Work Assignment
Phase 3

João Pedro Dias Faria
Dept. de informática - UC de Computação Paralela
Universidade do Minho
Braga, Portugal
pg53939

Rui Pedro Guise da Silva
Dept. de informática - UC de Computação Paralela
Universidade do Minho
Braga, Portugal
pg54213

***Abstract*—his document serves as an auxiliary support for the Parallel Computing course project, which corresponds to the academic year 2022/2023.**

***Index Terms*—C, SeARCH, OpenMP, CUDA, Parallelization.**

I. Introduction

In the concluding phase of our project, we have undertaken an ambitious initiative to enhance the execution time of our previously introduced algorithm. Our primary objective is to optimize the program further by implementing more efficient parallelization techniques, such as increasing the number of available threads without compromising on overhead or distributing algorithmic tasks among processes.

This endeavor aligns with our broader strategy, as our exploration extends to the domain of Graphics Processing Units (GPUs), renowned for their proficiency in parallel computations. GPUs create an environment where large, uninterrupted data blocks are efficiently handled, allowing numerous threads to operate simultaneously with minimal overhead. To harness this potential, we are incorporating CUDA (Compute Unified Device Architecture) into our development process.

Furthermore, this report serves as a detailed chronicle of our journey through these complex enhancements. It comprehensively details the technical challenges encountered, the strategic implementations pursued, and the resulting improvements in program performance.

II. WA1

The initial stage of this practical assignment involved implementing optimization techniques to minimize execution time. These techniques encompassed alterations to algorithms, enhancements in ILP (Instruction-Level Parallelism) and vectorization. Through complexity analysis and profiling, we identified two functions that consumed nearly 100% of the execution time: 'Potential()' with 54.92% of the total computational effort and 'computeAccelerations()', considering that it is executed approximately 201 times by the 'VelocityVerlet()' function, being responsible for 41.57% of the program's execution time.

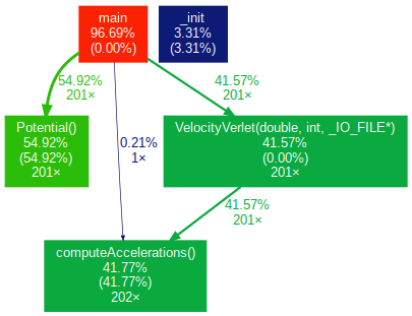

Fig. 1. Results from gprof2dot for the non-optimised code

In this initial phase, we simplified mathematical expressions to decrease the number of operations and modified the algorithms used. The most significant change in this first phase was the merging of the two functions into a single one, named 'computeAccelerationsOPT'.

III. WA2

In the second phase, after optimizing the sequential code, we explored parallelization techniques using OpenMP. To identify the function that takes up the most execution time, we used perf record on the sequential code from the first phase. As anticipated, the most time-consuming function is 'computeAccelerationsOPT,' and it was this function that applied parallelization. In summary, these were the changes made during this phase. We used **#pragma omp parallel for** on the outer loop so that each thread is responsible for a set of particles. All other variables are declared within this loop to remain private to each thread.Additionally, we used reduction for the variables 'a' and 'P', where a copy of the 'P' 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 addition was made because these variables are used outside the loop.The last change made in this phase was to add **schedule(dynamic, 40)** since the threads responsible for the last particles had less workload compared to the first ones.

IV. Tests and metrics to be evaluated

To assess the efficiency of our algorithm, we will employ two key performance metrics: execution time, which measures the total duration of algorithm execution, and Level 1 cache misses, which quantify the number of times data is retrieved from slower memory due to its absence in the CPU's primary cache. These metrics are strongly correlated with the overall performance improvement of an application, as reducing execution time and minimizing cache misses directly expedite processing and enhance overall system responsiveness.

The tests will be performed two computing platforms: the SEARCH Cluster, which has been provided by *Universidade do Minho* and has been extensively utilized in the earlier phases of this project, along with a personal laptop equipped with the following specifications:

Lenovo Legion	
OS	Linux Mint 21.2
CPU	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
RAM	16GB
GPU	NVIDIA GeForce GTX 1650 Ti

For each set of conditions, our team will execute 3 tests and calculate the average of the metrics, to further reduce variances inside the same machine.

