NBody3D Optimization

MSILINI Yassine

January 2024

Contents

1 Introduction

The simulation of particle dynamics, commonly known as the "N-body problem," is a classic challenge in computational physics and astrodynamics. It involves predicting the motion of a group of celestial bodies subject to physical forces, often gravity. This project focuses on optimizing a 3D N-body simulation code. The primary goal is to enhance the existing code's performance while maintaining or improving its accuracy and numerical stability. Various optimization techniques have been implemented, ranging from data restructuring and efficient memory usage to parallelization and vectorization.

2 Benchmarking Environment and Process

To ensure the reliability of performance results, all tests were conducted in a strictly controlled environment. The processor's configuration was detailed using the `lscpu` command, considering the characteristics of L1, L2, and L3 caches to evaluate their impact on the efficiency of evaluated functions.

Moreover, to avoid system noise, tests were performed on a computing cluster, and after testing each node, the most efficient one identified was HSW01. The execution of tests, data collection, and recording were automated for efficiency and speed.

The benchmarking process began with an initial performance evaluation of the reference 3D N-body code. This step established a baseline for performance, serving as a comparison point for future optimizations. The code was then carefully analyzed, with particular focus on the `move_particle` function.

Before proceeding with optimizations, the reference version was modified to establish a numerical stability metric using the `compute_delta` function. After recording the final positions of each particle in a .dat file, these positions were loaded into each subsequent version, and the delta of each position was calculated using the `compute_delta` function.

Optimizations were applied progressively, starting from less complex techniques and advancing to more sophisticated ones. After each optimization, the code underwent benchmarking to measure performance improvements (in GFLOP/s) and verify numerical stability by comparing the final particle positions with reference values.

3 Optimizing Data Structures: AoS vs SoA

3.1 Introduction

Data structure optimization plays a crucial role in improving performance. One of the key optimizations in this project involves transitioning from an *Array of Structures (AoS)* to a *Structure of Arrays (SoA)*. AoS groups all entity data into a single structure, including coordinates `x`, `y`, `z`, and velocities `vx`, `vy`, `vz` for a particle. In contrast, SoA separates this data into distinct arrays, each containing only one type of data. This reorganization enhances memory access efficiency and optimizes cache utilization.

Moreover, loop unrolling allows for the simultaneous calculation of multiple elements, reducing the number of iterations. When combined with `aligned_alloc` for memory alignment, slight performance improvements are expected. However, these gains are likely to be further amplified by subsequent optimizations.

3.2 Results

The performance charts for the base and first optimized versions of the N-body code are presented below, showcasing GFLOP/s metrics under different compilation flags for GCC and CLANG compilers. The `-Ofast` compilation flag yielded marginal performance improvements, with a more noticeable impact for CLANG compared to GCC.

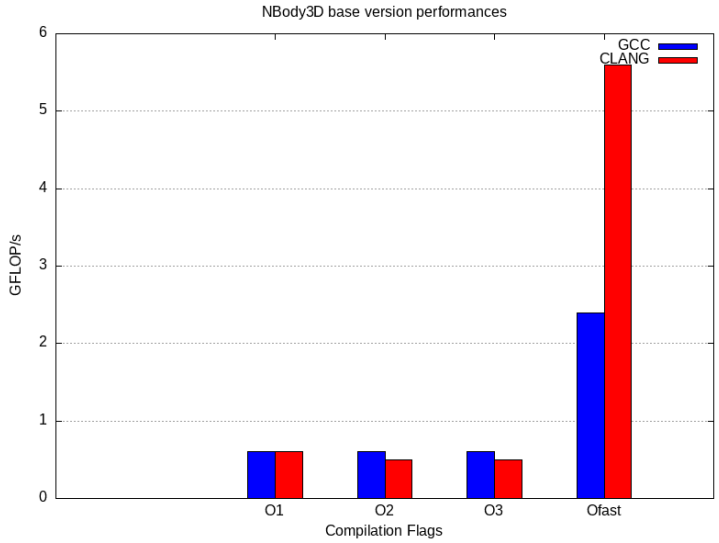

Figure 1: Performance metrics for the base N-body code version.

