Parallel Computing
Simple Molecular Dynamics

Gustavo Pereira
Informatics Department
University of Minho
Braga, Portugal
pg53723@alunos.uminho.pt

Nelson Almeida
Informatics Department
University of Minho
Braga, Portugal
pg52697@alunos.uminho.pt

***Abstract*—This paper presents an analysis and optimization program aimed at enhancing the performance of a simple molecular dynamics simulation code designed for the study of Argon gas atoms. The original code, FoleyLab/MolecularDynamics: Simple Molecular Dynamics, employs a generic approach to simulate the dynamic behavior of particles over discrete time steps, adhering to Newton's laws of motion. The given code lacks efficiency due to its exhaustive loops that result in a heavy computational load leading to substantial execution times. In this paper, we will analyze the code and apply optimizations utilizing CUDA programming following the previous assignment enhancements.**

I. Single-Threaded Program Optimization

A. Problem Statement

A Molecular Dynamics simulation presents itself to be a computationally intensive task due to the number of calculations required for modeling particle interactions, force computations, time integration, large data handling, accuracy needs, simulation size, and boundary conditions. For the first phase of this work assignment, we aimed for optimizations based on concepts such as ILP (Instruction Level Parallelism) and Memory Hierarchy in a single-thread context. Our initial approach involved the implementation of fundamental high-level optimizations, being the groundwork for subsequent enhancements through techniques such as Loop Unrolling and Vectorization. Since the input value was constant, our performance assessments were centered around key metrics, including execution time, cycle count, and the CPI (Cycles per Instruction) metric.

B. Methodology

Upon analyzing the single-threaded program using profiling tools like gprof and gprof2dot (results from this snapshot can be seen in figure X), we found out that the most called functions were Potential, called 201 times occupying 64.29% of the execution time, and computeAccelerations, called 202 times spanning over 35.63% of the execution time. These two together use 99.92% of the execution time, making it obvious that these were the hot spots that should be optimized.

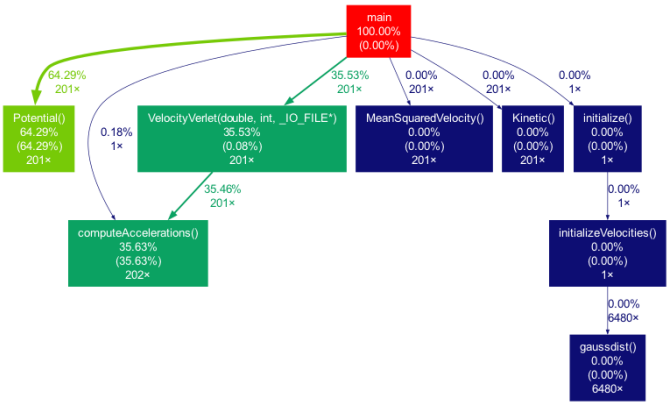


Fig. 1. Call graph for the source code

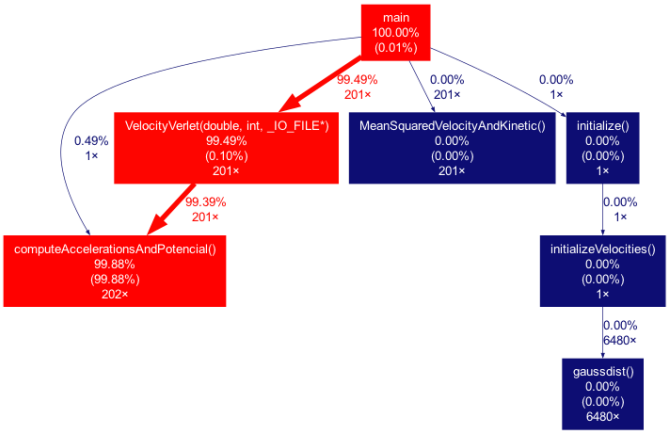


Fig. 2. Call graph for the optimized code

Taking a closer look, we can see that the mentioned functions consist of nested for loops with fixed $N * N$ iterations that, given our fixed N value of 2,160, result in 4,320 iterations per call, giving us a total of 1,740,960 iterations in total. In the case of *computeAccelerations()*, we have an extra loop to set the accelerations matrix to zero which gives us plus N*3 iterations, and inside the N*N loop, we have two other for loops that run 3 times each.

