Optimizing Molecular Dynamics Simulation for Argon Gas

Eduardo Figueiredo
MEI: Parallel Computing
Universidade do Minho
Barcelos, Portugal
Email: pg52679@uminho.pt

Gonçalo Senra
MEI: Parallel Computing
Universidade do Minho
Barcelos, Portugal
Email: pg52683@uminho.pt

***Abstract*—In this article, we're exploring ways to make a computer program that simulates how argon gas behaves work better and faster. We use special tools to check the program's code and see how well it works. We'll tell you what we discovered and how we can make it even better. We also talk about modifying the program so that it can do multiple things at the same time. This allows programs to work more efficiently, run faster, and avoid problems with the information they use. See how much faster your program is when you make these changes to your code, and compare it to the expected speed in different situations. We'll also take a closer look at programming methods that can help you better simulate molecular motion using CUDA.**

I. Introduction

This study focuses on analyzing and improving a computer program that simulates how argon gas behaves. The program is located in the "Simple Molecular Dynamics" section of the Foley Lab/Molecular Dynamics. At first, we worked on making the regular program work faster by organizing similar functions and optimizing how it does certain tasks, aiming to reduce the time it takes to run and improve its overall performance.

In the second phase of the project, we moved from making the program work on one task at a time to making it work on multiple tasks simultaneously. We used a tool called OpenMP for this, but we faced limits on how many things the computer could handle at once, determined by the number of available computing cores. To overcome this limitation, in the next phase, we turned to parallel programming with CUDA, utilizing the processing power of GPUs. This allowed us to conduct comprehensive tests using various metrics and on different computer systems, leading to robust conclusions. These methodological iterations aim to significantly enhance the efficiency and performance of the simulation program in question.

***Index Terms*—Optimizing, Instructions, Execution time, Performance, Code, Accelerations, Flags, Threads, Data race, Parallel, Parallelization, Multi-threaded, Speedup, Real gain, Optimal gain, Reduction, OpenMP, SeARCH, Hotspot, Loops, Potential, Power operations, Flags, Vectorize**

II. Sequential Code Optimization

A. Optimizing the Molecular Dynamics Simulation Code

This section discusses various strategies for optimizing argon gas's molecular dynamics simulation code.

*1) **Abbreviations and Acronyms**:*

a) SSE4: Streaming SIMD Extensions 4

b) AVX: Advanced Vector Extension

*2) **Optimizing the Potential Calculation**:* The original code of the Potential() function calculates the potential energy in a system of interacting particles via the Lennard-Jones potential. The optimizations we made, which can be seen in Figures 9 and 10, are as follows: We eliminated the loop named "k", which traverses three columns of table "r". This way we have reduced the number of instructions. In loop 'j' we notice that there are repeated calculations. In other words, the square of the difference between "r[i]" and "r[j]" is the same as the square of the difference between "r[j]" and "r[i]. We simplified the code by considering all pairs of 'i' and 'j' and then multiplying the result by 2. To reduce cache misses, we stored the values we accessed in the 'i' loop in variables so that we could use them in the second loop. To minimize the number of instructions, we eliminated a calculation that was performed N*N times and was equal to 4 times epsilon. Instead, we moved this calculation outside of the loops. These optimizations were made to improve the efficiency and performance of the code.

*3) **Optimizing the Compute Accelerations**:* After the pow() functions removal, the second target of optimization was the deletion of the loops named 'k' (as we can see in Figures 11 and 12), which were causing a significant increase of instructions. Beyond the optimization mentioned before, in every single iteration of the loop named 'i' we stored the matrix line 'i' in temporary variables ('ri0', 'ri1', 'ri2'), so that the program could avoid cache misses and consequently have to fetch the values from memory.

*4) **Combining Potential Calculation and Accelerations**:* In the original code, the calculation of the potential energy and the calculation of the particle acceleration are performed in separate steps. To optimize the code, we created a function that combines functions optimized for potential energy and acceleration(Figure 13). Taking advantage of the fact that

these functions are identical regarding loops and memory array accesses, we combined them into a single function, thus reducing the number of instructions and memory accesses.

*5) **Optimizing the Power Operations**:* The original program involved several power operations that consumed a significant part of the execution time (Figure 14), due to the computational cost of lookup operations. Because of that, we decided to replace all the pow() calls, (except the numbers that were powered to 1/n), with basic multiplications. This replacement reduced the number of instructions, and the execution time decreased significantly.

*6) **Eliminating Unnecessary Loops**:* While optimizing the code, we discovered unnecessary loops that were negatively affecting performance. These loops, named "k", are abundant in native code. Since 'k' loops only have a maximum range of 3, we chose to remove them and replace them with repeated instructions. This adjustment helped reduce the total number of instructions and improve efficiency. Finally, in the 'initializeVelocities()' function, we merged some loops, further reducing the number of instructions.

