Computação Paralela

Phase 3

Ricardo Lopes Lucena
Departamento de Informática
Universidade do Minho
Braga, Portugal

Xavier Santos Mota
Departamento de Informática
Universidade do Minho
Braga, Portugal

***Abstract*—This document delves into our group's efforts to enhance the provided code from our professors. We analyze various metrics and draw conclusions from our optimizations.**

***Index Terms*—Optimization, clock cycles, execution time, code readability, parallelization, openMP, threads, CUDA**

I. Introduction

In the final phase of our comprehensive report, we present the culmination of our group's iterative efforts to elevate the performance of the provided code. Building upon the enhancements achieved in the initial and second phases, where we meticulously refined and parallelized the code, this document focuses on the integration of CUDA implementation for further performance gains. We thoroughly analyze the impact of CUDA acceleration on our computational strategies, employing a diverse set of metrics to measure and interpret the improvements. This conclusive phase not only underscores the evolution of our practical work but also provides a holistic perspective on the cumulative optimizations undertaken throughout the project. By synthesizing insights from each phase, we derive comprehensive conclusions that encapsulate the entirety of our group's contributions to the codebase, emphasizing the iterative nature of our approach and the resulting advancements in computational efficiency.

II. Phase I

A. A priori assessment of the code

Upon initial examination and utilizing the 'perf' tool, our group promptly pinpointed the two primary functions within the code responsible for nearly the entire execution time. These functions are 'potential' and 'computeAccelerations.' In contrast, the 'velocityVerlet' function incurred minimal overhead, while all other functions had negligible impact on performance.

The thought of the group, after analising the entire code was to start making adjustments to one of the two functions refered above that took most of the execution time. Since the "Potential" function took more time than the other we decided we would start there.

B. Changes in the code

In this section we will be showcasing the strategies that we applied so that our code would transform into something more efficient with less execution time. It should be noted that the time that the program took to run was 240 seconds (around 4 minutes) in the SeARCH engine.

The following optimizations, shown in the next subchapters, were implemented in the 'Potential', 'computeAccelerations' and velocityVerlet functions

1) "Pow" and "Sqrt" function removal: The first thing we noticed was that there were many calls made to the *pow* and *sqrt* functions. Since they were inside two loops that went from 0 to 2160, it meant that they were called many times, which meant they could be one of the main sources of the heavy execution time in the program since calling these functions involves a function call and additional overhead.

The main change was made in this snippet of code:

```
rnorm=sqrt(r2);
quot=sigma/rnorm;
term1 = pow(quot,12.);
term2 = pow(quot,6.);
```

The 'pow' function was replaced with manual multiplication, and to maintain the same result while removing the "sqrt" function, all we needed to do is divide the exponent by 2. This is because the square root of a value raised to the 12th power is equivalent to that value raised to the 6th power. After making these changes, the code execution time was significantly improved.

```
quot = sigma/(r2);
term2 = quot *quot * quot;
term1 = term2 *term2;
```

2) Loop Unrolling: The loop unrolling optimization strategy capitalizes on maximizing the use of the L1 cache. Given that the cluster's CPU has 64 bytes per row, and each double takes up 8 bytes, we can infer that each memory access loads 8 doubles into the cache. This strategy was primarily employed in code blocks like the following:

```
for(k=0; k<3; k++){
r2+=(r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
```

By implementing this optimization in this context, we can efficiently access 6 doubles without relying on main memory, making optimal use of the cache. Ultimately, this results in a reduction in the program's execution time.

The optimized version of the code unrolls the loop as follows:

```
r2+=(r[i][0]-r[j][0])*(r[i][0]-r[j][0]);
r2+=(r[i][1]-r[j][1])*(r[i][1]-r[j][1]);
r2+=(r[i][2]-r[j][2])*(r[i][2]-r[j][2]);
```

This unrolled version minimizes cache misses and contributes to improved execution time.

