
University of California Berkeley

Computer Science Department

CS 267

# Assignment 2.3: Parallel Particle Simulation on GPU with CUDA

*Submitted by*
Cristobal Pais
Alexander Wu
CS 267
March 9th
Spring 2018

# Contents

# Assignment 2.3: Parallel Particle Simulation on GPU with CUDA

C.Pais, A.Wu

March 9th 2018

**Abstract**

An optimized code for a GPU multi-threaded particle simulation algorithm in the context of efficient algorithms implementations in C is developed for the Pittsburgh Supercomputing Center's (PSC) Bridges cluster. It is programmed using the NVIDIA's parallel computing platform and programming model CUDA. The optimization process is performed in a cumulative approach based on a theoretical analysis of the vanilla implementation provided with complexity $O(n^2)$ identifying its bottlenecks and potential improvements for decreasing its complexity up to a linear complexity $O(n)$. The solution approach consists of a simple mapping between the particles system and a grid/matrix structure allowing us to decrease the number of interaction checks performed per simulation step by defining the concept of neighbors cells, checking only a limited amount of interactions. A set of methods that maps the particles into a certain cell of the mesh are implemented as the main data structure. Best serial implementation reaches a linear complexity $O(n)$. The optimal multi-threaded GPU implementation developed reaches average speedup factors of 36.27, 2.97, and 2.27 for instances including $[500, 1000000]$ particles, improving the performance of our previous optimized serial code, shared memory implementation (OpenMP) and distributed memory approach (MPI), respectively. Further improvements are discussed.

## 1 Introduction

A simple two-dimensional bounded particle system where particles can interact with each other as well as hit the walls of the bounded regions is implemented. This small but challenging problem is useful for understanding more complex systems implementations used in different areas such as biological sciences, physics, and engineering, as well as test our abilities to decrease the complexity of a naive simulation approach and exploit the potential parallelism of the code via the CUDA GPU approach, a parallel programming paradigms that exploits the NVIDIA's graphic cards GPUs.

In this project, a discretized time simulation of a box with a constant average density of particles (scaled) is implemented. Each simulation consists of $T = 1000$ time steps where each particle interacts with other particles that are within a particular radius/zone per time step. The vanilla implementation provided has two main components: (1) Compute forces: forces are calculated for each particle taking into account the interactions with its neighbors, calculating its acceleration, after this step (2) Move particles: the movement of each particle is calculated based on its acceleration and current velocity including the potential effect of the existing walls of the region (inelastically bouncing).

The challenge of the current assignment is to implement an efficient GPU multi-threaded version of the particle simulation system by using optimal data structures, eliminating unnecessary steps/computations, and parallelization tricks reaching a linear complexity performance $O(n)$. On the other hand, we seek for high-performance results with respect to the speedup attained by our algorithm when comparing it to our optimized serial implementation as well as the shared (OpenMP) and distributed (MPI) memory approaches.

A C program containing CUDA instructions will be developed by the team using the code provided with the assignment - as well as our previous implementations - main starting point: the optimized serial version using the efficient grid data structure that allowed the team to obtain a linear complexity $O(n)$ in the problem.

The main solution developed by the team is based on keeping a data structure that allows us to check as the minimum number of particles as possible when computing the forces and moving the particles inside the

simulation. Once the CUDA GPU version of the code is optimized and the data structure parallelization is studied, a comparison between the previous implementations, i.e. optimized serial, distributed (MPI) and shared memory (OpenMP) approaches will be performed.

The structure of the report is as follows: In section 2, the main optimization methodology is described as well as the member's contribution and hardware description. Section 3 describes the tested optimization approaches and their potential impact. In section 4, results are discussed for the different optimization techniques, as well as defining and describing the main algorithm developed. A performance comparison with all previous approaches is included as a summary of the complete project, concluding the advantages/disadvantages of each implementation. Finally, section 5 contains the conclusions and future work of the project.

# 2 Methodology

The main methodology of this project consists of performing an optimization of the original particle simulation code (in C) provided with the course material, analyzing the different programming techniques/tricks that can be implemented in order to obtain a better performance, allowing us to reach an $O(n)$ complexity for the serial version while seeking improved running time for the parallel code. The performance is measured using the *autograder* code provided with the assignment when running instances with a different number of particles, simplifying our analysis for both implementations: the slope of the serial version, as well as the speedup factors of the parallel code, are automatically calculated from the summary of our results.

In the previous reports, we analyzed the "AS-IS" performance of the simulator original files provided with the statement of the assignment. Once the initial results were obtained, different data structures were tested in order to achieve the $O(n)$ linear running time for the serial, OpenMP, and MPI versions. In this case, a CUDA GPU C code version is implemented using the previous codes as a starting point.

Several experimental instances differing in the $n$ value are tested, with $n \in [500, 1500000]$ for testing the performance of our code. Depending on the techniques applied for optimizing the original algorithm, a series of parameters are optimized for each approach, performing the experiments - mainly following a brute force rule - and keeping the best solutions achieved.

## 2.1 Contribution

During the development of the project, each member of the team develops its own experiments, sharing his results in a common Github repository for checking the current state of the implementations while re-using code from other members code. In addition, cross-testing of other members' code is performed. After performing a series of experiments, the best implementation is selected as the main submission file.

Based on the experience and preferences of the members of the group, the members' contributions and tasks can be summarized as follows:

- **Cristobal Pais**
  - Optimized CUDA GPU version
  - Editor and main writer of the current report.
- **Alexander Wu**
  - Extra optimization of the CUDA GPU version.
  - Contribution to the report.

## 2.2 Hardware & Software

The optimization of the particle simulation algorithm is developed for a specific hardware and runtime environment. In this case, we are not implementing our code in the Cori supercomputer from The National Energy Research Scientific Computing Center (NERSC). Instead, we implement our code in the Pittsburgh Supercomputing Center's (PSC) Bridges cluster.

All experiments, benchmarks, and performance results for the GPU code are implemented using the following hardware and software:

### 2.2.1 Hardware

**PSC Bridges Regular Shared Memory GPU partition (16 nodes)**

- 2x Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
  - L1-Cache: 32Kb
  - L2-Cache: 256Kb
  - L3-Cache: 40Mb-shared
- 2x Nvidia K80 GPUs.
  - Number of processor cores: 2496
  - Memory clock 2.5 GHz
  - Base core clock: 560 MHz
  - Memory size 12 GB GDDR5

### 2.2.2 Software

- CentOS Linux 7.3.1611
- Built with PrgEnv-intel and PrgEnv-gnu/ gcc version 4.8.5 / nvcc
- Default Compiler and flags: nvcc -O3 -arch=compute_37 -code=sm_37

## 2.3 Assumptions

Based on the statement of the problem, our implementation of the particle simulator program satisfies the following assumptions:

- A bounded square region contains the entire system. Interactions between particles and walls are inelastic.
- Density of the whole particle system is kept constant when modifying the number of particles inside the grid, scaling its size.
- Interactions between particles occur within a bounded region.
- Linear scaling between work and processors.

# 3 Solution Description

In this section, we present the main strategies applied for optimizing the original GPU version (Vanilla version provided with the assignment) of the particle simulator. First, as done in previous parts of this project, the strategy applied to obtain a serial version of the code with an asymptotic complexity of $O(n)$ is summarized, consisting of a matrix/grid data structure for modeling the relevant interactions between particles and their neighbors instead of checking the $n-1$ particles for every element. Then, a specific analysis of the CUDA GPU implementation is performed, indicating the main modifications and techniques used for getting the best possible performance.

More details about the original data structure used for the parts 1 and 2 of the project (serial, OpenMP, and MPI implementations) can be seen in our previous reports.