Inside the main loops, one can find mathematical operations for the formulas that mimic the properties of our Molecular Dynamics simulation. Translating those formulas into code

leads to the overuse of multiplication, division, exponentiation, etc. These calculations can lead to a greater number of cycles since a division takes 36 more cycles than a simple addition operation instruction which takes up to 3 cycles. With that in mind, we simplified the mathematical formulas to reduce the number of division or exponentiation operations.

Given the intensive use of costly mathematical operations and nested for loops, as seen previously, our first optimization was reducing the number of heavy mathematical operators by reformulating mathematical equations, adopting new strategies to reduce the number of repetitive loop iterations, and merging functions that use the same type of loops, which was the case for *Potencial()* and *ComputeAccelerations()* functions. (the before and after values can be seen in the table below)

Besides these, other optimization techniques can be applied to decrease execution time, such as Loop Unrolling and Vectorization. These take advantage of concepts such as ILP (Instruction Level Parallelism) and Memory Hierarchy and can be employed by the gcc compiler when specific flags are used. For these to work we had to facilitate our code, which meant turning our matrices into vectors and converting our nested loops into N*3 loops.

C. Results and Evaluation

These optimizations resulted in a large decrease in the total number of cycles (since the cycle-consuming operations were reduced) and a reduction in the total number of instructions thanks to our merged functions.

TABLE I
CYCLE AND INSTRUCTION METRICS COMPARISON

Total	Before	After
Instructions	1,236,991,944,534	260,326,290,217
Cycles	766,640,483,170	163,770,254,165

Furthermore, we explored gcc compiler flags such as -O2, -O3, combined with the "-funroll-loops" and "-ffast-math" flags. These optimizations allowed us the utilization of loop unrolling methods within the code and a compiler-driven math optimization. To understand which optimizations presented the best performance, we collected the execution times associated with each optimization. These tests were conducted using the source code on the cluster.

As expected, the execution time exhibits a decrease with the escalation of optimization levels, with the best performance achieved using the -O3 -funroll-loops -ffast-math flags combined.

TABLE II
PERFORMANCE COUNTER COMPARISON

Metric	**Source**	**Optimized**
Cache misses	28473957	11795
Instructions	46255998416	32537845062
Cycles	782545448209	19379819430
Time Elapsed	269.721383703 seconds	6.015 seconds

It's worth noting that our study delved into the impact of specific compiler flags, including -O3, -ffast-math, -ftree-vectorize, -mavx, -mfpmath=sse, and -march=x86-64. The judicious application of these flags was pivotal in achieving optimal performance. Among these, the -O3 flag, when complemented by the -ffast-math and -ftree-vectorize flags, demonstrated exceptional results.

II. SHARED MEMORY PARALLELISM (OPENMP)

A. Methodology

To optimize the performance of our program, we targeted the most time-intensive function, `computeAccelerationsAndPotential`, which consumed 99.88% of the overall execution time, through performance analysis tools such as `perf report` and insights gained from prior assignments.

```
Samples: 29K of event 'cpu_core/cycles:Pu/', Event count (approx.): 31147308251
Overhead  Command     Shared Object          Symbol
  99,88%  MDseq.exe   MDseq.exe              [.] computeAccelerationsAndPotencial()
```

Fig. 3. Execution Overhead Analysis

Upon examining the sequential code, we identified a potential optimization concerning the nested loops. Specifically, we eliminated conditional statements within these loops, transforming the loop complexity from $N \times 3 \times N \times 3$ to $N \times 3 \times N$.

However, the introduction of parallelization presented a critical issue related to potential data races. Simultaneous access and modification of the same array `a` by multiple threads could yield incorrect results. To rectify this, we devised a solution using an auxiliary matrix, `aTemp`, where each thread accesses its own array, mitigating potential data races. For this, we initialized the matrix with `malloc`, leveraging the default value of 0 as the neutral element for operations with this auxiliary array, eliminating the need for additional loops to set these values to 0.

```
int numThreads = omp_get_max_threads();
double** aTemp;
aTemp=(double**)malloc(numThreads * sizeof(double*));
for (i = 0; i < numThreads; i++) {
    aTemp[i]=(double*)calloc(N * 3, sizeof(double));
}
```

This parallelized section addressed data race concerns but introduced a new challenge: aggregating values from the auxiliary array into the original array of accelerations (`a`). The order of these floating-point operations mattered, necessitating an additional nested loop, this time involving the number of threads (`numThreads`) multiplied by `numThreads * N * 3`.

To resolve the data race issue with the variable `PEA`, altered by each thread, we employed the OpenMP primitive reduction, ensuring a swift and efficient solution to the problem.

