Parallel Computing - Work Assignment 3

Gabriela Santos Ferreira da Cunha
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg53829@alunos.uminho.pt

Nuno Guilherme Cruz Varela
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg54117@alunos.uminho.pt

***Abstract*—This document consists in presenting and justifying the decisions that were made during the optimisation of a code that simulates particle movements over time steps, using the Newton law.**

***Index Terms*—optimisation, profiling, performance, sequential, instruction level parallelism, shared memory parallelism, openmp, thread, cuda, gpu**

I. Introduction

In an era where computational demand escalates alongside the complexity of tasks, optimising code efficiency is not just beneficial; it is imperative. In the practical work assignments of Parallel Computing course, it was proposed to explore optimisation techniques applied to a program and to improve and evaluate its final performance (execution time). Thus, a three-phased approach was followed, each phase targeting a different level of computation.

The journey started with Instruction Level Parallelism (ILP), where we refined code to exploit the potential of modern processors fully and achieve maximum efficiency at the instruction level. This phase went beyond traditional ILP techniques by incorporating a focus on memory hierarchy and vectorisation. By aligning the code with the processor's memory architecture, we reduce cache misses and memory latency, significantly enhancing data access speed. Concurrently, vectorisation is employed to transform operations into vector instructions, allowing the simultaneous processing of multiple data points with a single instruction.

The second phase elevated the code optimisation process from single-core execution to multi-core processing efficiency using OpenMP directives. In this phase, we adapted the code to exploit the capabilities of multi-core architectures by parallelising processes and distributing workloads across available cores.

In the final phase, we were presented with 3 distinct options: i) refine the existing OpenMP implementation; ii) create a new version tailored for GPU accelerators; iii) develop a new version designed for distributed memory systems. Therefore, we explored GPU acceleration with CUDA, extending our optimisation techniques to harness the massive parallel processing power of NVIDIA's GPUs. The code was adapted and restructured to run efficiently on the GPU, focusing on optimising kernel execution, memory transfers and effective utilization of the GPU's parallel architecture.

In this report, we will address all relevant information concerning all assignments but give special emphasis to the third or final phase, since the development of the other phases was always accompanied by a report presenting, detailing and justifying the decisions.

II. Code Profiling

Before starting the optimisation process, it is crucial to identify performance bottlenecks, inefficient segments and areas where improvements can yield significant benefits. Without code profiling, optimisation efforts can be misdirected, focusing on parts of the code that do not significantly impact overall performance. In order to make our optimisation task more targeted and productive, we may analyse the initial code provided that simulates particle movements over the time. Using the Newton law, the code follows the next approach:

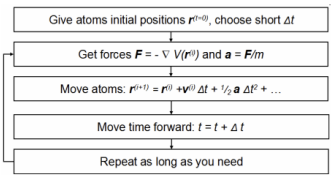

Fig. 1: Approach to simulate particle movements.

Initially looking at the code, we observe that `Potential` and `VelocityVerlet` functions iterate over all pairs of particles, exhibiting a $\mathcal{O}(N^2)$ complexity, where N is the number of particles. These functions are the ones which potentially will be the hotspots of the program. However, to substantiate this observation and accurately pinpoint the performance bottlenecks, we employed the use of a profiling tool - `gprof`.

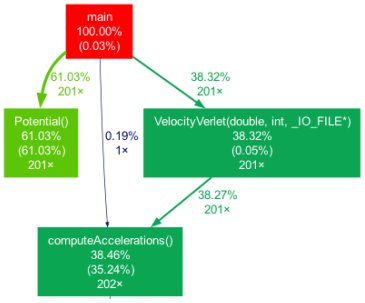

Fig. 2: Call graph using *gprof* tool.

The presented graph confirms our expectations as those functions are responsible for the high execution times. The `Potential` function spends 61.03% of execution time, while `computeAccelerations` is responsible for 35.24%, totalling 96.27%. According to Amdahl's Law, optimising the parts of a program where the most time is spent yields the greatest overall performance improvements. So, in the context of this code, focusing our efforts on these functions aligns with Amdahl's principle. By improving efficiency in this sections we can expect a substantial impact on the overall performance.

III. Sequential Version

A. Optimisation flags

To improve ILP, we started by testing compilation with optimisation flags: `-O2`, `-O3` and `-Ofast`. The optimisation level 2 includes various optimisations such as inlining functions, loop optimisations and instruction scheduling. The optimisation level 3 includes more aggressive optimisations and can lead to better performance but at the cost of longer compilation times. The `Ofast` sacrifices some adherence to strict language standards in exchange for maximum performance, so it may perform optimisations that could change the semantics of the code.