## 3.1 GPU grid

For the first 3 implementations (Serial, OpenMP, and MPI), we used a grid data structure that consisted of a $2-D$ matrix of dynamically resizing arrays to maintain quick access to local particles. However, we found that copying many dynamically sized arrays to GPU memory didn't scale as well as we wanted to. As a result, we looked towards different methods to avoid dynamically resized objects, yet still, maintain the $O(n)$ overall runtime per iteration.

The data structure we ended up using was very similar to an adjacency list. We maintain three buffers in GPU memory

- a buffer $dParticles$ of length $n$, a copy of the $particles$ into the GPU.
- a buffer $subMesh$ of length $gridLength \times gridLength$, where each entry is an integer $i$ between 0 and $n-1$, inclusive. We label each "square" in our grid: $i$ being $k^{th}$ entry of the buffer corresponds to the $i^{th}$ particle in $dParticles$ being the head of the adjacency list for the grid square labeled $k$.
- a buffer $adj$ of length $n$, where each entry is an integer $i$ between $-1$ and $n-1$, inclusive. If the $k^{th}$ particle is at the end of its containing adjacency list, then $adj[k] = -1$. Otherwise, $adj[k] = j \in [0, n-1]$, where $j$ represents the index in $dParticles$ of the entry in the adjacency list that follows the particle indexed at $k$.

This structure is inspired in examples from [1], [3], and [5], as well as the experience of the team members with simulation problems using a similar approach for modeling the underlying data structure.

## 3.2 GPU CUDA Algorithm

Using the previously described grid data structure as well as our knowledge obtained from the previous parts of the current project, we develop a pure CUDA code of the particle simulator that is able to reach a linear asymptotic complexity as well as obtaining significant speedups in comparison to our previous implementations (Serial, OpenMP, and MPI).

Therefore, our algorithm works as follows:

i) *Resources Initialization*: First, we allocate buffers on the GPU and populate them accordingly. Particles are initialized (in serial) and are being copied to the GPU using the $cudaMemcpy()$ function.

   A synchronization between all threads is performed in order to create the underlying mesh structure.

ii) *Grid initialization*: The work loop starts out by setting up our GPU grid. We start by running the CUDA function *clear*, which wipes out all the adjacency lists. The $clear()$ function is embarrassingly parallel, allowing us to easily take advantage of our parallel implementation.

iii) *Filling the grid*: Then, the $clear()$ function is followed by the CUDA function $push2mesh\_gpu()$, which populates the adjacency list of each grid square according to the state of the particles inside $dParticles$.

This is nearly embarrassingly parallel, except for the fact that some careful synchronization has to be done when updating the adjacency list pointers. We decide to use an atomic Compare-and-swap in NVIDIA's helper function *atomicExch*() (see [3]) between the current head of the list and the particle we intend to add to it due to its simplicity of implementation as well as for the great performance achieved during our tests.

iv) *Updating the grid*: We decided to use a "brute force" update approach instead of being clever about lazily updating (like we already implemented in our previous serial, OpenMP, and MPI versions) because our update code has no branching, maximizing throughput while minimizing code complexity. Future implementations would likely take this step into account as an initial optimization point for further improvements.

v) *Particles' interaction*: For the actual particle interaction, we naively perform the *apply_force*() on all the particles in the GPU. This step is embarrassingly parallel. We follow this by performing *move*() - identically to the Vanilla implementation - on the GPU. We do not update the state of the grid in the move function, instead, we only update the location of each particle, so this step is also embarrassingly parallel.

vi) *Release resources*: After the main simulation loop is completed, we free up the corresponding resources and finalize the code execution.

### 3.2.1 Other techniques tested

Besides the previously discussed characteristics/techniques included in the submission code, we tested different approaches in the road to our final implementation. The main approaches tested were the following:

i) *Thrust library*: as described in its website [3], "Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL). Thrust's high-level interface greatly enhances programmer productivity while enabling performance portability between GPUs and multicore CPUs". We tried some implementations using this library since it adds a simple support of features like vectors and iterators in CUDA, giving us a series of tools for developing high-performance applications without a significant extra effort.

The initial idea was to develop and code the underlying grid/mesh structure using the extra functions/features provided by the library, following the examples from the library's website. However, we were not able to obtain the maximum performance from this library due to our lack of experience using CUDA so we decided to keep our code as simplest as possible.

In addition, as informed via the course Piazza's website the idea of this part of the project is to use pure CUDA code as much as possible and therefore, we dropped our initial work with this very powerful library, leaving it as future work implementations.

ii) *Original grid*: Initially, we tried to adapt our original grid/mesh class developed for the optimized serial version in the initial part of this project. However, we were not able to successfully modify it in order to obtain a significant performance boost using the GPU approach with CUDA: a very poor performance code was obtained after a series of efforts by the members of the group, leading us to decide programming a new mesh "structure" from scratch based on our previous implementation and knowledge of parts 1 and 2 problems.

Thus, we decided to implement a specific grid structure for the current GPU approach, allowing it to take the maximum advantage of its parallel implementation as well as programming characteristics without incurring in a significant programming effort.

# 4 Results & Discussion

In this section, we present the main results obtained from a series of experiments including the optimization techniques introduced in section 3.

## 4.1 Preliminary results: Vanilla version

Using the code from the first parts of this project as the basic structure of our GPU implementation as well as the vanilla implementation provided with the assignment statement, we obtain a series of preliminary results for benchmarking our optimized code.

### 4.1.1 Vanilla GPU

When running the AS-IS/Vanilla version of the code provided with the assignment, we can clearly see in Figure 1 that the implementation is not very efficient and taking advantage of the GPU parallelism: the complexity of the simulation routine is $O(n^2)$ due to extra work checking all interactions between a particle and the $n-1$ inside the simulation environment. Clearly, this naive approach is not efficient. Therefore, the first optimization step will consist of adapting the underlying data structure of our previous implementations (mesh/grid) to the current GPU approach, as discussed in section 3.

When running the Vanilla version against our serial optimized code, we obtain a very poor average speedup factor around 0.35 for small and large instances. This factor tends to increase and reach its best value around $n = 10,000$ particles where the speedup factor is close to 0.6, after this threshold, it significantly decreases reaching a very poor speedup value around 0.1 for the largest tested instance with $n = 150,000$ (in order to avoid out-of-quota problems since the Naive GPU implementation takes around 20 minutes for $n = 150,000$), as seen in Figure 2 where speedup factors are plotted against the different values of $n$ tested. Therefore, the code is not taking advantage of the parallel GPU implementation and its performance is even worse than our serial optimized code, as expected since its complexity is around $O(n^2)$ (slope = 1.346650 using autograder with $n <= 150,000$) while our optimized serial approach is able to to get an $O(n)$ complexity due the underlying data structure implemented, as discussed in section 3 and previous reports of this project.

