Work Assignment - Phase 3

Afonso Ferreira
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
pg52669@alunos.uminho.pt

Catarina Costa
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
pg52676@alunos.uminho.pt

I. Introduction

Throughout the Parallel Computing subject, we delved into various segments within the field. In our first assignment we learned on optimizing a single-threaded molecular dynamics simulation code, using code analysis tools such as gprof and perf to identify critical areas and evaluated the execution times. Our primary goal, in this project was to better the execution speed without compromising simulation accuracy.

In our second assignment, we seamlessly transitioned to the implementation of shared memory parallelism using OpenMP, where we navigated challenges like critical zones and data races, refining our parallel approach. Our strategy aimed not only to accelerate computations but also to overcome nuanced hurdles, resulting in a robust and efficient parallelized codebase.

In the latest project phase, we extended our optimization efforts to include CUDA for GPU parallelization. By leveraging the parallel computing capabilities of GPUs, we tailored our molecular dynamics simulation code to harness the power of CUDA cores. This addition significantly accelerated computations, expanding our toolkit and providing a versatile and efficient solution for parallel processing. This systematic approach, encompassing optimization of the single-threaded program, shared memory parallelism with OpenMP, and GPU acceleration with CUDA, underscored the adaptability and scalability of our efforts, ultimately improving both theoretical and practical aspects of molecular dynamics simulation.

II. Phase 1: Code Optimization

A. Code Analysis

The group began the work assignment by following a systematic approach, starting with a thorough analysis of the code. We recognized the importance of understanding the simulation and its underlying mechanisms before diving into the optimization process. This initial analysis allowed us to gain insights into the code structure and identify potential areas for optimization right from the start.

To further aid our optimization efforts, we utilized the code profiler *gprof*, which was suggested as a valuable tool. By running the code through *gprof*, we were able to generate a call graph that provided a visual representation of the code's execution flow. This call graph proved to be instrumental in identifying the specific blocks of code that required our immediate attention.

Figure 1 showcases the call graph obtained from the *gprof* analysis. This visual representation made us recognize that the two most important functions to optimize were first the *Potential()* and secondly, the *computeAccelarations()*.

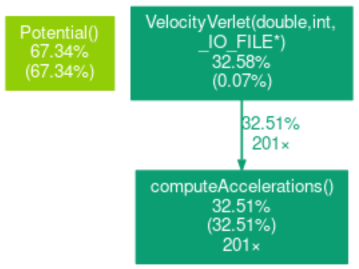


Fig. 1: Before Optimization

By combining our initial analysis with the insights gained from the *gprof* analysis, we established a solid foundation for our optimization efforts. This comprehensive approach allowed us to make informed decisions and prioritize our optimizations effectively, ultimately leading to significant improvements in the code's performance and efficiency.

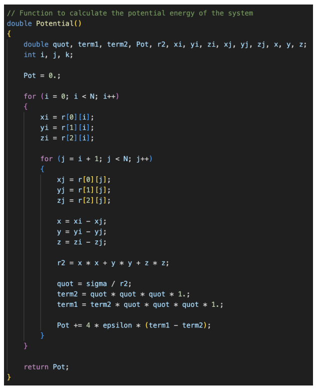
B. Optimizations

In our pursuit of optimization, we first focused on enhancing the performance of the crucial function, *Potential()*. This decision was influenced by its prominent presence in the *gprof* call graph, indicating its considerable impact on the overall execution time. Figure 2 (a and b) visually presents the optimization of the function *Potential()* through a side-by-side comparison of the before and after states.

Subsequently, the function *computeAccelerations()* was identified as the next candidate for optimization.