B. Less cycles

After an observation in the `Potential` function, we concluded that there were unnecessary cycles because the inner loop was starting in the index 0. For example, for the pairs $(1, 3)$ and $(3, 1)$ the computation is the same, so we may simplify and deal with this two pairs in a single iteration multiplying by 2. The inner cycle is now initiating in $i + 1$. This way, both memory hierarchy and ILP were improved by saving a lot of memory accesses that eventually could generate misses and unnecessary calculations.

C. Arithmetic operations

As the exponentiations were all with relatively small integer exponents, we replaced `pow` calls with direct multiplications. Calling `pow` and `sqrt` functions involves overhead related to parameter passing, stack operations and the function call itself, so we tried to avoid calling these functions due to their performance bottleneck.

Still regarding arithmetic operations, we also focused on reduce the number of divisions as it is the most complex basic operation, requiring more clock cycles to finish.

According to Lennard-Jones potential, ϵ (field depth) and σ (distance where the field is null) are specific for each material but commonly set to 1 in simulations. Then, by identity property, there is no need to multiply them. We also intended to remove the use of the `sqrt` operation due to its inefficiency. Hence, we made the following simplification to calculate the potential:

$$\phi(r) = 4\epsilon \left(\left(\frac{\sigma}{\sqrt{r^2}} \right)^{12} - \left(\frac{\sigma}{\sqrt{r^2}} \right)^{6} \right) = 4 \left(\frac{1}{r^{12}} - \frac{1}{r^6} \right) \quad (1)$$

It is important to remember that the number of iterations was halved so in each iteration we are calculating $2 \times \phi(r)$. Since $r^6 = (r^2)^3$ and $r^{12} = (r^2)^3 \times (r^2)^3$, we can put $\frac{1}{(r^2)^3}$ in evidence. For each iteration, the potential is now calculated by the following equation:

$$\psi(r) = 2 \times \phi(r) = 8 \times \frac{1}{(r^2)^3} \times \left(\frac{1}{(r^2)^3} - 1 \right) \quad (2)$$

D. Memory hierarchy

Increasing the data and temporal locality of a program can have a huge impact on its performance. In this case, the initial code has already a good data and temporal locality since the inner cycles iterate over all the numbers greater than the one that is fixed on the outer loop. Despite that, we could still improve in terms of memory hierarchy, reducing memory accesses, also contributing to minimize cache misses. For that, we used an accumulator to update accelerations in `computeAccelerations`, avoiding repeatedly accesses to the array.

E. Joining functions

One of the major changes to improve the execution time was joining the function that computes the accelerations to the one that calculates the potential energy. Initially, we verified if it was possible to make this improvement, confirming that the information `Potential` needed was not updated after `computeAccelerations` was called. This change could be made because both functions were iterating over the same data structure, `r`, and some of the calculations were the same, namely the calculation of the square of the distance (r^2) and its square root. As a result, we saved a lot of memory accesses and calculations.

F. Vectorisation

Another optimisation was introducing data parallelism. This technique involves executing the same operation on different pieces of distributed data simultaneously, being particularly effective for tasks where the same set of instructions needs to be applied to each element of a large dataset.

At the beggining, we started to vectorise the code manually with the objective of getting the maximum benefit of it. In order to do that, we had to transpose the matrices we were using. This way, data is stored contiguously in memory: all x elements together, all y elements together and all z elements together. This is beneficial for vectorisation because modern CPUs can load contiguous data into vector registers more efficiently than non-contiguous data. This also contributes to memory hierarchy, since there are fewer cache misses and better utilization of the cache lines, leading to faster data retrieval. We also had to implement memory alignment which ensures that the starting address of the data is a multiple of 32 (size of a vector of doubles in AVX). Aligned data can be loaded into vector registers without performing any additional operations to account for potential misalignment. Accesses are typically faster because they do not cross cache

line boundaries, which can cause additional cycles to resolve the misalignment.

However, manual vectorisation worsens the code readibility. Thus, we decided to remove it and put that responsibility on compiler by using the `-ftree-vectorize` and `-mavx` flags. We used the AVX (256-bit vectors) extension to take better advantage of vectorisation instead of using SSE (128-bit vectors). For that, we kept the transposed matrices but did not need the data to be aligned anymore.