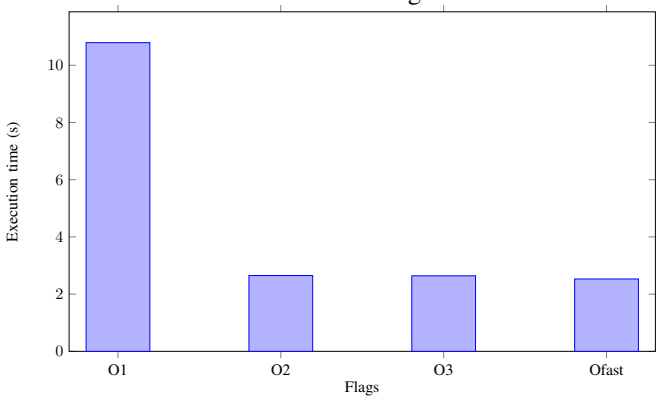
B. Analysis of Optimizations

*1) **Flags**:*

*a) **-pg, -g**:* These flags enable debugging information in the executable file, the -pg flag adds the execution time of each function

*b) **-fno-omit-frame-pointer**:* When enabled, the compiler keeps the frame pointer in the call stack, making debugging and tracking function calls easier. This is useful for accuracy and visibility in the call stack. Therefore, when using this option, the compiler does not optimize away the frame pointer, making it visible and accessible for debugging and analysis purposes.

*c) **-O2**:* This flag represents a moderate level of optimization, we chose this level of optimization due to the great balance between execution time and compilation time. Additionally, we tried the -O3 and -Ofast flags, but we didn't get a much better execution time and the compilation time increases, if we consider that these flags use much more aggressive optimizations than the -O2 flag, (which normally leads to bigger binary files), with all these facts we decided that the best flag to use is the -O2.



*d) **-ftree-vectorize, -msse4, -mavx**:* The flag -ftree-vectorize enables tree vectorization, this means that the compiler will attempt to generate instructions to process multiple data elements in parallel, the -msse4 flag enables support for SSE4 instructions, which are used to accelerate parallel data operations. On the other hand, the -mavx flag enables support for AVX instructions, providing additional resources for the parallel processing of data vectors, a feature commonly used throughout the code. When used together, these flags deliver optimized performance on the processor, improving overall code efficiency.

III. Optimizing the Molecular Dynamics Simulation Code with OpenMP

A. Hotspots identification

This section discusses the identification of the code blocks with the highest execution time, although we tested them in various locations. First, we identified the following hotspot, however after parallelizing in additional locations and not observing a noticeable speedup gain, we decided not to parallelize them.

*1) **ComputeAccelerationsPotential Function**:* In the last work assignment, we were asked to optimize this Molecular Dynamics program without parallelization, so one of the strategies adopted was to merge the two most complex functions (ComputeAccelerations() and Potential()) into one (computeAccelerationsPotential), which saved us a bunch of unnecessary cycles. That solution left us with a major code block that was consuming 99.43% as we can see in the call graph 16 of the execution time and, as a consequence of this fact, the target of parallelization had to be this function.

B. Parallelization

In this section, we explain the formal details of the techniques and 'tags' used to transform a single-threaded program into a multi-threaded program with OpenMP.

*1) **#pragma omp parallel**:* In the first place, we selected a section for parallelization, this tag (with braces), is responsible for indicating the place where the threads will be created/launched and terminated during the execution.

*2) **#pragma omp for schedule(dynamic) reduction(+:Pot)**:*

*a) **for**:* This directive name indicates an assignment of loop iterations to threads. In this directive, we had to make a choice between putting it in the first 'for' loop, or in the second loop. We decided to place it in the first one because the second loop has more dependencies than the first, so positioning it in the first loop will allow the threads to run more independently which will make the parallelization more efficient. Another reason that led us to place it in the first 'for' loop was the scalability would be more uniform, as the number of instructions per cycle in the first loop collectively is more identical among threads.

*b) **schedule(dynamic)**:* The keyword schedule indicates how the loop iterations will be distributed to the threads, in this case, we chose dynamic distribution so that the threads request a new chunk to handle every time they finish the current chunk. We could also specify the chunk size, but we didn't get better execution times due to those changes, so we kept the default size. Additionally, we conducted tests with other scheduling types, such as static, auto, and guided. However, in each experiment with these configurations, we noticed that the execution time increased, or data race conditions arose among the shared data between threads. This analysis led us to use the dynamic distribution option, which provides a satisfactory balance between performance and avoiding data race problems.

*c) **reduction(+:Pot)**:* This is a crucial step in code parallelization and data race avoidance, the reduction tag will create copies of the indicated variable for each thread. At the end of the parallelized section, the operation indicated before ':', (in our case "sum"), will be applied to all copies and store the result in the original variable. This technique is used to avoid multiple threads' access to the same variable and consequently prevent the corruption of the indicated variable. Furthermore, the execution time will be improved because we don´t have to use any synchronization technique and hold back threads that could be running their tasks independently.

