a speedup of only 2.4x from SIMD intrinsics).

References

[1] Intel. Intel® intrinsics guide, 2024. [online] Available from: https://www.intel.com/content/www/us/en/docs/intrinsics-guide.

[2] OpenMP. Openmp 6.0 api syntax reference guide, 2024. [online] Available from: https://www.openmp.org/wp-content/uploads/OpenMP-RefGuide-6.0-OMP60SC24-web.pdf.

[3] J. Reinders. Simd intrinsics aren't so scary, but should we use them?, 2017. [online] Available from: https://www.infoworld.com/article/2254236/simd-intrinsics-aren-t-so-scary-but-should-we-use-them.html.

[4] W. Schools. Loop unrolling, 2023. [online] Available from: https://www.geeksforgeeks.org/loop-unrolling/.

[5] P. Software. Intel core i5-7500 @ 3.40ghz, 2025. [online] Available from: https://www.cpubenchmark.net/cpu.php?cpu=Intel%2BCore%2Bi5-7500%2B%40%2B3.40GHz&id=2910.

[6] J. Wilczek. What is simd in digital signal processing?, 2022. [online] Available from: https://thewolfsound.com/simd-in-dsp/#disadvantages-of-simd.

[7] J. Yliluoma. Guide into openmp: Easy multithreading programming for c++, 2016. [online] Available from: https://bisqwit.iki.fi/story/howto/openmp/.