IV. Parallel Versions

A. OpenMP

Modern CPUs have multiple cores, each capable of running separate threads. Thread parallelism allows programs to utilize these cores concurrently, significantly improving performance over single-threaded applications. By distributing tasks across multiple threads, this can lead to more efficient use of the CPU, as it minimizes idle time and maximizes the CPU's workload capacity. To implement thread parallelism, we used OpenMP directives which is an easy way to parallelise a program, as it requires minimal changes to the code.

A process splits into several threads, all sharing the same memory space. Each thread represents a separate path of execution, but since they belong to the same process, they can access the same data and resources. Our hotspot, that is now computing accelerations and potential, has 2 nested `for` cycles which iterate over all pairs of particles. For each iteration, both potential accumulator and accelerations array are updated. We must pay attention to these updates because they will lead to data races.

The selected approach consists in exploring parallelism in the outer cycle. In this approach we start by introducing the directive `#pragma omp parallel for`. This allows the compiler to distribute the loop iterations among multiple threads. However, as we mentioned before, we have to take a look at the potential and accelerations structures because they will be shared by the threads and this will cause data races.

The OpenMP offers 3 directives to solve data races: `reduction`, `atomic` and `critical`. We aimed to use the `reduction` due to its better performance comparing to the other ones.

1) Atomic and Critical Directives: The first approach to deal with the data races was using `atomic` and `critical` directives because of their simplicity. Although, as we expected, it was not a good strategy due to the overhead associated with synchronisation mechanisms. The excessive use of these directives leads to performance degradation because threads spend more time waiting for access to the shared resources than actually performing useful work. In this case, the parallelised version had higher execution times than the sequential one as the referenced directives were used excessively.

2) Reduction: To solve this question in an efficient way, we decided to use a `reduction` in the potential accumulator and in the accelerations array by introducing the directive `#pragma omp parallel for reduction(+:potentialAcc, a)` which avoids the data races. The `reduction` primitive creates a private copy for each thread and in the end of the parallel section the results are reduced into a single variable.

In OpenMP, we can instruct the CPU to use scheduling strategies which determine how the iterations or tasks are allocated to threads during the execution of a parallel region. Static scheduling is commonly used as the default for parallel loops. This means the compiler determines the distribution of loop iterations at compile-time. The loop iterations are divided into chunks, and each thread is assigned a chunk of iterations to process.

On the other hand, in dynamic scheduling, the distribution of loop iterations is determined at runtime. Threads request work from a shared pool, and the runtime system assigns them a small unit of work. Though there is some overhead associated with the runtime system managing the distribution of work dynamically, this approach ensures better results in situations where the workload of iterations is uneven. Thus, `schedule(dynamic)` was added to the primitive.

B. CUDA

GPUs are designed for parallel processing, with hundreds to thousands of smaller cores. CUDA enables harnessing this parallelism for accelerating compute-intensive applications far beyond what is possible with CPUs.

As noted above, the hotspot of the program is `computeAccelsandPotential`, which was the result of combining the `computeAccelerations` and `Potential` functions in the first phase. As a result, this function has been replaced by the launch of the kernel that will be executed by the various threads on the GPU. This function is divided in 3 sections:

- **memory transfer between the host and GPU:** the algorithm needs to perform reads from array `r`, so the kernel will need its data;
- **launch of the kernel:** we need to specify the number of blocks and number of threads per block, which will be discussed later; the kernel arguments should be specified too;
- **memory transfer between the GPU and host:** the potential and accelerations that are the results of the kernel need to be passed back to the host.

It is important to note that the structures used in the kernel are only allocated at the beggining of the program. This allows us to reduce the number of calls to the `cudaMalloc` function.

The group considered two approaches to implement the algorithm in CUDA:

- the first one is related to the previous OpenMP version in which the outer loop is parallelized by assigning a thread to each particle i; each thread deals with the interactions of a specific particle i with all the other particles;
- the other approach is to parallelize both loops, so we will have to assign a thread to each pair of particles (i, j); each thread is responsible for calculating the interaction between a specific pair of particles.

This second approach consists in a solution that would probably result in a very high overhead, as it would require a greater synchronisation between the threads. Therefore, we decided to follow the approach which assigns a thread by each particle i.