*3) **Matrix 'a' reduction and #pragma omp critical**:*

*a) **Matrix 'a' reduction**:* In the context of reducing the 'a' matrix, each thread in the parallel section maintains its private copy of the 'private_a' array. This private array is used to independently compute contributions to the global matrix without causing contention or data races between threads. After a parallel loop, these private copies are combined and the global matrix is updated with the aggregated results. We implemented this private array because we were using an earlier version of OpenMP that did not support the direct reduction of arrays. Newer versions (4.0) allow direct reduction of arrays. However, to overcome this limitation, we manually created and maintained a private copy during parallel processing to ensure that the global matrix was updated correctly.

*b) **#pragma omp critical**:* This is used to create a critical section, allowing only one thread at a time to execute the block of code. In our case, it helps protect the process of updating the global matrix a. Without this critical section, multiple threads could modify the shared matrix simultaneously, potentially leading to data races. Critical sections ensure that updates to the global matrix occur synchronously, preventing conflicts and maintaining the integrity of shared data.

C. Analysis of Parallelization

Further, we will analyze the practical and theoretical advantages of program parallelization compared with sequential execution.

*1) **Execution time evolution**:* The execution time of the program diminishes every time the number of threads is increased, especially when the number of threads is low after the number of threads reaches 20-25 the execution time stops improving, this behavior was expected and could be explained by the machine's limitations, and the high demand on the SeARCH's capacities. On top of that, the code isn't fully optimized which can explain the fact that the speedup curve didn't reach the optimal gain curve (more detailed explanation about speedup in the next subsection).

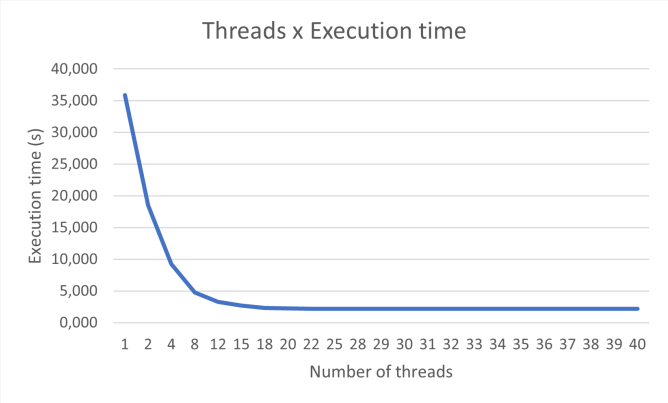


Fig. 1. Threads X Execution Time

*2) **Speedup evolution**:* By dividing the parallel execution time by the sequential execution time, the speedup is computed. Amdahl's law establishes the theoretical maximum gain in the case of 20 virtual cores and 20 physical cores, implying that 20 should be the ideal acceleration. However, because of racial disparities and Amdahl's law, which restricts the performance of contiguous program components, the actual performance benefits can be less in real-world situations. Suboptimal gains, stagnation, and slowdowns can be caused by practical issues including resource contention, search system load, and scheduling challenges, particularly after 25 threads. These elements draw attention to how complicated parallel computing is. The benefits of parallelization can be limited by Amdahl's rule, which can lead to severe sequential corruption. Therefore, it's crucial to take into account all relevant practical considerations and optimize.

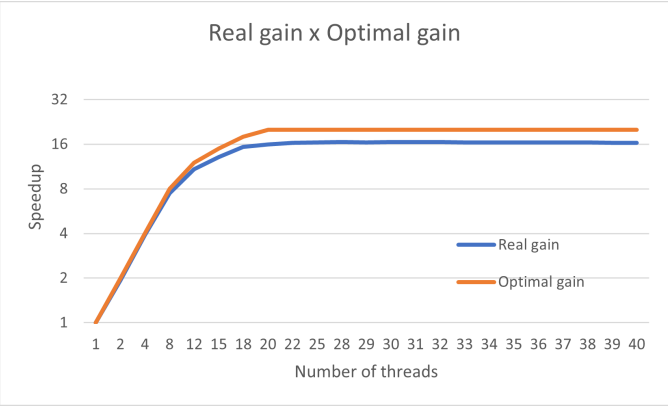


Fig. 2. Real Gain X Optimal Gain

IV. Optimizing the Molecular Dynamics Simulation Code with CUDA

A. Abbreviations and Acronyms

a) CUDA: Compute Unified Device Architecture
b) MPI: Message Passing Interface
c) GPU: Graphics Processing Unit

B. Assignment Objective

This assignment's objective is to decrease the program's execution time by implementing a parallel version of the code, in our case using GPU accelerators, and understand the problems and benefits that this way of parallelization introduces.