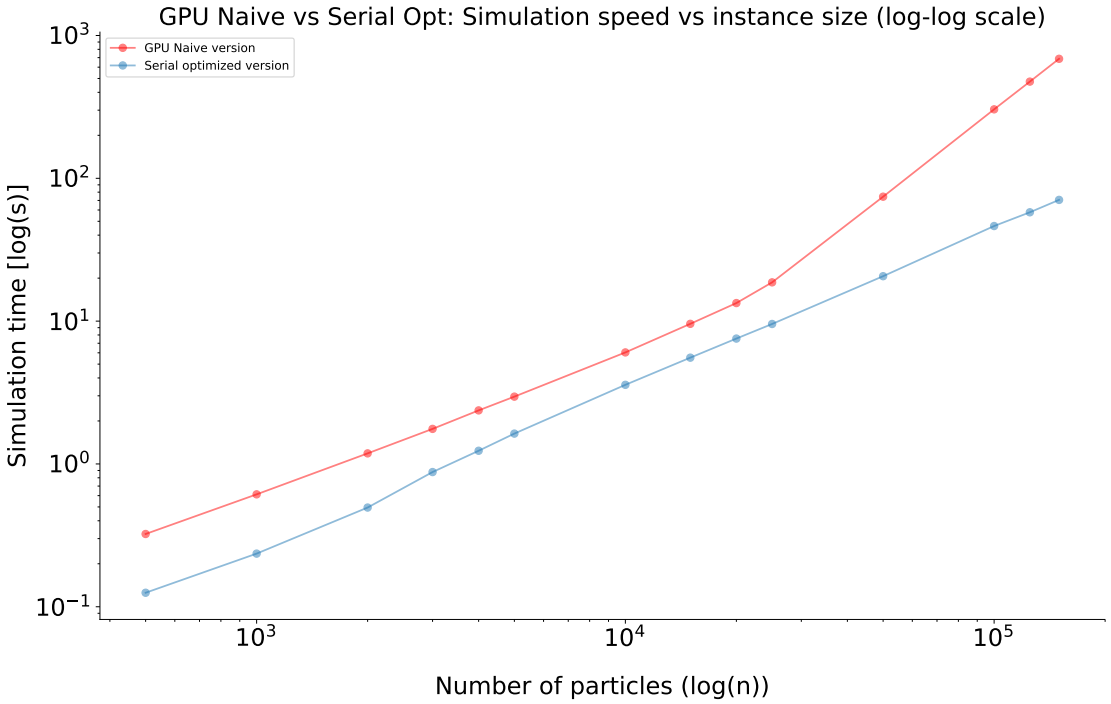


Figure 1: Performance of the Vanilla GPU code with respect to the number of particles $n$ simulated (log-log scale).

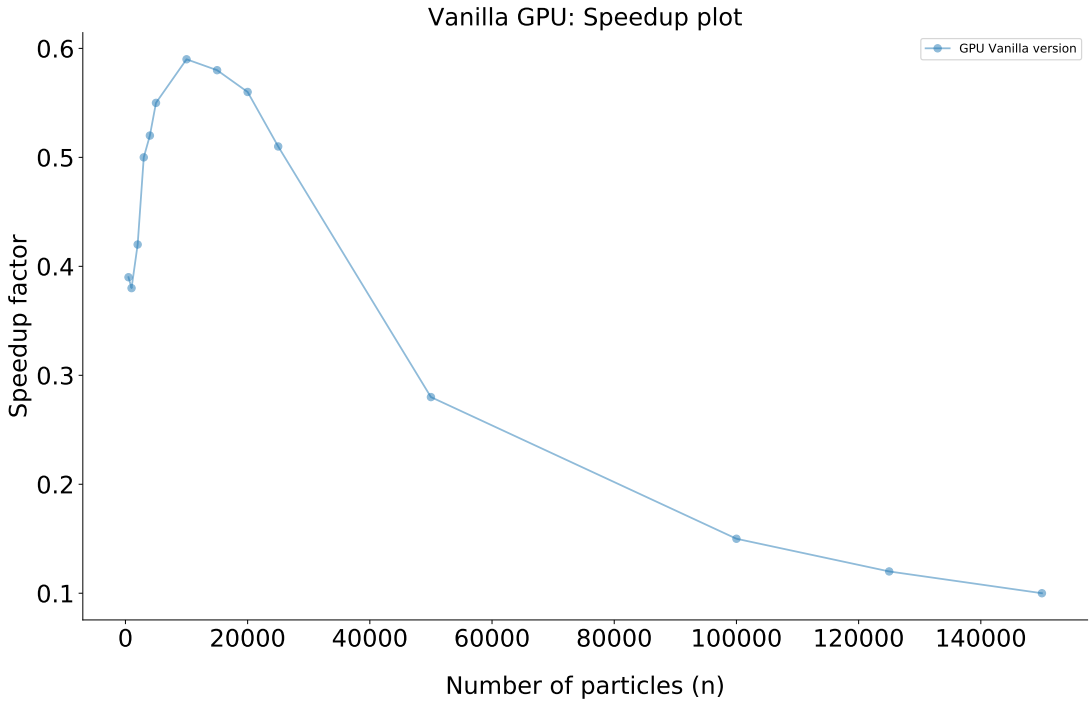


Figure 2: Speedup factors evolution for the Vanilla GPU implementation for different $n$ values with respect to the optimized serial version.

Comparing the Vanilla version with respect to our optimized implementation using the mesh approach for the GPU code, we can see in Table 1 and Figures 3, 4 how our optimized version is able to reach a linear complexity (in fact, almost sublinear), obtaining a significant performance boost with respect to the Vanilla implementation.

Table 1: Comparison results between the Vanilla/Naive and optimized GPU implementations, $n \in [500, 150000]$. Larger instances are not included due to potential quota problems with the Vanilla GPU implementation times.

| **Size $n$** | **Time GPU Opt [s]** | **Time GPU Vanilla [s]** |
|---|---|---|
| 500 | 0.059 | 0.323 |
| 1000 | 0.062 | 0.612 |
| 2000 | 0.065 | 1.185 |
| 3000 | 0.065 | 1.759 |
| 4000 | 0.077 | 2.370 |
| 5000 | 0.079 | 2.959 |
| 10000 | 0.100 | 6.037 |
| 15000 | 0.147 | 9.573 |
| 20000 | 0.176 | 13.380 |
| 25000 | 0.218 | 18.685 |
| 50000 | 0.449 | 74.362 |
| 100000 | 0.986 | 304.053 |
| 125000 | 1.329 | 475.390 |
| 150000 | 1.685 | 686.676 |
| **Estimated Slope** | 0.784688 | 1.346650 |

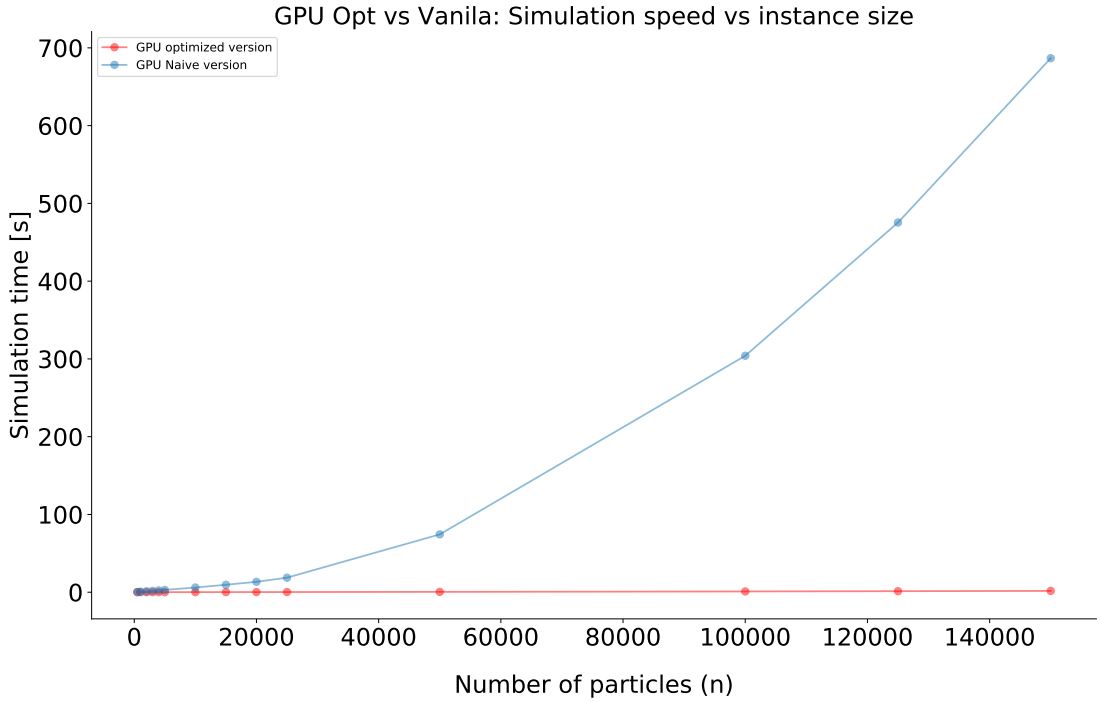


Figure 3: Performance of the optimized GPU code versus the vanilla GPU implementation.