Regarding thread execution scheduling, considering options of static, dynamic, or guided scheduling, and acknowledging the higher overhead of dynamic and guided scheduling, we tested their use with a focus on overall performance. Consequently, we opted for dynamic scheduling.

```
#pragma omp parallel num_threads(numThreads)\
private(privateVars)
{
    threadNum = omp_get_thread_num();
    #pragma omp for reduction(+:PEA)\
    schedule(dynamic, numThreads)
    for (i = 0; i < N * 3 - 4; i += 3) {
        for (j = i + 3; j < N * 3; j += 3) {
            // auxiliary computing

            // storing the values
            // in the auxiliary matrix
            aTemp[threadNum][i] += tempx * f;
            aTemp[threadNum][j] -= tempx * f;
        }
    }
}
```

B. Performance improvements

After the implementation, we used Amdahl's law to get an expected speedup for our program. The speedup according to Amdahl's Law is given by the following formula:

$$\text{Speedup} \leq \frac{1}{(1-P) + \frac{P}{S}} \tag{1}$$

Since we only parallelized the hot spot which took 99.88% of the overall execution time, replacing the P in the formula above we get the following equation:

$$\text{Speedup} \leq \frac{1}{(1-0.9988) + \frac{0.9988}{S}} \tag{2}$$

This gives us the expected speedup concerning the number of running threads.

TABLE III
PERFORMANCE METRICS FOR DIFFERENT NUMBERS OF THREADS

Num. Threads	Expected Speedup
1	1
2	1.997
4	3.985
8	7.933
10	9.893
16	15.717
20	19.554
32	30.852
40	38.211

It's clear that the expected speedup is different from linear since we aren't parallelizing the complete program but only the targeted hotspot.

C. Results and Evaluation

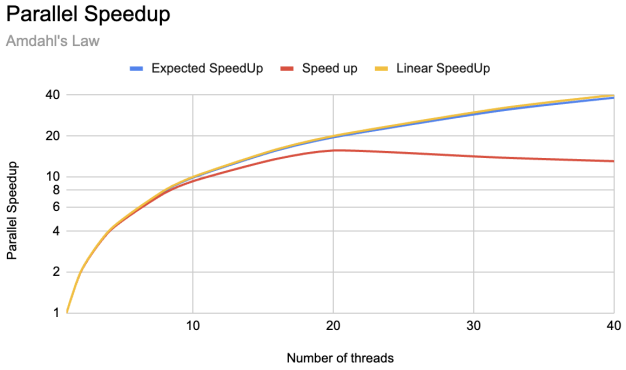
Having a theoretical basis of what the expected speedup is, we started testing the parallel implementation for the same number of threads to compare whether it matches the expected value.

We calculated our speedup by dividing the baseline which was our sequential implementation by our parallel time.

The data above can be better visualized in the following graph.

TABLE IV
PERFORMANCE METRICS MEASURED FOR DIFFERENT NUMBERS OF THREADS

Num. Threads	Avg. execution time	Speed Up
1	35,071	1,00
2	17,633	1,99
4	8,902	3,94
8	4,595	7,63
10	3,767	9,31
16	2,577	13,61
20	2,240	15,65
32	2,528	13,87
40	2,684	13,06



Analyzing the graph, it is evident that up to 20 threads, the speedup aligns well with the expected values. However, beyond this point, a significant loss in performance is observed. Considering that the cluster used for testing has 20 *Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz*, which supports multithreading, one would expect a continuous increase in speedup. Several factors contribute to this loss:

- **NUMA Architecture:** The CPU has NUMA nodes, and understanding the node distribution is crucial for optimizing parallel code. In our case, NUMA node0 includes CPU(s) 0-9 and 20-29, while NUMA node1 includes CPU(s) 10-19 and 30-39.
- **Thread Overhead:** Beyond a certain point, the overhead introduced by managing a higher number of threads may offset the gains from parallelization.
- **Memory Access Patterns:** The efficiency of parallelization can be affected by memory access patterns, especially in a NUMA architecture. Understanding and optimizing for these patterns is essential.

Further investigations into these aspects could provide insights into optimizing the parallelization strategy for better scalability.

III. GPU Acceleration with CUDA

A. Problem Statement

While OpenMP directives allow for parallelism and can augment the performance of computations on CPUs, GPUs equipped with CUDA (Compute Unified Device Architecture) provide a distinct advantage in terms of processing power and parallelism. This specialized architecture is tailored for handling highly parallel tasks, making it exceptionally well-suited for computationally intensive applications like molecular dynamics simulations.

