Computação Paralela - WA3

Mestrado em Engenharia Informática

Afonso Marques

Departamento de Informática
Universidade do Minho
Mirandela, Portugal
pg53601

Renato Gomes

Departamento de Informática
Universidade do Minho
Porto, Portugal
pg54174

***Abstract*—The program to analyse and optimise is part of a simple molecular dynamics simulation code applied to atoms of argon gas (original code version in Foley-Lab/MolecularDynamics: Simple Molecular Dynamics). The code follows a generic approach to simulate particle movements over time steps, using Newton laws of motion. In this report we shall refer to the program as MD.**

The code uses Lennard Jones potential to describe the interactions among two particles (force/potential energy). The Verlet integration method is used to calculate the particles trajectories over time. Boundary conditions (i.e., particles moving outside the simulation space) are managed by elastic walls.

This third and final installment of the project aims to fully optimize the MD code, preferably using the CUDA module (Compute Unified Device Architecture) which is a parallel computing architecture that enables developers to use NVIDIA Graphics Processing Units (GPUs) for general-purpose computing, not just graphics.

***Index Terms*—algorithm, performance, single thread, optimization, metrics, multiple threads, OpenMP, parallelization, CUDA, NVidia, GPU**

I. Introduction

The main goal of this third assignment was to further improve the performance of the *MD* code using one of three alternatives: improve the existing OpenMP parallelization, use the MPI module or use the CUDA module.

We started off by correcting the previous assignment that we delivered making sure that this time there were no data races while running the code. We achieved this by changing a couple of things in our WA2 code starting with making sure that the generated threads would access shared data with caution and order so that the intermediate and final values of Kinetic Energy, Potential and Total Energy would not be messed up by the end of the run. We also joined the *Potential* and *computeAcceletaions* functions after noticing that they accessed the same memory positions. This allowed us to reduce the number of instructions and slightly improve the execution time. Still the overall performance with the OpenMP directives proved to be somewhat disappointing because even with the improvements and no signs of data races, the execution time increased quite a lot, even surpassing the 15 seconds mark in some instances, when using 32 threads.

Relating to the WA3 itself, we wanted to do an implementation with CUDA pretty much from the start, because of all the options, we felt more comfortable with this one in terms of our abilities and knowledge. It also proved to be much more effective and fulfilling in terms of results, getting an average of 4 to 5 seconds in execution time with no data races in sight and N set to 5000.

II. Algorithmic Complexity

1) *Potential()*: The potential function involves three nested loops. The outer two loops iterate over all particles in the system, resulting in N^2 iterations. Inside the innermost loop, there are several mathematical operations like addition, subtraction, multiplication, and power calculations. These operations are not nested within the loops and are generally considered to be constant time. So, the time complexity of the *Potential()* function is O(N^2).

2) *computeAcceletaions()*: This function also involves nested loops. The outer loop iterates over all particles, and the two inner loops iterate over the components for each pair of particles. The most computationally intensive part is the calculation of forces (f) and updating the accelerations. Inside the inner loops, there are constant time operations as well as calculations that depend on the number of dimensions (3 in this case). Overall, the time complexity of this function is also O(N^2).

III. WA2 - Fixes to OpenMP Optimizations

In this section we will enumerate what changes we made to our WA2 implementation in order to eliminate the data races. The changes were made mostly on the *Potential* and *computeAccelerations* functions.

A. OpenMP optimizations used in the fixed version

We noticed various points in the *Potential* and *computeAcceletaions* functions where similar calculations where being made, plus both were looping over the same memory positions, so we though that maybe combining them together into a single function would be the better way to go. We achieved this and came up with our new version of these functions, *computeAccelerations_Potencial*. This one uses a combination of OpenMP instructions (in a single line) which we will dissect.

- ***parallel for***: this indicates that the following loop must be executed in parallel. Each loop iteration can be executed by a different thread.
- ***private(j, f, rSqd, rij)***: this private clause specifies that each thread must have its own copy of these variables. This is necessary to ensure that there are no data conflicts between different threads during parallel execution.
- ***reduction(+:Pot, a[:N*3])***: this reduction clause is used to perform reduction operations such as sum (+). In our case, we are performing a reduction on the variables 'Pot' and 'a', where 'Pot' is a scalar and 'a' is a three-dimensional matrix. This means that each thread maintains its own copy of 'Pot' and 'a', and at the end of the loop, these copies are combined according to the sum operation.
- ***schedule(dynamic,48)***: this schedule clause controls how loop iterations are distributed among threads. 'dynamic' indicates that iterations are dynamically assigned to threads, and 48 is the block size used for dynamic assignment. This means that each thread will receive a block of 48 iterations at a time until all iterations are completed.

