Simulation of Storms of High Energy Particles in CUDA

Bruno Carmo, *57418*, Sahil Kumar, *57449*

Abstract—In this report, we investigate the behavior of a parallel program when optimizing a simulation problem. This problem revolves around the simulation of high-energy particle bombardment on an exposed surface such as the surface of a space vessel in outer space. We approached the problem by using a methodology consisting of identifying potential bottlenecks and hotspots and resolving them using different parallel program patterns and NVIDIA's CUDA API. After obtaining the optimized parallel program we benchmarked it against the given sequential program and analyzed the results of our experiments. We conclude that in a real-world environment, the optimized parallel version would bring clear benefits in the usage of the simulation program.

Index Terms—High-performance computing, Nvidia CUDA, parallel programs.

1 Introduction

Parallel computing is often referred to as an execution model that leverages parallel computers to execute multiple operations or execution flows simultaneously. High-performance computing consists of using this kind of process used on highly parallel (and computing powerful) hardware to process massive multi-dimensional datasets and solve complex problems at high speeds.

In this report, we present our solution for the project assignment of the course of High-Performance Computing. This project consists of writing a CUDA or OpenGL program that implements a simulation of high-energy particle bombardment on an exposed surface, such as the surface of a space vessel in outer space. This program is based on a provided sequential code that simulates the effects of high-energy particle impacts on a cross-section of the surface, represented by an array of control points. The program computes the energy accumulated on each point after the impact of several waves of particles and, after each wave, it identifies what control point received the highest energy, that presents the higher risk of being damaged.

Our objective in this project is to is to parallelize the simulation using CUDA, in order to optimize its performance. We also analyze the results and compare them with the sequential version to make sure of its accuracy.

2 Simulation

The particle waves used for the simulation are provided in the form of text files and stored in an array that stores structures with the field for the details of each particle, the index of the impact point, and its energy value in thousandths of a Joule.

The program executes the simulation, after reading the particle-wave files, in three major phases for each wave:

1) **The bombardment phase**: for each particle in the wave, the program converts energy to Joules, determining energy accumulation at each array position. Upon impact, energy is transmitted to the contact point and neighboring points. The accumulated energy considers distance-based attenuation, with lower accumulation for farther points. A minimum energy threshold, that is material-dependent, is required upon impact; points below this threshold, due to their distance, do not accumulate any energy.
2) **The relaxation process**: During the relaxation process, the material responds by distributing its charge. Control points are updated by averaging values from the previous point, the point itself, and the next one. To preserve the original values necessary for updating neighbors, the array values are initially copied to a secondary ancillary array. Old values are then read from the ancillary array, and new values are written in the original array.
3) **Maximum energy point location**: In the last stage the program locates the point with the maximum energy value and saves its position and value.

After all the waves have been processed, the maxima and positions for all waves are printed.

3 Methodology

In order to understand and analyze the performance of our program we used the *Gprof* profiling Tool. These kinds of tools are usually used to identify most of the code's major performance bottlenecks, hotspots, and where most of the resources are being used to run the program to understand where and how time is being spent.

Using this we discovered that the *update()* function, which is called to update the energy value of a single point in the impact layer, is where the program spends most of its time. Further analysis of the program's code helped us understand which parts of the code were good candidates for parallel acceleration using CUDA.

3.1 Code Optimization

The code that we were able to identify and parallelize using CUDA were:

3.1.1 Initialize layer to zero

Using the Map parallel control pattern we were able to successfully parallelize the *for* used to initialize the layer vector to zero.

3.1.2 Update the energy value for each cell in the layer

Again, using the map pattern, we eliminated the *for* used to traverse the impact layer cells, and compute its energy update according to the impact storms.

3.1.3 Copy values to the ancillary array

Just like in the earlier sections, we used the map pattern to copy the values of the layer vector to an auxiliary vector used in the next subsection.

3.1.4 Update layer using the ancillary values

In the relaxation process, each control point is updated with the average value of three points: the previous one, the point itself, and the next one. To avoid destroying the original values in the array, it is used the copy of the array obtained in the previous subsection.

As each thread accesses 3 positions of the copied layer array we opted to use the stencil parallel control pattern, using a shared temporary array available across all the threads in a block and a neighborhood of 1.

3.1.5 Locate the maximum value in the layer, and its position

Lastly, we identified that we could optimize the *for* loop used to find the maximum energy stored in the control points of the impact layer and its index position after each wave. We believe that this code could be optimized using the reduction parallel control pattern but unfortunately, we could not properly implement it.

4 Experimental Analysis

After parallelizing the simulation using CUDA, we needed to check if our solution did indeed optimize its performance and maintain the correctness of the program. For that, we needed to compare the performance of our solution with the most optimal sequential version of the simulation or with the optimized simulation in CUDA using only 1 GPU thread. For simplicity and because we already had the sequential version, we benchmarked our solution against it in two different computational environments.