Exploring GPU acceleration via CUDA offers the potential to unlock an extensive level of parallelism within these simulations. GPUs are engineered with numerous cores optimized for parallel computation, enabling them to execute a vast number of computations concurrently. Leveraging CUDA allows for the exploitation of this massive parallel processing capability, thereby promising substantial performance gains compared to traditional CPU-based multi-threaded computations like the previous implementation.

B. Methodology

Building upon our optimization efforts in the preceding phases, our primary focus shifted towards identifying and parallelizing critical sections of code using CUDA. The central element for parallelization was the *computeAccelerationsAndPotential()* method, chosen due to its significance in executing major computations during runtime.

To facilitate effective parallelization, we decided to revert the vector conversion to the original matrices format. This alteration was crucial as it empowered each particle to independently calculate its x, y, and z coordinates. This step was strategically undertaken to address and eliminate data races that could potentially occur during the execution of the *computeAccelerationsAndPotential* algorithm.

CUDA Kernel(`computeAccelerationsPotential`)
The CUDA kernel is responsible for computing accelerations and potentials between particles in a molecular dynamics simulation was a focal point of our methodological approach. Breaking down its key components:

- **Shared Memory (`sharedRk`):** Declared with `__shared__`, this shared memory array is dedicated to caching particle positions (`rk`) within each block. Its purpose is to mitigate global memory access latency.
- Shared Memory Initialization (`sharedRk[threadIdx.x][k] = rk_[i][k]`): During this step, each thread loads a portion of the positions array `rk` into shared memory. This preloading facilitates faster access to particle positions during subsequent calculations.
- **Calculating Relative Positions (`rij`):** The computation involves determining the x, y, and z components of relative positions between the current thread's particle and others in the system.
- **Loop Over Other Particles (`for (int j = 0; j < hN; j++)`:)** The outer loop iterates over all particles in the system, with each thread calculating relative positions using cached data from shared memory.
- **Force Calculation and Potential Update:** This step includes the computation of force and potential between particles based on relative positions, employing Lennard-Jones potential calculations. The results are stored in local variables (`vPot_local` and `ak_local`) specific to each thread.
- **Storing Results in Global Memory (`Pot[i]`, `ak[i][0]`, `ak[i][1]`, `ak[i][2]`):** The computed results are stored in global memory arrays (`Pot` and `ak`) for subsequent utilization.
- **Shared vs. Global Memory:** Shared memory (`sharedRk`) optimization is directed towards enhancing access to particle positions within a block. However, potentials (`vPot_local`) are not shared due to the pairwise nature of Lennard-Jones potential calculations across the entire system. Sharing potentials would introduce synchronization and communication overhead, diminishing efficiency.

C. Results and Evaluation

TABLE V
Normalized Performance Metrics for Different Particle Quantities and Thread Configurations with GPU Acceleration

Num. Particles	Avg. exec. time (128 Threads)	Avg. exec. time (256 Threads)	Avg. exec. time (512 Threads)
5000	1.272	2.564	2.976
10000	3.153	3.586	3.5642
15000	6.945	7.025	6.786

TABLE VI
Performance Metrics Measured for Different Particle Quantities: OpenMP vs. CUDA

Num. Particles	OpenMP (sec)	CUDA (sec)
5000	3.2	2.5245
10000	8.692	4.7488
15000	17.12	9.469

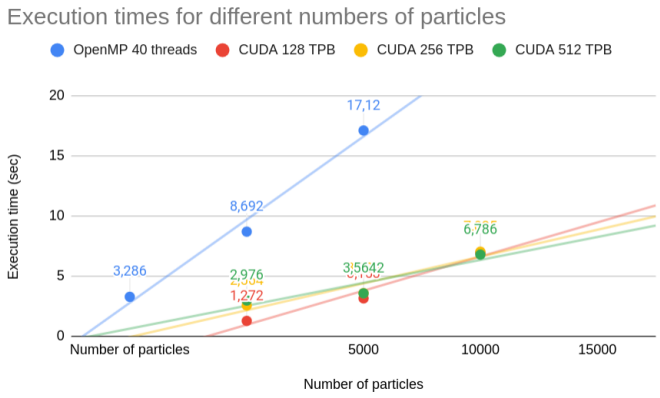


Fig. 4. Execution times for different particle numbers

IV. Result Analysis

A. Small Particle Quantities

For simulations involving a low number of particles, the benefits derived from GPU acceleration exhibit limitations. The granularity of parallelization achievable with GPUs may not align optimally with the computational demands of a smaller particle count. Moreover, the overhead incurred during data transfers between the CPU and GPU tends to offset the potential computational gains, leading to a diminished speedup. In such scenarios, the advantages of GPU parallelism may not be fully realized, and alternative optimization strategies or hardware configurations may need consideration.