V. Key Blocks

# Overhead	Samples	Command	Shared Object	Symbol
90.96%	1627	MDseq.exe	MDseq.exe	[.] computeAccelerationsOPT
9.01%	154	MDseq.exe	[unknown]	[k] 0xffffffffa158cbc0
0.01%	4	MDseq.exe	[unknown]	[k] 0xffffffffa0e6d5f8
0.01%	1	MDseq.exe	[unknown]	[k] 0xffffffffa0ee4b4f
0.00%	1	MDseq.exe	[unknown]	[k] 0xffffffffa1026a73
0.00%	1	MDseq.exe	libc-2.17.so	[.] __random
0.00%	1	MDseq.exe	[unknown]	[k] 0xffffffffa158c48a
0.00%	1	MDseq.exe	MDseq.exe	[.] initializeVelocities
0.00%	1	MDseq.exe	ld-2.17.so	[.] check_match.9525
0.00%	1	MDseq.exe	[unknown]	[k] 0xffffffffa0ffb653
0.00%	1	MDseq.exe	[unknown]	[k] 0xffffffffa159608e
0.00%	1	MDseq.exe	[unknown]	[k] 0xffffffffa0fa9edc
0.00%	1	MDseq.exe	ld-2.17.so	[.] __libc_memalign
0.00%	1	MDseq.exe	[unknown]	[k] 0xffffffffa1005afc
0.00%	1	MDseq.exe	[unknown]	[k] 0xffffffffa1213df1
0.00%	1	MDseq.exe	[unknown]	[k] 0xffffffffa0eea540
0.00%	3	perf	[unknown]	[k] 0xffffffffa0e6d5f8

Fig. 2. Output obtained using *perf record*

By analyzing the computationally intensive functions in our previous sequential version, it becomes evident that 'computeAccelerationsOPT' is the predominant function of this algorithm, responsible for a significant portion of the execution time.

Through the parallelization of this function utilizing OpenMP (Open Multi-Processing), we achieved a notable reduction in the algorithm's execution time. However, it persists as the most computationally intensive component within our application, indicating that the potential for substantial execution time enhancement lies in the pursuit of more advanced parallelization techniques.

VI. First version developed

A. Strategy

For our first version, we began by implementing CUDA only in the heaviest-weight functions, namely 'computeAccelerations' and 'computeAccelerationsOPT'. For this implementation, we needed to allocate arrays for the particle positions and accelerations, 'r' and 'a' respectively, as well as a variable 'P' related to the value of 'Pot'. Thus, the new variables 'da', 'dr', and 'dP' emerged. This accounts for all the necessary memory for this implementation and is carried out in the 'initializeKernel' function. In the 'initializeKernel' function, we took the opportunity to copy the particle positions to the device. We decided not to copy the array of accelerations since its initialization would be performed by a kernel.

After the kernel initialization, we focused on implementing 'computeAccelerations' with CUDA. In the OMP version, the 'a' matrix was initialized at the beginning of the function 'computeAccelerations' , and then its values were updated. Using CUDA, we couldn't perform this entire function with just one kernel call because we couldn't guarantee synchronization across all blocks. Thus, we couldn't ensure that all values were initialized before being updated. The solution found was to split the function into two kernel calls, one for initialization and another for updating, resulting in the functions 'initializeacelarationKernel' and 'newacelarationKernel' .This solution was found after some research and is commonly referred to as 'CPU synchronization'. This synchronization is performed by the host, which waits for all threads to finish before calling the next kernel.

We also use CUDA to assist whenever the 'VelocityVerlet' function wants to call the 'computeAccelerationsOPT' function.

This was replaced by two kernel calls,continuing with the same thought presented earlier, one is the same 'initializeacelarationKernel', and another to update their accelerations and calculate the new value of 'dP','newpotacelarationKernel'. Before these two calls, we need to copy the updated positions of the particles to the device, and after the calls, copy the new updates and the value of 'Pot' back to the host.