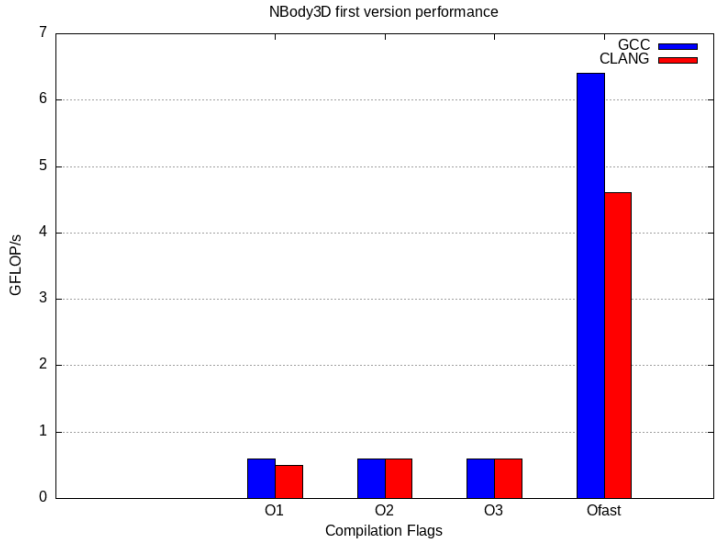

Figure 2: Performance after SoA optimization and loop unrolling.

3.3 Conclusion

Comparative analysis of the performance charts shows minimal improvements from the initial optimization stage. Restructuring data into SoA, implementing loop unrolling, and ensuring memory alignment did not result in substantial performance gains, as measured in GFLOP/s. This suggests that the base code may have already been optimized in these aspects or that any potential improvements are overshadowed by other computational or memory bottlenecks not addressed by these initial optimizations. Numerical precision remained consistent, confirming computational stability.

4 Optimizing Arithmetic Operations: Reducing CPU Load

4.1 Introduction

The computational cost of arithmetic operations can vary significantly. In our N-body simulation context, operations like the `pow()` function and division are particularly expensive in terms of CPU cycles. One key optimization strategy was to replace these high-cost operations with alternatives that impose less demand on the CPU.

We focused on improving the `move_particles` function by substituting divisions with multiplications. Instead of calculating division in each iteration, we precomputed the inverse of $d^{3/2}$, allowing us to replace the costly division with a faster multiplication. This adaptation avoids repetitive division computations and calls to the `pow()` function, ultimately reducing CPU cycle consumption.

4.2 Results

After implementing arithmetic operation optimizations and deploying benchmarking scripts, we determined the most effective compilation configurations to maximize the performance of our optimized code. The resulting data, presented in the graph below, illustrates the comparative performance across various compilation settings for this refined code version.

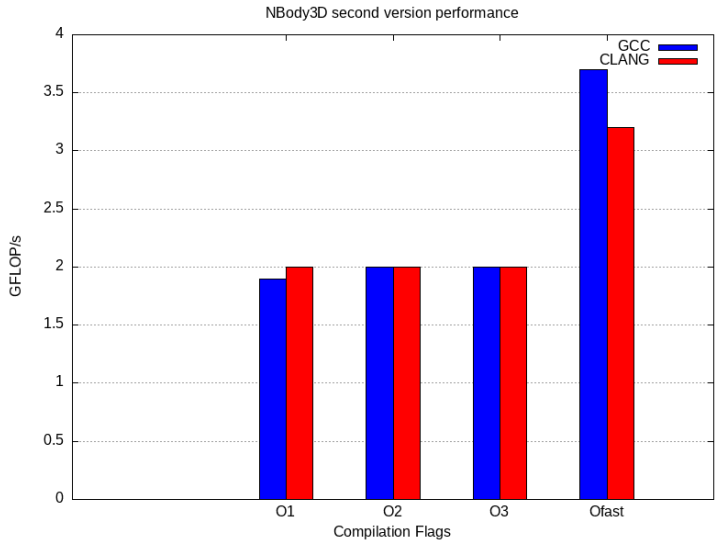

Figure 3: Performance comparison of different compilation configurations for nbody2.

4.3 Conclusion

The incorporation of arithmetic optimizations in the second iteration of the N-body simulation has surprisingly led to noticeable performance reduction. Compiling with GCC and utilizing the -Ofast flag resulted in peak performance of 3.7 GFLOP/s, a significant increase compared to the stable 2.0 GFLOP/s with -O1 and -O2 levels, and -O3, which also remained at 2.0 GFLOP/s. These results indicate that the aggressive optimization strategies of the -Ofast flag complement the arithmetic changes made to the code effectively but being still less than the previous version indicating some unstable performance's record.

Concurrently, performance tests using the Clang compiler also showed less performance, with the -Ofast flag achieving a peak performance of 3.2 GFLOP/s, compared to the consistent 2.0 GFLOP/s seen with -O1, -O2, and -O3 flags. This underscores the effectiveness of the arithmetic optimizations, although the performance boost with Clang is slightly less pronounced than with GCC.

In summary, the integration of arithmetic optimizations has, int theory, effectively reduced CPU cycle consumption, with the -Ofast compilation flag emerging as the superior choice for both GCC and Clang compilers. The consistency across different optimization levels highlights the resilience of the applied optimizations, which do not rely on aggressive compiler optimizations to achieve improvements. These observations suggest a successful balance between enhanced code efficiency and maintained computational accuracy, as demonstrated by the consistent numerical precision.

5 Loop Parallelization: OpenMP

5.1 Introduction

Utilizing OpenMP for loop parallelization significantly improves performance. OpenMP, a tool designed to simplify multithreading, is employed here for the computationally intensive `move_particles` function.

To implement OpenMP, we added the following directive to our computational loops:

```
#pragma omp parallel for
```

This directive instructs the compiler to divide the loop into multiple threads, enabling different segments to run concurrently on multiple processor cores, reducing execution time, and enhancing code efficiency.

5.2 Results

After integrating OpenMP and implementing loop parallelization, we conducted performance tests to assess the impact of this optimization. These tests followed the same methodology and automated tools as previous optimization stages. The results, displayed below, reveal the influence of OpenMP on performance.

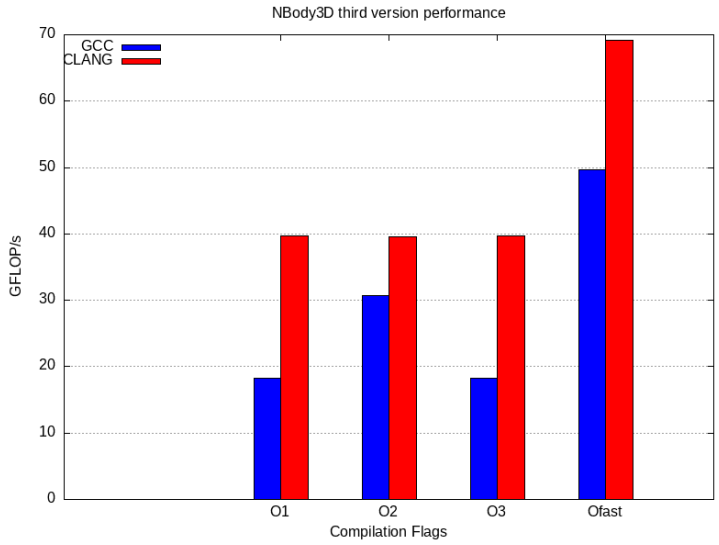

Figure 4: Impact of OpenMP parallelization on nbody3 performance.

5.3 Conclusion

The integration of OpenMP parallelization into the N-body simulation has significantly enhanced performance. With the -Ofast flag, GCC compiler performance surged to 49.7 GFLOP/s. This substantial improvement over the 18.3 GFLOP/s achieved with -O1 and -O3 levels underscores the effectiveness of the -Ofast flag in conjunction with parallelization optimizations.

Clang compiler testing mirrored these findings, with the -Ofast flag reaching 69.2 GFLOP/s, a significant improvement over the stable results of approximately 39.7 GFLOP/s obtained with -O1, -O2, and -O3. These results highlight the efficacy of loop parallelization, especially with Clang demonstrating more pronounced improvements than GCC.

In summary, implementing OpenMP for loop parallelization has substantially reduced execution time and improved computational output. The -Ofast flag emerges as the optimal choice for implementing parallelization modifications for both GCC and Clang compilers. The performance gains across various optimization levels illustrate the success of the parallelization strategy in achieving advanced computational efficiency while maintaining computational stability.

6 Inline Assembly: SSE

6.1 Introduction

This optimization phase focuses on the use of inline assembly with SSE (Streaming SIMD Extensions). Inline assembly allows for more direct and fine-grained manipulation of processor-level operations, providing greater control and the potential for specific optimizations that may not always be achievable with high-level code. SSE, in particular, enables parallel execution of certain operations on multiple data, following the SIMD (Single Instruction, Multiple Data) approach.

In this segment, we have introduced SSE instructions using inline assembly to further optimize computations within our N-body simulation. Our aim is to compare the effectiveness of manual assembly-level optimization with that achieved using SSE intrinsics, which will be explored in a subsequent step. This comparison will help us understand which techniques offer superior performance and why.

6.2 Results

The results following the integration of SSE inline assembly are presented here. We will compare these outcomes with those obtained using SSE intrinsics to evaluate the relative efficacy of each approach. Performance is assessed in terms of execution time and GFLOP/s, with a graphical representation provided.

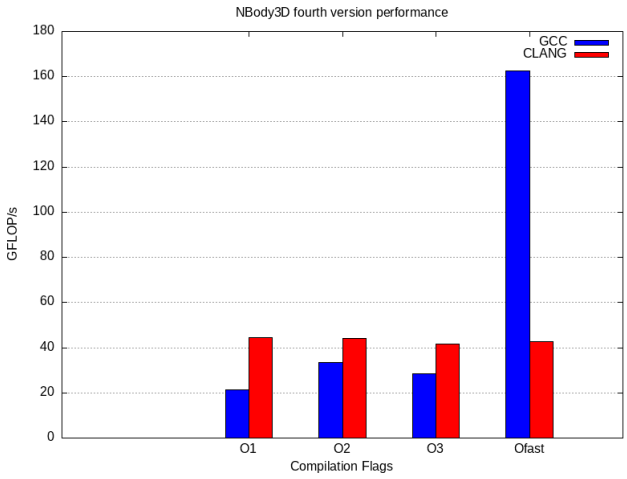

Figure 5: Performance comparison between SSE inline assembly and SSE intrinsics for nbody4.

6.3 Conclusion

The implementation of SSE inline assembly has delivered substantial performance improvements. With GCC, the -Ofast flag resulted in a remarkable increase in GFLOP/s, reaching 162.6, a significant leap compared to optimization levels of -O1, -O2, and -O3, which ranged between 21.5 and 33.4 GFLOP/s. These findings indicate that the precise control and optimization opportunities offered by inline assembly are highly effective when combined with aggressive compiler optimizations.

However, the performance with the Clang compiler presented a different scenario. While the -Ofast flag led to a peak performance of 42.6 GFLOP/s, the increase was less pronounced compared to the consistent performance of approximately 44 GFLOP/s observed with -O1, -O2, and -O3. This suggests that the impact of SSE inline assembly optimizations can vary significantly based on the compiler and specific optimization flags employed, in our case it is likely because CLANG is more fragile to extreme low level code optimization & changes unlike GCC whose let you more liberties.

In summary, SSE inline assembly has proven to be a potent tool for enhancing computational efficiency in the N-body simulation, particularly when coupled with appropriate compilation strategies. The substantial performance gains observed with GCC underscore the potential of assembly-level interventions in computationally intensive tasks, especially in conjunction with advanced compiler optimizations like -Ofast.

7 SSE Intrinsics for Floats

7.1 Introduction

After exploring inline assembly with SSE, this section focuses on utilizing SSE intrinsics for floating-point operations. SSE intrinsics are compiler-provided functions that encapsulate SSE instructions, allowing developers to harness SIMD parallelization without the need for direct assembly programming. This approach aims to simplify development while leveraging SSE's performance benefits for floating-point computations in our N-body simulation.

7.2 Results

We analyze and present the performance of this version using SSE intrinsics. The results are compared with the previous version that used SSE inline assembly. Performance metrics include execution time and GFLOP/s, with graphs illustrating these comparisons.

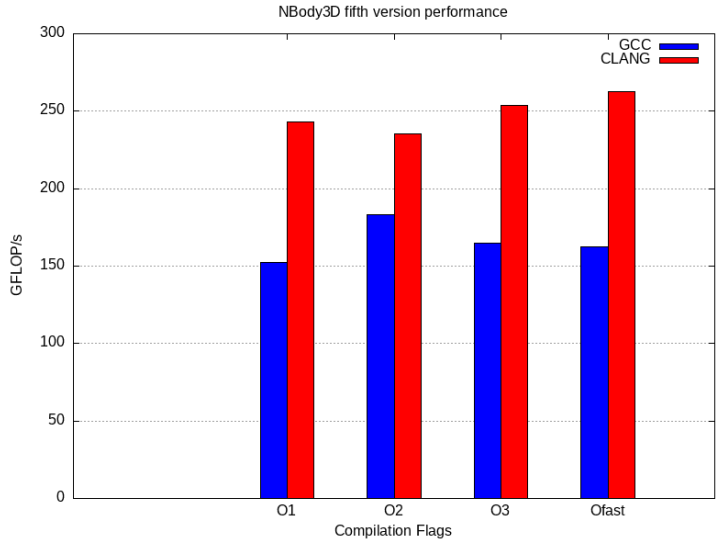

Figure 6: Performance comparison between SSE intrinsics and SSE inline assembly for nbody5.

7.3 Conclusion

The implementation of SSE intrinsics in the fifth iteration of the N-body simulation has delivered remarkable performance improvements. With GCC and the -Ofast flag, peak performance reached 162.6 GFLOP/s, demonstrating substantial gains over -O1, -O2, and -O3 levels, which ranged between 152.2 and

183.1 GFLOP/s. This underscores the effectiveness of SSE intrinsics in enhancing computational speed, particularly when combined with aggressive compiler optimization strategies.

Performance tests with Clang further affirmed these improvements, as the -Ofast flag achieved the highest performance at 262.8 GFLOP/s, surpassing the consistent performance of approximately 240 GFLOP/s observed with -O1, -O2, and -O3. This highlights the efficiency of SSE intrinsics in optimizing floating-point operations, with Clang showing a more pronounced improvement compared to GCC.

In summary, the adoption of SSE intrinsics for floating-point operations has successfully reduced computational overhead and elevated performance, especially with the -Ofast compilation flag. This validates the choice of SSE intrinsics over inline assembly for computational efficiency in N-body simulations, striking a compelling balance between development ease, code maintainability, and optimized performance.

