Parallel Computing - Work Assignment (Phase 3)

José Moreira
Software Engineering
University of Minho
Braga, Portugal
pg53963@alunos.uminho.pt

Santiago Domingues
Software Engineering
University of Minho
Braga, Portugal
pg54225@alunos.uminho.pt

***Abstract*—This document aims to identify, explore and explain all the stages covered during the realization of the assignment developed during the curricular unit of Parallel Programming. The concepts approached will contribute to the deep comprehension of the domain, allowing the clarification of the most important aspects.**

***Index Terms*—parallelization, CUDA, GPU, accelerators, metrics, performance**

I. Introduction

This document serves as the report for the third and last phase of the **Parallel Computing** work assignment, whose goal was to improve the previous use of the OpenMP directives with the implementation of a new version using accelerators (GPUs) or using distributed memory with MPI, in order to enhance the performance of the program.

On further discussion, the group decided to execute the version using accelerators because we identified a greater potential for performance enhancement through parallel processing using GPUs. In this fase, our focus was to restruct the program to effectively integrate GPU acceleration, optimize critical sections, and offload computationally intensive tasks to the GPU cores.

Throughout this phase, rigorous testing and benchmarking were conducted to assess the efficacy of the accelerated version. We strategically combined a set of tests to assess some aspects like the performance of the program under different input sizes, different machines and different computational loads to analyze the program's scability.

II. Code Analysis

The program, which simulates the behaviour of argon gas atoms, follows fundamental principles of physics, including Newton's laws and the Lennard Jones potential model.

Through the first two assignments, the group made some improvements, using different techniques of optimization taught in the curricular unit mentioned before. Those steps made could offer a solid initial step for the last phase of the assignment.

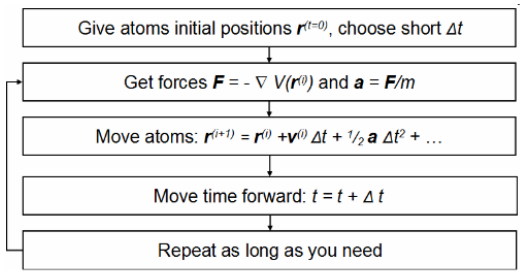

Fig. 1. Particle movements simulation over time steps, using the Newton law

III. First phase

On the initial phase of our assignment, the main focus was on improving the given code, using a single threaded program. So, the most important step was on improving the sequential code given. Profiling it, evaluating its performance based on some important aspects, like CPU execution time, number of instructions, number of clock cycles, heaviest functions in terms of calls and self execution time, allowed us to identify the heaviest parts of the code and the focus of the improvement yet to be made. Some of these steps were made using **gprof**, where we only represent the heaviest ones, that made us inspect, optimize and evaluate.

TABLE I
Initial gprof profiling

Function	%time	Seconds	Calls
Potential()	61.62	22.64	201
ComputeAccelerations()	38.46	14.13	202

Once the group identified the most important sources of optimization, the modifications started. We are going to list the most important steps made, so we can correlate easier with the results that emerged from it:

1) addition of some important flags to the Makefile: **-g -fno-omit-frame-pointer -pg -O3 -march=native -mavx -ftree-vectorize -msse4**
2) simplification of calculations in **Potential()** and **ComputeAccelerations()**
3) merge of **Potential()** and **ComputeAccelerations()**

TABLE II
GPROF PROFILING

Step	Seconds	Instructions (10^{12})	Cycles (10^{11})
0	247.1275	1.2568	7.9962
1	175.4322	92.2153	60.2115
2	9.2251	2.2412	2.7532
3	5.075	2.0801	1.6194

All the results were obtained through the mean of ten measurements, where the highest and the lowest values were dropped. As expected, the step 0 belongs to the initial phase of the assignment, corresponding to the original value.

This first and important phase made us understand how some heavy maths calculations can have a big impact on the code. Once it was a simple and initial phase of the assignment, it could give the group the notion of dealing with this kind of simple problems, allowing us to understand how to reduce, drastically, the number of instructions made by a program, giving a really easy and fast way of improving code performance or, in a most specific way, improving the sequential part of a program that is, a lot of times, the bottleneck of a algorithm.

IV. SECOND PHASE

In this phase our goal was to improve the previous phase with the use of the OpenMP and its directives. Our group felt compelled to embark on an in-depth research endeavor aimed at thoroughly exploring and comprehending the functionalities of OpenMP. Armed with this knowledge, we successfully applied its directives to our program. This strategic integration allowed us to capitalize on OpenMP's features, leveraging its directives to optimize parallelism and improve the program's overall performance.

