Parallel Computing
Work Assignment Phase 3

Ana Rita Poças (*pg53645)* e Orlando Palmeira (*pg54123)*

***Abstract*—At this phase of the practical assignment, we will use the optimized code from the previous phases (simple molecular dynamics simulation). The main goal is to explore parallelism techniques in order to improve the performance of the given program.**

***Index Terms*—CUDA, GPU's, Parallelism, Code Optimization, OpenMP**

I. Introduction

The core objective of this project is to investigate and assess a range of strategies and tools aimed at optimizing an algorithm. This project provided a chance to examine various parallelization models, with the goal of enhancing our algorithm's performance.

In this document, we scrutinize the progression through the project's numerous stages, from the early development of the algorithm in its sequential form and its subsequent enhancements, to the adoption of shared memory parallelism in its later phase, and finally, the application of the CUDA programming model.

During these stages, we will conduct a thorough evaluation of the outcomes derived from each strategy and solution deployed, and engage in a detailed discussion regarding the efficacy of these implementations.

II. WA1 - Optimizations in sequential version

The first phase of this practical work consisted of applying optimization techniques to reduce execution time. These techniques include: **algorithm changes**, **ILP** improvements, taking advantage of the **memory hierarchy** and **vectorization**.

When we analyzed the code for this phase, through complexity analysis and profiling, we found that there were two functions that took up practically 100% of the execution time: **Potential** (complexity: approximately $O(N^2)$, 74.6% exec. time) and **computeAccelerations** (complexity: approximately $O(N^2)$, 25.13% exec. time). Furthermore, the code had an execution time of approximately 200 seconds and 1.3×10^{12} instructions.

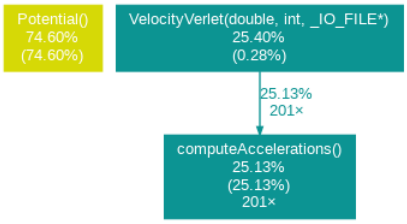

Fig. 1. Results from gprof2dot for the non-optimised code

A. Implemented optimizations

*1) **Algorithm changes**:* These changes consisted of reducing the amount of computation, as well as simplifying the mathematical calculations performed by the two previously mentioned functions.

Firstly, the functions used the "pow" (computeAccelerations) and "sqrt" (computeAccelerations and Potential) functions. The "pow" and "sqrt" functions are heavy due to the use of iterative methods to perform their calculations. Furthermore, its use was unnecessary because using the "pow" function performed a fixed exponent exponentiation that could be replaced by several multiplications. The use of the "sqrt" function was unnecessary, as the square root was canceled later when using its result in an even exponent exponentiation.

Secondly, both functions used complex mathematical expressions that were appropriately simplified through mathematical manipulation (covered in more detail in the Appendix A).

Thirdly, the inner loop of the Potential function had a number of iterations that was double of what was necessary ($j \in [0, N-1[, i \neq j$). It was possible to reduce the number of iterations by half by changing the calculation logic (discussed in more detail in Appendix B).

Finally, since the computeAccelerations and Potential functions perform the same calculations ($(r[i]-r[j])^2$) in portions of their code, it was possible to merge them into a single function for these calculations to be reused.

*2) **ILP Improvements**:* The improvements we implemented in terms of ILP consisted of unrolling loops of fixed size (control variable ranging from 0 to 2). The loop unrolling enhances ILP by reducing loop control overhead and increasing the number of instructions available for parallel execution. This technique decreases the loop iterations (in our case, we removed the loops completely) by expanding operations within each iteration. It allows more efficient utilization of the processor's execution units and improves instruction scheduling by filling gaps caused by dependencies or latencies. Additionally, it reduces branch prediction failures, as there are fewer loop jumps.

*3) **Memory hierarchy**:* Regarding the memory hierarchy, we proceed to create variables that store temporary values that are added to the matrix "a" (depending on the index i) in the inner loop of the computeAccelerations function. As the index "i" does not change throughout the inner loop iterations, these variables can store the sum of the acceleration values and at the end of the loop add them to the matrix 'a'. This technique **reduces the number of accesses** to the matrix 'a' and takes advantage of the **memory hierarchy** through **temporal locality**.

