ENCE464 Assignment 2

Michael Zhu, Sehyun Kim, Group 41

1 Architecture overview

The computer architecture use is the machines in ESL (Embedded System Lab) with 2021 sticker label. The CPU installed is Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz with 8 cores and 16 threads. Figure 1 below shows the computer architecture of the machine.

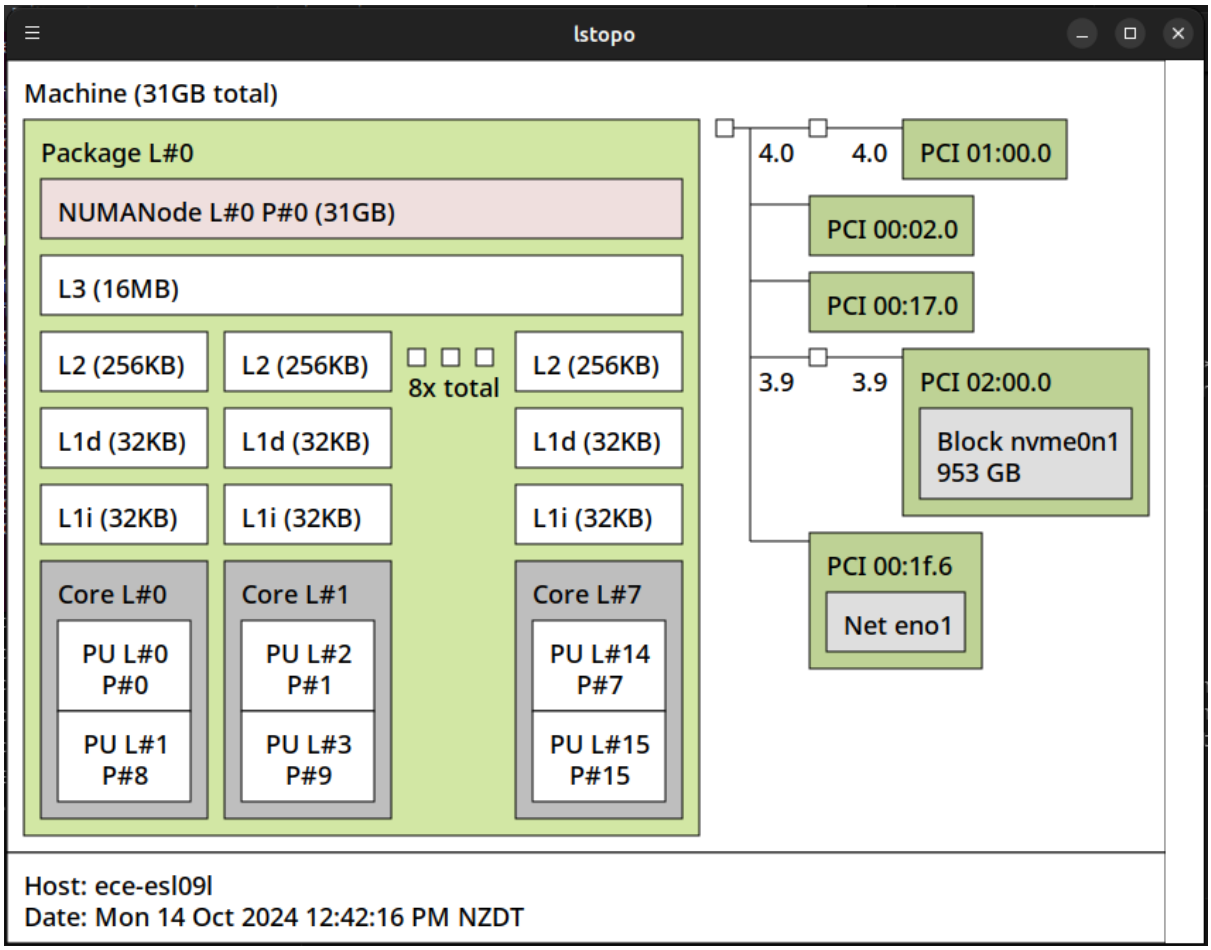


Figure 1: Computer Architecture Overview ESL Lab 2021 Model

The cache has three levels L1, L2, and L3. There are two cache types In Level 1 (L1d, L1i). L1d cache stores data that the CPU core frequently access and L1i cache stores instructions reducing the time needed to fetch from slower memory. Both level 1 cache types have 8 instances with a storage of 32 KB each.

Level 2 cache acts as an intermediate cache between L1 and the slower L3 cache. It is larger than L1 in terms of storage and allows L1 to store more data and instructions with a size of 256 KB and 8 instances.

Level 3 cache is shared among all cores and has a large storage capacity whilst being higher latency. There is 1 instance of a 16 MB L3 cache for the CPU which allows for efficient data sharing among cores.

The machine has 32 GiB RAM which is more than sufficient to handle the data load of the application. RAM (Random Access Memory) is used by the CPU as a temporary storage area for data and instructions.

2 Multithreading

The program takes advantage of multiple cores using block multithreading. The workload of the 3D grid is divided into smaller blocks. Each thread is assigned a specific range of z-axis slices which is processed independently. This ensures that the 3D voxel grid is evenly distributed among threads. This maximizes the utilization of CPU cores therefore enhancing performance and scalability.

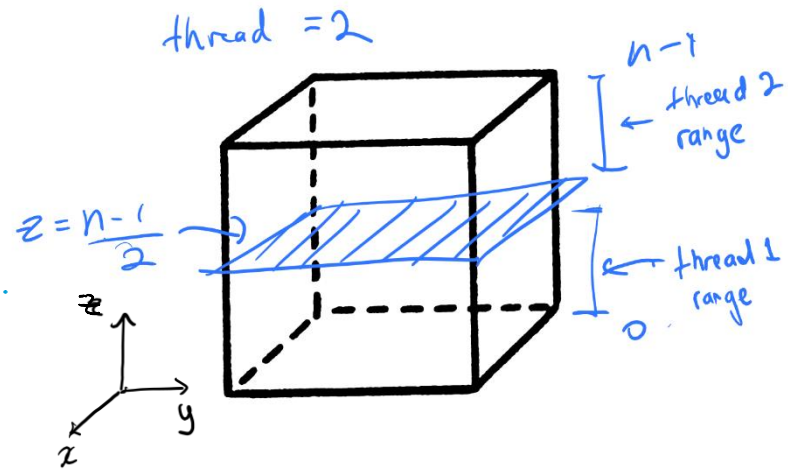


Figure 2 : Implemented multi-threading method

The threads divide the task management into x, y, z axis. The range of x and y is the full range of the cube for each thread whereas the z axis is divided into slices depending on how many threads have been initialized. The range of each thread is the full range (n-1) divided by the number of threads. An example with 2 threads can be seen in the diagram below.

Barrier synchronization was implemented in the program to ensure that threads are properly synchronized to avoid race conditions when they update shared resources. pthread_barrier_wait() is used to synchronize threads after each iteration. This ensures that all threads finish updating their part for the current iteration before proceeding to the next iteration [3]. This is important for accuracy of the solution since threads depend on data computed by other threads in the previous iteration.

Figure 1 below shows multithread performance data for voxel N = 101, 201, 301, 401, 501, 601, 701, 801, and 901 with a constant interaction of 100.

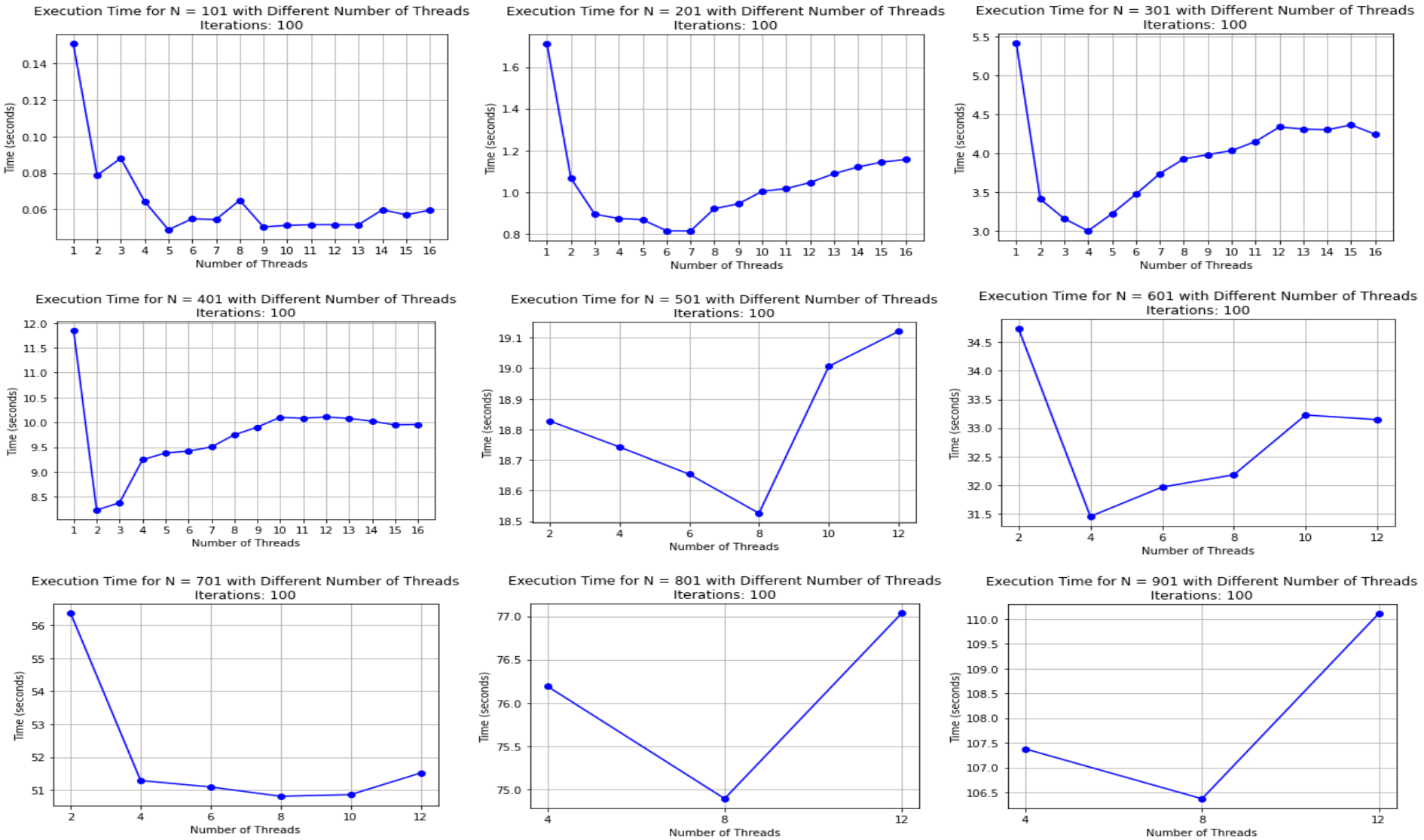


Figure 3 : Multithreading timings comparison

There is a clear trend where execution time decreases significantly as the number of threads increases until an optimal number of threads is reached. After the optimal number is reached the execution time starts to increase.

The reason for the significant decrease in time at the start is because the CPU is underutilized. This is especially obvious between one to two threads because the CPU has eight cores and by doubling the number of threads the tasks can be computed in parallel which now utilizes more than one core. This is true up until the optimal number which depends on the number of voxels.

Once optimal threads are hit the returns diminish as thread workers are outweighed by the overhead expenses to manage a thread. The CPU must constantly switch between threads as well as the workload per thread becomes too small thus reducing efficiency.

Larger datasets mean that the threads must synchronize more to avoid conflicts because of shared memory. This synchronization overhead can negate the benefits of parallelism [2] therefore more threads do not make processing large datasets faster.

3 Cache

Table 1 below shows the memory usage during different sections of the algorithm.

Table 1: Cache usage of different parts of the algorithm for different values of dimension N

	N = 301	N = 401	N = 501
Corners	2,529,292 (~0.0%)	3,359,292 (~0.0%)	4,189,292 (~0.0%)
Faces	753,689,300 (8%)	1,340,919,300 (6.2%)	2,071,249,200 (5.4%)
Inner Cube (Bytes)	8,895,458,399 (92%)	20,616,588,300 (93.9%)	39,715,718,300 (94.6%)

The results show that the inner cube consistently takes up most of the memory. This shows how computationally expensive the inner cube is and where the program could optimize.

Table 2 below shows the percentage of misses in each cache.

Table 2 : Cache hit miss % comparison

N = 301	Before	After
I1	2.5%	~0.0%
LL	10.4%	2.9%
LLi	1.3%	~0.0%
LLd	18.9%	6.6%
D1	27.8%	17.3%

I1 is level 1 instruction cache, LL is last level cache, LLi is last level instruction cache miss, LLd is last level data cache miss, and D1 is level 1 data cache. These regions represent distinct parts of the computational domain, and their contributions to memory usage. It can vary significantly due to their positions, dimensions, and size. Cache optimizations were applied to improve runtime efficiency, primarily by altering memory access patterns and maximizing cache usage to reduce cache misses as shown in the table above. The cache optimization methods used were Loop Interchange and Array Merging.

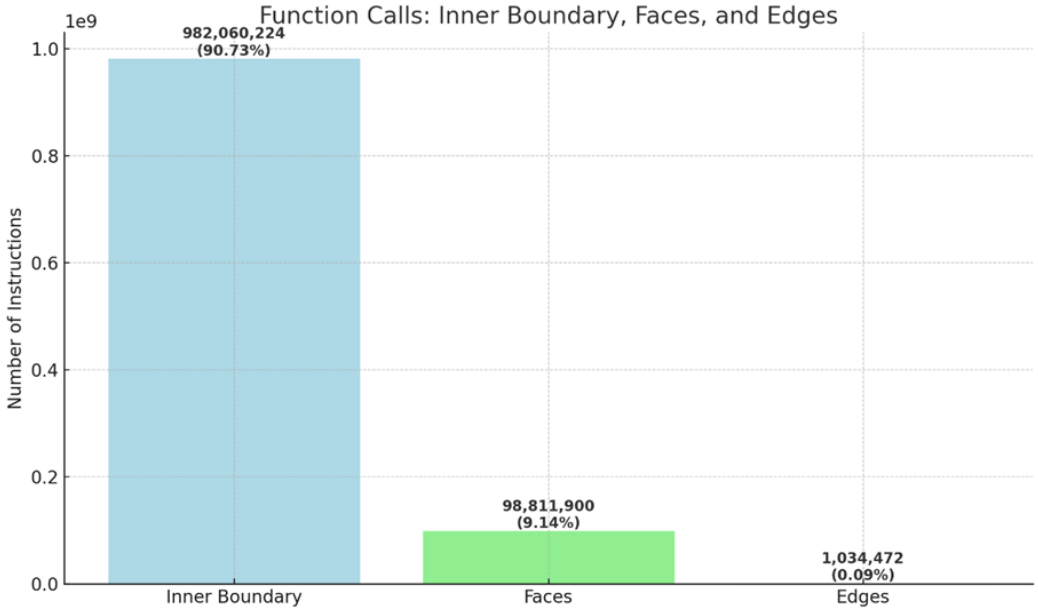
Loop Interchange [2]:
Improved spatial locality by computing in a row-wise manner as the array in memory is in row major order. Row major order is when the information is stored row first and then a column after the row has been filled out. This implementation causes the cache line to be used in successive loop iterations and hence a faster reading from the CPU [2]. The code implementation starts with a loop for the z-axis slice, which increments by n^2 in terms of array index. and serves as the outermost loop. After handling any exception cases, the y-axis loop begins, which increments by n in terms of array index, followed by the x-axis loop, which increments by 1. This order ensures that the traversal of memory access is contiguous, optimizing data handling and processing flow.

Array Merging [2]:
Improved spatial locality by storing both next and current arrays for the cube next to each other to be contiguous in memory [2]. Because these two arrays are accessed together this improves the average memory access time. This was achieved by creating an array as a variable which would have hold each array within its index.

4 Profiling

The profiling results from gprof indicated that the worker function took accounts for 100% of the runtime of the program. This indicates that programs spend all its time in the Poisson algorithm function. The entire algorithm is implemented in a single worker function because this design minimizes the overhead of function calls and allows for continuous processing without interruption. This method removes an entire for loop compared to splitting boundaries, faces and edges into their own functions. This is beneficial in numerical computing where loop overhead can significantly impact performance.

Using callgrind tool we can see how many function calls has been made at which location in the worker function. This allows for a more in depth and accurate approach in terms of gauging processor usage compared to other methods such as timing using clock. The result for function calls for N = 101 for 100 iterations with 8 threads can be seen in the following figure 4.



Most of the instructions is done within the inner boundary, this can be further analysed by breaking the inner boundary to each line within the code. The breakdown analysis of the inner boundary in figure 5 below shows that the “delta * delta * source([IND(x,y,z,n)])” takes up most of the computational power.

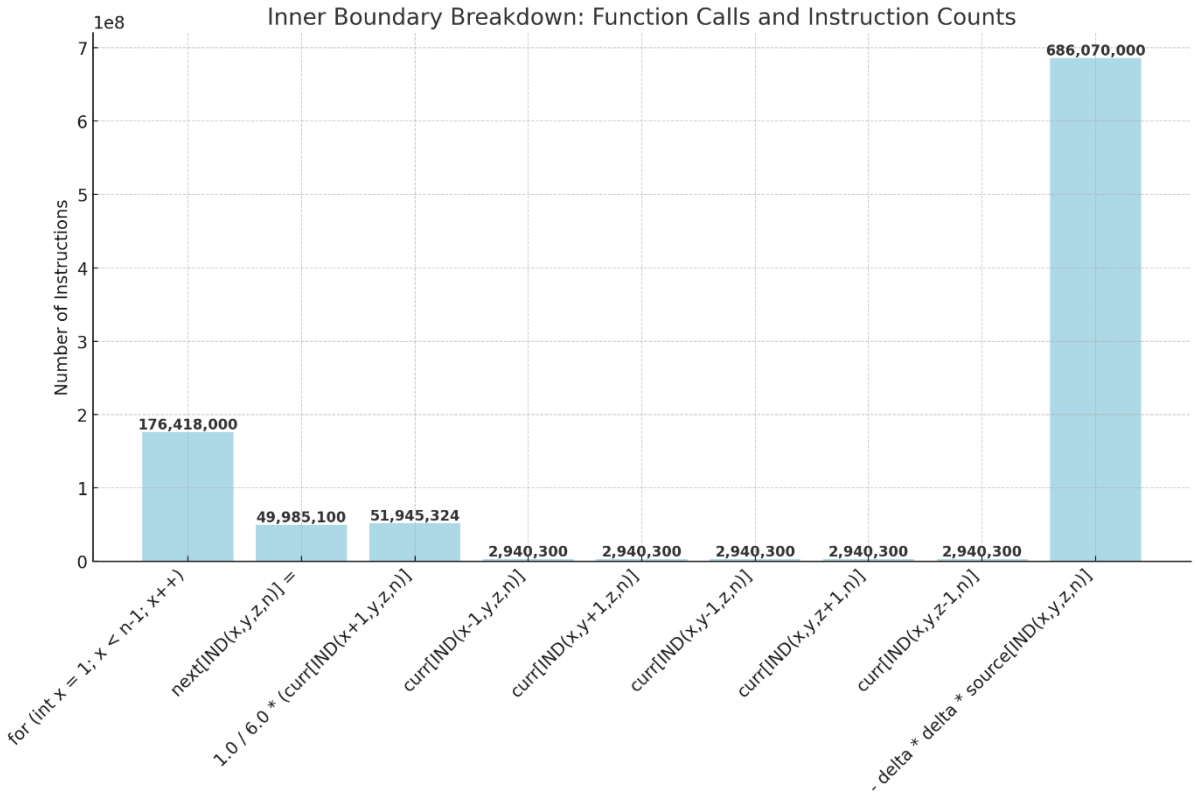


Figure 5 : Number of function calls in inner boundary

This high cost can be attributed to several factors. Firstly, there is two multiplication operations

$$delta * delta * source[IND(x, y, z, n)]$$

which are computationally intensive compared to addition or subtraction. This is made worse by repeating this multiplication it several times as this is all inside nested loops. It is likely it is being ran at n^3 times for a three-dimensional problem which is a very large computational cost.

Additionally, the access to $source[IND(x, y, z, n)]$ involves indexed access to a large 3D array. If the array is not efficiently cached or if the access pattern leads to cache misses. The CPU also needs to fetch data from the slower main memory which further increases execution time.

5 Compiler optimisation

Four different compiler optimization settings O1, O2, O3, and O4 were chosen alongside the default compiler configuration. They were tested for N dimension 101, 201, 301, 401, 501, 601, 701, 801, and 901, 100 iterations, and with 8 threads.

O1 is a basic optimization level that reduces code size and performs a few optimizations without sacrificing compile time [4].

O2 is a more aggressive optimization version of O1 where it aims to push performance further whilst avoiding risky transformations [4].

O3 is the general highest optimization level that chooses to include more aggressive optimization that might increase compile time or increase binary size for performance gains [4].

Ofast includes all optimizations from O3, plus more aggressive, non-standard-compliant optimizations to further boost performance in exchange for precision [4].

The evaluation of the different compiler optimization settings can be seen graphed in the figure 6 below.

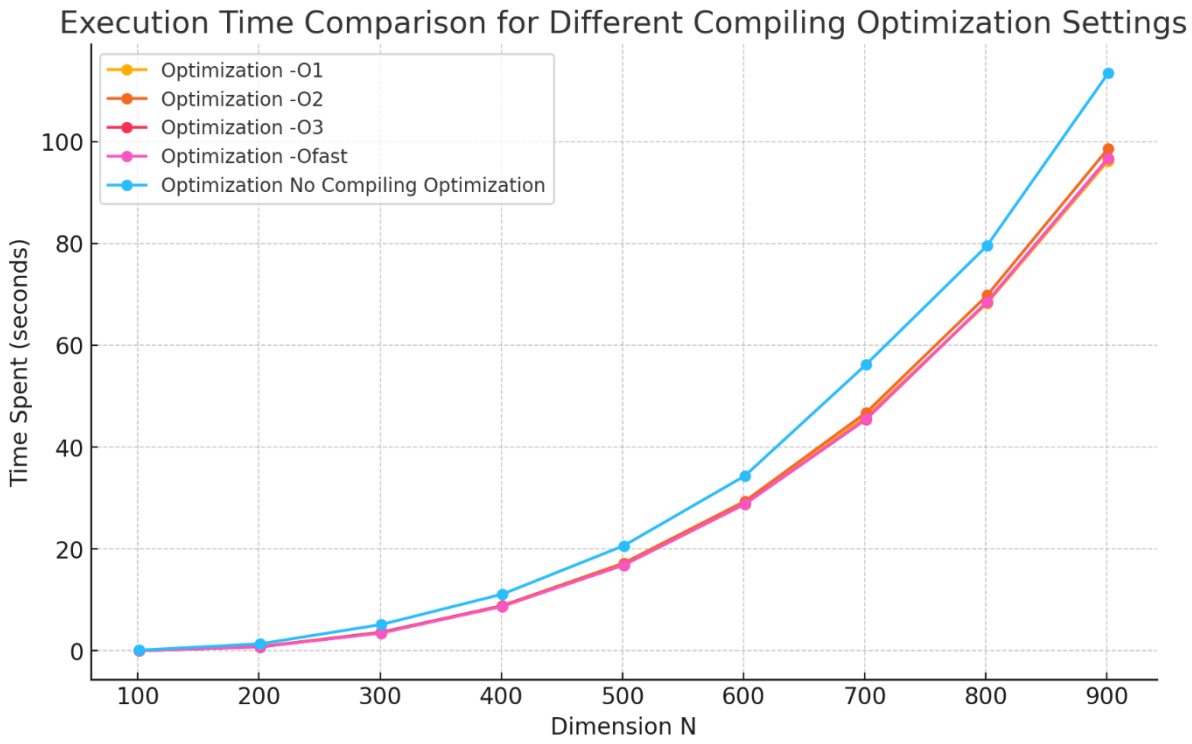


Figure 6 : Execution time comparison optimization settings

The results show a massive improvement of around 10% between no compiler optimization and the optimized compiler at a dimension size of N = 901. However, the different optimization settings for the compiler all seem very close in performance. This could be due to the optimization handled within O1 such as simplifying instructions, basic loop transformation, and reducing code size being the majority of the optimization the code needed and further aggressive optimization methods such as simplifying calculations is not effective for this application.

References

[1] Bhuyan, A. (n.d.). *Advantages and Disadvantages of Using Multiple CPUs vs. a Single Large CPU in Parallel Computing*. Retrieved from Medium: https://aditya-sunjava.medium.com/advantages-and-disadvantages-of-using-multiple-cpus-vs-a-single-large-cpu-in-parallel-computing-e4bb0085f73d#:~:text=Overhead%20in%20Communication,frequent%20data%20sharing%20or%20synchronization.

[2] Kowarschik, M., & Wei, C. (2003, April). *An Overview of Cache Optimization Techniques and Cache - Aware Numerical Algorithms.* Retrieved from https://www.researchgate.net/publication/2863424_An_Overview_of_Cache_Optimization_Techniques_and_Cache-Aware_Numerical_Algorithms

[3] The Open Group . (2004). *The Open Group Base Specifications Issue 6*. Retrieved from https://pubs.opengroup.org/onlinepubs/009696899/functions/pthread_barrier_wait.html

[4] gcc gnu. (n.d.). *3.11 Options That Control Optimization*. Retrieved from https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html