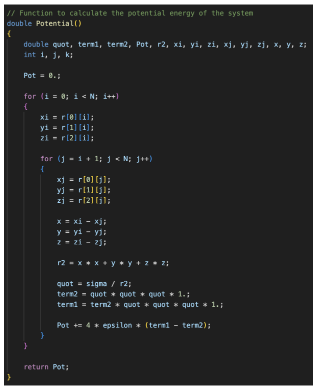
To optimize these functions, we decided to:

- **Optimizing *pow()* and *sqrt()* functions** - We identified the usage of *pow()* and *sqrt()* functions in the *Potential()* function as possible bottlenecks. *pow()* can be computationally expensive, so we replaced it with more efficient manual calculations. We also were able to completely remove the *sqrt()* function for optimal performance. Here's an example that demonstrates how we eliminated the square root calculation while preserving mathematical integrity.



```c
// Function to calculate the potential energy of the system
double Potential()
{
    double quot, term1, term2, Pot, r2, xi, yi, zi, xj, yj, zj, x, y, z;
    int i, j, k;

    Pot = 0.;

    for (i = 0; i < N; i++)
    {
        xi = r[0][i];
        yi = r[1][i];
        zi = r[2][i];

        for (j = i + 1; j < N; j++)
        {
            xj = r[0][j];
            yj = r[1][j];
            zj = r[2][j];

            x = xi - xj;
            y = yi - yj;
            z = zi - zj;

            r2 = x * x + y * y + z * z;

            quot = sigma / r2;
            term2 = quot * quot * quot * 1.;
            term1 = term2 * quot * quot * quot * 1.;

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

    return Pot;
}
```

(a) Before



```c
// Function to calculate the potential energy of the system
double Potential()
{
    double quot, term1, term2, Pot, r2, xi, yi, zi, xj, yj, zj, x, y, z;
    int i, j, k;

    Pot = 0.;

    for (i = 0; i < N; i++)
    {
        xi = r[0][i];
        yi = r[1][i];
        zi = r[2][i];

        for (j = i + 1; j < N; j++)
        {
            xj = r[0][j];
            yj = r[1][j];
            zj = r[2][j];

            x = xi - xj;
            y = yi - yj;
            z = zi - zj;

            r2 = x * x + y * y + z * z;

            quot = sigma / r2;
            term2 = quot * quot * quot * 1.;
            term1 = term2 * quot * quot * quot * 1.;

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

    return Pot;
}
```

(b) After

Fig. 2: Potential() before and after optimizations

$$(\frac{\sigma}{\sqrt{r2}})^{12} = \frac{\sigma^{12}}{r2^6} = \frac{1}{r2^6}$$

- **Loop unroll** - The team explored the technique of loop unrolling. By expanding the loop body multiple times, we aimed to reduce the overhead of loop control and improve performance. We specifically targeted loops that were small enough (k = 3) to maintain code readability while still reaping the benefits of this optimization strategy.
- **Removal of the if condition** - The removal of an *if condition* within the *for loop* of the *Potential()* function. By carefully analyzing the code, we determined that this condition was unnecessary and removing it resulted in improvement of efficiency.
- **Avoid code redundancy** - Our analysis revealed the presence of repeated parcels throughout the code. To eliminate this redundancy, we isolated them in a variable, significantly improving the efficiency of the main functions.
- **Loop Optimization** - Minimizing operations, reducing memory access, and improving cache utilization in loops improves execution times and resource efficiency. We strategically declared and assigned variables before specific inner loops, reducing assignments for improved performance. The group enhanced efficiency and performance of *Potential()*, *computeAccelerations()*, and *VelocityVerlet()*. These optimizations improved execution speed and resource utilization, resulting in more efficient code, while maintaining legibility.
- **Spatial Locality** - Optimizing for spatial locality is crucial as it can greatly impact the cache performance of the program. To improve spatial locality, we transposed the four main 2D arrays (*position*, *acceleration*, *force*, and *velocity*) and made corresponding code adaptations. This resulted in a notable reduction in cache-references and cache-misses. Additionally, the group incorporated compiler flags to facilitate automatic vectorization, further enhancing performance in this context.
- **Compiler Flags** - These flags are settings and directives used to control the compilation process in C or C++ compilers. They influence code optimization, debugging information, warning levels, and more. In our Makefile, we used the following flags:

  -pg: Enables profiling support to collect information on function calls and execution time, aiding performance analysis.