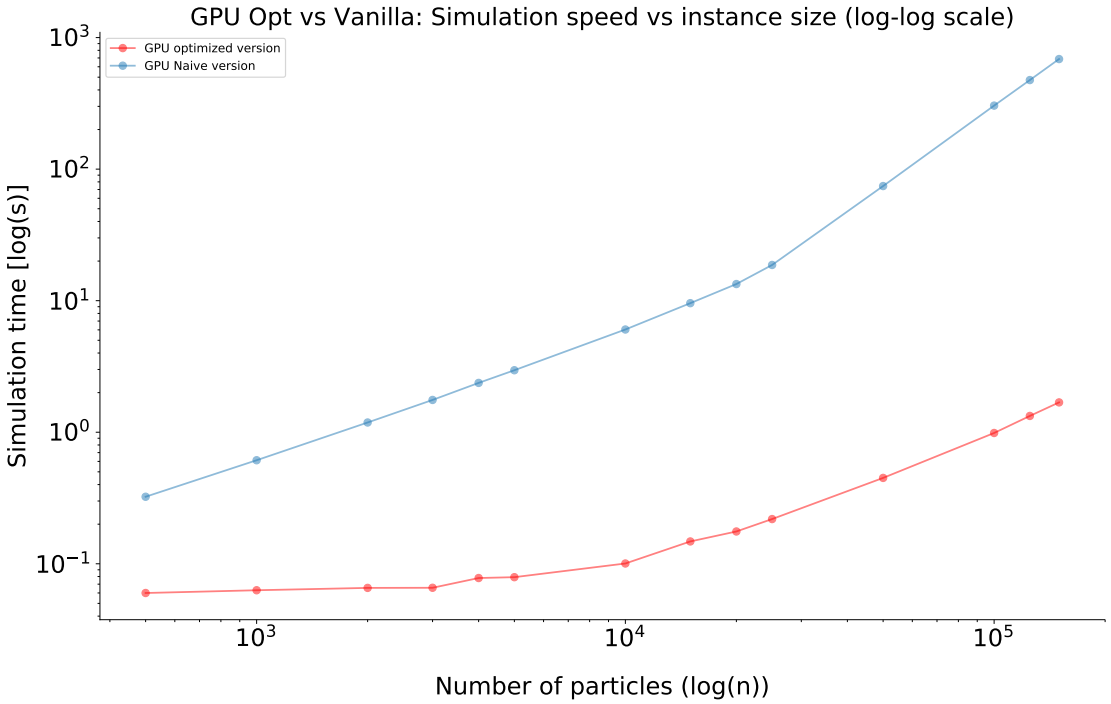


Figure 4: Log-log scale plot with the performance of the optimized GPU code versus the vanilla GPU implementation.

## 4.2 Optimized CUDA GPU version

Based on the ideas gathered from the single-threaded version of the previous reports, and following the techniques discussed in section 3, an optimized CUDA GPU multi-threaded version is developed.

Two main performance metrics are analyzed: (1) complexity $O(n)$, and (2) speedup factor comparison between our CUDA GPU version and the previous implementations: serial, OpenMP, and MPI.

### 4.2.1 Complexity

As with the single-threaded version of the code, we want our parallel code to reach an $O(n)$ asymptotic complexity. In order to check this performance metric, we fix the number of threads $p$ to a certain number and then we vary the number of particles to be simulated.

As we already discussed in the previous section, we can clearly see that our GPU implementation is able to reach the desired $O(n)$ complexity in Figures 5 and 6 as well as clearly improve the performance of the optimized serial code. Looking at the slope of the curve we can easily see that the slope tends to one. Checking the results obtained from the *autograder* utility provided with the assignment, we can check that the estimated asymptotic complexity (slope) is very close to 1 (Table 2) allowing us to conclude that the GPU implementation is satisfying the complexity $O(n)$ requirement.

Table 2: Summary results from serial and parallel implementation using 256 and 128 threads with $n \in [500, 1000000]$.

| **Size** $n$ | **Time 256C** [s] | **Time Serial** [s] | **Time 128C** [s] |
|---|---|---|---|
| 500 | 0.059 | 0.125 | 0.056 |
| 1000 | 0.062 | 0.235 | 0.059 |
| 2000 | 0.065 | 0.495 | 0.064 |
| 3000 | 0.065 | 0.877 | 0.065 |
| 4000 | 0.077 | 1.236 | 0.071 |
| 5000 | 0.079 | 1.631 | 0.072 |
| 10000 | 0.100 | 3.580 | 0.101 |
| 15000 | 0.147 | 5.548 | 0.140 |
| 20000 | 0.176 | 7.548 | 0.175 |
| 25000 | 0.218 | 9.549 | 0.211 |
| 50000 | 0.449 | 20.635 | 0.440 |
| 100000 | 0.986 | 46.332 | 0.985 |
| 125000 | 1.329 | 57.824 | 1.329 |
| 150000 | 1.685 | 70.614 | 1.685 |
| 200000 | 2.398 | 99.293 | 2.422 |
| 225000 | 2.754 | 117.499 | 2.781 |
| 250000 | 3.167 | 141.221 | 3.187 |
| 500000 | 6.938 | 485.961 | 6.980 |
| 1000000 | 14.659 | 1309.190 | 14.750 |
| **Estimated Slope** | 0.784688 | 1.179209 | 0.783242 |

Interesting is to notice that the serial optimized code is getting worse performance in the current cluster than the one obtained in Cori for previous reports, both in simulation times and the estimated slope value. A possible explanation for this pattern could be the fact that we are using a different compiler (*nvcc* instead of *gcc*) for the current implementation. In addition, both hardware and software are not exactly the same and thus, performance can vary from platform to platform.

Therefore, we are able to see the benefits of our modified serial implementation's data structure in the parallel GPU version.