These changes allowed us to eliminate the data races at the cost of execution time.

IV. Implementation of CUDA

Now let's focus on the third assignment itself, WA3.

As we mentioned in the summary, we decided from the start that we wanted to do an implementation with CUDA. In order to achieve this we had to create a new file, MDpar_CUDA.cu, as well as a new Makefile. The focus was again on the these two functions, *Potential* and *computeAcceletaions*, however, just like the fixed version of WA2 that we presented in the previous section, these two functions are combined into a single one.

The code in the CUDA version is pretty much the same as before, with some necessary changes to make it work. They are as follows:

- Kernel Functions: There is a __global__ kernel function named computeAccelerations_Potencial_KERNEL. This function is executed on the GPU and is responsible for computing accelerations and potential energy. The function utilizes CUDA's parallel processing capabilities, with each thread (indexed by i) handling computations for a different particle in the simulation. This parallel approach is highly efficient for molecular dynamics simulations, which involve calculations for many particles.
- Memory Management: We included cudaMalloc and cudaMemcpy operations. These are used for memory allocation on the GPU (cudaMalloc) and for transferring data between the CPU and GPU (cudaMemcpy). For example, memory is allocated for particle positions (r), accelerations (a), and potential energy (PE), and data is transferred between the host (CPU) and the device (GPU). This aspect is crucial because the GPU needs its own memory space and cannot directly access the CPU's memory.
- The ability to read a variable passed as an argument to the program. We needed to do this in order to run two testing scripts - one to see how the program would perform if the number of particles, N, increased by a couple of hundreds and thousands and the other to test the performance for different numbers of threads per block with 5000 particles. Basically, we export a GLOBAL_N_VALUE or GLOBAL_THREAD_VALUE with a certain number to the program which then reads it and updates the value of the global variable N ot tpb. If we don't export anything, the value of N is defaulted to 5000 particles and the value of tpb is defaulted to 288.

V. Impacts of CUDA

The CUDA implementation significantly improves the performance of the molecular dynamics simulation by utilizing the parallel processing power of GPUs. This enhancement allows for more efficient computation of particle interactions and dynamics, though it also introduces additional complexity in terms of memory management and program architecture. Some of the impacts are:

- Enhanced Performance: The use of CUDA allows the program to leverage the parallel processing capabilities of the GPU, leading to faster computations compared to a CPU-only approach. This is particularly beneficial for computationally intensive tasks like molecular dynamics simulations.
- Scalability: The program can handle larger and more complex simulations efficiently due to the GPU's ability to process many operations in parallel.

- Development Complexity: Writing and maintaining CUDA code can be more complex than traditional CPU-only programming. This includes managing data transfer between CPU and GPU, optimizing memory usage, and ensuring efficient parallel computations.

A. Results for CUDA implementation

In this section we expose the overall result of running the CUDA version of the program for 5000 particles and using 288 threads per block in the kernel. We made multiple runs until we reached the conclusion that the average execution time was around 4,14 seconds. We collected three of these runs and put the results in the Annexes. The following is a table with the execution time for 5 different runs.

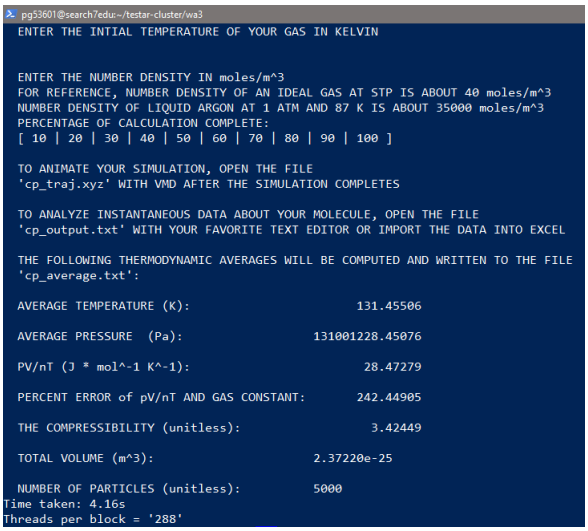

Fig. 1. One of the runs

TABLE I
TABLE WITH EXECUTION TIME OF *MDpar_CUDA* IN MULTIPLE RUNS

#Run	Execution Time
1	0 m 4.11 s
2	0 m 4.16 s
3	0 m 4.18 s
4	0 m 4.06 s
5	0 m 4.17 s

It's worth mentioning that the output of the MD program is not altered in any way in both *MDseq* and *MDpar_CUDA*. This includes the *cp_output.txt* and *cp_average.txt* files as well as the terminal's output.