The code had data races in the accelerations and potential data structures that had to be dealt with. The easiest option would be to handle these data races with functions provided by CUDA that allow atomic operations to be carried out, but obviously it would lead to worse results due to the synchronisation overhead between threads. This way, we decided to change the way we were iterating over the particles. Initially, the loop was iterating on the upper triangular matrix in order to reduce the iterations by half. However, this approach was doing operations in the accelerations structure for different particles in each loop iteration, what would lead to data races in the selected parallelism approach (thread for each iteration of the outer loop). In order to avoid these data races in the structure of the accelerations, we now iterate over the entire matrix (all (i, j) pairs), starting the inner cycle at index 0.

Relatively to the potential variable which is shared among all the iterations of outer loop, we used the shared memory to store the intermediate value of the potential and to be able to make a reduction. The strategy used to perform this reduction is the sequential addressing [1]. For every block of threads, we are applying a reduction on potential within the block and this value is stored in the global memory. After that, we combine all the potentials obtained in each block and we get the final value.

V. Results discussion

In this section, we will present and discuss the results obtained on the optimisations that were done.

A. Sequential Version Optimisations

On the following table, we present the obtained results applying the optimisations that were made to the sequential version of the code. These results were obtained on SEARCH cluster using `gcc 9.3.0` version. The number of instructions (#I), clock cycles (#CC) and L1 cache misses are measured with a billion as base reference and the execution time in seconds.

TABLE I: Sequential Version Results

Optimisation	#I (B)	#CC (B)	L1 Cache Misses (B)	Exec Time (s)	CPI
0)	1248.623	843.614	3.448	266.37	0.7
1)	1015.568	666.507	2.558	208.16	0.7
2)	993.146	776.211	4.224	248.39	0.8
3)	51.344	64.042	0.694	21.63	1.2
4)	308.237	187.658	0.680	62.38	0.7
5)	221.918	123.040	0.420	42.43	0.6
6)	103.955	121.883	0.516	39.32	1.2
7)	103.899	68.014	0.426	21.83	0.7
8)	70.952	46.005	0.324	14.84	0.6
9)	18.230	11.042	0.320	3.52	0.6
10)	6.095	5.641	0.326	1.87	0.9
11)	4.687	4.552	0.327	1.47	1.0

Since optimisation 4), each table line represents an optimisation that was implemented throughout the code at that time. This means, for example, in optimisation 6) the present optimisations were less cycles, less divisions and removal of `pow` and `sqrt` calls.

- **0) :** Initial code without optimisations
- **1) :** Initial code compiled with `-O2` flag
- **2) :** Initial code compiled with `-O3` flag
- **3) :** Initial code compiled with `-Ofast` flag
- **4) :** Without pow and sqrt calls, no optimisation flags
- **5) :** Less divisions, no optimisation flags
- **6) :** Less cycles on `Potential`, no optimisation flags
- **7) :** Memory access reduction, no optimisation flags
- **8) :** Join functions, no optimisation flags
- **9) :** Same as 8), compiled with `-Ofast` flag
- **10) :** Matrices transpose, compiled with `-Ofast -ftree-vectorize -mavx` flags
- **11) :** Manual vectorisation, compiled with `-Ofast -ftree-vectorize -mavx` flags

As we can see in the table I, the initial code was very poorly optimised in terms of Instruction Level Parallelism (ILP), memory hierarchy and vectorisation. Nowadays, the compilers are very effective and can have a lot of impact in improving the ILP of a certain program. This can be comproved by the optimisation 1) and 4) in which the `-Ofast` flag had a huge impact. In terms of memory hierarchy, the final version is much more improved than the initial as can be seen in the "L1 Cache Misses" column. The Intel Xeon E5-2695 V2 present in the cluster supports the AVX extension, so, by using it, it leads us to get the maximum benefit of vectorisation.

B. OpenMP Results

To measure the impact within the usage of OpenMP directives, we present the results obtained on SEARCH cluster using `gcc 11.2.0` version. In order to analyse the performance of a parallel computing system as the problem size or the number of processing elements increases, we can employ several techniques and methodologies.

Focusing on general performance, we should analyse scalability. Weak scalability focuses on keeping the problem size per PU fixed and increasing both the problem size and the number of PUs proportionally. The total amount of work per processing element remains constant, so does the execution

time ideally. Conversely, strong scalability focuses on keeping the problem size fixed and increasing the number of PUs to observe how the solution time changes. In this approach, ideal speedup is proportional to the number of assigned physical PUs. In an effort to provide a strong scalability analysis, the figure 3 represents the relationship between the number of threads that were used and the speedup ($\frac{Texec_{best\_sequential}}{Texec_{parallel}}$).