*4) **Vectorization**:* When it comes to vectorization, we first use the ”-ftree-vectorize” and ”-msse4” flags. The ”-ftree-vectorize” flag was employed to activate the tree vectorizer, enabling the automatic vectorization of loops by transforming scalar operations into vector operations. These vector operations utilize SIMD intructions, which potentially enhance the performance of the code. Additionally, we incorporated the ”-msse4” flag to instruct the compiler to utilize instructions from the SSE4 instruction set. These instructions support 128-bit vector operations, contributing to the overall performance.

In addition to the flags, we were able to eliminate an ”if” present in the loop responsible for calculating the Potential. The vectorization of loops benefits from uniformity. Introducing ”if” statements in loops creates variations in operations, breaking the essential uniformity for effective vectorization. This not only makes the control flow less predictable, impacting processing efficiency, but also complicates the task of compilers optimizing the code automatically. For these reasons, we put effort into analyzing the code in order to remove the ”if” statement (The ”if” was removed by optimizing the number of Potential iterations referred to in the penultimate paragraph in the ”***Algorithm changes***” section).

B. Results

With all the improvements mentioned throughout the previous section, the code managed to obtain an execution time of approximately 3.4 seconds with 1.93×10^{10} instructions. The optimizations carried out resulted in a speedup of 58.82 and a reduction in instructions of 67.36 (these results were measured with $N = 2160$). In this way we were able to see that there was a clear improvement in performance.

III. WA2 - OpenMP Parallelism

Having an optimized sequential version, we can now explore parallelism techniques.

We started by profiling the code to detect blocks with a lot of computing time (hot-spots) in order to define the places that need to be parallelized. In this case, the hot-spot of the code is located in the nested for loops (that perform all the mathematical calculations) of the computeAccelerations function (which took up close to 100% of the execution time and has a complexity of approximately $O(\frac{N^2-N}{2})$).

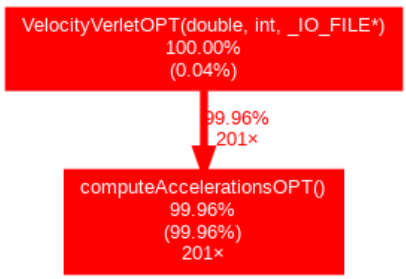

Fig. 2. Hotspot detected with *gprof*

This time, with $N = 5000$, the execution time of the sequential version is approximately 23 seconds.

A. Implemented optimizations

Considering de previous reasons, we will use the `#pragma omp parallel for` directive to parallelize the two nested *for* loops of the *computeAccelerations* function, and this application will take place in the outer loop. We have chosen to implement parallelism in the outer loop in order to reduce the granularity of tasks executed by each thread, which may result in performance improvement (small tasks introduce more(relative) overhead caused by creation and management of threads). However, there are variables in the loop whose usage should be exclusive to each thread (since their values won't be used outside the context of the parallelized section of the function). To address this, we have used ”private” within the directive to declare variables that need to be copied for each thread to avoid data races.

While most variables in the function are protected using ”private”, two of them cannot be treated this way (Pot and matrix 'a'), as their values are used outside the context of the parallelized loop. Thus, we can employ some techniques to solve this issue. The first option is to safeguard calculations associated with these variables with directives such as ”#pragma critical” or ”#pragma atomic”. This approach introduces some overhead due to the need for thread synchronization and, furthermore, contributes to reducing the portion of parallelized code, making it a less favorable option.

The second option is to use ”reduction”, where a copy of the 'Pot' and 'a' variables is made for each thread, and the values of these copies are aggregated into the original variables at the end of the loop. This technique is evidently better as it avoids the need for heavier synchronization mechanisms like ”critical” and ”atomic”, contributing to improved performance.

Taking all the previous explanation into account, the OpenMP directive used in the outer loop will be the following: `#pragma omp parallel for reduction(+:Pot,a[:N][:3]) private(variables of the loop)`. However, there are still other options that we can explore, which involve choosing the task scheduling approach for the threads.