3) Improved array access patterns: In both the 'potential' and 'computeAccelerations' functions, we optimized the array access patterns in two key ways:

Firstly, we harnessed the memory hierarchy by transposing the primary arrays, 'a,' 'r,' and 'v,' originally structured as

```
array[N][3] Into array[3][N]
```

This transformation significantly reduced cache misses during repeated array accesses since we now take advantage of the L1 cache. The original code didn't take full advantage of the caching mechanism, as every new access to an array value caused a cache miss, necessitating the retrieval of the required data line into the cache. To enhance this aspect of performance, we made an improvement by transposing the array structure. We reconfigured it into a 3-line by N-column format. This adjustment increased the likelihood that the subsequent item needed for array access was already loaded in the cache, ultimately resulting in improved overall performance.

Secondly, we substantially minimized the number of array accesses. This was achieved by storing values corresponding to array accesses dependent solely on the 'i' variable of the outer loop in local variables. These local variables were updated within the inner loop, thereby eliminating the need to access those positions within the inner loop on each iteration. Upon completion of the inner loop, the array values were updated based on the local variables. This approach greatly enhanced efficiency in array access.

```
for (i = 0; i < N-1; i++) {
    for (j = i+1; j < N; j++) {
        y = array[i];
```

Would change to

```
for (i = 0; i < N-1; i++) {
    x = array[i];
    for (j = i+1; j < N; j++) {
        y = x;
```

4) Optimization of mathematical equations: We enhanced the code's performance by substituting certain mathematical formulas with equivalent expressions that significantly improved computational speed.

As an example, consider the original formula:

```
f = 24 * (2*pow(rSqd, -7) - pow(rSqd, -4));
```

Additionally, following the modifications in subsection B, we streamlined the process by reducing the number of divisions, which can be computationally expensive, to just one division per iteration, we did this by first calculating the inverse of rSqd and using it to calculate pow(rSqd, -7) and pow(rSqd, -4). The revised formula looks as follows:

```
double invSqd = 1/rSqd;
double invHelp = invSqd * invSqd;
double invHelp2 = invHelp * invHelp;


f=24*(2*invHelp2*invHelp*invSqd-invHelp2);
```

This significantly improved performance.

5) Loop optimizations: In the 'potential' function, we eliminated the 'if' statement to maximize CPU pipelining resources utilization. To ensure that the logic of the 'if' statement placement remains consistent with

```
if(j != i)
```

we divided the original 'for' loop (ranging from 0 to N) into two identical loops. One loop iterates from 0 to 'i' while the other spans from 'i+1' to 'N.' Afterwards, we discovered that the 'r' matrix exhibited symmetry, eliminating the necessity for both loops. As a result, the 'potential' function now only performs half the number of cycles compared to its previous implementation.

This reconfiguration effectively preserves the 'if' statement's intended logic while improving code performance.

6) Vectorization and overall flags: To leverage vectorization for optimizing our code, we incorporate several compilation flags, including -mavx and -ftree-vectorize. These flags enhance performance by making efficient use of vector operations. In addition, we apply -Ofast, -funroll-all-loops, and -march=native. These combined flags help reduce execution time and improve the overall efficiency of our code.

7) Creating auxiliary functions: To enhance the overall code readability, we introduced auxiliary functions at various points in our code. An example of this can be found in the 'Potential' function, where we delegate the intensive mathematical computations to a separate function called 'contaPotential.' However, it's worth noting that this modification is not meant to have an impact on code's execution time.

C. Final Execution Time

Reaching the end of this phase, these were the obtained results, where the first line corresponds to the initial run, without

any code optimizations, while the second line represents the final results after all the optimizations have been applied.

#I	#CC	#CM
1247809985705	1061626728804	1392086
6633867623	7597833881	2168

Texec Inicial: 348,56s **Texec Final:** 2,45s

III. Phase II (OpenMP)

In this phase, by strategically leveraging parallel computing, we aim to optimize the performance of our codebase, achieving a balance between efficiency and scalability.

A. A priori assessment of the code

By our previous knowledge of the work developed on our previous phase, we know for a fact that the "Potential" and "computeAcceleration" is where most of our program's resource's are consumed, making both of them our primary focus for optimization.

Given that all the other functions have almost no weight on the entire code, we decided to make barely any changes on them.

What we also strived to change in this phase of the work was to correct things done in the past, such as unnecessary loop unrolling and changes that had no impact on the code and made code readability worse.