4.1 Computational Environments

We designed our solution and checked its correctness using the hardware in our local computational environment, our development machines (Nvidia RTX 3070). Afterward, as suggested, we ran our solution in the *bulbasaur* node of the DI-Cluster (Nvidia Quadro M2000). To compare the two computational environments, we present their GPU specifications in table 1 after consulting [1], [2] and [3].

We only compared the GPU specifications of the computational used because their CPU and RAM do not have a significant impact on the performance of our solution.

Observing the table, we can notice that our local environment has better resources than the node of the DI-Cluster. As such, it is expected that the performance of our solution should be better in our local environment, as confirmed by section 3.2.

GPU	Architecture	CUDA Cores	Memory
Nvidia RTX 3070	Ampere	5888	8 GB GDDR6
Nvdia Quadro M2000	Maxwell	768	4 GB GDDR5

GPU	Bus	Bandwidth	Clock(Base)
Nvidia RTX 3070	256-bit	448.0 GB/s	1.73 GHZ
NVIDIA Quadro M2000	128-bit	105.79 GB/s	0.872 GHz

TABLE 1
Hardware specification of computational environments used

In both computational environments and the following benchmarks we used 128 threads per block because we did not notice any significant difference when using other numbers of threads and for simplicity.

4.2 Benchmarks

Considering that the format of the wave test file was given we could make our test files. Unfortunately, because of a lack of time, we benchmarked our solution against the sequential program using only the test files given.

We executed our solution and verified its performance with different combinations of wave files, however for the sake of simplicity and understandability, we used the given test cases that are described as follows:

- **Test Case 1**: a general debugging test used to check the correctness of the solution. The test has 4 waves and each wave has different numbers of particles ranging from 5 particles to 8 particles.
- **Test Case 2**: a small workload test with 6 waves where each wave has 20 000 particles.
- **Test Case 3 to 6**: race condition tests. The tests check for proper communication in the borders of partitions and are executed one file at a time. Each test has only 1 wave with 4 particles.
- **Test Case 7**: an optimization test to check the performance of the solution. The test has 4 waves and each wave has 5 000 particles.
- **Test Case 8**: a reduction efficiency test to check the effectiveness of the solution. The test has 3 waves and each wave has only 1 particle, but it should be executed with 100 million control points.
- **Test Case 9**: a behavioral test used to check that the behavior of the sequential program in the array extremes has not changed even after the optimizations done in our solution. The test has only 1 wave with 3 particles and was executed with 16 and 17 control points.

Our solution passes all the tests except test case 8 and test case 9 (with 16 control points). For test case 8, our solution returns always 0 energy for each wave for the highest energy received. We assume that this happens because of

Testcase	CUDA Local	CUDA Cluster	Result
1	0.000245	0.000665	equal
2	0.21992	1.098957	equal
3	0.000145	0.000455	equal
4	0.000166	0.000462	equal
5	0.000156	0.000426	equal
6	0.000157	0.000436	equal
7	0.207344	2.883709	equal
8	-	2.278262	not equal
9	0.000145	0.000419	equal

TABLE 2
Comparison of runtimes of CUDA Local and CUDA cluster

Testcase	Sequential	CUDA	Speedup	Result
1	0.000015	0.000595	0.03	equal
2	27.004666	0.867189	31.14	equal
3	0.000004	0.000412	0.01	equal
4	0.000004	0.000443	0.01	equal
5	0.000004	0.000422	0.01	equal
6	0.000004	0.000453	0.01	equal
7	150.066735	2.856312	52.54	equal
8	3.554675	2.568420	1.38	equal
9	0.000004	0.000424	0.01	equal

TABLE 3
Comparison of runtimes of sequential version and CUDA version in the cluster

a memory access out of bounds given that the number of control points is also too big. However, this test only fails in our local environment and, unfortunately, we could not find the cause of such behavior. For test case 9 (16 control points), our solutions just return values that do not equal those of the sequential version meaning that our solution does not preserve the behavior of the sequential program in the array extremes.

Considering that the tests do not take too much time to execute, we decided to verify what would be the maximum number of control points we could feed to the program such that it passes all test cases mentioned above. Our solution supports up to 2 621 440 control points before breaking.

In the following subsections, we present the results of three different experiments: CUDA Local vs CUDA Cluster, the Speedup of the CUDA version against the sequential version, and the Speedup as a function of the number of control points.

4.2.1 CUDA Local vs CUDA Cluster

Considering the differences in the computational environments used, one experiment we thought was interesting was the comparison of the runtime of the CUDA version of the program in the local environment as opposed to the runtime of the CUDA version in the DI-Cluster for each of the test cases, as shown in table 2.