B. Larger Particle Quantities

Contrastingly, the advantages of GPU acceleration become more pronounced as the number of particles in the simulation increases. The inherent parallel processing capabilities of GPUs efficiently manage the substantial arithmetic computations inherent in molecular dynamics simulations. The scalability of GPU parallelism becomes evident in this context, leading to a notable improvement in overall execution time compared to the OpenMP-parallelized version. The optimized handling of parallel tasks on the GPU allows for a more effective distribution of computational workloads, resulting in a substantial speedup for simulations involving larger particle quantities.

The observed trends underline the importance of tailoring parallelization strategies to the specific characteristics of the computational workload. While GPU acceleration excels in scenarios with higher computational demands, careful consideration is required when applying it to simulations with smaller particle counts, where alternative optimization techniques may yield better results.

V. Overall Conclusion

Our comprehensive project involved a systematic optimization of a Molecular Dynamics simulation code to significantly enhance its performance using a multi-pronged approach: single-threaded optimization, shared-memory parallelism with OpenMP, and GPU acceleration with CUDA.

A. Optimization Cascade

Starting from the improvement of the single-threaded program given, focusing on fundamental high-level optimizations like Instruction Level Parallelism (ILP) and Memory Hierarchy considerations, we laid the foundation for subsequent enhancements employing techniques like Loop Unrolling and Vectorization.

Subsequently, we introduced shared-memory parallelism via OpenMP, concentrating on the critical `computeAccelerationsAndPotential()` function. Loop transformations and parallelization were employed, addressing data race challenges through innovative solutions like an auxiliary matrix (`aTemp`) and OpenMP `reduction` directives.

GPU acceleration with CUDA revolutionized our efforts, capitalizing on GPUs' unparalleled parallel processing capabilities. The CUDA kernel for `computeAccelerationsAndPotential()` showcased the potential for massive parallelism. However, we encountered some roanblocks, primarily regarding data transfer overhead between the CPU and GPU, highlighting the complexities of heterogeneous computing architectures.

B. Challenges, Insights, and Limitations

Along the way, we encountered and overcame a multitude of challenges. Data races, a common pitfall in parallel programming, demanded creative solutions, leading to the development of novel strategies like auxiliary matrices and reduction directives. Our scalability analysis revealed intriguing insights into the impact of NUMA architecture, thread overhead, and memory access patterns, emphasizing the importance of understanding these factors for optimal parallelization.

While OpenMP parallelization exhibited commendable scalability up to 20 threads, diminishing returns suggested the need for further architectural exploration. GPU acceleration delivered remarkable performance gains for larger datasets, underscoring its potential, albeit with limitations for smaller datasets due to CPU-GPU communication overhead.

C. Academic Contributions and Recommendations

We believe that our project makes a relevant academic contribution by providing a framework for optimization methodologies and parallelization strategies in the context of Molecular Dynamics simulations. The findings offer valuable insights into the trade-offs, challenges, and potential solutions associated with optimizing scientific computing codes for modern architectures that can be generalized for other purposes where there's a need to compute a large dataset.

For future research, we would recommend exploring architecture-specific optimizations, delving into advanced CUDA features, and pursuing hybrid approaches that leverage the strengths of both CPU and GPU. These avenues hold the promise of unlocking additional layers of performance enhancement and scalability in scientific computing applications.

D. Conclusion and Future Prospects

We are satisfied with the achieved results in all three work assignments aimed at optimizing a molecular dynamics simulation code. Beginning with sequential optimization techniques and progressing through shared memory parallelism using OpenMP, and later exploring GPU acceleration with CUDA, each phase contributed incrementally to enhancing the computational efficiency of the codebase.

The results obtained across all three phases signify tangible advancements in reducing execution times, paving the way for more efficient simulations. This comprehensive optimization endeavor has not only enriched our understanding but also emphasized the significance of diverse optimization techniques in enhancing computational performance.

This way, we think that there are still improvements and optimizations to be explored in every part of the assignments, that could lead us to better results, but overall, the study has provided valuable insights into the realm of computational optimization, emphasizing the pragmatic value of parallel computing paradigms and specialized hardware in advancing scientific simulations.