Parallel Computing-Work Assignment Phase 3

1st Hugo dos Santos Martins
Universidade do Minho
a95125
Barcelos, Portugal
a95125@alunos.uminho.pt

2nd João Bernardo Teixeira Escudeiro
Universidade do Minho
a96075
Valpaços, Portugal
a96075@alunos.uminho.pt

***Abstract*—This document is a brief report that explains all stages covered since the initial until the final code. It has a discussion about the optimization of a program that initially was sequential and all the steps the group has done to achieve the final product (CUDA).**

I. Introduction

In the pursuit of enhancing computational efficiency and accelerating performance, this report encapsulates a multi-phase optimization journey focused on leveraging vectorization and parallelism techniques. The primary objective in the initial phase was to optimize the given code through vectorization and instruction level parallelism, methodologies that aim to exploit parallelism in processor architectures. Building upon the success of vectorization, the second phase delved into the realm of parallelism, with a specific focus on OpenMP. In the third and final phase of this optimization objectives extends the parallelization efforts to a specialized computing environment – the Graphics Processing Unit (GPu) using CUDA. For each phase, a detailed account of the steps taken and the overall optimization achieved will be provided. The report aims to meticulously outline the specific actions taken in implementing vectorization, OpenMP parallelism, and the transition to GPU with CUDA, emphasizing the impact on the overall code performance in each iteration of the project.

II. Code Analysis

The provided code was about a molecular dynamics simulation app for argon gas atoms. We identified some bottlenecks that could be improved in order to parallelize the code and improve its performance. Almost all the functions had nested loops and some of them have the same iterations and so , this could be changed to avoid code redundancy.

III. Phase 1- Sequential code Optimizations

In the initial phase, as previously mentioned, our primary objective was to optimize the sequential code. To evaluate its initial performance on the provided cluster, we deliberately ran the code without alterations, using the **Perf** Tool to measure execution time, #cycles, and #instructions.

The code profiling revealed notable elevations, particularly in terms of the number of instructions (#I) and cycle counts (#CC), contributing to an extended execution time of approximately 330 seconds.

Upon confirming an excessively high execution time, our decision was to hone in on optimizing the code. Recognizing calculations involving exponentiation up to the power of 32, we implemented significant simplifications in maths calculations that could be simplified. These measures were taken to alleviate the computational burden and enhance overall code efficiency.

Following this assessment, our attention turned towards parallelizing instructions and matrix computations to harness optimal parallelism. Recognizing the potential for concurrent execution, we embarked on a strategic method to leverage parallel processing capabilities. This involved identifying independent tasks within the code, facilitating their simultaneous execution to enhance overall efficiency. In this stage the execution time had decreased to 56 seconds and the number of instructions was three times lower.

At this juncture, the strategic decision was made to incorporate flags similar to those employed in practical sessions. These flags, tailored to optimize parallel execution, played a pivotal role in our enhancement strategy. Specifically, we introduced flags designed to exploit parallelism, such as those governing loop unrolling and vectorization. This lies in instruction-level parallelism by expanding loop bodies, while the latter leverages SIMD (Single Instruction, Multiple Data) capabilities for efficient parallel processing.

In this phase, following the implementation of all modifications, we conducted a comprehensive reevaluation of the code's performance. The results reflected a substantial improvement, with the execution time dramatically reduced to approximately 11 seconds.

A potential improvement lied in the consolidation of the 'Potential()' and 'ComputeAccelerations()' functions, given that the 'Potential' functionality could be seamlessly integrated into the loops of the 'Compute' function. This proposition arises from the similarity in computations and loop structures between the two functions. The rationale behind this suggestion stems from the expectation that merging these functions would lead to a reduction in the total number of operations, enhancing computational efficiency.