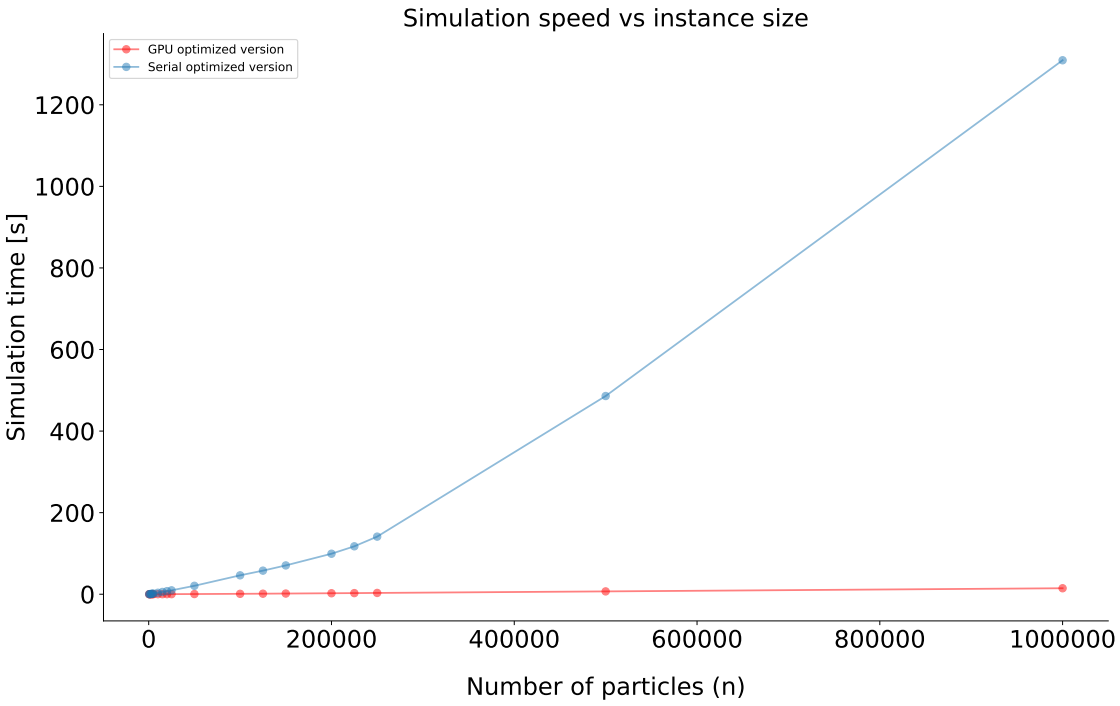


Figure 5: GPU vs Serial optimized versions comparison

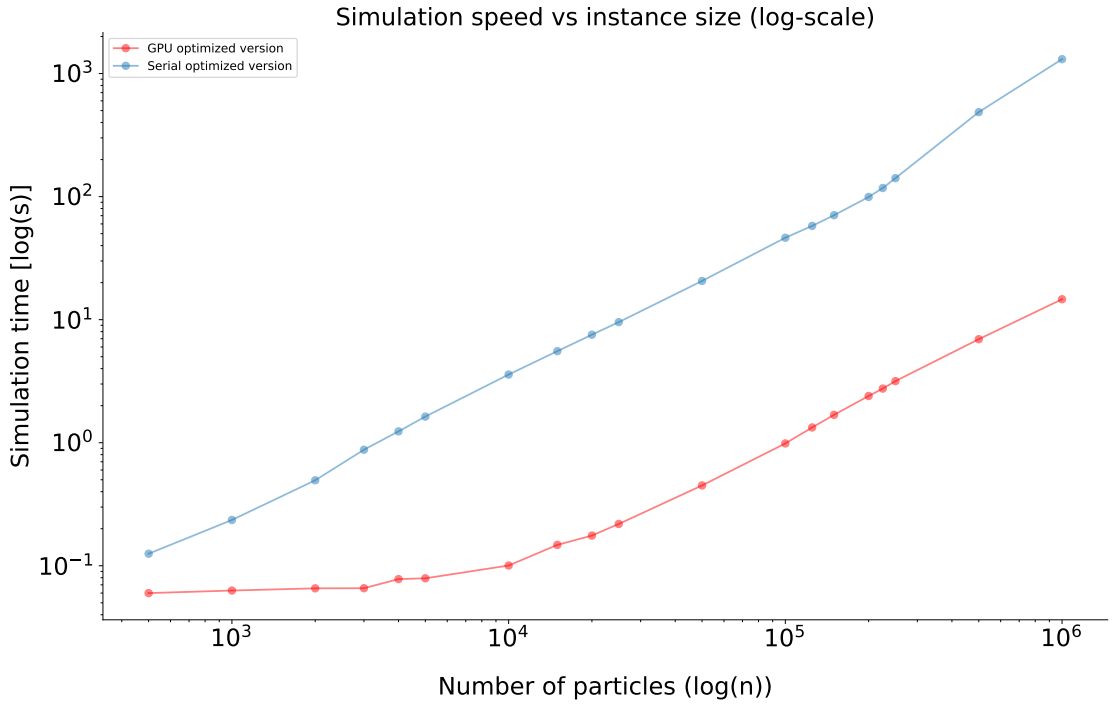


Figure 6: GPU vs Serial optimized versions comparison (log-log scale)

### 4.2.2 Speedup analysis

One of the classic challenges when implementing a parallel approach for a particular problem is to be able to reach good scalability of the speedup factors when solving different instances ($n$), used as key indicators of the performance of the overall parallel implementation. Therefore, the main objective of our implementation is to be able to obtain a significant performance boost with respect to the optimized serial version as well as our previous shared and distributed memory implementations.

For comparison purposes between our GPU parallel implementation and the all three previous solutions, we run a series of instances with $n \in [500, 1000000]$, calculating the speedup factor obtained by the CUDA GPU code. In order to perform a fair comparison between the parallel implementations, the maximum number of threads are used for each version: (1) 32 threads with OpenMP, (2) 32 threads in 32 nodes with MPI, and (3) 256 threads with our current CUDA code. Therefore, we perform a comparison between the best performance implementations, being able to obtain the real speedup factor values given our access to hardware resources.

As discussed in the previous section, running the Naive version of the CUDA GPU implementation (Figures 1 and 2), we obtain poor speedup results, even with the optimized serial implementation: an average speedup of 0.35 is obtained for instances with $n \in [500, 150000]$. The explanation is clear: the complexity of the Naive code $O(n^2)$ is clearly a bottleneck for exploiting the parallelism of the GPU version, obtaining worse results than our optimized serial implementation. Therefore, these implementations are not able to obtain a significant performance boost when adding more threads and thus, they are not really useful for large-scale problems due to their lack of ability to take advantage of the underlying hardware/resources available.

With our final implementation, we are able to obtain significantly better results. As can be seen for instances with $n \in [500, 1000000]$ in Figures 7 and 8, we can see how the performance of our optimized GPU code is significantly better than the optimized serial implementation, as well as the OpenMP implementation. In Figure 8 it is possible to observe how the gap between the Shared memory approach and both the MPI and the GPU versions is increasing with the size of the instance: the scalability of the implementations is explicitly observed. Differences with the optimized MPI code are not as significant as with the others, following our expectations from the previous part of this project since our MPI implementation was the most successful among the first three developed.

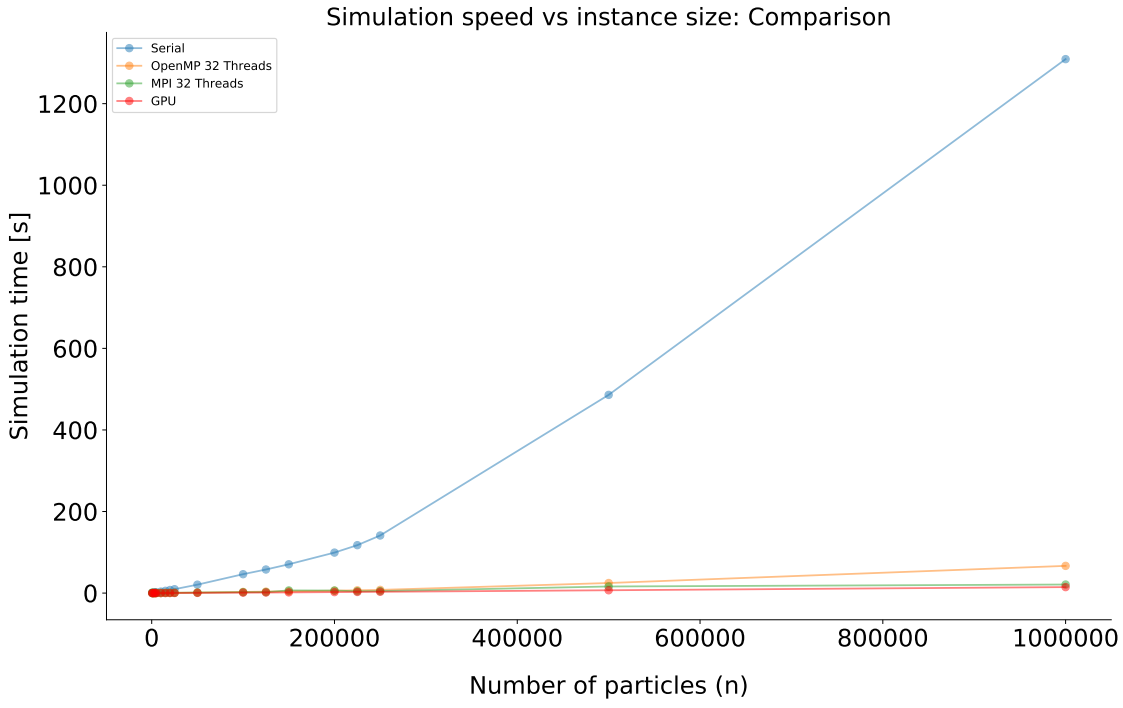


Figure 7: All final versions simulation times comparison.