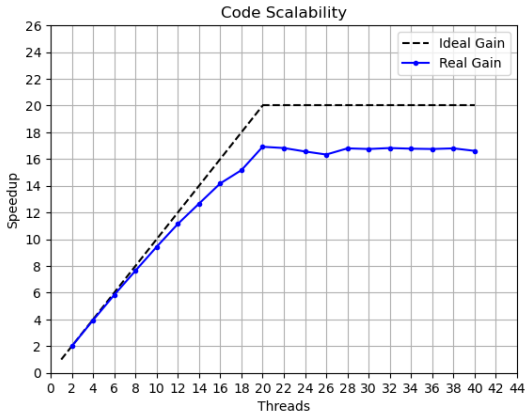

Fig. 3: Code scalability.

Firstly, we visualize that speedup tends to grow linearly until it reaches 20 threads. The number of cores of the machine does not allow this continuous linear relationship between the variables, since the ideal speedup is proportional to the number of assigned physical PUs. In the Search machine, we therefore expect the limit to be 20, as that is the number of available physical cores. We also conclude that the speedup is not ideal and this happens thanks to different reasons: percentage of serial work, memory wall, parallelism/task granularity, synchronisation overhead and load imbalance.

Percentage of serial work (Ahmdal's Law)

According to the Ahmdal's Law the maximum speedup is limited by the percentage of code non-parallelisable. Therefore, since we are only parallelising the hotspots and there are still some pieces that could not be parallelised, we are awared that we can not achieve the maximum speedup due to this serial work.

Synchronisation overhead

The speedup can also be limited by the synchronisation overhead of the primitives used. In the initial approach to avoid data races, we used the `critical` and `atomic` directives. However, the performance is much lower than the final version using the `reduction` primitive. This occurs because of the synchronisation overheads explained before. The only synchronisation overhead is the initialization of the copies and results combining in the end which is much smaller than the overhead introduced by primitives referenced before.

Load imbalance

Relatively to load balancing, it is important to understand that each iteration i of the outer loop will iterate over a new cycle starting at $i+1$. Consequently, the first iterations of the outer loop will have more work than the final ones. For this reason, the static scheduling is non-optimal on this algorithm. Taking a look at the figures 4 and 5, we confirm that we obtain smaller execution times with dynamic scheduling instead of the static one.

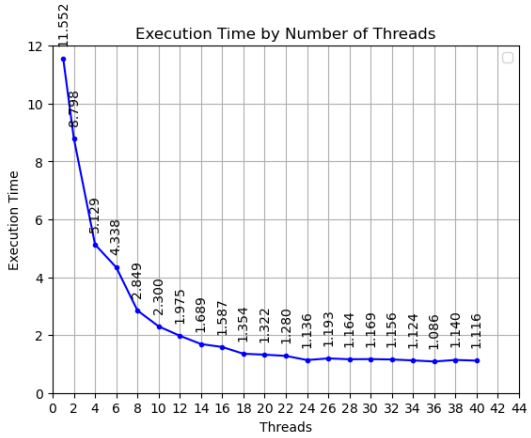

Fig. 4: Execution time by number of threads with static scheduling.

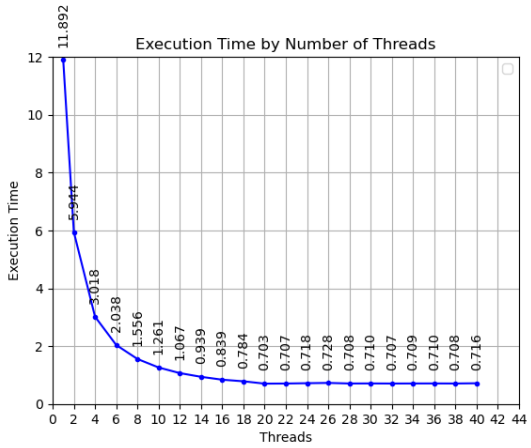

Fig. 5: Execution time by number of threads with dynamic scheduling.

To summarize, with the selected parallelism approach, we achieved gains on scalability that are not too far from the ideal. The best speedup was achieved using 20 threads and corresponds to 16.916.

C. CUDA Results

As a way to evaluate the scalability and analyse the results of CUDA solution, we created a bash script `test.sh` where is defined the number of particles list, the number of threads per block and the number of repetitions for each set of parameters. The execution was measured by the tool `time`, measuring the time of binaries generated by the Makefile.