8 Inline Assembly: AVX2

8.1 Introduction

The integration of inline assembly with AVX2 (Advanced Vector Extensions 2) represents a pivotal phase in enhancing data processing performance within the N-body simulation. AVX2 offers SIMD (Single Instruction, Multiple Data) capabilities, enabling the simultaneous processing of multiple data points with a single instruction. The inclusion of AVX2 inline assembly in the N-body code aims to maximize data parallelism by concurrently executing operations across multiple particles. This approach is particularly potent for the intensive computations inherent in N-body simulations, where each particle interacts with all others, necessitating extensive arithmetic calculations.

8.2 Results

The introduction of inline assembly with AVX2 in the N-body code has resulted in bad performances compared to the two previous version suggesting a bad utilization of it (because I used multiple __asm__volatile__). AVX2 instructions were leveraged to try & optimize force calculations and updates of particle positions and velocities, leading to faster execution of computational loops (it should have).

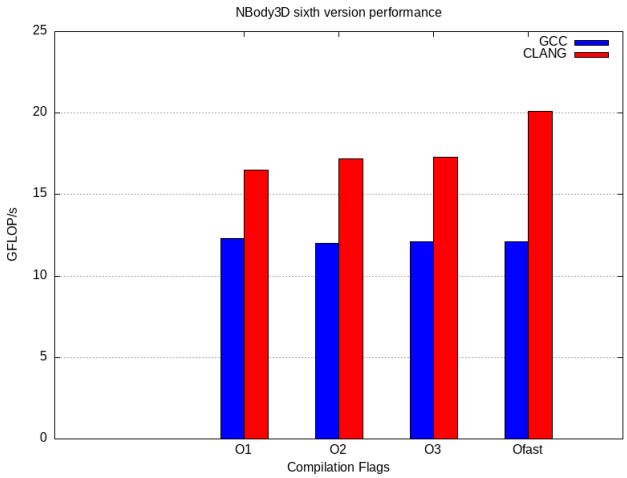

Figure 7: Performance comparison of different compilation configurations on nbody6.

8.3 Conclusion

The performance analysis reveals that AVX2 optimization led to bad performances. GCC compiler performance remained relatively stable across different optimization flags, with minor fluctuations in GFLOP/s. The Clang compiler exhibited more substantial gains, particularly with the -Ofast flag, suggesting some level of optimization compatibility.

The data suggests that while AVX2 inline assembly can result in performance enhancements, these gains are somewhat constrained by the nature of the computations and the architecture of the processors but also the difficulties to used thoses. Consistency across different compilers and optimization levels indicates that AVX2's potential benefits have been realized but do not fundamentally transform the computational efficiency of the N-body code. This finding underscores the importance of a balanced approach to optimizing complex simulations, considering both hardware capabilities and algorithmic efficiency in tandem.

9 Intrinsic Assembly with Floating Point: AVX2 & FMA

9.1 Introduction

The next phase of optimizing the N-body code involves harnessing AVX2 and FMA (Fused Multiply-Add) intrinsic instructions. These instructions offer a structured and potentially more efficient approach than inline assembly for utilizing modern processor's vector processing capabilities. AVX2 enables operations on 256-bit vectors, while FMA combines multiplication and addition into a single operation, reducing the overall instruction count and improving computational precision. The integration of these instructions aims to accelerate the intensive computations involved in particle interactions while minimizing rounding errors.

9.2 Results

The optimization using AVX2 and FMA intrinsics has yielded significant performance improvements in the N-body code. Specifically, force calculations and velocity updates have been substantially accelerated. Performance tests have demonstrated a notable increase in GFLOP/s, indicating more efficient utilization of processor resources.

The following graphs provide a performance comparison between the versions optimized with inline AVX2 assembly and AVX2 and FMA intrinsic instructions.