C. Why CUDA

Initially, we needed to choose a code parallelization tool, and we opted for *CUDA*, a parallel computing platform developed by *NVIDIA*. *CUDA* utilizes the power of *GPU* accelerators to parallelize tasks and reduce execution time. Alternatively, we could have chosen MPI, which allows developers to create processes that communicate with each other, improving shared memory performance and, consequently, optimizing execution time.
Another option was to use OpenMP to enhance code optimization and continue work on previous tasks. However, due to grade caps, this option was automatically excluded. After taking these factors into consideration and careful deliberation, we decided to use *CUDA*. This decision was also supported by one of us, who had an *NVIDIA* graphics card, making testing and experimentation easier, contributing to a more efficient learning curve and smoother implementation.
While MPI offers the benefits of process communication that improve shared memory performance, we wisely chose *CUDA* for parallel optimization due to the nature of the task, which requires intensive computation and our specific group environment.

D. How does CUDA work?

CUDA is a modern programming model designed for multicore *CPU* hosts coupled to many core devices. The device has extensive SIMD/SIMT parallelism, and the host and device do not share memory. This parallel computing platform developed by *NVIDIA* provides thread abstractions to handle SIMD, synchronization, and data exchange between small groups of threads.
In practice, *CUDA* allows the *GPU* to be used for parallel computing tasks alongside traditional rendering functions. Working with *CUDA* involves creating parallel programs called "*kernels*" that run on *GPUs*. These kernels are made up of threads, which are small units of work that can run concurrently. Threads are organized into blocks, and these blocks can be arranged into grids. Each thread runs an instance of the kernel and processes specific data.

*1) **Arquitetura**:*

*a) **sm_35**:* The sm_35 architecture compatible with *SeARCH* introduced some limitations, especially when performing operations such as *atomicAdd* with double values. Although it supports many important features, only floating point numbers are supported, so this limitation can affect certain important operations related to numerical precision. Double functionality had to be implemented manually, resulting in slower performance.

*b) **sm_70**:* The sm_70 architecture has several advantages over previous architectures. One of us had his RTX 2060 graphics, so we also used this architecture to simplify the implementation and achieve better performance. The sm_70 architecture provides the latest optimizations and hardware improvements that allow you to perform complex operations more efficiently. This was critical to achieving great results in terms of speed and efficiency.

E. Hotspots Identification

In this assignment, hotspot identification was quite easy, because that identification was already done in work assignment 2, so our focus was to parallelize the *computeAccelerationsPotential* function since that is the code block with more temporal complexity.

F. Parallelization with CUDA

In this section, we will provide a detailed explanation of the steps taken to create the parallel version of the code using CUDA. This involved modifying existing functions and creating new ones.

*1) **GPU Kernels**:* Kernels are "data-parallel portions of an application which run on many threads - SIMT" (Single Instruction Multiple Threads). In SIMT architecture, a kernel is a block of code that is run by a great number of threads, each operating on different portions of data.

In our implementation, we created two kernels, the first one "computeAccelerationsPotential" is the kernel that corresponds to the most intense function in terms of time complexity. This kernel is the key to achieving parallelization and optimizing the performance of the molecular dynamics simulation.
The other one, "computeAccelerations" is very similar to the other one, but it is executed just one time, nevertheless, we parallelized it because we got an execution time improvement.

*2) **Inside "computeAccelerationsPotential"**:* More specifically, inside the major kernel, to make the parallelization work, we replaced the first "for" loop, with a condition "$i < n$", that prevents the threads from accessing out-of-bounds memory and ensures that the threads operate only with valid data.
After that, each thread creates an array "ak" that stores the values of the incremented accelerations inside the second loop, and at the end of that loop, the values are added to

the matrix ”atomically” (more detailed explanation about atomicAdd afterward).
Inside the second loop, we also have ”atomicAdds” to sum the decremented accelerations and a pot variable at the end of the loop likewise the incremented accelerations are atomic added to the respective global variable.

*3) **Atomic Add Double**:* The ”atomicAdd” is a function responsible for making numeric additions atomically, or in other words, without interference from other threads, which prevents concurrent memory accesses and consequently data corruption. This function is especially useful inside CUDA kernels.

This is where our first problem began, currently, the ”cpar” partition in SeARCH cluster, runs CUDA in a ”Tesla K20” GPU which is sm_35 architecture, and that was a problem because in this architecture atomic operations with double precision numbers are not supported.
After some research, we found out (in CUDA documentation) that this operation could be performed with a custom function (that we call ”atomicAddDouble”) using ”atomicCAS” (a function that does an atomic compare-and-swap operation). This is a good solution for our problem but comes with the cost of some execution time increase.
We also tried to change the data types to float (single precision), but as we expected, precision was brutally affected, and even with a great execution time decrease we decided to exclude this approach.
It should be noted that afterward, we will perform some tests in an RTX 2060, which is sm_70 architecture and supports atomic operations with doubles, thus we will have both approaches (with and without atomicAdd) in the results section.