By adeptly incorporating OpenMP's directives, our objective was to harness its power in streamlining computational processes. This deliberate implementation aimed not only to improve performance but also to establish a more efficient and scalable framework for our program.

We began with the use of simpler and more fundamental directives to introduce the practical effects of OpenMP in the current program. For that we used directives such as **#pragma omp parallel for** and **#pragma omp critical**. Throughout the program, we recognized the necessity of employing advanced and more complex directives such as **#pragma omp parallel for schedule(dynamic, 40)** and **#pragma omp parallel private(j) for reduction()**, in order to explore all of the OpenMP capacities to improve our program.

With the need to ensure the effectiveness of our modifications, we conducted a series of tests, to assess the scability of the optimized code. The main goal was to enhance the execution time across varying thread counts. Testing was performed for all program phases, employing one, two, four, eight, twenty, and forty threads to cover a broad spectrum of scenarios. We meticulously analyzed the results to verify that the code modifications not only enhanced performance at distinct thread counts but also demonstrated scalability across various levels of parallelization. Every phase of the program underwent individual testing with diverse thread counts to confirm that scalability enhancements were not confined to specific phases.

With the purpose to ensure that the modifications improved the execution time of the program we compared the optmized code in **MDpar.cpp** with the original sequential version **MDseq.cpp**. This comparative analysis aimed to validate that the parallelized version of the code not only improved the execution time but outperformed the sequential version across different scenarios.

The subsequent step involved the organization of the data gathered from the comparison done above. To accomplish that, we constructed the following table:

TABLE III
RESULTS

Threads	Time (s)	#I	Cycles	Speedup (x)
2	14,026 +- 0,344	9,605	6,976	8,664
4	7,061 +- 0,169	9,617	7,005	17,210
8	3,603 +- 0,085	9,650	7,083	33,727
20	1,569 +- 0,034	9,840	7,557	77,400
40	1,458 +- 0,012	10,152	13,772	83,345
Sequential	121.517 +- 4.200	9.363	7.684	-

Note that all of these results were rounded to three decimal places and the values in the **#I** and **Cycles** columns of the table correspond to numbers in scientific notation (1E10). Is important to refer that the group made tests with fifty threads, but the results were worse in comparison to the forty threads test.

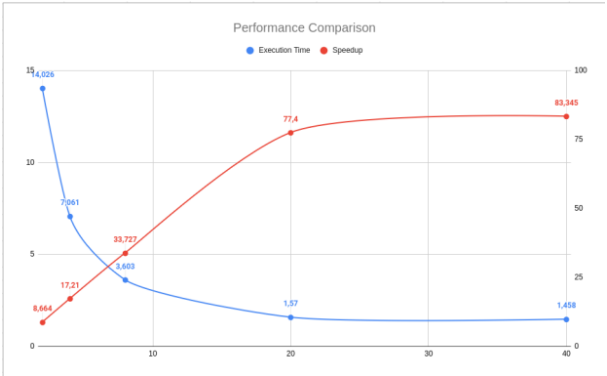

Fig. 2. Performance comparison

As it's possible to check, the optimizations of the parallel code were correctly implemented, as expected the total time for the parallel program execution was significantly reduced compared to the sequential version. In the best case scenario, which is the test with forty threads, we can see a speedup of

approximately 83, that means that the parallel version executes 83 times faster than the sequential version.

In this way were were able to assure that all of the optimizations done were very effective in the main goal of this phase, nevertheless regarding the fact that OpenMP works with shared memory there were some situations that needed a more detailed attention such as data races, that were a serious concern in the development of our program, because they could lead to inconsistent and incorrect results.

V. Third Phase

As mentioned before, the group had a lot of interest in exploring the **GPU** version, once it showed a lot of potential of improvement. All this phase aimed on a critical choice of a set of tests that could make us understand the behaviour of our implementation. All of those tests made possible the careful analysis of the scalabilty of the algorithm and all its dependent results.

A. CUDA

CUDA, or Compute Unified Device Architecture, stands as a groundbreaking framework developed by **NVIDIA**, revolutionizing the landscape of parallel computing. Initially introduced to enhance graphics processing, CUDA has evolved into a versatile platform that harnesses the immense power of GPUs for a broad range of computations beyond graphics rendering.