  -ftree-vectorize: Enables automatic vectorization of loops using SIMD instructions, improving performance for certain computations.

  -msse4 and **-mavx**: Enable SSE4 and AVX instructions, respectively, providing additional SIMD instructions for improved performance.

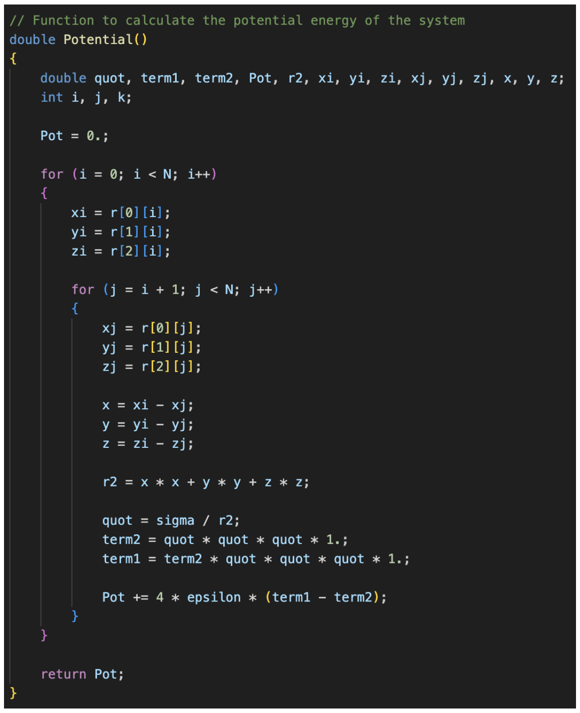
  -march=native: Generates code optimized for the machine's native architecture, utilizing specific features and capabilities of the processor.

  -mtune=native: Optimizes code for the machine's native architecture, focusing on performance on the target machine.

  -O3: Enables aggressive optimization levels, performing various optimizations to improve code performance.

C. Results

After all the optimizations, we got the following results:



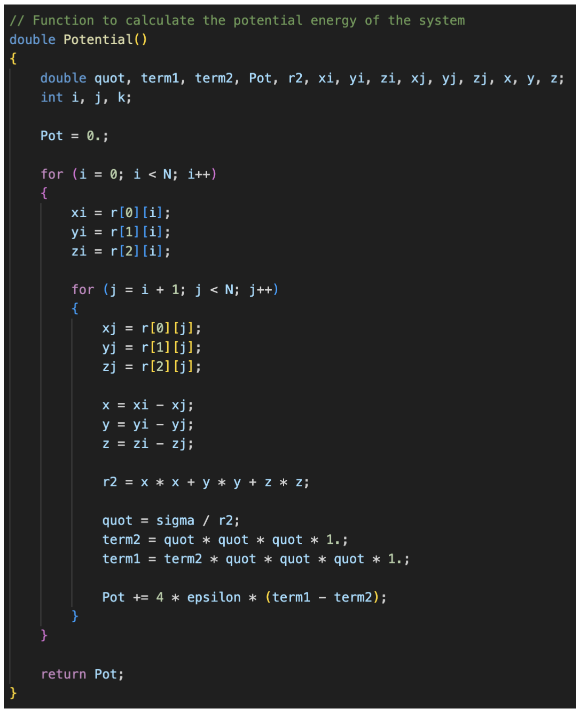
```c
// Function to calculate the potential energy of the system
double Potential()
{
    double quot, term1, term2, Pot, r2, xi, yi, zi, xj, yj, zj, x, y, z;
    int i, j, k;

    Pot = 0.;

    for (i = 0; i < N; i++)
    {
        xi = r[0][i];
        yi = r[1][i];
        zi = r[2][i];

        for (j = i + 1; j < N; j++)
        {
            xj = r[0][j];
            yj = r[1][j];
            zj = r[2][j];

            x = xi - xj;
            y = yi - yj;
            z = zi - zj;

            r2 = x * x + y * y + z * z;

            quot = sigma / r2;
            term2 = quot * quot * quot * 1.;
            term1 = term2 * quot * quot * quot * 1.;

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

    return Pot;
}
```

Fig. 3: After Optimization

III. Phase 2: Memory Parallelism (OpenMP)

A. Identify code Hot Spots

To enhance code performance, the group began by adjusting the number of particles (global variable *N*) to the required

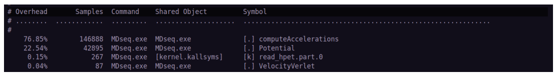


```
// Function to calculate the potential energy of the system
double Potential()
{
    double quot, term1, term2, Pot, r2, xi, yi, zi, xj, yj, zj, x, y, z;
    int i, j, k;

    Pot = 0.;

    for (i = 0; i < N; i++)
    {
        xi = r[0][i];
        yi = r[1][i];
        zi = r[2][i];

        for (j = i + 1; j < N; j++)
        {
            xj = r[0][j];
            yj = r[1][j];
            zj = r[2][j];

            x = xi - xj;
            y = yi - yj;
            z = zi - zj;

            r2 = x * x + y * y + z * z;

            quot = sigma / r2;
            term2 = quot * quot * quot * 1.;
            term1 = term2 * quot * quot * quot * 1.;

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

    return Pot;
}
```

Fig. 4: Best results obtained in the SeARCH cluster

count of 5000. Subsequently, we initiated the optimization process by identifying code blocks with the highest overhead, i.e., those consuming the most computation time.

To achieve this, we employed the profiling and tracing tool *perf* to establish a performance analysis setup, aiding us in identifying hot spots within the application. The results of this analysis are presented in the figure below, depicting the execution outcomes when using the -O2 compiler flag.



```
# Overhead       Samples  Command    Shared Object        Symbol
# ........  ............  .........  ...................  ..............................................................
#
    76.85%        146888  MDseq.exe  MDseq.exe            [.] computeAccelerations
    22.54%         42895  MDseq.exe  MDseq.exe            [.] Potential
     0.15%           267  MDseq.exe  [kernel.kallsyms]    [k] read_hpet.part.0
     0.04%            87  MDseq.exe  MDseq.exe            [.] VelocityVerlet
```

Fig. 5: Perfreport results

B. Optimizations

1) Sequential MDseq.cpp: After a comprehensive code review, the team identified an opportunity for improvement overlooked in the initial project phase. This enhancement involved consolidating two functions, *Potential()* and *computeAccelerations()*, into a unified function. Formerly handling the computation of total potential energy and forces separately, the merger not only enhanced code readability but also delivered a performance boost.

To accommodate this change, a new global variable, *Pot*, was introduced, necessitating adjustments to the main function to avoid errors.

Both functions, dealing with particle interactions and featuring an overlap in computations, were merged to eliminate redundancy, resulting in a more concise and efficient codebase.

The unified function not only streamlines maintenance but also adheres to software development best practices, fostering a cleaner, modular code structure. This strategic consolidation reflects our commitment to code optimization, anticipating positive impacts on both readability and potential runtime efficiency.

Besides this major change, a compiler flag, -Wall, was introduced to check for all warnings, leading to minor code adjustments, including resolving inconsistencies and removing unused variables.

2) Exploring parallelism with OpenMP in MDpar.cpp: To develop the parallel version of the application, we analyzed the previously mentioned sequential code implementation to identify performance bottlenecks in terms of execution time. Afterwards, we took this sequential implementation and introduced *OpenMP* directives and efficiently distributed the computational load among various parallel threads.

Our primary focus was on implementing the *OpenMP* directives within the loop cycles of the *computeAccelerations()* function, as this is where the most significant computational load is concentrated.

```
#pragma omp parallel for reduction(+:Pot) reduction
    (+:a[:MAXPART][:3]) private(j, rSqd, rSqd4,
    rSqd7, rij, f) schedule(dynamic, 40)
for (i = 0; i < N-1; i++)
{
    for (j = i + 1; j < N; j++)
    {
        rSqd = 0.;
        // Calculate the position of atom i relative
     to atom j
        rij[0] = r[i][0] - r[j][0];
        (...)
```

The group strategically positioned the *pragma* directive at this precise location due to its alignment with the peak computational demands of the code. A meticulous examination of this code segment led to the determination that only the outer loop possesses the capacity for seamless parallel execution.

By using the *pragma*, it was possible to parallelize the loop iterations where the computation of forces and accelerations between pairs of particles takes place. This section is a computationally intensive part of the code, and parallelization can lead to significant performance gains. The reduction clauses ensure that the shared variables *Pot* and *a* are correctly updated across all threads preventing data races.

We did not parallelize the inner loop because the calculation of particle accelerations depends on the results of the previous iterations. Specifically, the acceleration calculation for particle *i* depends on the positions and distances involving all particles with indices greater than *i*. Since these calculations have data dependencies, using the parallelism in this section, would require careful synchronization mechanisms and could potentially introduce race conditions, making it more complex and error-prone.

Reasons why we apllied the pragmas:

- **#pragma omp parallel for:** This directive instructs the compiler to immediately parallelize the for loop.

It implies that iterations of the loop can be executed concurrently by multiple threads.

- **reduction(+:Pot):** We used this clause since each thread will have its local copy of *Pot*, and at the end of the parallel region, these local copies will be combined into a single global value. This results in the reduction of the *Pot* variable in an addition context.
- **reduction(+:a[:MAXPART][:3]):** Similar to the previous clause, this one specifies a reduction for the acceleration array *a*. Its notation signifies that the reduction is applied to each element of the two-dimensional array. It's important to note that each element of the matrix is treated as an independent variable, and the local copies from each thread are combined at the end of the loop.
- **private(j, rSqd, rSqd4, rSqd7, rij, f):** We put this variables at private for each thread, so they could have their own local copies for each thread, preventing data races.
- **schedule(dynamic, 40):** We used dynamic scheduling because it allows the loop iterations to be distributed dynamically among the available threads. The parameter 40 indicates the number of loop iterations assigned to each thread at a time.

C. Results

To analyze the behavior of the code under different thread configurations, we executed the batch file *./script.sh* on the *SeARCH cluster*. The table below presents the execution times for the parallel and the speed up values for the different number of threads. Two graphs depict Scalability (Speedup vs Threads) and Efficiency (Efficiency vs Threads).

The two following graphs were obtained by running the MDpar.exe in the *SeARCH cluster*.

We also ran our sequential code on the *SeARCH cluster* a indefinite number of times. For reference, its best time was 48.914 seconds.

Number of Threads	Execution Time	
	Parallel(s)	SpeedUp(s)
4	12.222s	1.000
8	6.215s	1.9665
12	4.205s	2.9065
14	3.630s	3.3669
18	2.859s	4.2749
22	2.635s	4.6383
24	2.613s	4.6774
26	2.626s	4.6542
30	2.558s	4.7780
34	2.545s	4.8024
36	2.534s	4.8232
40	2.469s	4.9502

It was observed that from a specific juncture onward, augmenting the number of threads in the *OpenMP* parallelization does not yield a substantial improvement in speedup, as corroborated by the Efficiency Analysis below.

D. Reflections on the SeARCH cluster

During the conception of this project we applied the chosen parallelization approach by incorporating *OpenMP* directives. The code underwent an optimization specifically to the

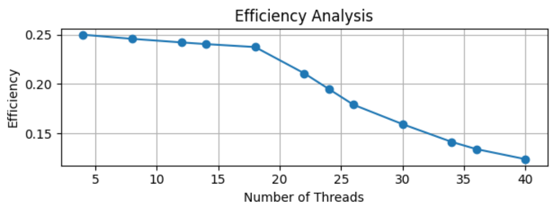


Fig. 6: Efficiency Analysis

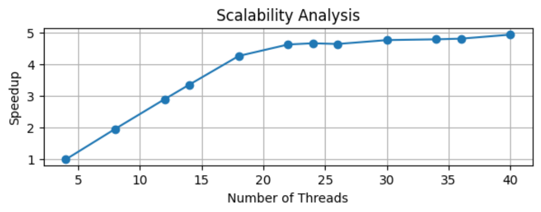


Fig. 7: Scalability Analysis

SeARCH cluster, taking into account the architectural nuances and the resources available on the compute node within the *cpar* queue.

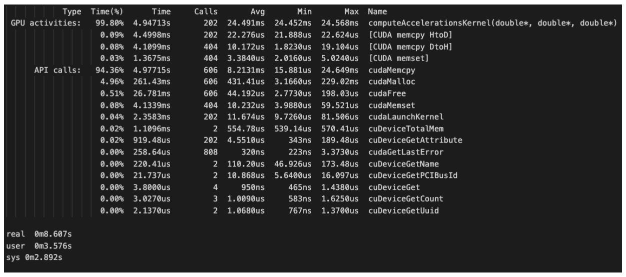
IV. Phase 3: CUDA

As previously mentioned, we decided to use the *CUDA* as our approach, to accomplish this work. *CUDA* is an *API* designed for parallel computing, capable of using *GPU*, and heterogeneous computing, created by *NVIDIA*, highly optimized parallelization data processing.

A. Analysis

Resuming from the previous project, the team has committed to further enhance the recently developed *computeAccelerations()* function — essentially the amalgamation part of the functions *Potential()* and *computeAccelerations()*. Significant emphasis has been dedicated to optimizing the utilization of blocks, threads per block, and memory access to achieve efficient parallelization. This meticulous approach is designed to ensure the seamless integration of these components, promoting enhanced performance and efficiency.

With the use of *nvprof* (Nvidia Profiler) and *nvvp* (Nvidia Visual Profiler), we analysed our profiling metrics to see how the *API* calls to *CUDA* were handled and how much overhead might have been introduced by this method. Due to the missing *GUI* in the *SeARCH Cluster*, needed for the use of *nvvp*, we will be showing the profilings from the Laptop test:



```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.80%  4.94713s       202  24.491ms  24.452ms  24.568ms  computeAccelerationsKernel(double*, double*, double*)
                    0.09%  4.4998ms       202  22.276us  21.888us  22.624us  [CUDA memcpy HtoD]
                    0.08%  4.1099ms       404  10.172us  1.8230us  19.104us  [CUDA memcpy DtoH]
                    0.03%  1.3675ms       404  3.3840us  2.0160us  5.0240us  [CUDA memset]
      API calls:   94.36%  4.97715s       606  8.2131ms  15.881us  24.649ms  cudaMemcpy
                    4.96%  261.43ms       606  431.41us  3.1660us  229.02ms  cudaMalloc
                    0.51%  26.781ms       606  44.192us  2.7730us  198.03us  cudaFree
                    0.08%  4.1339ms       404  10.232us  3.9880us  59.521us  cudaMemset
                    0.04%  2.3583ms       202  11.674us  9.7260us  81.506us  cudaLaunchKernel
                    0.02%  1.1096ms         2  554.78us  539.14us  570.41us  cuDeviceTotalMem
                    0.02%  919.48us       202  4.5510us     343ns  189.48us  cuDeviceGetAttribute
                    0.00%  258.64us       808     320ns     223ns  3.3730us  cudaGetLastError
                    0.00%  220.41us         2  110.20us  46.926us  173.48us  cuDeviceGetName
                    0.00%  21.737us         2  10.868us  5.6400us  16.097us  cuDeviceGetPCIBusId
                    0.00%  3.8000us         4     950ns     465ns  1.4380us  cuDeviceGet
                    0.00%  3.0270us         3  1.0090us     583ns  1.6250us  cuDeviceGetCount
                    0.00%  2.1370us         2  1.0680us     767ns  1.3700us  cuDeviceGetUuid

real  0m8.607s
user  0m3.576s
sys 0m2.892s
```

Fig. 8: Profiling using Nvidia Visual Profiler

B. Design and Implementation

To start, the first thing we need to figure out is the best number of blocks and threads for each block. We used a formula to ensure we consistently get an effective number of blocks: `NUM_BLOCKS = (N + NUM_THREADS_PER_BLOCK - 1) / NUM_THREADS_PER_BLOCK`. This way, it works smoothly even when N is not a perfect multiple `NUM_THREADS_PER_BLOCK`, so the situation is handled.

Having established this foundation, we proceeded to integrate *CUDA* into our *computeAccelerations()* function. The initial step involved reorganizing its structure. The group started by allocating memory on the *GPU* for the acceleration (da), position (dr), and potential energy (dPot) parameters. Simultaneously, the particle positions (r) were transferred from the host to the *GPU*.

Subsequently, the kernel (*computeAccelerationsKernel*) was invoked with specified values for `NUM_BLOCKS` and `NUM_THREADS_PER_BLOCK`, utilizing the previously mentioned formula to determine the optimal number of blocks. This strategic approach ensures a well-defined parallelization setup for the computational task at hand.

Inside the kernel, the outer loop was removed so we could use the thread ID (`i = blockIdx.x * blockDim.x + threadIdx.x`) instead. This way, each thread will take care of the calculations of the position of atom i relative to atom j and so on.

Initially we added six *atomicAddDouble* operations to the loop (specified by 'for (j = i + 1; j ¡ 5000; j++)'). We added the *atomicAddDouble()* function. This function ensures that the addition operation on a double-precision variable at the specified memory location is performed atomically, preventing race conditions in parallel execution environments.

To enhance efficiency and minimize array accesses, the code was optimized, employing sum_acc0, sum_acc1, and `sum_acc2` variables to accumulate contributions for each dimension inside the loop j. This adjustment enables the *atomicAddDouble* operations for the accumulated values to be performed outside the loop j. Consequently, the overall number of atomic operations and memory accesses is reduced, significantly optimizing the parallelized acceleration calculation.

In this parallelized implementation, each thread keeps its own local variable (`localPot[threadIdx.x]`) in shared memory to gather intermediate results during the potential calculations for each pair of particles processed within the loop j. Following the completion of the loop, a parallel reduction operation takes place within the block. The values accumulated by individual threads are combined, and the final result is stored in `localPot[0]`. Subsequently, the result of this reduction operation, symbolizing the overall potential, gets stored in the Pot array on a per-block basis (`Pot[blockIdx.x] = localPot[0]`). This assignment effectively streamlines the potential calculation into a per-block operation, thereby improving computational efficiency through the consolidation of results at the block level.

In the end, the acceleration and *Pot* arrays (*localPotBlock*) are copied from the *GPU* to the host. And then it runs cudaFree to free up space from the *GPU*. In the end, there's a final calculation done by the *CPU*, and the function is done.

C. Testing and Results

For the scalability assessment, our group opted to conduct a series of tests using three distinct input sizes. We began with the initially specified number of particles (N) at 5000 and repeated the test for N = 2160. To thoroughly examine scalability, we introduced two additional input sizes: N = 10000 and N = 15000. Each input configuration underwent testing with varying thread counts, specifically 8, 16, and 32 threads. The outcome of these tests is presented in Figure 8, offering a comprehensive overview of the scalability trends observed across different particle counts and thread allocations.