*4) **Inside ”VelocityVerlet”**:* Inside this function, we launch de GPU kernel ”computeAccelerationsPotential”, which takes some positional arguments, so before this call, we did a ”cudaMemcpy” (copies the value of a variable to another) in the ”r” matrix, to a new ”d_r” matrix with the intent to be used in the kernel. We also have the ”cudaMemset” function to initialize two new variables ”d_a” and ”d_PE” to zero.

The kernel call has also a specific syntax for CUDA configuration ”<<<gridSize, blockSize>>>”, that specifies the number of thread blocks in the grid (gridSize), and the number of threads in each thread block (blockSize).

All variables used in this function are global and initialized in the ”main” function with the ”cudaMalloc” function and freed with cudaFree.

*5) **Block Size**:* The block size is a critical parameter for code parallelization in CUDA, its selection is an essential point in obtaining the best performance possible. After some exhaustive testing (results shown in the results section), to find the best block size, we think the best size is 16 (it has to be a power of 2, otherwise the data gets corrupted).

G. Testing Methodology and Performance Evaluation

*1) **Results**:* Tables and Graphs

	2000	4000	5000	10000	30000	50000
16	3,469	5,567	6,291667	16,65967	121,9187	334,1097
32	4,006333	6,115667	7,654	16,125	103,7847	277,7303
64	4,899333	6,649333	7,884333	16,382	97,899	264,7177
128	5,202667	6,689333	7,931	16,64467	94,46333	258,5783
256	5,949333	7,055	9,304667	17,76533	95,832	259,1

Fig. 3. Execution times with different input and block sizes in SeARCH cluster.

Fig. 4. Graph of the execution times with different input and block sizes in SeARCH cluster.

	2000	4000	5000	10000	30000	50000
16	0,744333	1,660333	2,187667	6,720333	55,698	156,97
32	0,660333	1,149	1,451333	4,3	31,55767	85,26
64	0,664333	1,149667	1,607333	4,393333	33,495	84,89
128	0,837333	1,352667	1,755	5,649667	36,371	86,25
256	1,254333	2,260333	2,667	6,184	40,544	91,57

Fig. 5. Execution times with different input and block sizes in RTX 2060.

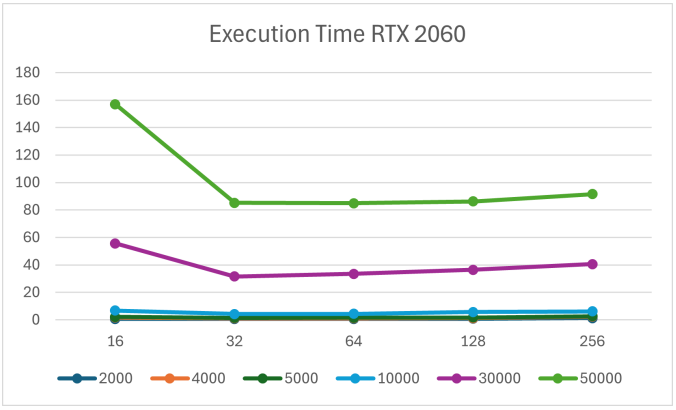


Fig. 6. Graph of the execution times with different input and block sizes in RTX 2060.

	2000	4000	5000	10000	30000
16	4,352	3,375333	3,725667	9,618333	49,464
32	4,817333	5,644667	6,037333	9,199667	43,853
64	5,34	5,728667	6,107	7,448	40,90433
128	5,930667	6,179667	6,146	9,341333	42,57567
256	6,053667	6,492333	6,328667	9,766333	43,30167

Fig. 7. Execution times with different input and block sizes in RTX 2060.

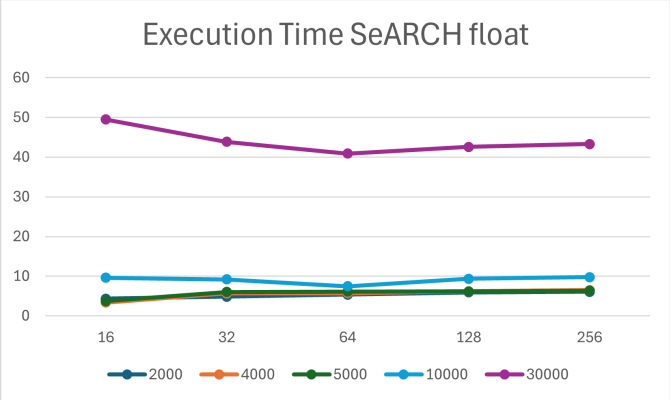


Fig. 8. Graph of the execution times with different input and block sizes in RTX 2060.