At its core, CUDA enables developers to offload computationally intensive tasks onto the massively parallel architecture of NVIDIA GPUs. Unlike traditional CPUs, which excel in sequential processing, GPUs excel in parallelism, allowing thousands of cores to work simultaneously on data sets. This parallel processing capability makes CUDA particularly well-suited for tasks like scientific simulations, data analysis, machine learning, and deep learning.

The key to CUDA's effectiveness lies in its programming model, which extends the widely-used C and C++ languages. Developers can write CUDA kernels, which are functions executed on the GPU, to perform computations in parallel. This seamless integration of parallelism into familiar programming languages facilitates the development of highly efficient and scalable applications.

CUDA has become an indispensable tool for researchers, scientists, and developers working on computationally demanding projects. Its impact is evident in various fields, from simulating complex physical phenomena to training sophisticated machine learning models. CUDA's influence extends beyond academic and research settings, as it has played a pivotal role in driving advancements in artificial intelligence, computer graphics, and scientific computing.

In conclusion, CUDA represents a paradigm shift in computing, unlocking the tremendous potential of parallelism on GPUs. Its adoption has paved the way for accelerated and scalable applications, marking a significant milestone in the journey towards achieving unprecedented computational capabilities. As technology continues to evolve, CUDA remains at the forefront, driving innovations that redefine the boundaries of what's possible in the realm of parallel computing.

B. Implementation

As the need of writing code to run on GPUs, the group had to start with learning a lot if directives, documentation and other stuff related to **CUDA**. After that important step, we had to measure the piece of the code that could lead to a better optimization of performance. This code had to be a software fragment with a lot of parallelization potential. The **ComputeAccelerationsPotential()** function showed how heavy it was in terms of execution time and, as well as seen in the second phase of the assignment, its enormous capacity of parallelization. As an agreement, we decided to work on that function, having the responsability of splitting it in two another main functions, one having the job of configuring all the stuff the GPUs needed to run and, the second one, being the function that would run in the kernel.

1) Host function: This specific function is the one that prepares the execution scenario of the kernel. It begins with memory allocation, through CUDA, does some content copying for the devices to run, calls the kernel function, responsible for making the heavy calculations and, in the end, copying the results to the host memory. It is really important once it can detect a lot of CUDA errors, allowing us to understand and correct our code. From another point of view, it is on that function that we can manipulate the grid of the GPUs, or, in other words, the number of existing threads in each block and the number of blocks in the grid. As a choice, the group decided to use one dimension therads, being, each one, responsible for working with a single line of the matrix in the function.

2) Device function: This one is called by the other function already specified. It is the **__global__** one that runs on GPU. As it was said before, this fragment of code is responsible for forcing each thread of doing an entire for cycle, being the execution member of a matrix line. As it can offer a lot of strategies, we decided to use a **__shared__** variable, r, reducing, by a lot, the number of accesses, per block, of the threads, to the global variable, **r**. Being another implementation, we decided to reduce the accesses of the thread to the global matrix, **a**, creating a local variable that has the responsability of keeping the iteration value, assigning it to the matrix position only at the end. In a first approach, the group was forced to use **atomicAdd** clauses to guarantee there were no data races damaging and putting in risk the integrity of the final results. For sure, this type of meaning words have a lot of impact in the performance of the threads, once they block them for a while. With a lot of brainstorming, the final decision was to stop crossing only a part of the matrix (optimization

that was made on the first phase of the assignment, where crossing only less then a half of it became really good in terms of performance), starting make the calculations in all the positions of it. This update resulted in a really good improvement, as it was expected by the group. All of this improvements were made based on a lot of replicated tests that made us be sure of the changes.

C. Tests and Analysis

As one of the most important phases of this assignment, the evaluation and performance discussion, the group decided to implement some specific tests, with a good power of feedback. All of them are going to be explained in detail and discussed.

1) Threads per Block: The first and more obvious test we made was the variation of the number of threads in each **GPU block**. This was done using a number of threads that corresponds to a multiple of two.

TABLE IV
RESULTS

Threads	Real Time (s)	GPU activities (s)	API calls (s)
4	8.842	5.062	5.136
8	5.485	2.898	2.943
16	5.462	1.473	1.545
32	4.830	0.749	0.805
64	4.643	0.749	0.802
128	3.265	0.749	0.797
256	4.630	0.754	0.806
512	6.771	0.777	0.825

The analysis of execution times concerning the number of threads per block in **CUDA** provides valuable insights into the optimization of parallel computations. Observing the trend across different thread configurations reveals a nuanced relationship between parallelism and performance.