B. Scalability analisys for CUDA implementation

In order to test how the performance varied for different numbers of particles, we used a script file, *test-CUDA-N.sh*, which we executed multiple times to determine what the average execution time was for each scenario. We tested how the program would react for 5500, 5600, 6000, 7000 and 8000 particles.

We obtained the following results in the graph below. It's possible to see that with the increase of the number of particles the execution time also increased (which is to be expected) but the increase wasn't as big as we toughed it would.

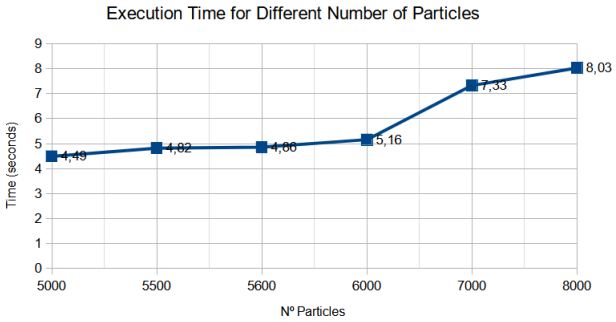

Fig. 2. Graph with execution times

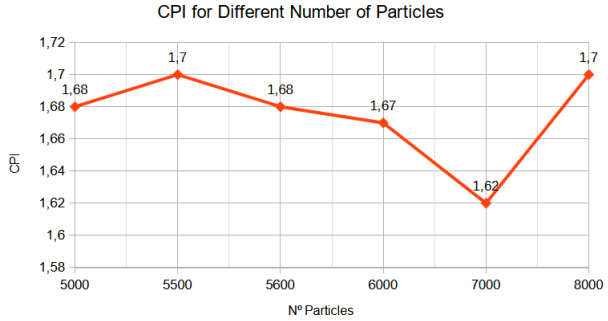

Fig. 3. Graph with CPI's

Essentially, while the CPI remained very much the same, the execution time increases slightly in the order of the hundreds and jumps when we reach the thousands.

We also did a speedud analysis where we ran a script, *test-CUDA-threads.sh*, where we altered the number of threads per block in the kernel of CUDA to see how the program would react. The outputs remained correct in every run.

The best time obtained while running the sequential version of MD was 50.37 seconds (see annexes). We will use this value to calculate the speedup for each number of threads per block using the following formula:

$$speedup = \frac{bestSeqTime}{CUDAExecTime}$$

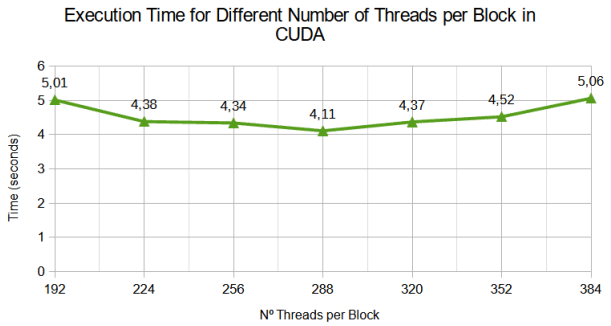

Fig. 4. Graph with execution time for Nº threads

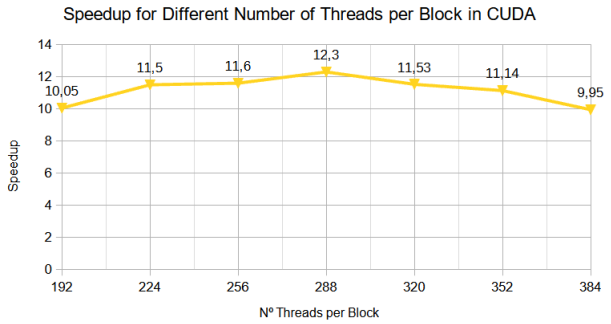

Fig. 5. Graph with speedup

TABLE II
TABLE WITH SPEEDUP RESULTS

#Threads per block	Speedup
192	10,05
224	11,50
256	11,60
288	12,30
320	11,53
352	11,14
384	9,95

Analysing the graphs it's pretty easy to see that the best number of threads per block is 288, as it's the one that allows for better execution time and more speedup in relation to the sequential version. We conclude that the best number for threads per block is 288 and it shall be used as the default value.