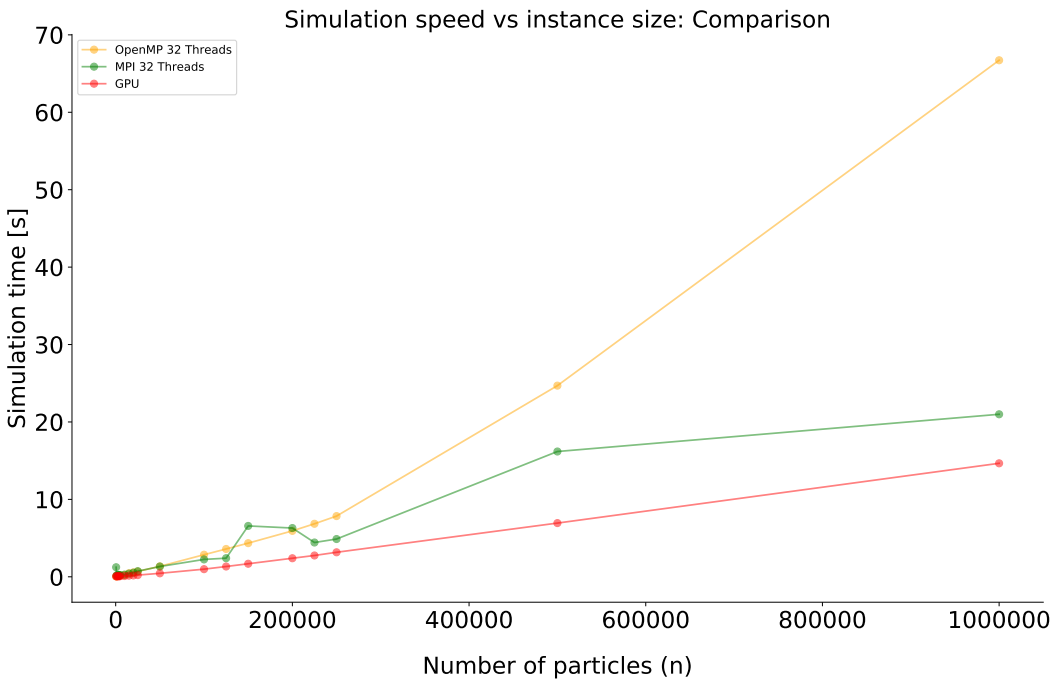


Figure 8: OpenMP, MPI, and GPU final versions simulation times comparison.

When comparing the speedup factors of the GPU version with respect to the serial, OpenMP, and MPI implementations, we can see in Figures 9 and 10 how we are able to obtain an average speedup factor around 36 with respect to the optimized serial implementation, as well as 3.53 and 2.54 with respect to the MPI and OpenMP versions, respectively. Interesting is to notice in Figure 10 that the average speedup factor obtained when comparing the GPU version with the MPI implementation is clearly biased by the - outlier - speedup factor obtained when $n = 500$ particles: a factor of approximately 20. This leads to a higher average speedup factor than the one obtained using the shared memory approach with OpenMP, however, from the graphs it is clear that the MPI version achieves a better overall performance than the OpenMP code both in solving times and speedup factors with respect to the GPU approach.

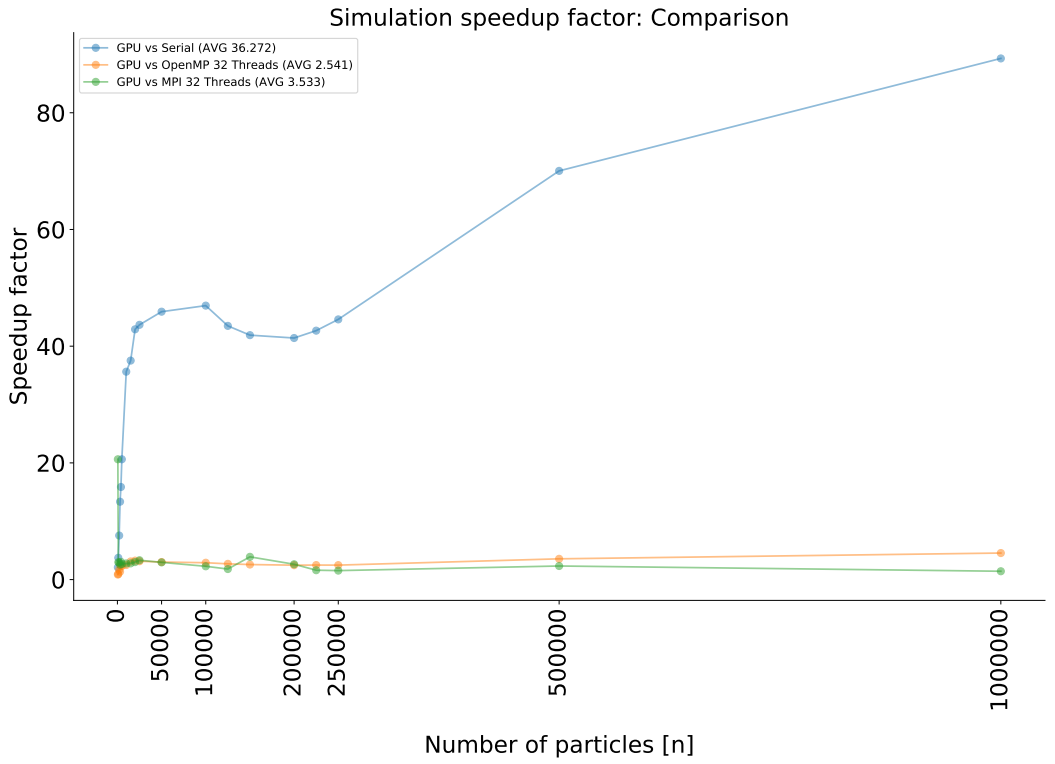


Figure 9: Serial, OpenMP, MPI, and GPU final versions speedup factors comparison.