In the 'newacelarationKernel' and 'newpotacelarationKernel' function, each thread represents a particle and calculates the difference in acceleration that its iteration with the particles in front of it in the array will cause. Since two threads can modify the acceleration of the same particle, we use a 'myatomicAdd' function to write to the 'da' array. The purpose of using this function is to avoid data races, which were prevented in the previous phase version by the reduction. In the 'newpotacelarationKernel,' we also use this atomic function to add the value to 'dP'.

B. Results

V1		
Test	**TExec (s)**	**L1 Cache Misses**
N=5000, SEARCH	7.666	-
N=5000, Laptop	3.208	7280797

A more in-depth analysis of the obtained results will be conducted in one of the final chapters.

C. Profiling

```
==392== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.32%  5.79093s       201  28.811ms  28.778ms  28.852ms  newpotacelarationKernel(int
                    0.52%  30.441ms         1  30.441ms  30.441ms  30.441ms  newacelarationKernel(int, d
                    0.07%  4.2183ms       202  20.882us  20.576us  29.953us  [CUDA memcpy HtoD]
                    0.07%  3.9046ms       403  9.6880us  1.2790us  18.785us  [CUDA memcpy DtoH]
                    0.02%  903.24us       202  4.4710us  4.0960us  5.1840us  initializeacelarationKernel
      API calls:   96.47%  5.84729s       605  9.6649ms  39.008us  30.559ms  cudaMemcpy
                    3.43%  207.95ms         3  69.318ms  7.7110us  207.94ms  cudaMalloc
                    0.07%  4.0627ms       404  10.056us  6.1720us  82.136us  cudaLaunchKernel
                    0.01%  581.57us         1  581.57us  581.57us  581.57us  cuDeviceTotalMem
                    0.01%  338.98us       101  3.3560us     211ns  133.52us  cuDeviceGetAttribute
                    0.00%  277.72us       809     343ns     194ns  15.555us  cudaGetLastError
                    0.00%  233.90us         3  77.967us  4.7830us  171.88us  cudaFree
                    0.00%  223.51us         1  223.51us  223.51us  223.51us  cuDeviceGetName
                    0.00%  9.0250us         1  9.0250us  9.0250us  9.0250us  cuDeviceGetPCIBusId
                    0.00%  1.5080us         3     502ns     222ns     829ns  cuDeviceGetCount
                    0.00%  1.2260us         1  1.2260us  1.2260us  1.2260us  cuDeviceGetUuid
                    0.00%  1.0370us         2     518ns     209ns     828ns  cuDeviceGet
```

Fig. 3. Profiling - version 1

From this analysis, we can identify relevant aspects of our implementation.

- The cudaMalloc API call has a small overhead,taking around 207.95 to process.
- The 'initializeacelarationKernel' was call 202 times, averaging 4.47 us per call, taking a total of 903 us to execute.
- The 'newpotacelarationKernel' takes 5.79 seconds to complete the 201 calls, averaging 28.811 ms per call. This elevated value is due to the use of 'myatomicAdd', which hinders the parallel approach.
- Similary, the 'newacelarationKernel' takes 30ms in its single call.

VII. Second version developed

A. Strategy

The use of our version of 'atomicAdd' for double adds significant overhead to our implementation. Therefore, in this new version, our main goal was to eliminate dependency on this function to calculate the new accelerations and continue without data races in our program.

With this goal in mind, we realized that, to avoid using 'atomicAdd', each thread could only make changes to the acceleration of the particle ('da') it corresponds to. Thus, a change in the algorithm was necessary, so that instead of each particle calculating its new acceleration and that of the particle it interacts with ,those in front in the array, each particle now only calculates its acceleration resulting from its interaction with all the existing particles. With this new implementation, each thread modifies different positions in the array,since they only change the positions of the corresponding particles, eliminating the need for the use of atomic add. The function 'newpotacelarationKernel' was renamed to 'newpotacelarationKernel1Part' with this new implementation.

To handle the 'atomicAdd' of the 'Pot' variable, instead of allocating space for a single double, we started allocating space for each existing particle. In the 'newpotacelarationKernel1Part' function, each thread adds the calculated potential from the iterations of its particle relative to those in front of it, in a position of the new array. This array is copied to the host, which adds all the values to determine the total potential after the new iteration.

With this new implementation, all dependencies on 'atomicAdd' were removed, achieving the goal of not relying on this function for our implementation, resulting in an improvement in the execution time.