*2) **Analysis of Results**:* In this section, we will present and discuss the results of the many tests run on different environments and with different conditions. Firstly we ran the program in *SeARCH* cluster (these partitions: compute-134-111, compute-134-112, compute-134-101) (GPU: Tesla K20), and we considered different input sizes (2000, 4000, 5000, 10000, and 30000) and different block sizes (16, 32, 64, 128, and 256). In the first step, we experimented with the *CUDA*-supported atomicadd function in Figures 7 and 8 using a matrix of floating point numbers (floats). Although the performance improved significantly, the accuracy of the results decreased significantly.

Next, we tested a more robust implementation in *CUDA*, as shown in Figures 3 and 4. We observed no data races regardless of the block size tested. When migrating to the local environment, we used the RTX 2060 GPU mentioned above. Better results were obtained (Figures 5 and 6) thanks to an architecture that supports sm_70 and allows atomic operations on double-precision floating point numbers using atomicAdd. After analyzing the results, we found that there is a relationship between the block size and the number of Argon gas particles (N). Increasing the value of N increased the block size and gave better results. This relationship can be explained by parallelism optimization on GPUs. Larger blocks encourage efficient collaboration between threads and minimize startup overhead. Additionally, we found that the block size should be a multiple of 32 for optimal performance, as shown by local graph analysis. However, note that this pattern may not apply to *SeARCH* clusters because the architecture is different.

These findings improve our understanding of the relationship between program parameters and performance and highlight the importance of making careful configuration decisions in his *CUDA* environment for effective optimization.

H. Numerical Precision Considerations

Beyond all the different tests run before, we also tested a different code implementation in the SeARCH cluster, we replaced the double-precision variables with floats so we could use ”atomicAdd” in the cluster (sm_35 architecture). The results of this experience, as mentioned before, were great if we wanted to decrease execution time at any cost, but the precision of the results was severely affected, which is an outcome that doesn't fit our solution, so we kept the experience results but discarded the implementation.

I. Recommendations for Future Work

In the future, we could try to implement this solution with MPI and take advantage of inter-process communication for code parallelization and optimization.

J. Final Conclusions

In summary, implementing this solution with CUDA parallelization decreased significantly the execution time. By using the parallelization capabilities of the GPU, we successfully achieved our goal. The efficient use of these resources, careful memory access, and optimized thread execution contributed to the overall acceleration of our algorithm. CUDA parallelization proved to be a valuable approach for enhancing the computational efficiency of our code.

However, when we compared the performance of programming with OpenMP on SeARCH and a local PC, we saw a clear difference. In the case of SeARCH, the OpenMP implementation was faster, probably due to the nature of the cluster environment and the efficient coordination of multiple processors. However, on my local PC, CUDA parallelization performed better than his OpenMP implementation. This is due to the unique features of modern GPUs such as the RTX 2060, which are optimized for parallel computing and can efficiently perform large-scale operations compared to traditional processors.

To conclude we think that these assignments were very enlightening since more and more complex problems tend to appear the need for code parallelization knowledge is urgent, so we find this course and assignments very useful for our future.

REFERENCES

[1] Computer Organization and Design: The Hardware/Software Interface, David Patterson and John Hennessy, 5th Ed., Morgan Kaufmann, 2013
[2] Course slides
[3] http://search6.di.uminho.pt/wordpress/?page_id=55
[4] https://en.wikipedia.org/wiki/Parallel_computing
[5] https://www.nvidia.com/content/PDF/kepler/tesla-k20-active-bd-06499-001-v03.pdf
[6] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

V. ANNEXES

```c
double Potential() {
    double quot, r2, rnorm, term1, term2, Pot;
    int i, j, k;
    Pot=0.;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if (j!=i) {
                r2=0.;
                for (k=0; k<3; k++) {
                    r2 += (r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
                }
                rnorm=sqrt(r2);
                quot=sigma/rnorm;
                term1 = pow(quot,12.);
                term2 = pow(quot,6.);

                Pot += 4*epsilon*(term1 - term2);
            }
        }
    }
    return Pot;
}
```

Fig. 9. Original function Potential()

```c
double Potential(){
    double quot, r2, rnorm, Pot, term3, term6, term12, aux, r0i, r1i, r2i,term1,term2;
    int i, j, k;
    double var = 8 * epsilon; //(2 * (4 * epsilon));
    Pot = 0.;
    for (i = 0; i < N; i++){
        r0i = r[i][0];
        r1i = r[i][1];
        r2i = r[i][2];
        for (j = i + 1; j < N; j++){
            r2 = 0.;
            aux = r0i - r[j][0];
            r2 += aux * aux;
            aux = r1i - r[j][1];
            r2 += aux * aux;
            aux = r2i - r[j][2];
            r2 += aux * aux;
            rnorm = sqrt(r2);
            quot = sigma / rnorm;
            term3 = quot * quot * quot;
            term6 = term3 * term3;
            term12 = term6 * term6;
            Pot += term12 - term6;
        }
    }
    Pot = Pot * var;
    return Pot;
}
```