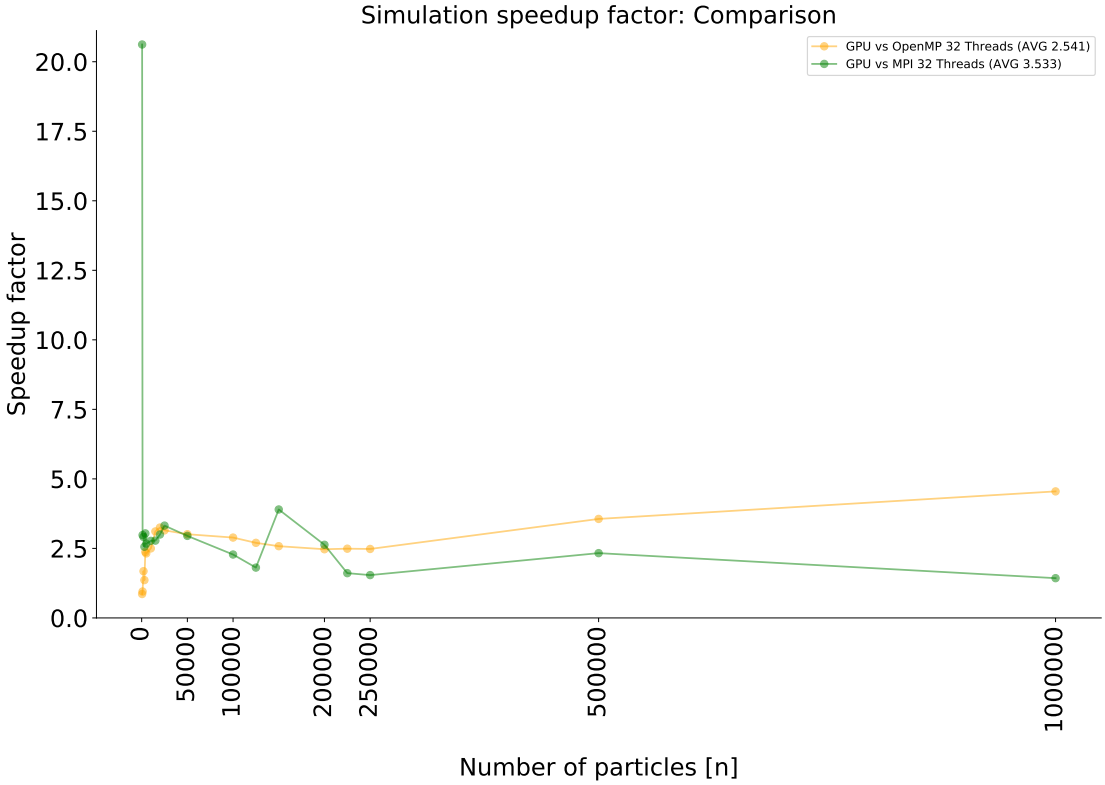


Figure 10: OpenMP, MPI, and GPU final versions speedup factors comparison.

Focusing on the bigger instances $n \in [50000, 1000000]$, we can see in Figures 11 and 12 how the average speedup factor with respect to the serial version is significantly increased reaching a value around 52. On the other hand, as expected from our previous discussion about the MPI and OpenMP results, the average speedup factor for the shared memory approach is increased from 2.54 up to 2.97, following the increasing gap trend observed with larger instances in Figure 10, while the average factor obtained by the distributed memory approach is decreased from 3.53 to 2.27, being consistent with our observations.

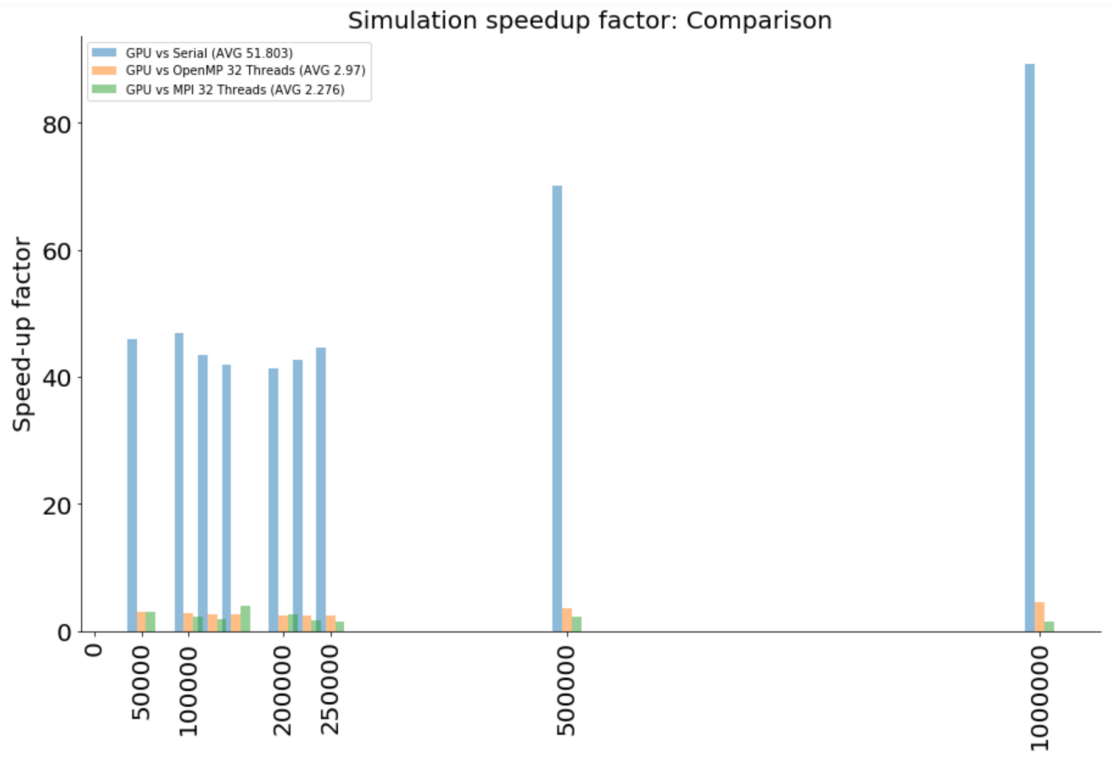


Figure 11: Speedup factors for big instances $n \in [50000, 1000000]$ all versions comparison.

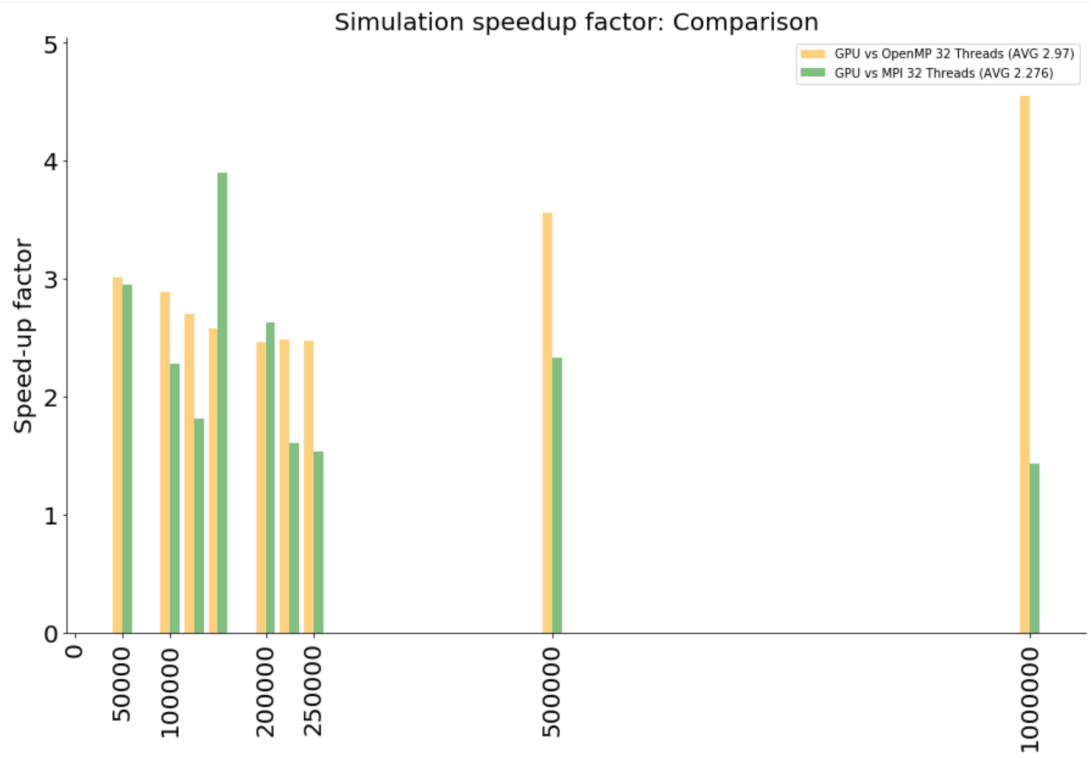


Figure 12: Speedup factors for big instances $n \in [50000, 1000000]$ OpenMP and MPI versions comparison.