B. Results

V2		
Test	**TExec (s)**	**L1 Cache Misses**
N=5000, SEARCH	5.127	-
N=5000, Laptop	3.199	7754848

We can see that our search execution time had an improvement from 7.666 seconds to 5.127 seconds

C. Profiling

```
==25226== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.71%  1.12302s       201  5.5871ms  5.5752ms  5.6041ms  newpotacelarationKernel1Part
                    0.43%  4.8818ms       403  12.113us  6.0480us  18.592us  [CUDA memcpy DtoH]
                    0.40%  4.5149ms         1  4.5149ms  4.5149ms  4.5149ms  newacelarationKernel(int, do
                    0.38%  4.3663ms       202  21.615us  21.280us  22.336us  [CUDA memcpy HtoD]
                    0.08%  871.20us       202  4.3120us  4.1920us  4.8320us  initializeacelarationKernel(
      API calls:   82.41%  1.16280s       605  1.9220ms  47.972us  5.6590ms  cudaMemcpy
                   17.03%  240.34ms         3  80.112ms  8.5290us  240.32ms  cudaMalloc
                    0.36%  5.0280ms       404  12.445us  6.8000us  75.971us  cudaLaunchKernel
                    0.08%  1.1139ms         2  556.95us  518.13us  595.77us  cuDeviceTotalMem
                    0.07%  972.36us       202  4.8130us     298ns  207.68us  cuDeviceGetAttribute
                    0.02%  264.84us         2  132.42us  59.882us  204.95us  cuDeviceGetName
                    0.02%  240.27us       809     297ns     192ns  7.9050us  cudaGetLastError
                    0.01%  184.64us         3  61.546us  4.1110us  168.45us  cudaFree
                    0.00%  19.042us         2  9.5210us  3.9450us  15.097us  cuDeviceGetPCIBusId
                    0.00%  3.6730us         4     918ns     394ns  1.4040us  cuDeviceGet
                    0.00%  2.5940us         3     864ns     429ns  1.4130us  cuDeviceGetCount
                    0.00%  1.6800us         2     840ns     590ns  1.0900us  cuDeviceGetUuid
```

Fig. 4. Profiling - version 2

- The cudaMalloc API take around 240.34ms to process. This increase is due to the fact that we allocate space for more values.
- The 'initializeacelarationKernel' average 4.31 us per call, taking a total of 871 us to execute.
- 'newpotacelarationKernel1Part' takes now 1.12 seconds to complete the 201 calls,averaging 5.5871 ms per call.So, we can see that removing the atomics resulted in an improvement of approximately 5.17 times, for this function.
- This improvement was also observed in the 'newacelarationKernel' function, which now takes 4.5 ms in its call.
- The CUDA memcpy from device to host and from host to device took 4.88 ms and 4.36 ms, respectively.

VIII. Third version developed

A. Strategy

In this latest implementation, our goal is to avoid constant copying between the host and device and vice versa, and to

make a better implementation in the calculation of 'dP', using shared memory.

To avoid constant copying between the host and device, we need all arrays related to particles to be updated within the device, so that it always has their values updated. To achieve this, we need to allocate space for the velocity array and start executing the 'VelocityVerlet', 'MeanSquaredVelocity', and 'Kinetic' functions on the device.

The 'VelocityVerlet' function was divided into several functions for reasons similar to those mentioned in the implementation of the 'newacelarationKernel' and 'newpotacelarationKernel'. These new functions include the 'posvelKernel','updatevelKernel' and 'elasticKernel',in addition to those already created, 'initializeacelarationKernel' and 'newpotacelarationKernel1Part'.

While implementing these new functions, we faced the same challenge that 'dP' had caused us, as we needed to reduce the values calculated by the threads to a single value. Since one of the objectives of this latest implementation was to take advantage of shared memory in the calculation of 'dP', we used the same solution for each of the problems.

This solution involves each block having a shared array of size equal to the number of threads per block used. Instead of entering its value into the global array, each thread inserts the value into the shared memory array at the position equivalent to its ID in the block. We synchronize the threads of the block using the __synchronize() function to ensure that all values are in the array. After this synchronization, we can perform a parallel reduction, with the total value calculated by the block residing in the first position of the array. The first thread of each block writes the obtained result to the global array, which is initialized not with a size of N but with the number of blocks used, in this implementation.

This implementation is used in the function 'elasticKernel','MeanSquaredVelocitykernel','Kinetickernel',each one with a specific global array being 'dPress','dmvs' and 'dKE' respectively. The space for these arrays is allocated during the 'initializeKernel' and is influenced by the number of blocks to be used. Following the same reasoning, 'dP' now allocates space only for the number of blocks instead of the number of particles.

The aggregate value of all these global arrays is calculated by a single thread in the function 'reduce'. The summed value of each one is stored in a specific position of the results array.

Since the values of 'Press', 'mvs', 'KE' and 'P' can be obtained on the host simultaneously, so instead of having a variable for each value, we this 'results' array where each position represents one of the necessary values. To have space on the device to store these results in the 'initializeKernel', we also allocate space for the respective array,as mentioned earlier. This way, we avoid multiple calls to 'memcpy'.

After each reduce call, we copy the 'dresults' array to the host to obtain the values we have just calculated.

B. Results

Final		
Test	**TExec (s)**	**L1 Cache Misses**
N=5000, SEARCH	2.981	-
N=5000, Laptop	3.202	5396658

We can see that our search execution time had an improvement from 5.127 seconds to 2.981 seconds to

C. Profiling

```
==38303== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.41%  1.13956s       201  5.6694ms  5.6553ms  5.6915ms  newpotacelarationKernel1Part
                    0.44%  5.1088ms       201  25.416us  25.217us  25.921us  reduce(double*, double*, dou
                    0.39%  4.5095ms         1  4.5095ms  4.5095ms  4.5095ms  newacelarationKernel(int, do
                    0.17%  2.0001ms       201  9.9500us  9.7280us  10.369us  posvelKernel(int, double*, d
                    0.15%  1.7397ms       201  8.6550us  8.3200us  9.4720us  elasticKernel(int, double*, 
                    0.12%  1.3561ms       201  6.7460us  6.5920us  6.9760us  MeanSquaredVelocitykernel(in
                    0.12%  1.3364ms       201  6.6480us  6.4320us  7.2010us  Kinetickernel(int, double*, 
                    0.11%  1.2392ms       201  6.1650us  6.0160us  6.5920us  updatevelKernel(int, double*
                    0.07%  824.98us       202  4.0840us  4.0000us  4.8000us  initializeacelarationKernel(
                    0.03%  293.95us       202  1.4550us  1.2800us  18.592us  [CUDA memcpy DtoH]
                    0.00%  42.560us         2  21.280us  20.864us  21.696us  [CUDA memcpy HtoD]
      API calls:   83.41%  1.15291s       204  5.6515ms  73.102us  5.7340ms  cudaMemcpy
                   15.53%  214.70ms         8  26.837ms  3.7090us  214.66ms  cudaMalloc
                    1.01%  13.932ms      1610  8.6530us  5.8890us  80.124us  cudaLaunchKernel
                    0.02%  282.89us         1  282.89us  282.89us  282.89us  cuDeviceTotalMem
                    0.01%  153.64us       101  1.5210us     149ns  61.346us  cuDeviceGetAttribute
                    0.01%  149.45us         8  18.681us  2.7020us  120.61us  cudaFree
                    0.00%  66.937us       206     324ns     220ns  3.6640us  cudaGetLastError
                    0.00%  29.650us         1  29.650us  29.650us  29.650us  cuDeviceGetName
                    0.00%  9.1890us         1  9.1890us  9.1890us  9.1890us  cuDeviceGetPCIBusId
                    0.00%  1.6310us         3     543ns     251ns     907ns  cuDeviceGetCount
                    0.00%     997ns         2     498ns     196ns     801ns  cuDeviceGet
                    0.00%     506ns         1     506ns     506ns     506ns  cuDeviceGetUuid
```