For the scheduling of thread tasks, we decided to explore static scheduling and dynamic scheduling. Static scheduling has the advantage of not introducing much overhead in thread management, as the blocks of iterations assigned to each thread are predefined and do not change over time. On the other hand, dynamic scheduling allows load balancing between threads, enabling a thread that finishes its block of iterations to request more ”work” to reduce the load on threads performing heavier tasks. However, dynamic scheduling introduces overhead in thread management.

With this, we anticipate that the use of dynamic scheduling can contribute to better code performance. The reason for this is that the number of iterations in the inner loop (executed by each thread) directly depends on the value of the i variable from the outer loop, which may result in unbalanced thread loads. For example, with N=5000 and i=0, the inner loop will have 4999 iterations. If i=4500, then the inner loop will have 499 iterations. There is an evident imbalance in the load distribution, and this can be minimized with the use of

dynamic scheduling.

B. *Strong scalability analysis*

To confirm our predictions, we conducted a comparison of the two approaches through a strong scalability analysis:

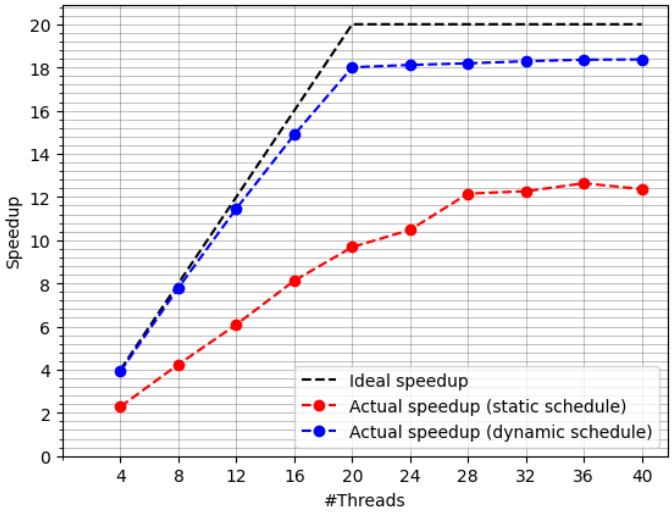

Fig. 3. Comparison dynamic vs static scheduling

As shown in the graph above, the version with dynamic scheduling performs significantly better (it's very close to the ideal speedup). Due to the uneven load that would exist between threads with the static scheduling approach, the overhead imposed by thread management in dynamic scheduling is easily offset by the load balancing advantages.

C. *Performance analysis*

From the graph presented in Figure 3, we can observe that our solution failed to achieve the ideal speedup. It was expected that the solution would not reach optimal performance, and there are some reasons for this to occur.

1) ***Reduction overhead****:* Firstly, we must consider the overhead imposed by the use of the 'reduction' clause, particularly on the matrix 'a'. At the creation of threads, we are making multiple copies of the matrix 'a' (with dimensions 5000×3). At the end of the parallelized loop execution, the sum of all elements of the matrix is calculated to obtain the result. All these operations impose additional load on the execution.

2) ***Memory wall****:* In terms of memory bandwidth, we believe this factor has some impact on the performance of our solution. As seen in Appendix C, the increase in the number of threads contributes to a slight increase in CPI. This is a possible indicator (without absolute certainty) that the algorithm might be somewhat constrained by memory bandwidth, since the CPU might be waiting for data from the memory (there are multiple memory accesses, mainly in matrix 'a'). Furthermore, we analyzed the arithmetic intensity (Appendix D, evolution of #I/LLC.Miss with the number of threads) and we've realized that the arithmetic intensity decreases when the number of threads increases. This might tell us that the algorithm of our program is memory bounded, which contributes to deteriorate the performance.

3) ***Load imbalance****:* When it comes to load imbalance, we acknowledge the possibility that our solution may be affected by this factor. In fact, we attempted to minimize this issue with the use of 'schedule(dynamic)', but this mechanism may not be perfect. Additionally, dynamic scheduling imposes thread management overhead, which contributes to not reach the ideal speedup.