Therefore, we are able to conclude that the current CUDA GPU implementation is the best one among all four versions tested during the project, obtaining the best solving times - and thus, speedup factors. However, as we will discuss in the next section, it has some drawbacks and limitations with respect to the hardware architecture needed as well as a potentially negative impact in the programming complexity of the code depending on the algorithm/program that is being parallelized.

Then in second place, we have the optimized distributed memory approach using MPI: its performance using 32 nodes (and threads) is comparable to the one obtained by the optimized GPU version. However, its main limitation is the fact that it needs a significantly higher amount of resources (32 nodes) and thus, it is not particularly efficient when managing the resources available inside a high-performance computer, as well as expecting significant longer waiting times in the shared queue.

Based on its solving times and speedup factors, we classify the OpenMP version in the third place. However, its simplicity in both implementation and coding efforts, as well as the efficient use of resources (using only one node with 32 threads inside), rise it as a very attractive and efficient choice. When comparing the time invested by the team for programming this version versus the MPI approach - about 3-4 times longer for getting the final MPI version -, the OpenMP version seems like a better option for limited time projects and testing ideas or concepts before developing a distributed memory approach for gaining more performance.

Finally, as expected, the optimized serial version of the code is the one with the worst performance, since it is not taking into account any parallelization, not taking advantage of the resources available in the cluster.

### 4.2.3 Comparison Summary

In order to summarize the major insights obtained from the entire project, we briefly describe the most significant differences between all our implementations as well as the major advantages/disadvantages of using CUDA and the current GPU architecture:

i) Serial implementations are easy to write and have the lowest code complexity, however, their performance is limited by the processor capacity as well as the structure of the algorithm: no simple performance boosts can be easily obtained.

ii) Shared-Memory implementations (such as OpenMP) can be quick to write and fast due to the simplicity of adding certain *pragmas* to the original code in C allowing embarrassingly parallel sections to be easily solved, but run into scaling issues across multiple nodes. They also have race conditions and false sharing/cache consistency issues generating potential - and significant - performance loss.

iii) Message Passing implementations (such as MPI) scale well across many nodes, but require very careful maintenance of messages (pack/unpack, async), as well as the high overhead of communicating over the network. Therefore, their implementation can be challenging and significantly time demanding for achieving good performances.

iv) GPU implementations (such as CUDA) have exceptionally high bandwidth, but this bandwidth may not be feasible for code with lots of branches. In addition, a new set of functions and code syntax should be studied and tested in order to obtain an efficient and well-optimized simulation code.

### 4.2.4 CUDA GPU: Advantages & disadvantages

In order to provide some insights and measure the usefulness of the CUDA GPU programming approach, we include a brief analysis of some of its advantages and disadvantages, based on the group's experience as well as from [1] and [3] concepts/analysis:

- One of the strengths of the CUDA is the availability GPU-accelerated libraries. NVIDIA GPU-accelerated libraries provide highly-optimized functions that perform 2x-10x faster than CPU-only alternatives. Using drop-in interfaces, you can replace CPU-only libraries such as MKL, IPP, and FFTW with GPU-accelerated versions with almost no code changes. The libraries can optimally scale your application across multiple GPUs.
- One of the benefits of CUDA over the earlier methods is that a general-purpose language is available, instead of having to use pixel and vertex shaders to emulate general-purpose computers. That language is based on C with a few additional keywords and concepts, which makes it fairly easy for non-GPU programmers to pick up.
- CUDA cannot benefit every program/algorithm: the CPU is good in performing complex/different operations in relatively small numbers (i.e. $< 10$ threads/processes) while the full power of the GPU is unleashed when it can do simple/the same operations on massive numbers of threads/data points (i.e. $> 10.000$). The benefits of GPU programming vs. CPU programming is that for some highly parallelizable problems, you can gain massive speedups (about two orders of magnitude faster). However, many problems are difficult or impossible to formulate in a manner that makes them suitable for parallelization.
- They are very fast at applying the same instruction to multiple data points (SIMD). However, if you add branching (an if), the two branches will be serialized (first the if on all data that takes it, then the else on all data that doesn't - so still quite parallel, but not entirely). Also, if you have very few data points, the overhead of uploading to the GPU and downloading the result will likely dominate the overall execution time - it might be cheaper to just execute on the CPU instead.
- CUDA is dependent on NVIDIA's GPU card, therefore, a specific hardware architecture is needed, making it not as portable as CPU-oriented code. In addition, the use of recursive functions is limited to certain cards such as NVIDIA's Fermi, adding an extra limitation to the programmers' code.

# 5 Conclusions & Future Work

1. $O(n)$ complexity codes for the single and multi-threaded versions of the two-dimensional particle simulator were successfully implemented based on a different and more efficient underlying data structure than the vanilla version.
2. The main strategy applied for improving the Vanilla implementation consists of noticing that the original problem can be modeled in a different way without impacting its expected behavior. Eliminating and replacing extra computations like checking all the interactions between each pair of particles inside the simulation region, by the notion of particle's neighborhood - for interacting purposes - improves the performance while not losing the underlying logic of the simulator.
3. A successful GPU multi-threaded CUDA's code has been implemented obtaining average speedup factors around 36 when using instances with $n \in [500.1000000]$. This implementation is over-performing all our previous attempts/approaches, showing us the great potential of developing a pure GPU code using libraries as CUDA for optimizing performance.
4. In order to obtain the best performance of the GPU approach, we needed to re-design the underlying grid structure, adapting it to the CUDA's code style as well as the multi-threaded structure of the code. Re-using our original implementation was not successful, leading to very poor results in terms of simulation time and complexity: the initial lack of experience of the members of the group in CUDA was one of the bottlenecks for this implementation.
5. Strong and weak scaling factors can be analyzed in the future by limiting the number of threads and/or nodes that are going to be used during the simulation, replicating the results obtained by both the OpenMP and MPI versions of the simulator.
6. Different and potentially better data structures can be tested and implemented in order to reach better performance in both the serial and parallel versions: linked lists and/or static arrays would be useful options for developing an alternative approach.
7. A lazily updating approach for the grid structure can be implemented as a future work step in order to obtain a significant performance boost without impacting the complexity of the code's programming style. Similar to our previous implementations, it is possible to only check and update those entries that have been modified instead of performing a full $clear()$ task.

# 6 References


1. Bell, N., & Hoberock, J. (2011). Thrust: A productivity-oriented library for CUDA. In GPU computing gems Jade edition (pp. 359-371).
2. C++ benchmark – std::vector VS std::list VS std::deque, https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html
3. GPU Accelerated Computing with C and C++, Nvidia, https://developer.nvidia.com/how-to-cuda-c-cpp
4. Lecture notes and slides from course CS267, Computer Science Department, University of California Berkeley, 2018.
5. Numba: NVIDIA GPU programming using Python, https://devblogs.nvidia.com/numba-python-cuda-acceleration/
6. Nvidia, C. U. D. A. "Programming guide." (2018).
7. Sanders, J., & Kandrot, E. (2010). CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional.