Vectorising Molecular Dynamics (MD) Simulation

Afonso Franco
Universidade do Minho
Braga, Portugal
pg53595@alunos.uminho.pt

Alice Teixeira
Universidade do Minho
Braga, Portugal
pg52670@alunos.uminho.pt

***Abstract*— The virtual simulation of physics can often times be a time-consuming, resource-heavy procedure. In this paper, we utilize AVX instructions in order to vectorize a molecular dynamics simulation, evaluating the quality of the results obtained.**

***Index terms*—optimization, programming, strenght-reduction, loop-invariant code motion, molecular dynamics**

I. Introduction

The journey of our project in the realm of computational simulations has reached its pivotal Phase 3. This phase represents the culmination of our collective efforts and the application of advanced computational techniques to enhance the efficiency and performance of our simulations. The primary objective of Phase 3 is to further improve the efficiency of our code.

Building on the groundwork laid in the previous phases, where we focused on optimizing and parallelizing the code, Phase 3 delves into the realm of further optimizing computational algorithms. Our goal is to leverage modern processing capabilities to handle complex simulations more effectively and efficiently. This involves a strategic re-evaluation and restructuring of our computational approach, aiming for a more streamlined and performance-oriented design.

Throughout this phase, we are committed to investigating the impact of these enhancements on the overall performance of our simulations. The report will provide insights into the methodologies we adopted, the challenges we faced, and the results we achieved.

II. Previous work

A. Phase 1

In the first phase of this project , we seeked to optimize the single threaded MDS code.

Before optimization, the code's performance was measured using various compiler flags, resulting in a significant reduction in execution time. The optimization techniques used in the paper are described in detail, focusing on minimizing repetitive calculations and reducing arithmetic operations, such as replacing expensive functions like 'pow' and 'sqrt' with more efficient operations.

The results of the optimizations are presented, showing a remarkable improvement in performance, with a 6500% decrease in execution time and significant reductions in instructions and cycles.

B. Phase 2

Before parallelization, the paper establishes a baseline for the code's performance, which takes 28.9 seconds for a simulation with 5000 particles. Various profiling tools are used to identify code hotspots.

The paper then discusses the implementation of parallelization using OpenMP, specifically focusing on optimizing the "PotentialAndAcceleration" function. Theoretical speedup calculations based on Amdahl's law are presented, showing that the ideal speedup should be directly proportional to the number of threads used, up to a maximum of 40x.

Practical results obtained by running the parallelized code with varying thread counts are presented in a table and graph. The analysis shows that the realistic speedup stabilizes after 24 threads, indicating memory bandwidth limitations as the primary factor. The paper concludes that while parallelization significantly improves efficiency, memory bandwidth constraints must be considered when implementing parallel computing solutions.

In summary, the paper successfully parallelizes the molecular dynamics simulation, achieving substantial speedup. However, it also highlights the importance of addressing memory bandwidth limitations when optimizing resource-intensive scientific simulations through parallelization.

III. Attempt at CUDA Implementation

This chapter addresses our efforts to implement the molecular dynamics simulation using CUDA, NVIDIA's parallel computing platform. The primary motivation for this approach was to leverage GPU computing for enhanced computational performance.

The initial phase of the CUDA implementation involved translating key computational routines into CUDA kernels, suitable for execution on NVIDIA GPUs. This process was technically challenging, given the complexity of parallel programming and the specific requirements of GPU architectures. However, the initial implementation was successful in terms of execution.

Bellow we will talk about the different aproaches/ideas we had while trying to implement this simulation in CUDA efficiently.

A. Block's shared memory

An approach we contemplated in our CUDA implementation was the use of shared memory within each block to cache the acceleration values. Shared memory in CUDA is a faster type of memory accessible by all threads in a block and offers an efficient way to share data between these threads. The idea was to use this shared memory to temporarily store and accumulate the acceleration values for particles being processed by a block.

The potential benefit of this approach was the reduction in global memory accesses, a common bottleneck in GPU computing. By caching acceleration values in the faster shared memory, we could potentially enhance the performance of our simulation, especially considering the frequent read and write operations associated with updating particle accelerations.

However, a significant limitation of this approach was identified: the particles corresponding to indices i and j in the force computation loop could reside in different blocks. In such scenarios, the acceleration data needed for a particle in one block could be present in the shared memory of another block, making it inaccessible. This limitation meant that caching in shared memory would not always be effective.

While in many cases particles might be processed within the same block, ensuring efficient use of shared memory, the inability to guarantee this for all particle pairs reduced the overall effectiveness of this strategy.

B. Changing the j loop

Another approach was one that involved altering the iteration pattern in the 'j' loop of the force calculation. Traditionally, in a CPU-based approach, the 'j' loop runs from (i + 1) to N, ensuring each interaction is computed once. However, in our CUDA implementation, we modified this loop to iterate from 0 to N.

This change in iteration pattern resulted in N squared iterations, significantly increasing the computational workload. However, it also allowed us to circumvent the challenge of concurrent write operations to the acceleration array, a, at the j index. In molecular dynamics simulations, the force (and hence acceleration) between two particles is equal and opposite. This principle was utilized in our modified approach.

To account for this, we implemented a conditional operation within the loop: if j was less than i, the acceleration would be subtracted, while if j was greater than i, it would be added. This strategy aimed to ensure that when the roles of i and j were reversed in different threads, the correct cumulative acceleration would be applied to each particle.

This approach, though conceptually sound in addressing the issues of parallelism in CUDA, led to an increased number of computations. Every particle interaction was essentially computed twice, which is a deviation from the more efficient half-matrix computation in traditional molecular dynamics simulations. The rationale behind this approach was to simplify the parallel computation model and avoid race conditions in memory writes, which are a common challenge in CUDA programming when dealing with shared data structures.

This subsection of our CUDA implementation journey provides an insight into the challenges and trade-offs encountered when adapting algorithms for parallel execution in GPU environments. The goal was to harness the parallel processing power of GPUs while ensuring the integrity and correctness of the simulation's output.

C. Conclusion on CUDA

Subsequent testing revealed discrepancies in the simulation results produced by the CUDA-based version when compared to those obtained from the CPU-based version. The inconsistency in the output values necessitated a thorough investigation. Despite extensive debugging and analysis, the precise cause of these discrepancies remained unresolved. Factors such as parallel execution anomalies, memory access patterns, and synchronization issues within the GPU architecture were considered, but a conclusive solution was not attained.

Given the critical importance of accuracy in simulation results and the constraints of the project timeline, the decision was made to discontinue the CUDA implementation. This decision was guided by the necessity for reliable and accurate simulation outcomes over exploratory implementation. The experience provided insights into the intricacies of GPU-based computing and highlighted the challenges as-

sociated with adapting complex simulations to parallel computing environments.

Bellow is the final CUDA code that we developed. This version of the code is just a Proof of concept and doesn't address data races on the *a* array with the block shared memory (which would then be reduced in the CPU part of the code into a single array).

```
double PotentialAndAcceleration(double dt) {
    double *a_dev, *r_dev, *Pot_dev;
    double Pot = 0.0;

    // Allocate device memory
    cudaMalloc((void**)&a_dev, N * 3 * sizeof(double));
    cudaMalloc((void**)&r_dev, N * 3 * sizeof(double));
    cudaMalloc((void**)&Pot_dev, N * sizeof(double));

    // Copy data from host to device
    cudaMemcpy(r_dev, r, N * 3 * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemset(Pot_dev, 0, N * sizeof(double), cudaMemcpyHostToDevice);
    // Set a_dev to 0
    cudaMemset(a_dev, 0, N * 3 * sizeof(double));

    // Set grid and block sizes
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    // Launch the kernel
    computeAccelerationsAndPotentialKernel<<<blocksPerGrid, threadsPerBlock>>>(a_dev, r_dev,
Pot_dev, sigma, epsilon_8);

    // Copy results back to host
    cudaMemcpy(a, a_dev, N * 3 * sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&Pot, Pot_dev, sizeof(double), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(a_dev);
    cudaFree(r_dev);

    // Accumulate the potential energy
    for (int i = 0; i < N; i++) {
        Pot += Pot_dev[i];
    }
    cudaFree(Pot_dev);

    return Pot * epsilon_8;
}
```

Figure 1: The function that launches the CUDA Kernel

```
__global__ void computeAccelerationsAndPotentialKernel(double *a_dev, double *r_dev, double
*Pot_dev, double sigma, double epsilon_8) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N - 1) {
        double ai0 = 0.0, ai1 = 0.0, ai2 = 0.0;
        double rI[3] = {r_dev[i * 3], r_dev[i * 3 + 1], r_dev[i * 3 + 2]};
        double Pot_local = 0.0;

        for (int j = i + 1; j < N; j++) {
            double quot, term2;
            double rij[3];
            double rSqd = 0.0;

            for (int k = 0; k < 3; k++) {
                double rijk = rI[k] - r_dev[j * 3 + k];
                rij[k] = rijk;
                rSqd += rijk * rijk;
            }

            double rSqd_3 = rSqd * rSqd * rSqd;
            double rSqd_7 = rSqd_3 * rSqd_3 * rSqd;
            double f = (48.0 - (24.0 * rSqd_3)) / rSqd_7;

            double tmp = rij[0] * f;
            double tmp1 = rij[1] * f;
            double tmp2 = rij[2] * f;
            ai0 += tmp;
            ai1 += tmp1;
            ai2 += tmp2;
            // Have to fix this data race
            a_dev[j * 3]  -= tmp;
            a_dev[j * 3 + 1] -= tmp1;
            a_dev[j * 3 + 2] -= tmp2;

            quot = sigma * sigma / rSqd;
            term2 = quot * quot * quot;
            Pot_local += term2 * (term2 - 1.0);
        }

        Pot_dev[0] = Pot_local;
        // Have to fix this data race
        a_dev[i * 3] += ai0;
        a_dev[i * 3 + 1] += ai1;
        a_dev[i * 3 + 2] += ai2;
    }
}
```

Figure 2: This is one of the versions of the CUDA Kernel code

IV. SIMD

Recognizing the challenges and inaccuracies encountered with the CUDA-based implementations, we resolved to concentrate on optimizing our code using CPU-based vectorization techniques.

The final approach involved refining our algorithm to better leverage the capabilities of modern CPUs, specifically focusing on manual vectorization using **Advanced Vector Extensions** (AVX). This method allowed us to process multiple data points simultaneously, significantly enhancing the computational efficiency while maintaining the integrity of the simulation outcomes. By prioritizing reliability and precision, we aimed to achieve a balance between computational performance and the accuracy of our simulation results.

A. *Implementation*

In the final implementation phase, our focus shifted towards leveraging CPU-based vectorization. This involved a series of modifications to the existing simulation code to optimize it for SIMD processing.

One of the key changes was the restructuring of the data arrays. The original code utilized multidimensional arrays to represent particle positions and accelerations. In the revised approach, these were transformed into separate one-dimensional arrays - rx, ry, rz for positions, and ax, ay, az for accelerations. This change enhanced data locality and alignment, crucial for efficient SIMD processing.

The core computational routines of the simulation, particularly those involved in calculating particle interactions, were then adapted to use AVX instructions. Functions like *mm256_load_pd and* mm256_store_pd were employed to load and store data in 256-bit wide registers, allowing operations on four double-precision elements simultaneously. Computational operations, such as calculating distances and forces between particles, were implemented using AVX functions like *mm256_add_pd and* mm256_mul_pd, which perform element-wise addition and multiplication, respectively.

These modifications resulted in a significant enhancement in the computational efficiency of the simulation. By processing multiple data points in parallel, the execution time was considerably reduced, while maintaining the accuracy and integrity of the simulation results. This successful implementation of CPU-based vectorization demonstrates the potential of SIMD processing in optimizing computational tasks in scientific simulations.

V. Performance scaling data

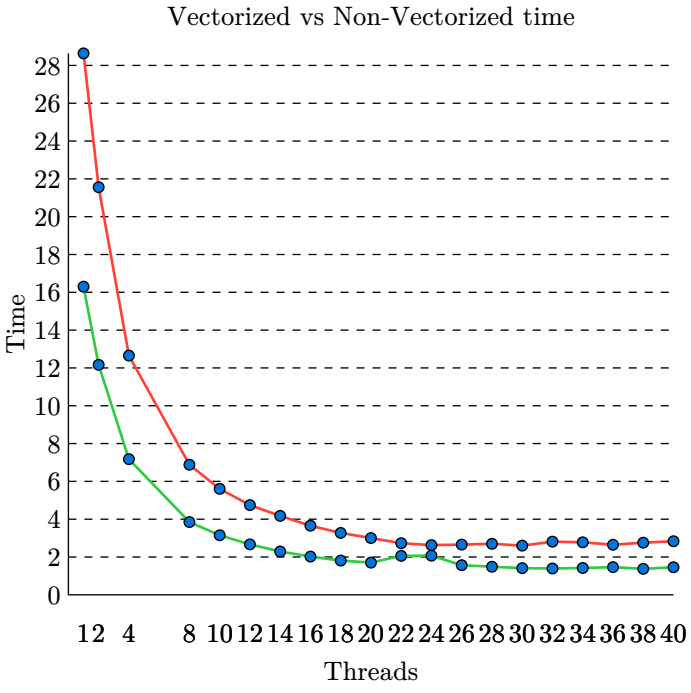
The table below provides a concise summary of the execution times observed for different numbers of threads, focusing solely on the 'real' time, which represents the total elapsed time.

Threads	Time (s)
1	16.298
2	12.162
4	7.174
10	3.150
12	2.667
14	2.290
16	2.023
18	1.811
20	1.702
22	2.061
26	1.566
28	1.485
32	1.392
34	1.419
38	1.373

And bellow is a section of the graph of execution time to number of threads graph.

Green represents the vectorized code's time, and red represents phase 2's code.

Green represents the



VI. Analysing results

The performance improvement in Phase 3, which incorporated CPU-based vectorization using Advanced Vector Extensions (AVX) alongside OpenMP, was remarkable compared to the non-vectorized OpenMP implementation in Phase 2. The execution times in Phase 3 were significantly reduced, about half of what was observed in Phase 2. This demonstrates the substantial impact of SIMD (Single Instruction, Multiple Data) operations when integrated into parallelized computational tasks. The combination of OpenMP for threading and AVX for vectorization effectively harnessed the CPU's capabilities, leading to more efficient processing and quicker execution times, particularly beneficial for the complex calculations involved in molecular dynamics simulations.

In the detailed analysis of the execution times across varying thread counts, we observed a plateau in performance improvements beyond 28 threads. While increasing the number of threads substantially reduced execution times initially, the benefits diminished after crossing this 28-thread mark. This observation is indicative of the limitations imposed by factors such as the CPU's memory bandwidth and architectural constraints, which set a practical upper limit to the efficiency gains from parallelization. The use of OpenMP in conjunction with SIMD vectorization in Phase 3 thus emerged as a highly effective approach, maximizing CPU computation efficiency up to this point of optimal thread count. This strategic combination of threading and data-level parallelism proved to be an essential factor in achieving the enhanced performance seen in the vectorized implementation of Phase 3.

1) *Comparing to theoretical performance gain:*

Understanding the theoretical versus actual performance improvements in our AVX vectorization implementation, which processes four particles at a time, provides valuable insights into the practical application of SIMD operations in molecular dynamics simulations.

Theoretically, the capability of AVX to handle four particles simultaneously should lead to a substantial increase in computational throughput. Ideally, this would mean a four-fold improvement in processing speed compared to handling a single particle at a time. This theoretical efficiency gain assumes optimal conditions where the CPU can seamlessly parallelize these operations without significant overhead or constraints.

However, in our practical results, the performance enhancement, while substantial, did not fully match the theoretical quadrupling of speed. This divergence can be attributed to various real-world factors. For instance, the efficiency of memory access, the effectiveness of CPU cache utilization, and the overhead associated with orchestrating

parallel operations can all impact the performance gains from SIMD operations. Additionally, parts of the simulation that remain unvectorized contribute to the overall execution time, thereby reducing the relative impact of the vectorized sections.

Despite these considerations, the implementation of AVX to process four particles at a time demonstrated a significant boost in performance, affirming the value of SIMD operations in computational physics. This practical experience underscores that while theoretical models provide an upper limit of expected performance, actual results may vary due to a range of systemic and architectural factors. Nonetheless, the approach validates the effectiveness of vectorization in enhancing computational efficiency in scientific simulations.

VII. Conclusion

In conclusion, this paper documents the journey of optimizing and parallelizing a molecular dynamics (MD) simulation, with a particular focus on the utilization of Advanced Vector Extensions (AVX) for CPU-based vectorization. The project progressed through various phases, from initial code optimization and parallelization using OpenMP to an attempt at implementing CUDA-based GPU acceleration.

The key findings and takeaways from our project can be summarized as follows:

A. Optimization and Parallelization:

In Phase 1, we successfully optimized the single-threaded MD simulation code, achieving a remarkable reduction in execution time through various optimization techniques. Phase 2 extended these improvements by parallelizing the code using OpenMP, demonstrating substantial speedup while highlighting memory bandwidth limitations.

B. CUDA Implementation:

Our attempt to implement MD simulation on CUDA GPUs presented challenges related to memory access patterns and synchronization issues, ultimately leading to discrepancies in simulation results. As a result, we decided to discontinue the CUDA implementation in favor of accuracy and reliability.

C. CPU-Based Vectorization with AVX:

Phase 3 marked the implementation of CPU-based vectorization using AVX instructions. This approach significantly improved computational efficiency by processing multiple data points simultaneously. The restructuring of data arrays and the use of AVX instructions led to a notable reduction in execution times.

D. Performance Scaling:

The performance scaling analysis revealed that the combination of OpenMP threading and AVX vectorization was highly effective, with performance improvements observed up to a certain thread count limit. Beyond this limit, practical constraints such as CPU memory bandwidth and architecture limited further efficiency gains.

E. Practical vs. Theoretical Performance:

While the theoretical potential for AVX vectorization to quadruple processing speed was not fully realized in practice due to various real-world factors, the implementation demonstrated a substantial performance boost. This underscores the practical value of SIMD operations in scientific simulations.

In summary, this project showcases the significance of advanced computational techniques in optimizing and parallelizing scientific simulations. The integration of AVX vectorization into our MD simulation code exemplifies how modern processing capabilities can be harnessed to enhance computational efficiency. Furthermore, it emphasizes the importance of balancing theoretical performance expectations with real-world limitations and the need for accuracy and reliability in scientific simulations.

Future work in this area could explore hybrid approaches that combine CPU-based vectorization with GPU acceleration to further improve the efficiency of molecular dynamics simulations. Additionally, ongoing advancements in hardware and software technologies may offer new opportunities for optimizing and parallelizing computational physics simulations, paving the way for even more efficient and accurate scientific research in the future.