Fig. 5. Profiling - version 3

- The cudaMalloc API take around 214ms to process to allocate all the space necessary. In this way, although more arrays are allocated, the new approach with shared memory helps with the time because we allocate smaller-sized arrays.
- The 'initializeacelarationKernel' average 4.08 us per call, taking a total of 824 us to execute.
- 'newpotacelarationKernel1Part' and 'newacelarationKernel' take approximately the same time as in the last version.
- The 'reduce' average 25.416 us per call,taking a total of 5.1ms.
- The 'posvelKernel' average 9.95 us per call,taking a total of 2ms.
- The 'elasticKernel' average 8.65 us per call,taking a total of 1.73ms.
- The 'MeanSquaredVelocityKernel' average 6.74 us per call,taking a total of 1.3ms.
- The 'KineticKernel' average 6.64 us per call,taking a total of 1.33ms.
- The 'MeanSquaredVelocityKernel' average 6.74 us per call,taking a total of 1.3ms.
- The CUDA memcpy from device to host and from host to device took 293.95 us and 42.56 us, respectively.

IX. Tests and analysis

A. General comparison

To better understand the evolution of the execution time throughout this work, we started by presenting the results for a number of particles equal to 5000 (N=5000). It was also suggested in this phase to use a personal computer to analyze how our program behaves on another machine. Thus, we once again analyzed how the execution time varies for the various versions and compared the results with those obtained on the search machine. The laptop used has the specifications mentioned earlier. Presenting the execution time for the three versions created in this phase.

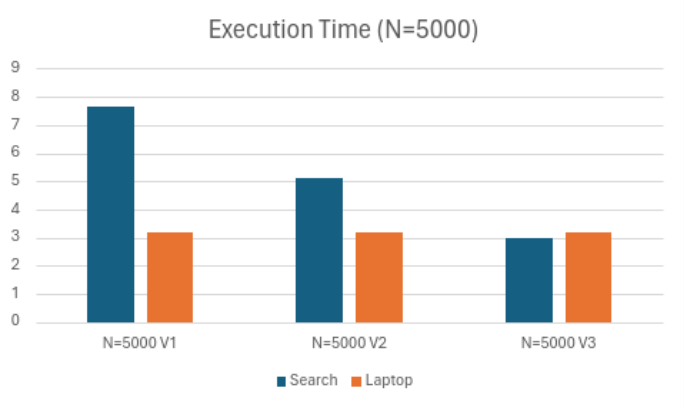

Fig. 6. Execution time for the last version (N=5000)

	Search	Laptop
N=5000 V1	7,666	3,208
N=5000 V2	5,127	3,199
N=5000 V3	2,981	3,202

Fig. 7. Table execution time for the last version (N=5000)

B. L1 Cache Misses

We also wanted to explore other metrics in addition to those used in previous phases. Therefore, we decided to analyze the L1 cache misses between the same versions compared earlier. However, we were unable to make a proper comparison with the results obtained on the Search for this type of metric, as we encountered some difficulties in using the metric on the Search cluster. Nevertheless, we decided to retain this metric, confirming by reading the bar chart presented below that our efforts to improve the final program were indeed successful, resulting in a reduction in the number of L1 cache misses from version 2 to version 3.

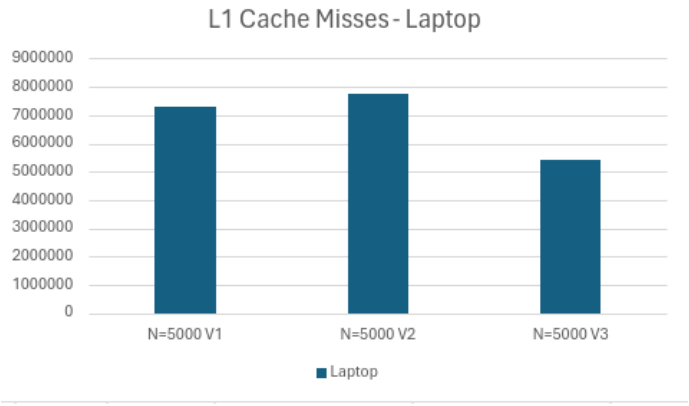

Fig. 8. L1 cache misses on laptop

	Laptop
N=5000 V1	7280797
N=5000 V2	7754848
N=5000 V3	5396658

Fig. 9. Table L1 cache misses - Laptop

The reduction in L1 cache misses in the last version is related to the fact that more calculations are performed on the GPU. It's normal for the number of cache misses by the CPU to decrease for this version.