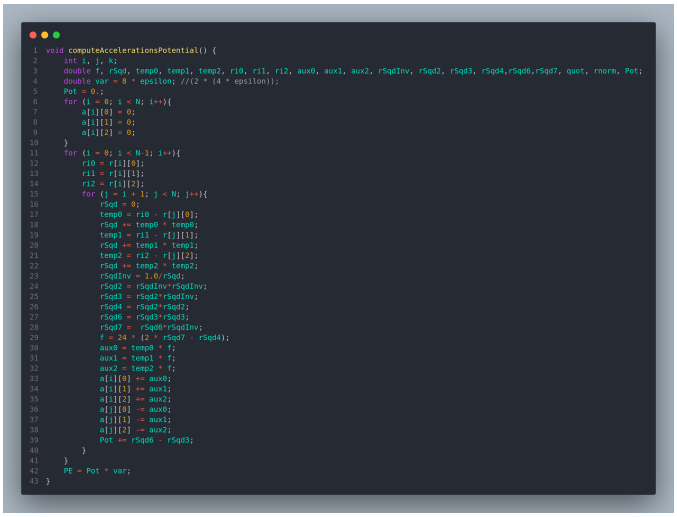
Fig. 10. Optimized function Potential()

```c
void computeAccelerations() {
    int i, j, k;
    double f, rSqd;
    double rij[3]; // position of i relative to j
    for (i = 0; i < N; i++) {  // set all accelerations to zero
        for (k = 0; k < 3; k++) {
            a[i][k] = 0;
        }
    }
    for (i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j
        for (j = i+1; j < N; j++) {
            rSqd = 0;
            for (k = 0; k < 3; k++) {
                //  component-by-componenent position of i relative to j
                rij[k] = r[i][k] - r[j][k];
                //  sum of squares of the components
                rSqd += rij[k] * rij[k];
            }
            f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4));
            for (k = 0; k < 3; k++) {
                //  from F = ma, where m = 1 in natural units!
                a[i][k] += rij[k] * f;
                a[j][k] -= rij[k] * f;
            }
        }
    }
}
```

Fig. 11. Original function computeAccelerations()

```c
void computeAccelerations(){
    int i, j, k;
    double f, rSqd, temp0, temp1, temp2, ri0, ri1, ri2, aux0,aux1,aux2,rSqdInv,rSqd2,rSqd4,rSqd7;
    for (i = 0; i < N; i++){
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }
    for (i = 0; i < N - 1; i++){
        ri0 = r[i][0];
        ri1 = r[i][1];
        ri2 = r[i][2];
        for (j = i + 1; j < N; j++){
            rSqd = 0;
            temp0 = ri0 - r[j][0];
            rSqd += temp0 * temp0;
            temp1 = ri1 - r[j][1];
            rSqd += temp1 * temp1;
            temp2 = ri2 - r[j][2];
            rSqd += temp2 * temp2;
            rSqdInv = 1.0/rSqd;
            rSqd2 = rSqdInv*rSqdInv;
            rSqd4 = rSqd2*rSqd2;
            rSqd7 =  rSqd4*rSqd2*rSqdInv;
            f = 24 * (2 * rSqd7 - rSqd4);
            aux0 = temp0 * f;
            aux1 = temp1 * f;
            aux2 = temp2 * f;
            a[i][0] += aux0;
            a[i][1] += aux1;
            a[i][2] += aux2;
            a[j][0] -= aux0;
            a[j][1] -= aux1;
            a[j][2] -= aux2;
        }
    }
}
```

Fig. 12. Optimized function computeAccelerations()



```
void computeAccelerationsPotential() {
    int i, j, k;
    double f, rSqd, temp0, temp1, temp2, ri0, ri1, ri2, aux0, aux1, aux2, rSqdInv, rSqd2, rSqd3, rSqd4,rSqd6,rSqd7, quot, rnorm, Pot;
    double var = 8 * epsilon; //(2 * (4 * epsilon));
    Pot = 0.;
    for (i = 0; i < N; i++){
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }
    for (i = 0; i < N-1; i++){
        ri0 = r[i][0];
        ri1 = r[i][1];
        ri2 = r[i][2];
        for (j = i + 1; j < N; j++){
            rSqd = 0;
            temp0 = ri0 - r[j][0];
            rSqd += temp0 * temp0;
            temp1 = ri1 - r[j][1];
            rSqd += temp1 * temp1;
            temp2 = ri2 - r[j][2];
            rSqd += temp2 * temp2;
            rSqdInv = 1.0/rSqd;
            rSqd2 = rSqdInv*rSqdInv;
            rSqd3 = rSqd2*rSqdInv;
            rSqd4 = rSqd2*rSqd2;
            rSqd6 = rSqd3*rSqd3;
            rSqd7 =  rSqd6*rSqdInv;
            f = 24 * (2 * rSqd7 - rSqd4);
            aux0 = temp0 * f;
            aux1 = temp1 * f;
            aux2 = temp2 * f;
            a[i][0] += aux0;
            a[i][1] += aux1;
            a[i][2] += aux2;
            a[j][0] -= aux0;
            a[j][1] -= aux1;
            a[j][2] -= aux2;
            Pot += rSqd6 - rSqd3;
        }
    }
    PE = Pot * var;
}
```