All runs were performed in the SEARCH cluster using `gcc 7.2.0` and `cuda 11.3.1` versions. The following tests were taken with 3 repetitions and 128 threads per block.

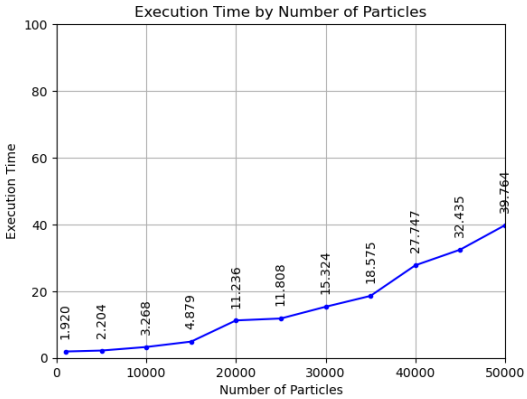

Fig. 6: Execution time by number of particles.

Observing the figure 6, we can see that the time it takes to run the algorithm increases as the number of particles grows, but the relationship is not linear. In spite of the algorithm's parallelization, its time complexity remains fundamentally the same - quadratic -, meaning that the interactions between particles grow proportionally to the square of the number of particles.

Choosing the number of threads per block to run in the kernel is a challenging step, because it depends on the hardware and situations faced. In the case of this GPU, Tesla K20, the ideal number of threads per block is 128 as we can see in the table below.

TABLE II: Execution Time per Number of Threads per Block

Number of Threads	**Execution Time** (s)
32	25.712
64	17.004
128	15.535
256	17.338
512	16.577
1024	18.510

D. Overall Analysis

Finally, we could not miss a comparison between the different gains obtained in each phase of this code optimisation. This comparison also suggests a reflection about the evolution of technologies, since some time ago it was not possible to take advantage of some features, such as using a graphic card to potentially improve a program performance. Therefore, in the following graph, we compare the results obtained from the ILP-level optimisations, the results obtained with the adoption of OpenMP directives and the results obtained within the implementation of CUDA kernel. The results were obtained on SEARCH cluster, under the same conditions mentioned before for each type. The OpenMP results were obtained using 20 threads.

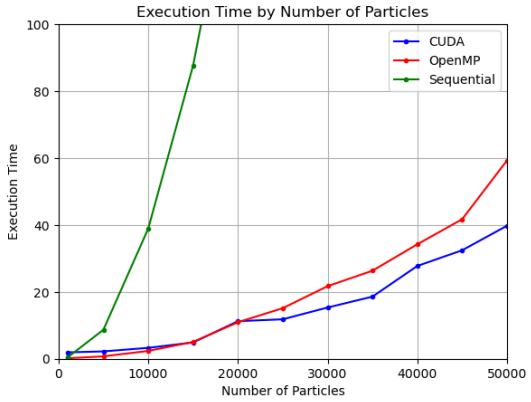

Fig. 7: Execution time by number of particles.

In the image above, we can clearly see the power of parallel computing against the sequential computing. The parallel versions have much lower execution times comparing to the sequential one.

Relatively to the parallel versions, we can conclude that the higher the number of particles, the greater is the difference between CUDA and OpenMP. Interestingly, for a smaller number of particles (up to about 10000), OpenMP seems to outperform CUDA, suggesting that the overhead of using the GPU (data transfer between the CPU and GPU, kernel launch, etc.) may not be justified for a smaller number of particles. For a higher number of particles, CUDA has better results than OpenMP which is related to the number of threads that GPU can provide, since the number of threads OpenMP was using was fixed to 20. The curve for CUDA is flatter than for OpenMP, indicating better scalability with increasing numbers of particles.

VI. Conclusion

The practical assignments of the course effectively illustrated the power of targeted optimisation. Each phase built upon the previous, showcasing that while techniques like ILP and OpenMP directives offer substantial improvements, especially for smaller workloads, the true potential for large-scale computational efficiency lies within the realm of GPU acceleration. Our results show a clear trend: as the complexity and size of the data increase, so does the advantage of using GPU over traditional CPU methods. This reinforces the notion that optimisation is not a one-size-fits-all solution but rather a strategic selection of tools and techniques based on the specific demands of the task at hand.

References

[1] Mark Harris. Parallel reduction in cuda. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.