After joining the two functions and streamlining certain mathematical computations, we achieved a notable reduction in execution time, clocking in at 5 seconds. This optimization endeavor resulted in a substantial decrease in the number of instructions to $21 * 10^9$ and cycles to $16 * 10^9$. The

successful convergence of functions and simplification of mathematical calculations not only significantly improved the code's efficiency but also contributed to a marked reduction in both the number of instructions executed and the overall cycle count. These achievements underscore the effectiveness of the optimization strategies employed, signaling a great advancement in computational performance of the program.

Fase	Tempo (s)	Instruções	Ins/Ciclo	Ciclos
Original	333	2.81×10^{12}	1.23	1×10^{12}
Reduce Calculations	119.52	6.20×10^{9}	1.68	3.69×10^{9}
Parallization	52.43	3.00×10^{9}	1.84	1.62×10^{9}
Flags	12.44	5.50×10^{9}	1.66	3.3×10^{9}
Vectorization	11.05	4.50×10^{9}	1.41	3.2×10^{9}
Function Union	9	3.10×10^{9}	1.17	2.6×10^{9}
Optimizations	5.69	2.10×10^{9}	1.32	1.6×10^{9}

TABLE I
RESUMO DAS MÉTRICAS DE DESEMPENHO MEDIDAS NA PRIMEIRA FASE DE OTIMIZAÇÃO.(N=2160)

IV. PHASE 2 - PARALLEL CODE IMPLEMENTATIONS

In the second phase of the practical project, the primary objective was to build upon and optimize the work done in phase 1. The focus shifted towards implementing the code with parallelism, leveraging the OpenMP library. This approach aimed to enhance the efficiency and performance of the existing codebase by introducing parallel processing techniques, ultimately contributing to a more streamlined and faster execution of the program. By threading and parallelizing relevant sections of the code, the project aimed to achieve better resource utilization, marking a significant step towards achieving the overall objectives of the practical assignment.

In our initial strategy, we opted to employ **gprof** to identify functions significantly impacting our code's performance. Our analysis revealed that the computeaccelerations() function emerged as the most computationally intensive, consuming approximately 90 percent of the code execution time (4seconds +/-). Recognizing the critical role this function played in overall performance, our team shifted its focus towards optimizing computeaccelerations() to achieve more efficient execution.

In the targeted function, a loop was identified for initializing an array 'a' with zeros, and later, this array was modified throughout the computation of Potential Energy. To enhance performance, we introduced the OpenMP **pragma omp parallel for** to parallelize both the initialization and computation loops. The objective was to leverage multiple threads for concurrent execution and reduce overall function execution time. However, as anticipated, data race issues surfaced due to shared data access during array value modifications. Despite the significant reduction in execution time, the obtained results were incorrect. Recognizing the occurrence of data races, the group investigated potential problematic areas, pinpointing one in the modification of the 'a' array values. This realization prompted a reassessment of the parallelization strategy to address data race concerns and ensure accurate computation results.

To fix data race issues in modifying the 'a' array values, we initially used **atomic** primitives but found a better solution with the **reduction** pragma to the array **a**. This approach synchronized updates during parallel execution, addressing concurrent access challenges related to additions. Incorporating the reduction pragma balanced efficiency and correctness, ensuring accurate results while benefiting from parallelization performance gains. The runtime schedule pragma was deemed more efficient due to its adaptive nature, providing dynamic load balancing during program execution. Unlike static scheduling, where the number of iterations is fixed for each thread at the start, runtime scheduling dynamically adjusts the workload distribution based on the actual runtime performance of each thread. This adaptability minimizes idle time and optimizes the allocation of tasks, promoting better resource utilization and load balancing. As we can see in Figure 1 and in Table 2, the final results of execution time we got When changing the Number of Threads for Number of Atoms of 5000 is :

Number of threads	32	35	37	40
Tempo	1.841	1.821	1.817	1.836

TABLE II
FINAL EXECUTION TIMES IN CLUSTER WHEN CHANGING NUMBER OF THREADS.