4) ***Task granularity****:* We believe that the solution's performance is not affected by task granularity, as the parallelism approach adopted in our solution already has the maximum task size possible to achieve in the optimized function.

5) ***Synchronization overhead****:* Finally, we also believe that performance may not be affected by synchronization overhead since no synchronization directives like 'atomic' and 'critical' are used anywhere in the code.

D. *Correction of undetected data race.*

Unfortunately, in the previous phase of this work, code was delivered with a data race in the variable `j` of the inner loop of the nested loops of the computeAccelerations function.

In this phase, we decided to apply a correction to this data race by making the variable j only created at the time of inner loop initialization (`for(int j = i+1; j < N-1; j++)`).

E. *Results*

At the end of the WA2 phase of practical assignment, we were able to develop a faster version than the sequential one, due to the use of parallelism. However, the adopted solution did not reach the ideal speedup, but managed to achieve performance close to that level.

With $N = 5000$, we were able to reduce the time from 23 seconds (sequential version) to approximately 1.3 seconds (with 40 threads).

IV. WA3 - GPU'S (CUDA) PARALLELISM

CUDA, which stands for "Compute Unified Device Architecture", is a parallel computing architecture developed by NVIDIA. This technology is used to accelerate processing on GPUs (Graphics Processing Units). In simple terms, CUDA allows programmers to harness the processing power of GPUs to perform parallel computations, which is particularly useful in tasks related to graphics, physics simulations, and other scientific computations.

As seen in the previous phase (WA2) through the analysis of the sequential code, the function that consumes the most computational time is *computeAccelerations* (it takes 99.96% of the execution time). Therefore, it is in this function that we will apply parallelism techniques (as we did with OpenMP) with the CUDA programming model.

Firstly, in CUDA we need to implement a kernel function that implements the logic/calculations of the parallelized function to be executed on the GPU. This function (kernel) requires some preparation operations to be performed, such as memory allocation and copying all data to the GPU (kernel launcher).

A. Implementation of CUDA code

The original computeAccelerations function reads the matrix 'r', reads and writes the matrix 'a' and calculates the Potential value. Therefore, before executing the kernel (computeAccelerationsKernel) it is necessary to allocate memory (*cudaMalloc*) for these two matrices ('a' and 'r') for the GPU, as well as make copies (*cudaMemcpy*) of the data present in these matrices to the GPU before executing the kernel . Furthermore, so that each thread calculates the Potential value safely, that is, without the occurrence of data races, we decided to allocate an array on the GPU (of size N = number of threads actually working) where each thread stores its value of the Potential and, at the end of the kernel execution, this array is reduced (through a summation) to obtain the effective value of the Potential. In this way, we were able to make the Potential calculation be carried out in parallel.

As far as the kernel implementation is concerned, we kept all the logic and calculations performed in the same way as in the sequential version (optimized). The only difference is the absence of the external loop since the kernel represents an iteration of this loop that will be executed by each thread.

If there were no kind of concurrency control in the kernel, we would have concurrent writes in the Potential calculation and in the 'a' matrix updates. Regarding the Potential calculation, we create an array where each thread will have a specific position to write to (explained previously). Regarding the matrix 'a', we use atomic operations to prevent multiple threads from writing to the same position in the matrix. The function that provides atomic operations (*atomicAdd_double*) for doubles is not provided directly by the CUDA library used in this practical assignment. To do this, we had to use a function provided in the NVIDIA documentation that allows atomic sums over doubles.

At the end of the kernel execution, the kernel launcher retrieves data relating to matrix 'a' and the Potential calculation from the GPU. The GPU matrix 'a' is directly copied to the host matrix 'a' and the Potential array is also copied to an array on the host which is later reduced (with sums) to obtain the real Potential value.

B. Some justifications regarding implementation

In this implementation, we use a one-dimensional CUDA grid. The reason for this is the fact that the tests we carried out with a two-dimensional grid turned out to be quite slow. Using a one-dimensional grid, we have N threads performing atomic writes to matrix 'a' while with a two-dimensional grid we would have approximately $\frac{N^2}{2}$ threads performing these atomic writes. The two-dimensional grid would greatly increase the impact caused by the use of atomic operations (for example, with $N = 5000$ and a two-dimensional grid, we'd have 12.5×10^6 threads performing atomic writes).

Regarding the Potential calculation, we chose to create an array to allow its parallel calculation (with the respective reduction after the kernel) since if we used a single variable with atomic updates, we would have N threads writing concurrently to that variable generating a performance bottleneck with a significant impact.

C. Tests and analysis of results

To analyze the performance of the CUDA implementation, we performed some tests. First, we start by analyzing the impact of the number of threads per block on program performance with a fixed value of N (5000). We then analyzed, using the **nvprof** tool, which operations were the heaviest. Finally, we perform performance tests with different values of N along with increasing the number of GPU threads (weak scalability analysis).

Threads/block analysis

Threads per block	Execution Time (s)
16	6.432s
32	7.819s
64	7.812s
128	7.834s
256	8.228s
512	9.213s
1024	14.019s

Table 1: Execution time evolution with the number of Threads per Block (N=5000)

Based on the table above, we can see that the best number of threads per block is 16, with N=5000. This can be explained by the GPU model employed in the cluster, which features 13 SMX (Streaming Multiprocessors), necessitating each block to execute within the same SMX. Furthermore, the strategic allocation of multiple blocks per SMX proves beneficial in mitigating diverse latencies (memory, for instance).

Since using 16 threads per block was where we obtained the best results, the next tests and analyzes will be carried out with this parameter.

Heaviest operations analysis

With the help of nvprof, we were able to find out which operations took the longest, and we obtained the following results:

Operation	Total time	Calls	Average
kernel	4.55142s	202	22.532ms
memcpy HtoD	8.8808ms	404	21.982us
memcpy DtoH	5.0837ms	404	12.583us
cudaMemcpy	4.60259s	808	5.6963ms
cudaMalloc	282.32ms	606	465.88us

Table 2: Profilling results (N = 5000, nvprof)

With the results in the table above, we can draw some conclusions. Firstly, the operations that take up the largest percentage of execution time are the kernel and memcpy's. The kernel takes up a lot of computing time since, with regard to the program logic, it is the function that has the greatest computational complexity. Furthermore, this function presents a load imbalance between threads since the value of the variable `j`, which controls the kernel loop, depends directly on the thread id/index i (this problem was addressed in the section about implementation in OpenMP). In addition, the kernel performs frequent atomic writes to the matrix 'a'. The purpose of these operations is to ensure that the program

runs correctly (without data races), but they have a significant impact on performance.

Regarding the impact of memcpy's, the problem is due to the frequent calls we make to this function. Copying data between the GPU and the host is slow and in our implementation we execute it on every kernel run.

Weak scaling analysis

For the weak scalability analysis, we increased the value of N together with the number of GPU threads (with 16 threads per block) and obtained the following results:

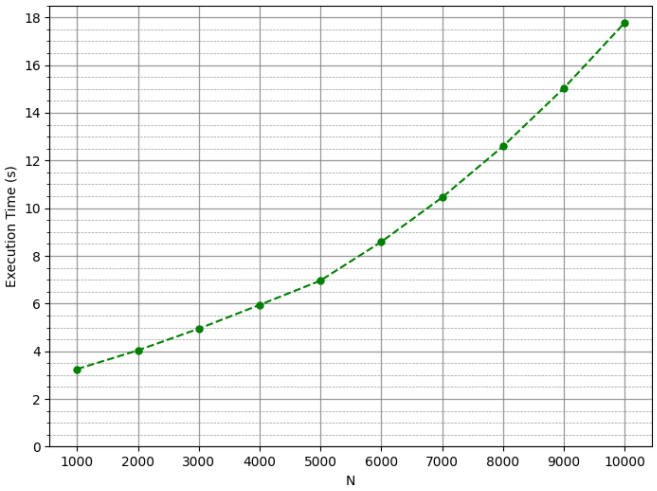

Fig. 4. Weak scaling chart

As we can see, the execution time continues to rise even if there is an increase in the number of threads relative to the size of N, which indicates that our implementation does not have a satisfactory weak scalability and there are some reasons for this to happen. As previously stated, the kernel implementation generates load imbalance and uses atomic operations. With the increase in N we are further increasing the load imbalance between threads (since the number of the iterations of the loop in the kernel depends on the value of N) and, in addition, we are increasing the size of matrix 'a' which generates an increase in the number of atomic operations performed on this matrix.

D. Possible improvements

Although we do not have the performance that would be expected using GPUs, we believe we know what could have been done to improve the implementation. Firstly, we can mention the use of shared memory, which is a fast-access, low-latency memory region that is shared among the threads of a block in a CUDA kernel. Additionally, an approach that would improve performance would be to have the calculations of the "VelocityVerlet", "Kinetic" and "MeanSquaredVelocity" functions (executed in the main function loop) executed on the GPU. This way, we could avoid the countless data transfers (cudaMemcpy) that occur between the host and the GPU on each kernel launcher run.

Along with the previously mentioned improvements, we could try to reduce the number of atomic operations through reduction operations on the matrix 'a'.

E. Solutions comparison

In this practical assignment, we were able to explore three code optimization approaches.

The best version implemented was the implementation with OpenMP, then it was the version with accelerators (GPU's) and then the version with optimizations to the sequential version. The sequential version has the worst performance, as it does not implement any parallelism technique. Regarding the versions with parallelism, it was expected that the version with GPUs would have a higher performance than the version with shared memory parallelism (OpenMP). However, given the complexity of the proposed problem, it was not possible to implement a version that took advantage of all the potential provided by GPU computing.

OpenMP provides an abstraction layer that reduces the complexity of implementing reduction operations in the code, which allowed us to avoid atomic operations and any kind of directive that would create critical regions in the code. Furthermore, the OpenMP version was not hampered by factors that impacted the CUDA implementation, namely data exchanges (cudaMemcpy) between the host and the GPU.

V. Conclusion

In summary, throughout the three assignments that we've had in this semester, we were able to analyze the importance of parallelism on code optimization. By using the different techniques that we've learned, we were able to verify and see for ourselves how much it improved the code that was initially provided by the professors.

In the first assignment, the techniques applied in the code (algorithm changes, ILP improvements, memory hierarchy,vectorization) resulted a much better execution time in comparison to the original code, which was our main goal.

In the second assignment, by applying OpenMP, in the code developed in the first phase, and we were able to obtain an even better execution time.

Finally, in the third assignment, we delved into the CUDA programming model. While we didn't surpass the OpenMP version, we still managed to outperform the sequential version. Moreover, the complexity of this task provided us with a lot of motivation to study this programming model in order to strive for the best possible solution.

APPENDIX A
WA1 - SIMPLIFICATIONS OF THE MATHEMATICAL OPERATIONS

A. Potential

1) : $Pot = \sum_{i=0}^{N} \sum_{j=0,j\neq i}^{N} 4 \times epsilon \times (term1 - term2)$

$= \sum_{i=0}^{N} 4 \times epsilon \times \sum_{j=0,j\neq i}^{N} (term1 - term2)$

$= 4 \times epsilon \times \sum_{i=0}^{N} \sum_{j=0,j\neq i}^{N} (term1 - term2)$

2) : $term1 = quot^{12} = ((quot^3)^2)^2 = (((\frac{sigma}{rnorm})^3)^2)^2 = (((\frac{sigma}{\sqrt{r2}})^3)^2)^2 = \frac{(sigma^6)^2}{(r2^3)^2} = (\frac{sigma^6}{r2^3})^2 = term2 \times term2$

$term2 = quot^6 = (quot^3)^2 = ((\frac{sigma}{rnorm})^3)^2 = ((\frac{sigma}{\sqrt{r2}})^3)^2 = \frac{(sigma^3)^2}{r2^3} = \frac{sigma^6}{r2^3}$

B. computeAccelerations

1) : $f = 24 \times (2 \times rSqd^{-7} - rSqd^{-4}) = 48 \times rSqd^{-7} - 24 \times rSqd^{-4} = \frac{48}{rSqd^7} - \frac{24}{rSqd^4} = \frac{48}{rSqd^7} - \frac{24 \times rSqd^3}{rSqd^7} = \frac{48 - 24 \times rSqd^3}{rSqd^7}$

2) : $rSqd3 = rSqd \times rSqd \times rSqd$ $(rsqd3 = rSqd^3)$
$rSqd7 = rSqd3 \times rSqd3 \times rSqd$ $(rsqd7 = rSqd^7)$

APPENDIX B
WA1 - DECREASING OF ITERATIONS IN POTENTIAL CALCULATION

We've noticed that the code inside the loops i and j ($i, j \in [0, N[$) runs only when $i \neq j$. Thus, we can see the indices **i** and **j** in the following table:

i,j	0	1	2	...
0	×	A	B	C
1	A	×	D	E
2	B	D	×	F
...	C	E	F	×

Legend:
- Cells with ×: Code does not run.
- Cells with letters: Code runs

Each cell in the table above represents an iteration in the function for specific i's and j's. The cells with equal letters have the same value of $r2 = \sum_{i=0}^{N-1} \sum_{j=0,j\neq i}^{N-1} \sum_{k=0}^{2} r[i][k] - r[j][k]$. Hence, we can perform the execution of the j loop from $i + 1$ to N, which reduces the #instructions significantly. However, in the end of the function, we need to calculate $8 \times epsilon$ instead of $4 \times epsilon$, because we perform the half of the calculations (half of the sum in the variable **Pot**).

APPENDIX C
CPI EVOLUTION WITH OPENMP

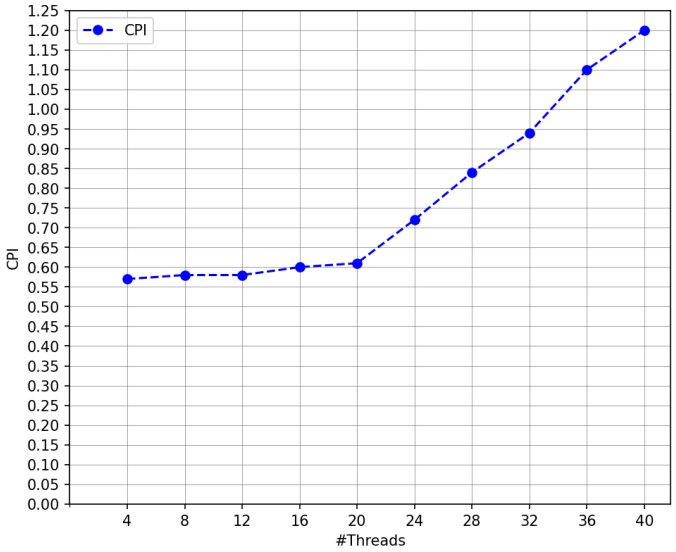

Fig. 5. CPI evolution with the number of threads

APPENDIX D
ARITHMETIC INTENSITY (#I/LLC.MISS) WITH OPENMP

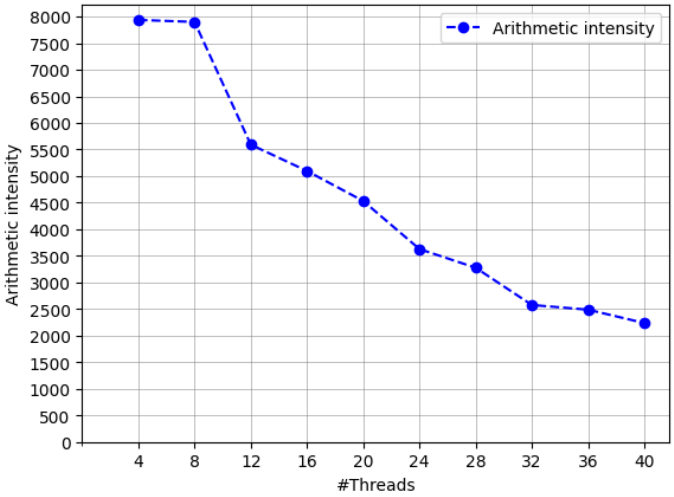

Fig. 6. Arithmetic intensity with the number of threads