B. Small overall changes to the code

Before diving into the parallelization part of this paper, the team decided to fix some issues within our code that were referenced by our teachers.

In the first phase of the work we loop unrolled certain parts of the code that barely had any impact on the execution time. Besides that, loop unrolling is something that the compiler does automatically with the correct flags, which made this change useless, only worsening code readibility. Because of this, we reverted those changes.

Besides that, we noticed that since the two *for* cycles inside of the ***Potential*** and ***computeAcceleration*** functions are the same and only their content varies, we decided to join both of them into a single function: ***PotentialEcomputeAccelerations***. This change reduced the number of cycles done and showed a great improvement in execution time.

C. Parallelism Analysis

Before starting to dwelve in the code, the teachers told us to increase the number of atoms (N) to 5000. Since in the previous phase the N was set to 2160, we knew we would witness an huge increase in execution time.

The team acknowledged that we should be carefull on where we use the *pragma* methods, since they carry a heavy overhead and they will not always be beneficial for our code. We recognized that applying *pragma omp parallel for* to every individual loop would not be advantageous, considering that many of these loops have relatively short execution times.

So, our only focus is on the one function that takes an huge toll on our execution time, which is *PotentialEcomputeAccelerations*, which is only composed of a nested loop.

D. Applying Parallelization

To start things off we started to apply a

```
#pragma omp parallel for
```

on the outter for cycle. The first thing we noticed was that there were two variables that were being written on during the cycles, which were the ***Pot*** and ***transpostaA*** variables. To ensure that there were no data races between all threads we added

```
reduction(+:Pot) reduction(+:transpostaA[:3][:N])
```

so that when the threads would end, the values inside these variables (each thread has a Pot variable) would be summed into a single variable. The usage of *reduction* instead of pragma *atomic* or *critical* arises since they impose a heavy overhead, and the last one makes the thread stop and wait for others.

Furthermore, we had many variables that were important to be private for each thread. Instead of using *private(x)*, we opted for declaring each required variable within the loop, eliminating the need for the explicit *private(x)* directive and mitigating additional performance overhead.

Finally, the selection of an appropriate scheduling strategy plays a crucial role in achieving efficient workload distribution among threads. In this context, we chose to employ the following OpenMP directive:

```
schedule(dynamic, 50)
```

The use of schedule(dynamic, 50) is particularly beneficial in scenarios where the workload of each iteration within the loop is not uniform. By opting for a dynamic scheduling strategy, the runtime system dynamically assigns iterations to available threads, enabling a more balanced distribution of the computational load. This is especially advantageous when certain iterations require significantly more processing time than others, which is exactly our case.

The parameter 50 in the dynamic schedule denotes the chunk size – the number of iterations assigned to each thread at a time. By setting an appropriate chunk size, we aimed to strike a balance between minimizing the overhead of task scheduling and ensuring that each thread receives a sufficient workload to maintain high parallelization efficiency.

E. Scalability

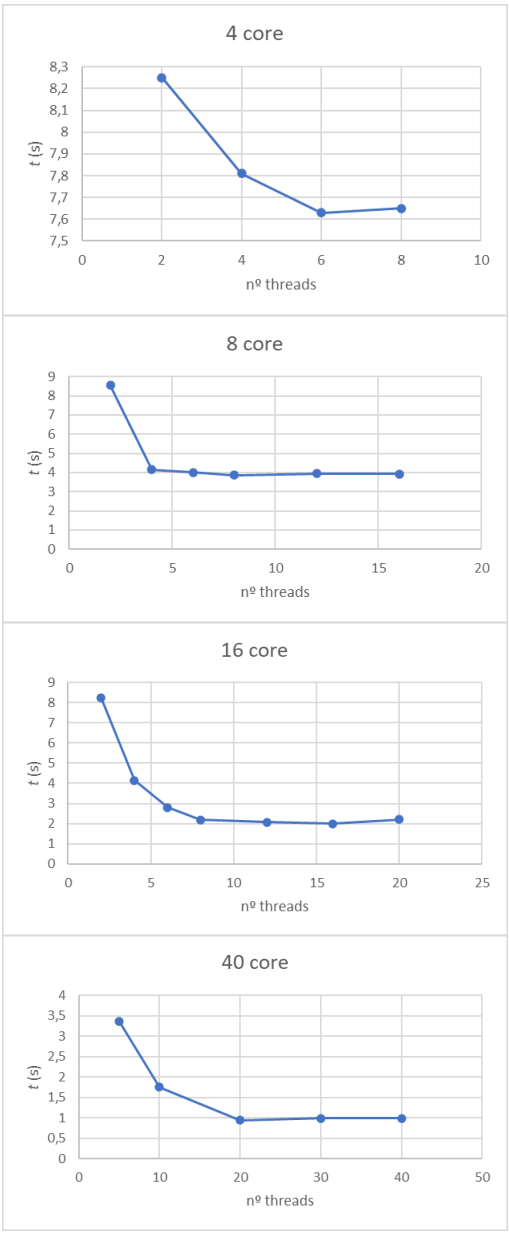
In our comprehensive testing of the new code, we closely examined the impact of both the number of cores and the number of threads on the program's scalability.