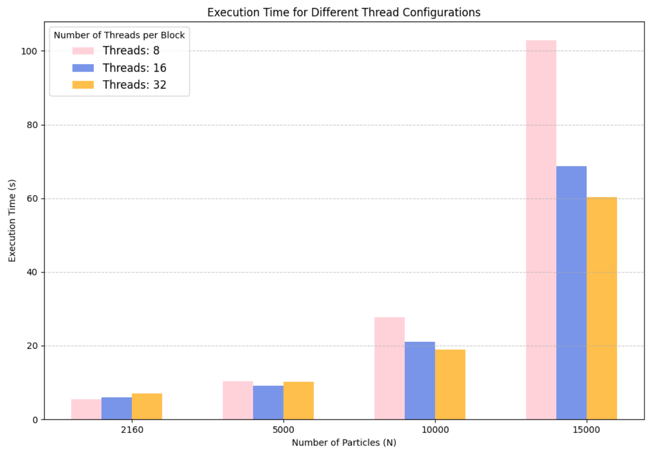


Fig. 9: Number of Threads and Execution Times for different number of particles

Upon scrutinizing the figure, a clear trend emerges: for the targeted particle size of N = 5000, the optimal number of threads per block appears to be 16. It is important to remark that the execution time for this number of threads was 8,607s.

Shifting focus to N = 2160, our group identified that a more conservative approach, utilizing 8 threads per block, yielded optimal results. Extending our analysis to larger particle sizes, specifically N = 10000 and N = 15000, the pattern indicates an advantageous configuration with 32 threads per block. This recurrent pattern suggests that, for particle counts exceeding 15000, an even higher thread count, possibly 64 threads, might yield superior performance.

A noteworthy observation our group acknowledges is the importance of adhering to multiples of 32 when determining the optimal thread block size for *CUDA*. Given that *CUDA* kernels issue instructions in warps (sets of 32 threads), aligning with this multiple ensures efficient execution. Additionally, it's crucial to recognize that excessively small thread block sizes may underutilize the *GPU*, as managing numerous small blocks incurs administrative overhead that could outweigh the benefits of parallelization.

Furthermore, shared memory usage plays a pivotal role. Particularly for operations like reductions, an overconsumption of shared memory may restrict the number of concurrently running thread blocks on the *GPU*. This limitation emphasizes the need for a thoughtful and adaptable approach to thread configuration. By considering both the nature of the workload and *GPU* constraints, optimal performance can be achieved across a diverse array of computational scenarios.

The group initially anticipated that *CUDA* parallelization would yield a more significant improvement in execution time compared to *OpenMP*. However, upon examining the results, we encountered unexpected outcomes. It became evident that the *CPU* within the *SeARCH Cluster* demonstrated remarkable power, while the *GPU* didn't exhibit comparable performance. As a result, the potential superiority of *CUDA* over *OpenMP* was hampered by the limitations of the cluster's *GPU*.

Adding to the complexity, the group faced constraints in conducting experiments beyond the *SeARCH Cluster* due to the absence of discrete *GPU*s among its members. The absence of such hardware prevented us from exploring performance variations on different *GPU* architectures. Given the opportunity to experiment on systems with discrete *GPU*s, we anticipate that *CUDA* and *OpenMP* results would likely show more comparable performance, offering a clearer perspective on the relative advantages of each parallelization approach.

V. Conclusion

Overall, the project and its three distinct phases, each with an unique and specific purpose, allowed us to learn new skills and deepen our knowledge in the parallel computing field.

We learned advanced optimization techniques, utilized sophisticated performance measurement tools, and explored parallelization at both *CPU* (with *OpenMP*) and *GPU* (with *CUDA*) levels.

This experience enhanced our understanding of code efficiency and provided valuable skills in tackling complex problems.

Overall, we enjoyed the challenges and gained significant insights throughout the project.