Starting with a minimal number of threads (4 per block), the execution time is noticeably high, indicating underutilization of the **GPU's** parallel processing capabilities. As the thread count per block increases, a substantial reduction in execution time is observed until a certain point. The most significant improvement occurs when transitioning from a small to a moderate number of threads (4 to 8 per block), demonstrating the initial benefits of parallelization.

However, as the thread count continues to increase, diminishing returns become apparent. The most substantial reduction in execution time is achieved when moving from 64 to 128 threads per block, signifying an optimal configuration for the specific hardware. Beyond this point, further increases in the number of threads result in fluctuations and, in some cases, deterioration in performance.

The analysis suggests the importance of striking a balance between maximizing parallelism and avoiding resource oversaturation. The sweet spot appears to be around 128 threads per block, where the execution time is minimized, and the GPU resources are efficiently utilized. This finding underscores the critical role of experimentation and fine-tuning when optimizing CUDA algorithms.

The relationship between the number of threads per block and execution time is complex, and the optimal configuration depends on various factors, including the specific GPU architecture and workload characteristics. A thoughtful and iterative approach to tuning thread configurations is paramount to unlocking the full potential of parallel computing in CUDA.

2) Speedup: Speedup is a performance metric that quantifies how much faster a parallel program executes compared to its sequential counterpart. It is calculated as the ratio of the sequential execution time to the parallel execution time. In this part of the job, we took the sequential execution time measured in the second phase of the assignment so we could make our comparisons and based our approach in **Amdahl's law**.

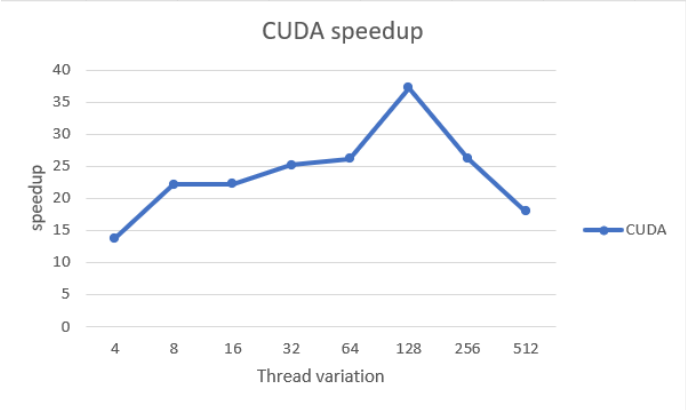

Fig. 3. CUDA speedup

Just like the last table, we could understand that all hardware has its capacities and it will start to perform worse once it reaches them. The speedup, as it was expected, gave us a really good feedback of our parallel **CUDA** implementation.

3) CUDA vs OpenMP: As a second approach, we found relevant comparing different implementations. This kind of job could help us understand how, sometimes, depending on the problem, the choice of an architecture is really decisive in terms of performance.

By the graph given, we can take a lot of conclusions. In first place, we can see **OpenMP** runs in less time than **CUDA** in all the observations made. This can mean that OpenMP is more effective when we are working in a not very large scale. Another thought that can sustain our argument is that, with the increase of the value of the variable **N**, the CUDA execution time starts to soften, while the OpenMP value of its execution starts increasing a lot. We would like to make more tests (with a bigger N value), but it wasn't possible once the algorithm didn't allow us. Concluding this analysis, we consider the dimension of the problem assumes itself as a really important factor in the choice of a implementetion method.

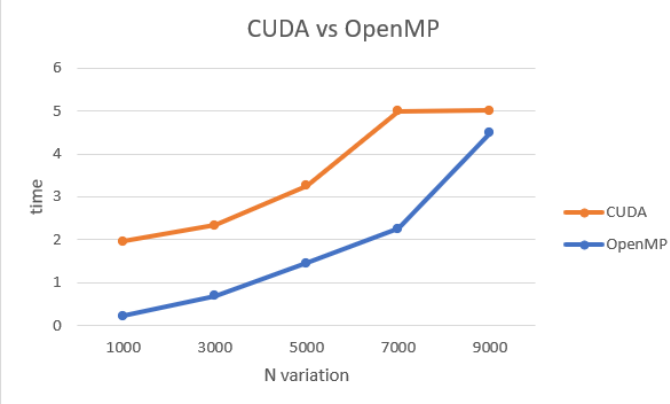

Fig. 4. CUDA vs OpenMP

D. Scalability efficiency

Scalability efficiency is a measure that evaluates how well a parallel program benefits from an increase in the number of processors, threads, or processing units. It is typically expressed as the ratio between the achieved speedup and the number of processors or processing units added. We searched for the speedup values calculated before so we could make a scalability efficiency calculation without any deviaton, getting the best analysis from it.