Initially, we observed substantial improvements in program speed with the addition of more cores. However, as expected, there was a point where the speed gains reached a plateau. This plateau suggests that while more cores initially enhance parallel execution, certain factors, such as communication

overhead or algorithm constraints, limit further improvements beyond a certain core count.

Interestingly, within each test variation holding the core count constant, we noticed that increasing the number of threads also contributed to a speedup. This trend continued until a critical point where the number of threads equaled half the number of cores. At this point, the program's speed exhibited a slight decline compared to configurations with fewer threads. We hypothesize that this phenomenon is a result of increased contention for resources, leading to diminished efficiency.

Our testing further revealed a critical turning point where the addition of more cores ceased to provide significant improvements in execution time. This observation aligns with Amdahl's Law, which posits that the speedup of a parallel program is constrained by the sequential portion of the code. As the sequential fraction becomes increasingly prominent, the potential for further speed gains diminishes.



F. Closing thoughts about Phase II

We successfully strategically applied OpenMP directives, addressed previous issues and consolidated functions while maintaining code readability and improving execution time.

Our analysis revealed significant speed improvements with more cores, reaching a plateau —a phenomenon consistent with Amdahl's Law. Interestingly, increasing threads contributed to speedup until a critical point, where due to factors like communication overhead, led to a slight decline in efficiency.

In essence, our parallelization efforts proved successful, providing insights into the delicate balance between cores and threads and the impact of Amdahl's Law on program speedup.

IV. Phase III (CUDA)

In the third and final part of this project, we were given two choices for parallel programming. We could either use MPI (Message Passing Interface), a standard for coordinating communication between processes in distributed systems, or CUDA (Compute Unified Device Architecture), a platform for harnessing the computational power of GPUs in parallel computing tasks. The group chose the latest option, because CUDA shows a greater performance on bigger loads of work than MPI. Besides that, it was the option that the group had more interest in.

A. A priori assessment of the code

Since we are already familiar with our code the group already knew what were its main flaws and hot-points in terms of execution. Since all the execution time and resources go when the program is executing the *potentialEcomputeAccelerations*, we knew that that would be the only function that would undergo some changes.

B. CUDA Implementation

Before explaining our implementation, it is important to have in regard how CUDA works. First of all we need a function on our device that would be capable of sending everything necessary to the GPU. Secondly, we have to specify what the GPU has to do by creating another function (kernel). After the GPU does the changes to the variables sent before from the device, the same function that sent those variables to the GPU, retrieves them.

The first thing the group decided to do was to find a way to send all the data necessary to the GPU. Instead of sending the two matrix *transpostaA* and *transpostaR*, we build a single array with the lines from the matrixes. Since each array only has 3 lines with N columns, the size of the array we build is N*6. This option is better than sending 6 arrays of N size (3 for each matrix), since transferring a larger contiguous block of data is more efficient than transferring multiple smaller blocks due to factors like memory alignment and transaction overhead.

With this, we only need to send two things to the GPU, the array we build, and a variable so the GPU can send the value of the potential. We alocate memory for each variable using *cudaMalloc*, as follows:

```
    cudaMalloc ((void**) &dr, bytes);
cudaMalloc((void**)&result, sizeof(double));
```

After that, we use *cudaMemcpy* to copy the array we built before into the variable that goes to the GPU (dr), and *cudaMemset* to set the potential value to 0.

Before calling the kernel we need to calculate the value of *threads_per_block* and the number of *blocks* we desire, having always in account that:

```
threads_per_block*num_blocks > N
```

Since we have this restriction, to calculate the values of these variables, we choose a number of threads per block that is a multiple of 32, since it is the best value to run due to the underlying hardware architecture of NVIDIA GPUs and how thread execution is managed. After having the number of threads per block we use the following formula:

```
(N+threads_per_block-1)/threads_per_block
```

For example, if the number of threads per block is 256, the number of blocks would have to be 20.

When the GPU is done running our code, we retrieve the values back and make sure that the matrixes are updated with their correct values. To end the process, we free the memory that was previously allocated using *cudaFree*.

Secondly, we had to build the kernel. Most of the code inside the kernel is the same as the *potentialEcomputeAccelerations* in our previous phase. The main differente is that we had to set it up for multiple threads to run it. So firstly we wrote

```
int id = blockIdx.x * blockDim.x + threadIdx.x;

if (id < tam-1)
```

making it so that each thread only runs once. Another difference from the *potentialEcomputeaccelerations* is that we had to change the function to do the necessary calculus using a single array instead of two matrixes. Since the first line of the matrixA allocates the first 5000 elements of the array, the second line the elements between 5000-10000, and so on, we use *array[j+5000]* for example, to access an element of the second line of the matrixA.

Another critical aspect within the function involves the utilization of shared memory (declared using the shared keyword) to store the calculated potential values from each thread within a thread block. Before the function concludes, the thread with an ID of 0 in each thread block takes the responsibility to sum up the potential values computed by individual threads within their respective block. Subsequently, an atomicAdd2 operation is employed to accumulate this block-level potential value to the overall potential value that will be read by the host (CPU).

V. Result Analysis

Before entering in this section it is important to note that the group build a script in order to aid us in this task. The script runs the code using different N, different number of threads per block and different number of blocks.

With that script, we were able to test out our program with different values for all those variables refferenced above. The results can be visualized in the table below:

Results			
N	Blocks	Threads Per Block	Time(s)
1000	21	256	4.322s
1000	11	512	4.307s
1000	6	1024	4.196
2160	21	256	5.224
2160	11	512	5.053
2160	6	1024	5.021
5000	21	256	7.514
5000	11	512	7.867
5000	6	1024	10.233

Looking at the table we can see that the highest the value of ”N”, the higher the time goes, as it should be. However, we can also see that the higher the number of threads per block, higher the execution time.

It is also important that as said before, we use 256, 512 and 1024 (multiples of 32) because it is the best value to run due to the underlying hardware architecture of NVIDIA GPUs and how thread execution is managed. If the number of threads per block * number of blocks surpasses N by a lot, the execution time will rise a lot since we are creating threads that will never work and therefore be completely useless.

VI. Conclusion

Throughout this project, we continuosly improved our program either by improving the initial code doing loop optimizations and other simple techniques, then by using OpenMP, and finally by using CUDA. Even though the best results were obtained when using OpenMP, the group believes that CUDA could have showed better results if we had implemented it better (take more advantage of shared memory for example), and if we were working with bigger loads of data. This phase showed us that the usage of CUDA is crucial as it empowers developers to leverage the parallel processing capabilities of GPUs, significantly accelerating complex computations through the usage of our hardware. Looking at our work, we feel like we did a great job at optimizing this code. It is satisfatory and impressive for us to think that the execution time for example, went form 240 seconds to 7 seconds using CUDA, and 1 second using OpenMP, which showed us how powerfull these tools can be, and gave us knowledge that we can use in our future.

Besides that, we feel like we could have done a better job at exploring these tools since both CUDA and OpenMP have many small things that can make a real impact on our programs. We think that if we did more research we could have acquired better results overall. We should also have dedicated more time to work on the metrics in this final phase of our project, to really see where CUDA could thrive. Nonetheless, we are satisfied with our work and our results.