Fig. 13. Function computeAccelerationsPotential()

Physical cores	Threads	Exec time (s) 1	Exec time (s) 2	Exec time (s) 3	Mean (s)	Speedup
1	1	35,904	35,909	35,895	35,903	1
2	2	18,235	18,173	19,126	18,511	1,939496525
4	4	9,232	9,225	9,224	9,227	3,891044399
8	8	4,783	4,791	4,789	4,788	7,498990462
12	12	3,306	3,319	3,320	3,315	10,83036702
15	15	2,722	2,731	2,758	2,737	13,11752527
18	18	2,343	2,338	2,335	2,339	15,35176739
20	20	2,208	2,280	2,249	2,246	15,98753154
20	22	2,171	2,201	2,235	2,202	16,30210383
20	25	2,161	2,234	2,168	2,188	16,41139723
20	28	2,173	2,166	2,165	2,168	16,5602706
20	29	2,176	2,178	2,176	2,177	16,49433384
20	30	2,170	2,173	2,164	2,169	16,55263562
20	31	2,169	2,183	2,168	2,173	16,5196319
20	32	2,169	2,169	2,179	2,172	16,52723646
20	33	2,173	2,178	2,184	2,178	16,48171385
20	34	2,173	2,180	2,191	2,181	16,45904645
20	35	2,174	2,175	2,190	2,180	16,47163175
20	36	2,179	2,178	2,191	2,183	16,44899206
20	37	2,180	2,191	2,177	2,183	16,44899206
20	38	2,182	2,183	2,186	2,184	16,44145932
20	39	2,188	2,203	2,199	2,197	16,34415781
20	40	2,190	2,198	2,189	2,192	16,37646343

Fig. 15. Execution time

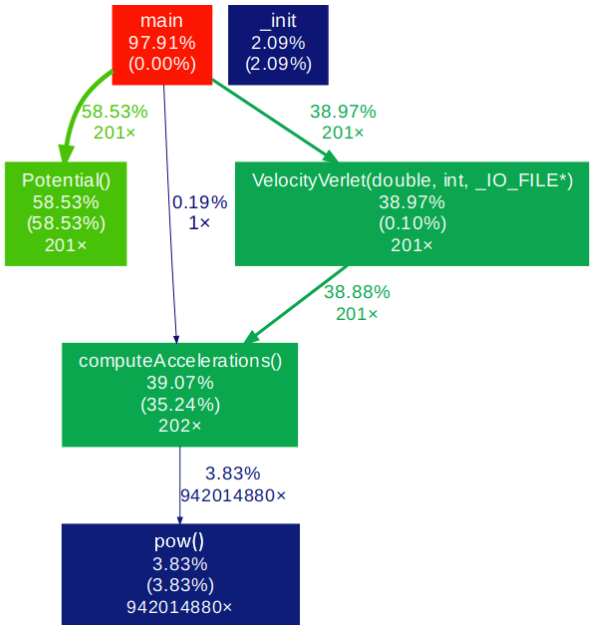


Fig. 14. Original call-graph

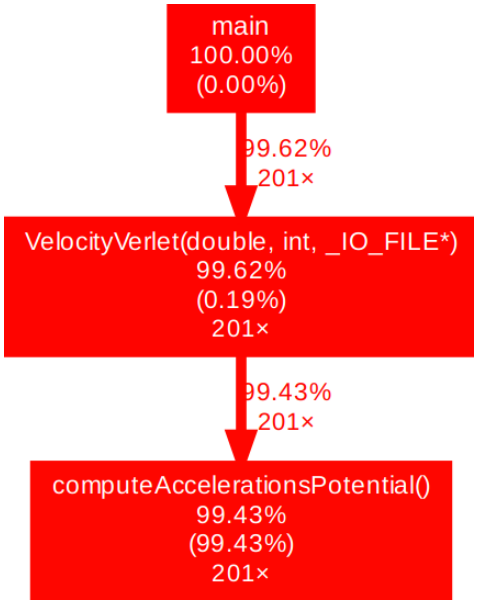


Fig. 16. Optimized call-graph