VI. Conclusion

In conclusion, this assignment aimed to assess the advantages of optimizing code through the use of the GPU's in a computer using the CUDA architecture to use the NVidia Cards in the context of the MD code execution. The process involved converting the existing code from the previous assignment to run in a CUDA environment.

This project allowed us to understand that CUDA offers substantial performance benefits for specific types of parallelizable and computationally intensive tasks, like molecular dynamics simulations. It enables more efficient and scalable simulations compared to traditional CPU parallelization, especially for large-scale or complex systems. However, these benefits come with increased complexity in programming, dependence on specific hardware, and considerations regarding cost and energy consumption. Therefore, the choice between CUDA and CPU parallelization depends on the specific requirements of the task, available resources, and the expertise of the development team.

VII. Annexes

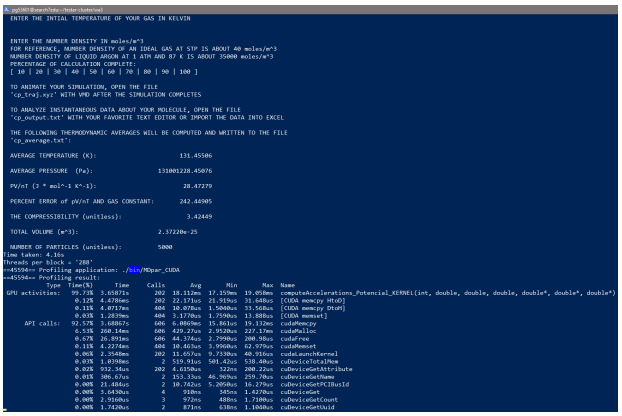

Fig. 6. Run 1 from MD with CUDA

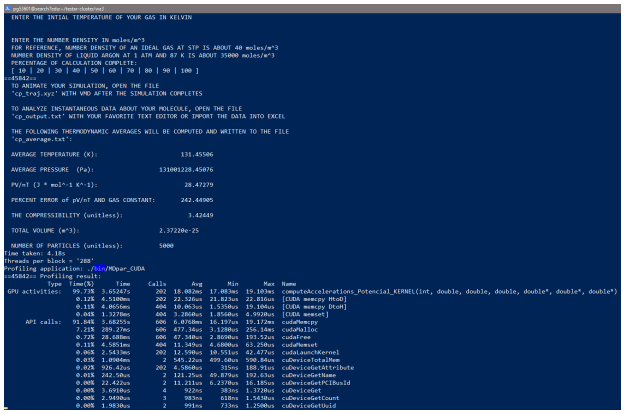

Fig. 7. Run 2 from MD with CUDA

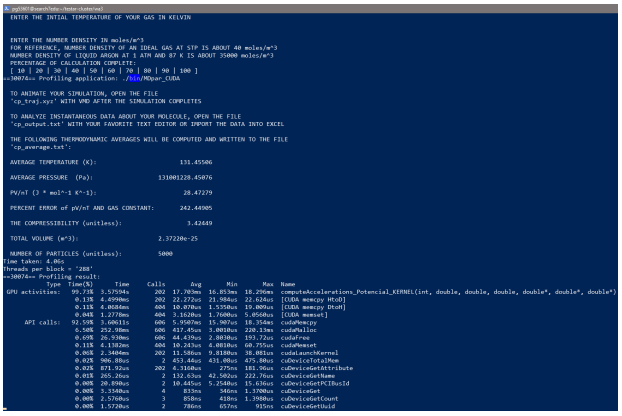

Fig. 8. Run 3 from MD with CUDA

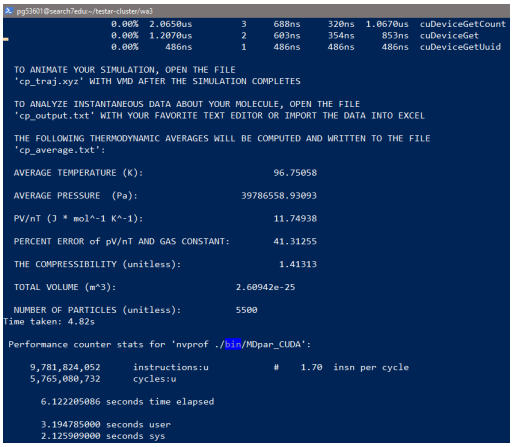

Fig. 9. Running 5500 particles

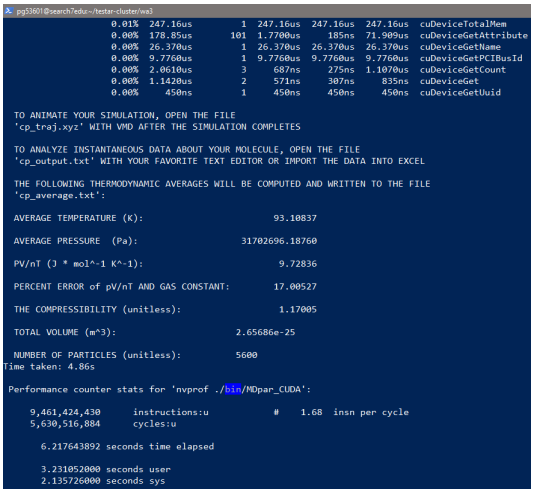

Fig. 10. Running 5600 particles

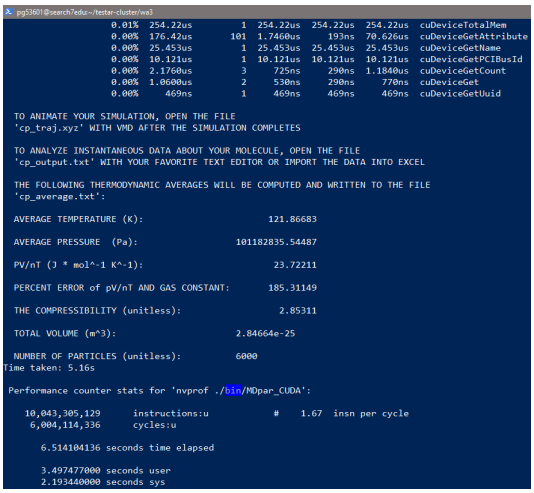

Fig. 11. Running 6000 particles

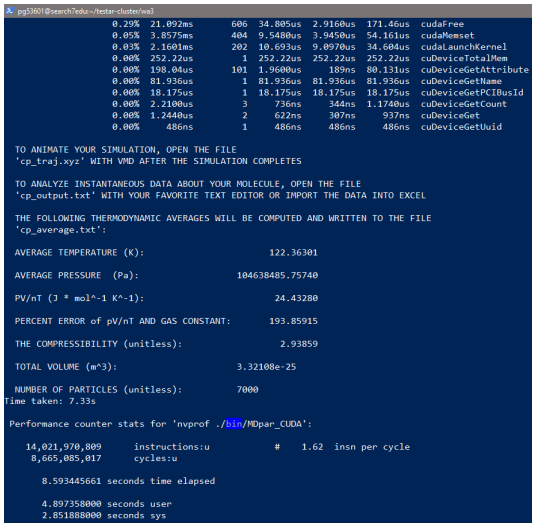

Fig. 12. Running 7000 particles

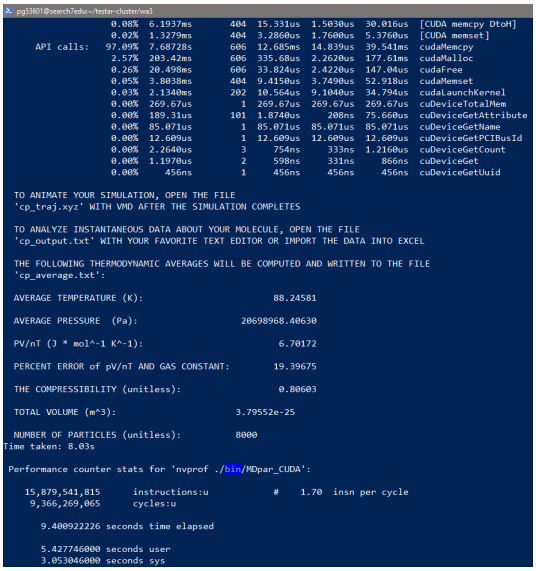

Fig. 13. Running 8000 particles

```
 AVERAGE TEMPERATURE (K):                        131.45506

 AVERAGE PRESSURE  (Pa):                   131001228.45076

 PV/nT (J * mol^-1 K^-1):                       28.47279

 PERCENT ERROR of pV/nT AND GAS CONSTANT:       242.44905

 THE COMPRESSIBILITY (unitless):                  3.42449

 TOTAL VOLUME (m^3):                         2.37220e-25

 NUMBER OF PARTICLES (unitless):             5000
Time taken: 50.37s

Performance counter stats for 'make runseq':

     189120066870      instructions:u            #    1,51  insn per cycle
     125457928568      cycles:u

    110,557909561 seconds time elapsed

     50,376665000 seconds user
      0,005077000 seconds sys
```

Fig. 14. Result from sequencial version of MD