C. Scalability analysis

To better analyze the scalability of our program, we tested how the execution time varies with changes in the number of particles. For each phase, we assessed its scalability based on the value of N and the required execution time. Once again, we performed this operation both on the Search and on our personal laptop, and obtained the results presented below.

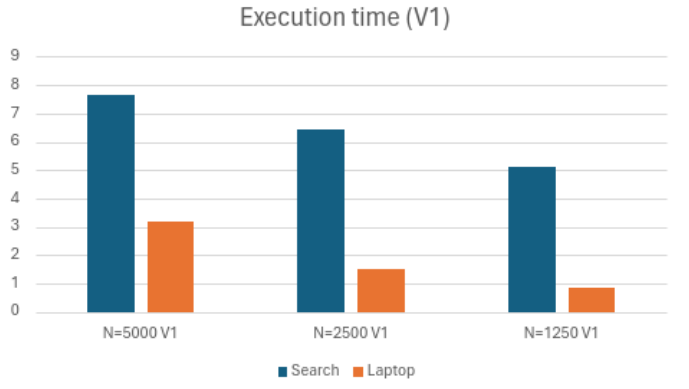

Fig. 10. Execution time - version 1

	Search	Laptop
N=5000 V1	7,666	3,208
N=2500 V1	6,431	1,504
N=1250 V1	5,132	0,87

Fig. 11. Table execution time - version 1

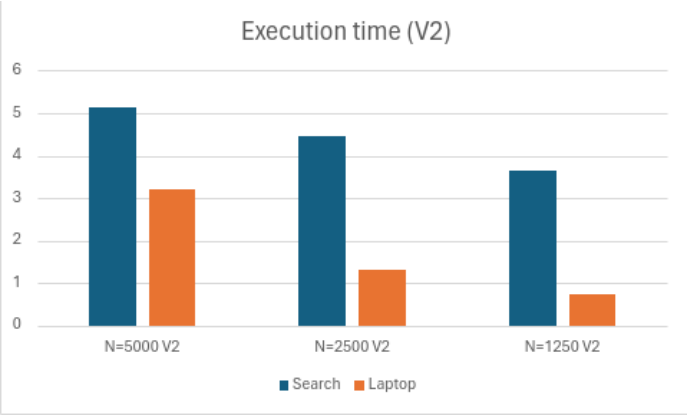

Fig. 12. Execution time - version 2

	Search	Laptop
N=5000 V2	5,127	3,199
N=2500 V2	4,445	1,311
N=1250 V2	3,638	0,735

Fig. 13. Table execution time - version 2

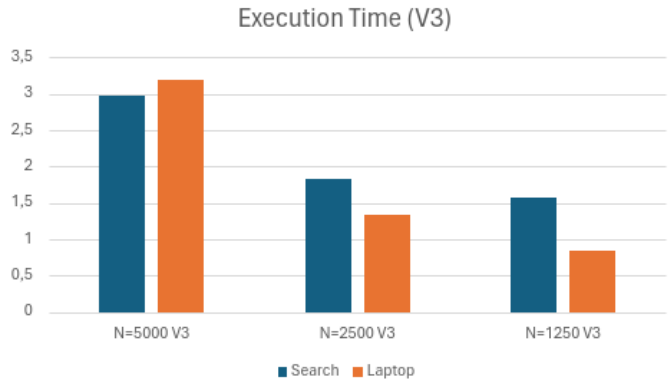

Fig. 14. Execution time - version 3

	Search	Laptop
N=5000 V3	2,981	3,202
N=2500 V3	1,834	1,338
N=1250 V3	1,57	0,833

Fig. 15. Table execution time - version 3

The reasons for the increase in execution time with the increase in the number of particles may be related to the competition for resources among threads. This competition may intensify as the number of particles increases, negatively impacting the ideal scalability.

X. Conclusion

In summary, although the execution time was not better than that obtained in the previous phase, we consider that we made a correct implementation of our problem using CUDA,being aware of the limitations that arose during its execution. This work helped us to better understand how to program using the GPU. This phase also helped to better understand the difficulty of performing parallel reductions and the challenges they pose to programmers in implementing these algorithms.