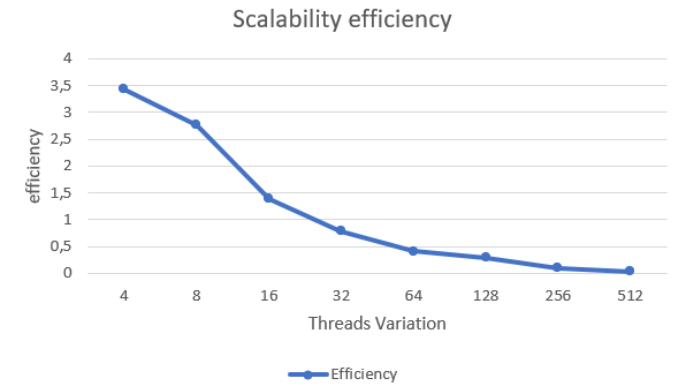

Fig. 5. Scalability Efficiency

There are a lot of reasons for justifying the consecutive decreasing of the speedup based on the number of threads used. It is expected, once we have to beat some bottlenecks, increasing a lot the number of logical processors used. In first place, despite not having abundant objective primitives of synchronization, there are a lot of things programmers can't control. The synchronization between threads can be one of them. In other hand, the overhead of creating extra threads can decrease significantly the efficiency of the machine. The process of handling with a lot of threads can be most heavy than the algorithm cost itself. In last place, but not less important, the limitations in the resources of the machine, like it was explained before, can decrease a lot the overall performance.

VI. Conclusion

In conclusion, our exploration of Parallel Computing, particularly through the phases involving CUDA implementation, has been a journey marked by rigorous testing, insightful analyses, and the harnessing of GPU acceleration.

The tests conducted, such as the variation of threads per block and the comparison between CUDA and OpenMP, provided a granular understanding of how different configurations impact performance. The examination of threads per block showcased the delicate balance required for optimal parallelization, revealing a peak in performance around 128 threads, beyond which diminishing returns became evident. This nuanced relationship emphasizes the importance of tailored experimentation and fine-tuning in CUDA algorithms.

The comparison between CUDA and OpenMP shed light on the context-dependent nature of parallelization choices. While OpenMP proved more efficient for smaller-scale problems, CUDA exhibited its prowess with larger data sets, showcasing the adaptability of parallel computing methodologies based on the specific characteristics of the problem at hand.

The speedup analysis, particularly when comparing CUDA with the sequential version, showcased the significant performance gains achieved through GPU acceleration. The visual representation of speedup provided a clear and compelling narrative of how parallelizing computationally intensive tasks on a GPU can lead to substantial improvements in overall execution time.

Moreover, the scalability efficiency analysis delved into the challenges inherent in scaling parallel programs. While the implementation of CUDA allowed us to exploit the power of GPU parallelism, factors such as synchronization overhead, thread creation costs, and resource constraints became more pronounced as the number of threads increased. This analysis underscored the importance of not only achieving speedup but also ensuring that scalability is sustained across a range of computational resources.

In summary, our tests and analyses, coupled with the CUDA implementation, have illuminated the intricate landscape of parallel computing. The insights gained from these experiments provide a solid foundation for future endeavors, emphasizing the need for thoughtful consideration of problem characteristics, hardware capabilities, and algorithmic nuances in the pursuit of optimal parallelization. As we continue to navigate the realms of high-performance computing, the lessons learned from this assignment will undoubtedly shape our approach to addressing complex computational challenges.

References

[1] Sobral J., "Molecular Dynamics simulations: A short overview," Univ. Minho, Accessed on January 6, 2024.

[2] Navarro Cristobal A, Hitschfeld Nancy, Mateu Luis, "Molecular dynamics simulations: advances and applications," Dovepress, Accessed on January 7, 2024.

[3] "Understanding NVIDIA CUDA: The Basics of GPU Parallel Computing," Accessed on January 9, 2023.