As expected, our local environment has a faster runtime compared to the DI-Cluster and maintains the correctness of the program in all test cases except test case 8. As mentioned earlier, unfortunately, we could not find the cause for the test case to fail in our local environment even though it passed in the DI-Cluster.

Observing the table, we notice that despite our environment being faster than the DI-Cluster the difference in runtimes is not too much significant for most cases. Only in test cases where the workload is heavier, such as test case 2 and test case 7, the difference gets significant with our local environment being sometimes 10x faster than the DI-Cluster.

4.2.2 Speedup of CUDA version against sequential version

Taking a real-world program optimization approach, we compared the runtime of the sequential version with the runtime of the CUDA version, to see if the CUDA version was worth it. We executed both versions in the DI-Cluster so that the results share the same context (the hardware) as our colleagues, as demonstrated in table 3.

Looking at the table, as expected, we can perceive that in the test cases where the objective is to feed a big workload to the program to check its optimization performance, like test cases 2 and 7, the CUDA version has a faster runtime than the sequential version which in consequence gains a bigger speedup. Considering this observation, the correctness of our solution, and the fact that runtimes do not have a great difference in other test cases that test other aspects (including race condition, etc.), we can say that our solution is indeed worth it for real-world program optimization.

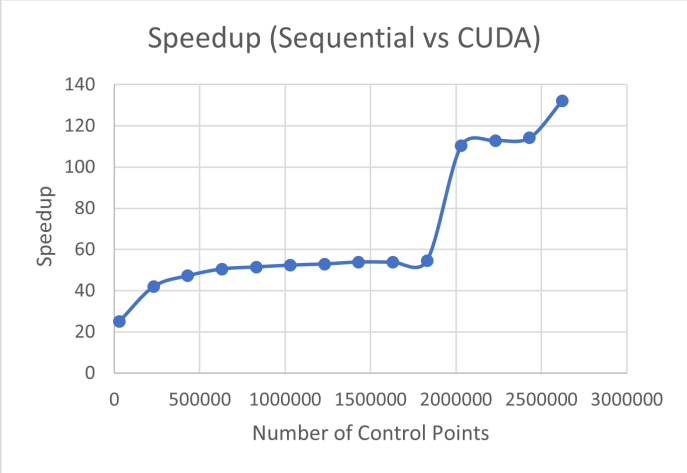

Fig. 1. Speedup gain of the CUDA version against the sequential version as a function of the number of control points in test case 7.

4.2.3 Speedup as a function of the number of control points

Another interesting experiment to do was to verify how the speedup behaves when we increase the number of control points. We chose to run the experiment using test case 7 because it is an optimization test that checks the performance of the optimized solution. Therefore, we executed our solution in the DI-Cluster and increased the number of control points starting from 30000 points until 2 621 440 points (our limit) with a step size of 200000 points. The results are shown in figure 1.

Observing the figure, as anticipated, the speedup increases as we increase the number of control points. Nevertheless, it does not change too much until we hit 2 030 000 points, because the runtime of the sequential version and the CUDA change equally or they do not change too much with the number of control points. However, one interesting behavior, is that the speedup has a great increase when we pass from 1 830 000 points to 2 030 000 points. This is due to the sequential not being able to maintain its previous runtime for more than 2 000 000 control points.

5 Conclusion

This report presented the methodology we used to complete the assignment given and the experimental analysis we made to verify our solution's performance, correctness, and behavior.

We managed to optimize most of the bottlenecks identified except the one that found the control point with the highest energy. Our solution also passed all our test cases in the DI-Cluster and only failed test case 8 in our local environment.

We concluded based on our experimental analysis that if this problem assignment was taken to a real-world application then the parallel optimization of the sequential program using GPUs (in this case Nvidia and CUDA) would surely bring benefit to the users of the program taking into account that it is faster and still maintains correctness.

References

[1] DI-Cluster, http://cluster.di.fct.unl.pt/docs/technical/
[2] NVIDIA Quadro M2000, TechPowerUp_2023, https://www.techpowerup.com/gpu-specs/quadro-m2000.c2837
[3] NVIDIA Quadro M2000, TechPowerUp_2023, https://www.techpowerup.com/gpu-specs/geforce-rtx-3070.c3674

Acknowledgments

We would like to thank Hugo Pereira (58010) for helping us with how to launch the solution in the DI-Cluster, some common bugs, and clarification about the sections of the project report.

We would also like to thank Diogo Almeida (58369) for helping us with some working methodology and Bash script knowledge.

Individual Contributions

We divided the work taking into account the number of code blocks we had to optimize. Considering that there were 5 code blocks to optimize, Bruno Carmo optimized 3 code blocks and Sahil Kumar optimized 2 code blocks. Sahil Kumar additionally created the script that checks the correctness of the solution and the speedup gain to balance the workload distributed. Therefore each member contributed with 50%.