V. PHASE 3- USE OF GPU ACCELERATOR

The assigned task for our group involved the analysis and improvement of a previously modified codebase from the prior phase. Among the optimization options considered were implementing the previous version using full GPU resources and creating a version utilizing a Message Passing Interface (MPI) for distributed computing. Notably, the potential improvement of the OpenMP version was intentionally excluded to avoid limitations on the group's overall evaluation. This decision reflects a strategic balance between the pursuit of comprehensive optimization and practical considerations, leaving room for exploration of parallel computing paradigms.

VI. OPENMP

In our earlier project phase, we implemented OpenMP, achieving goof results with a runtime of approximately 1.5 seconds and,without encountering any data races. The success of this implementation instilled confidence within the group, leading us to make a strategic decision in the subsequent phase. Given the proven reliability and effectiveness of our OpenMP version, we opted to forego further refinements, channeling our resources and efforts toward exploring alternative areas for potential enhancement. This strategic approach not only underscores our commitment to optimizing the project comprehensively but also reflects a nuanced understanding of resource allocation and prioritization in the dynamic landscape of software development.

VII. MPI -Message Passing Interface

In the evaluation of optimization options, the implementation of a version utilizing Message Passing Interface (MPI) for distributed computing was considered. However, after careful consideration, our group opted against this method due to a collective acknowledgment that we weren't entirely comfortable with the intricacies of implementing MPI. This decision reflects our commitment to pragmatic and achievable solutions, acknowledging the importance of working within the group's collective expertise and comfort level. While MPI presents opportunities for distributed computing, choosing not to pursue it at this juncture ensures that our efforts are concentrated on areas where we can confidently navigate and make notable improvements without introducing unnecessary complexity or risk.

VIII. CUDA-Computer Unified Device Architecture

Our decision for choosing CUDA over MPI in our project was primarily based in its efficacy for parallel computing on GPUs, a quality aligned with our specific computational requirements. Furthermore, our decision was notably swayed by the inclusion of CUDA in both theoretical discussions and practical classes through the semester .

To delve into how CUDA operates, it's essential to recognize that CUDA is a parallel computing platform developed by NVIDIA that enables developers to harness the computational power of GPUs for general-purpose processing. In a CUDA-enabled application, the CPU (host) and GPU (device) collaborate in a synchronized way. The host manages control flow and data transfer, while the device executes parallelized tasks. CUDA's architecture allows multiple threads to execute concurrently, each handling its designated computation. This parallel processing capability can significantly accelerate a diverse range of applications.

The first strategic action we undertook was a comprehensive analysis of the code inherited from the previous phase. This examination was not confined to a surface-level review but extended to a meticulous scrutiny aimed at identifying functions that had the potential to introduce performance bottlenecks in our code. Our specific focus was on discerning how these functions behaved under varying conditions, specifically in response to fluctuations in the number of atoms provided as arguments. This preliminary investigation set the stage for a nuanced understanding of potential performance challenges.

We promptly identified that the function imposing the most substantial time constraints on our result calculations was the computeAccelerations() function, as well as in the previous phases. Not only was this function invoked frequently, but it also presented notable challenges (nested loops with huge calculations) that significantly impacted the computation time during our analysis.

Our initial focus revolved around understand the nuances of CUDA's functioning, with a specific goal in mind – the restructure of the computeAccelerations function to harness the full potential of the GPU, thereby parallelizing the particle calculations. This strategic approach aimed not only to exploit the power of parallel computing but also to unlock unprecedented efficiency in our computational processes. The initial step involved splitting the computeAccelerations function into two separate functions. The first function is responsible for initializing memory and arrays, copying the necessary values from the host arrays (a and k) to the device, invoking the computeAccelerationsGPU function for calculations, and finally copying the results back to the host for further CPU processing. The second function, namely computeAccelerationsGPU, encompasses all calculations related to the original computeAccelerations function. This two-function approach not only streamlines the codebase but also optimizes the utilization of GPU resources, marking a strategic advancement in our quest for efficient parallelized particle computations.

The process of developing the two functions and troubleshooting for errors posed a significant challenge initially, as we were not entirely accustomed to working with CUDA. We encountered numerous errors, often stemming from basic mistakes such as inadvertently copying excessive information, leading to a substantial overhead in program performance. Another common issue was variations in results, indicating that the program was not behaving as expected.

In the initial versions, even after resolving these issues and obtaining correct values, we faced challenges in ensuring the program's stability. To mitigate potential problems, we incorporated the use of an atomic add function. This step was crucial to prevent issues related to concurrent memory access and to ensure the correctness of the computations and results.

At this stage we managed to get the correct results, but we realised a strange fact.The total execution time of the program with the CUDA (about 20 seconds with 16 threads /block) version took a lot more time when executing than the parallized version used OpenMP.We recognised that this fact was a bit strange, not only because normally it is expected the GPU performance to be better than the CPU, but also that we needed to do some upgrades in order to try to use the more GPU resources possible and in a more effective way. A critical aspect emphasized during our classes was the importance of utilizing shared memory among threads within the same block, a fundamental practice in parallel programming, particularly within the CUDA framework. The use of shared memory could be helpful because in the function that is set to do the calculations, all threads in the same block access the "r_Cuda" array , which is a copy of the original **r** array, a lot of times. The implementation of a shared memory for this **r** array could improve the code performance because shared memory provides fast and low-latency access compared to global memory accessing the **r** array).

Recognizing that the utilization of atomicAdd in our code presented a bottleneck, particularly in comparison to the more efficient primitive observed during the preceding phase, we sought a strategic reevaluation of our GPU code. Deliberating on this matter, we identified an opportunity to optimize the structure of our GPU code by incorporating auxiliary arrays. This approach not only facilitated a more efficient modification

of the array a values but also contributed to the alleviation of the performance bottleneck previously associated with the use of atomicAdd. This strategic enhancement was improving the overall efficiency of our GPU-accelerated computations.However, the execution time was still high in our point of view. We decided that it was important to add auxiliary arrays in order to optimize the memory access patterns, force accumulation, and potential energy computation, collectively improving the overall efficiency of the GPU-accelerated code. We decided to use the following auxiliar arrays :

- Shared Memory Optimization **shared**
- Accumulation of Forces **d_aux**
- Potential Energy Computation **vPot_local**

At this stage and using 512 threads per block (the best combination for threads we got) we got about 4 seconds usr time(this time can be higher than real one because this values were calculated at a point where the cluster was full).

The structure of the final CUDA implementations for the two key functions (ComputeAccelerations) and (ComputeAccelerationsGPU) are explained below:

A. Host Function (Initialization - computeAccelerations())

This function serves as the entry point of the GPU-accelerated computation. Key steps in this function include:

- Initializing arrays and variables on the host .
- Allocating memory on the GPU for the necessary arrays (r_Cuda, a_Cuda, and Pot_Cuda).
- Copying input data from the host to the device.
- Calculate the number of block to use based on the N
- Launching the GPU kernel (computeAccelerationsGPU) with an appropriate configuration of blocks and threads.
- Handling potential errors that might arise during kernel execution.
- Copying the results back from the device to the host.
- Accumulating the potential energy values.

B. Kernel Function (computeAccelerationsGPU())

As previously highlighted, the functionality of computing accelerations within our codebase is intricately linked to the invocation of the computeAccelerationsGPU CUDA kernel. This kernel undertakes the parallelized computation of accelerations and potential energy for a system of particles, capitalizing on the parallel architecture of the GPU to optimize performance. The ensuing discussion delves into a comprehensive breakdown of the key features and operations encapsulated within this critical CUDA kernel, shedding light on its pivotal role in enhancing the efficiency of our computational processes.The function can be divided into this points:

- Thread Indexing
- Shared Memory (Initialise the shared Array with the **r_Cuda** array values
- Parallel Computation(Calculate all the values needed for each Thread)
- Accumulation(Compute variables of this function **(Pot_Cuda)** and array **a_Cuda** based on local values calculated for each thread)

#Threads	Real Time (s)	CA(s)	MemCPY HtoD(ms)	MemCPY DtoH(ms)
1	28.424	24.0097	8.9061	5.0504
2	14.969	12.5166	8.6255	5.0681
4	9.160	6.75693	8.6269	5.0518
8	7.877	3.87725	8.9091	5.0344
16	6.262	1.97220	8.8995	5.0796
32	4.807	1.00286	8.9111	5.0423
64	5.094	1.00194	8.9084	5.0434
128	3.487	1.00188	8.6281	5.0593
256	3.481	1.00179	8.6234	5.0818
512	3.457	1.00236	8.6271	5.0581
1024	5.248	1.14664	8.9032	5.0403

TABLE III
EXECUTION TIME (CUDA) MEASURED WITH NVPROF AND TIME

IX. ANALYSIS OF THE CUDA CODE

Concluding our journey in code optimization, we manipulated the number of threads from 1 to 1024, observing its impact on the overall execution time. Our focus centered on crucial metrics, including the execution time of the computeAccelerations function on the GPU and the duration dedicated to data transfer between the host and the device. The results of these experiments are succinctly presented in Table 3, as well as in Figure 2 in visually engaging manner.

X. COMPARATIVE ANALYSIS OF THE DIFFERENT CODE VERSIONS

With the aim of comprehending the optimizations implemented in the code and the entire process from the initially provided code to the finalized version, we embarked on a series of comprehensive tests. These tests were conducted to observe variations in the profiling of the code and draw conclusions about the obtained results. This systematic approach allowed us to analyze the impact of each modification on the overall performance and efficiency of the code, providing valuable insights into the effectiveness of the optimizations applied throughout the developmental stages.

In this section, we will delve into tests conducted on the final product of each developmental phase, aiming to unravel the optimizations introduced through parallelism at both the CPU and GPU levels in our case study. These tests were designed to provide a comprehensive understanding of the performance enhancements achieved through parallel computing strategies. By scrutinizing the outcomes of these tests, we sought to discern the impact of parallelism on both the CPU and GPU, shedding light on the efficacy of the implemented optimizations in the context of our specific case study.

A. Execution Time

In this step we are going to compare the execution time since the first version (parallel code) until the last version (GPU version) , including the OpenMP one. Theoretically the best version we should achieve should be the GPU One. However, probably because we did not get to the best version of the code, or the GPU is not the best option in this case of study, the best execution times we got, with N=5000 was

#Particles(N)	CUDA(s)(512 T)	OpenMP(s)(32 T)	Sequential(s)
1000	2.428	0.118	1,348
2000	3.989	0.330	5,350
3000	4.590	0.665	12,028
4000	4.747	1.201	21,367
5000	3.650	1.577	33,381
6000	3.650	2.150	48,074
7000	5.400	2.836	65,437
8000	4.306	3.637	68,079
9000	6.125	5.655	86,099
10000	6.1609	6.951	105,757

TABLE IV
EXECUTION TIME (S) CUDA VS OPENMP VS SEQUENTIAL

the OpenMP version. Below we have a Table that shows the variations of time in all versions we implemented:

To obtain the results presented in this table, it was necessary to execute the code for various phases multiple times, adjusting the number of particles and even the defined **MAX_PART**. However, in terms of computational efficiency, this table stands out as the most crucial one. It has significantly contributed to our enhanced understanding of the program's performance and scalability as the number of atoms increases.

As we can see in the table , as well as in figure 3, the Sequential function in the worst in terms of execution time, its the worst one. as the level of N increases also the time execution increases. This can be explained by the total complexity of our global program that is given by this expression.

$$\theta(N) = N^3 + 2N^2 + 14N$$

When comparing the execution times between the OpenMP and CUDA implementations, we observed variations. For number of atoms ranging from 1000 to 8000, the OpenMP version with 32 threads proved to be more time-efficient than the CUDA version with 512 threads. This difference in performance may be caused by the overhead associated with managing a higher number of threads in the CUDA version. The cost of thread creation, synchronization, and coordination might have outweighed the benefits of parallelism for this N values.

Starting from n=9000, the trend shifted, and the CUDA implementation exhibited better performance than OpenMP. This change might be attributed to the increased parallelization capacity of CUDA, allowing it to efficiently handle larger problem sizes. The architecture of CUDA-enabled GPUs often excels in tasks with high degrees of parallelism, which could explain the improved performance as the number of particles increased.

Although we didn't perform a detailed analysis beyond n=10000, it is expected that the CUDA implementation's performance will continue to outshine OpenMP. This anticipation arises from CUDA's ability to leverage the parallel processing power of GPUs effectively. As the problem size increases, CUDA's architecture is likely to provide better scalability and computational efficiency compared to OpenMP on the CPU.

XI. CONCLUSIONS

In conclusion, our experimentation involved comparing the execution times of three different versions: a sequential implementation, an OpenMP parallelized version, and a CUDA-accelerated version. Surprisingly, the sequential approach exhibited the longest execution times, emphasizing the necessity for parallelization in our computational tasks. The OpenMP implementation demonstrated a notable improvement over the sequential version, showcasing the efficacy of parallel processing in reducing computation time.

Contrary to expectations, the CUDA-accelerated version initially did not outperform the OpenMP parallelized version in our specific implementation, especially for smaller values. However, as the problem size increased beyond a certain threshold, CUDA exhibited superior performance. This shift could be attributed to the parallel processing power of GPUs becoming more advantageous for larger workloads, showcasing the scalability and efficiency of CUDA in handling substantial computational tasks.

Overall, we are pleased with the final results, which demonstrate notable progress at each stage of the project. The CUDA implementation posed a significant challenge, marking our inaugural foray into GPU programming. The pivotal aspect of our work centered around the comprehensive comparison of the three versions. This allowed us to conduct a thorough scalability analysis, probing the behavior of our code across varying input values.

For future work, potential enhancements include delving into additional metrics such as memory usage or even the number of cache misses, and expanding the scope of comparisons between the CUDA, sequential, and parallel versions. This endeavor could provide a more comprehensive understanding of the computational landscape and inform further optimizations. We gained valuable insights into diverse methods for assessing code performance and identifying avenues for improvement, carefully considering the specific context of each situation. This experience has equipped us with a nuanced understanding of evaluating code efficiency and determining optimal strategies tailored to the given scenario.

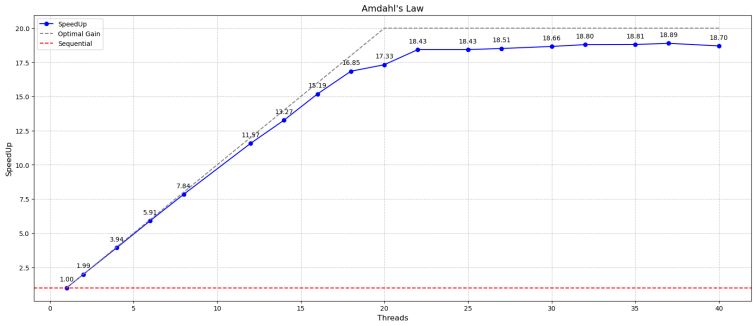

Fig. 1. Threads vs SpeedUp (OpenMP).

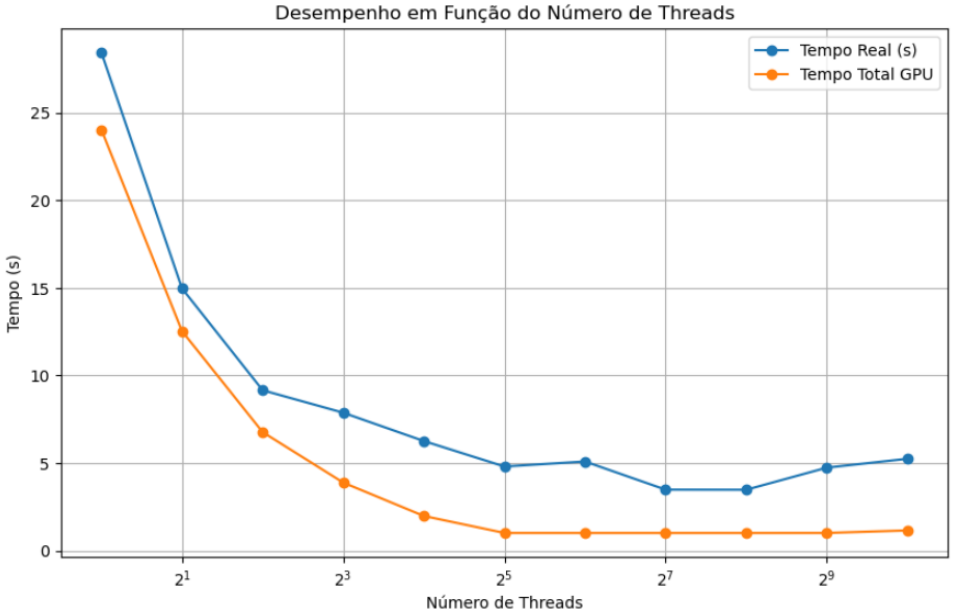

Fig. 2. Time vs Nº Threads(CUDA).

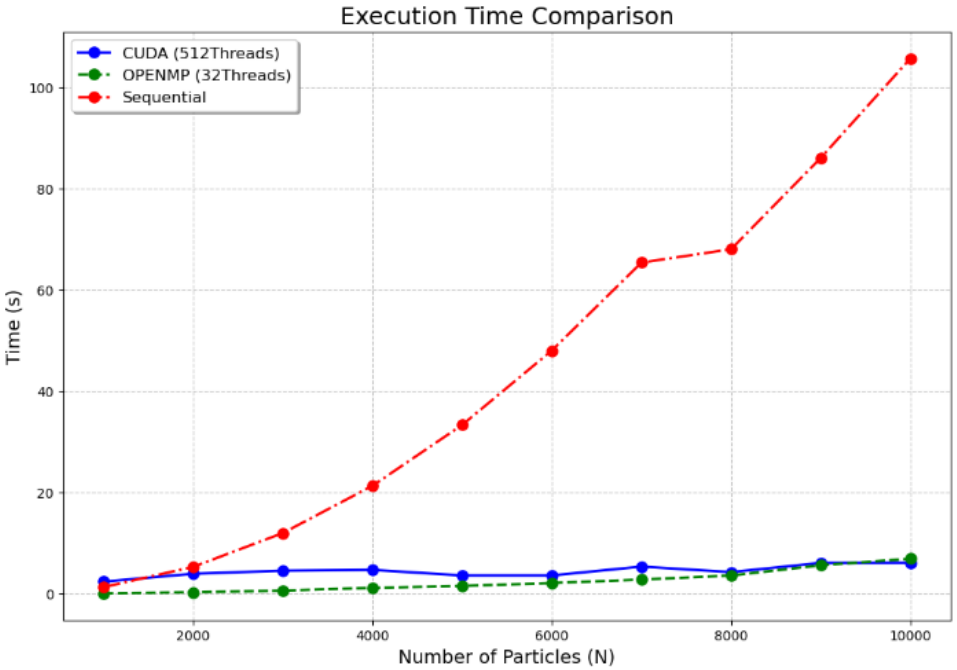

Fig. 3. Execution Time vs Number of Particles (Sequential vs CUDA vs OpenMP).