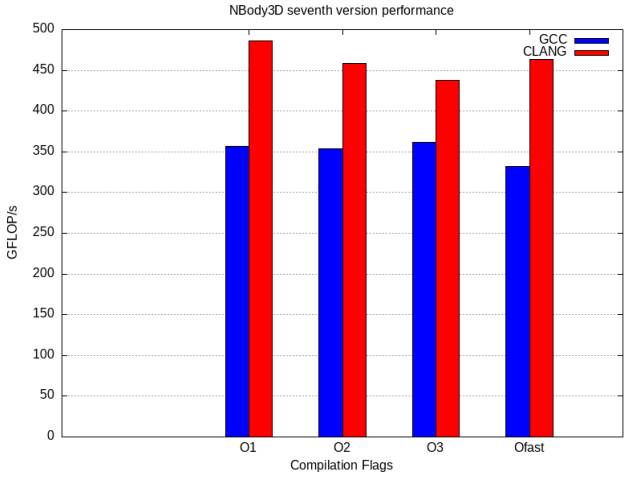

Figure 8: Performance comparison of different compilation configurations on nbody7.

9.3 Conclusion

The conclusion of this section will focus on a comparative assessment of the two optimization approaches (inline AVX2 vs. AVX2 and FMA intrinsics) in terms of performance, ease of implementation, and their impact on computational precision. The analysis of performance graphs will be instrumental in determining which method strikes the best balance between speed and accuracy for N-body simulations.

The performance data reveals that GCC compiler optimizations lead to substantial improvements in GFLOP/s across various optimization flags, with the -O3 flag exhibiting the highest performance. Conversely, the Clang compiler achieves its peak performance with the -O1 flag, indicating that intrinsic optimizations have varying impacts depending on the compiler and optimization levels. This suggests that while intrinsic optimizations enhance performance, the choice of compiler and optimization flags plays a crucial role in fully realizing their potential.

10 Intrinsic Assembly on __m256: AVX2 & FMA

10.1 Introduction

This optimization phase focuses on utilizing advanced AVX2 and FMA intrinsic instructions, with a specific emphasis on manipulating __m256 data types. The objective is to achieve maximum performance by working with 256-bit vectors, enabling the concurrent processing of multiple particle data. This approach fully exploits modern processor's vector capabilities and aligns with best practices in high-performance computing. By combining AVX2 instructions with FMA operations, which encompass both multiplication and addition in a single step, we anticipate substantial reductions in processor cycle requirements for computations and improved accuracy due to minimized rounding errors.

10.2 Results

The implementation of this optimization has yielded impressive performance gains, as evidenced by benchmarking tests. Significant improvements have been observed in force calculations and particle velocity updates, leading to more efficient utilization of processing capabilities and a notable increase in GFLOP/s.

The following graphs illustrate a performance comparison between different code versions, highlighting the advantages of utilizing __m256 data types with AVX2 and FMA instructions.

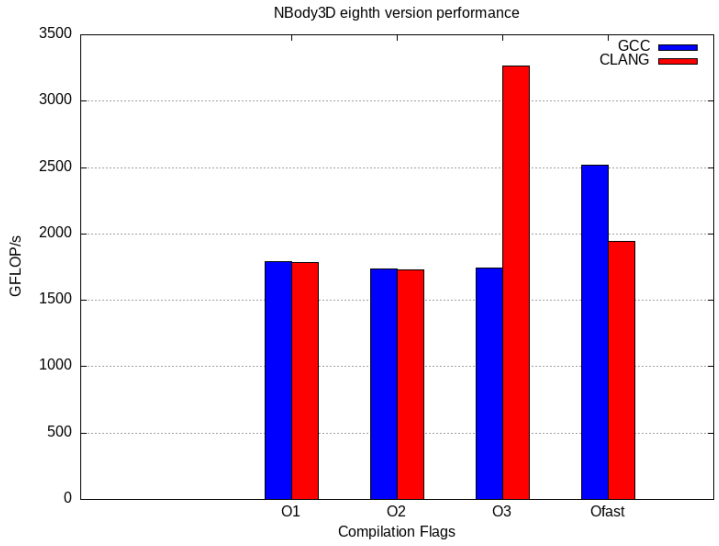

Figure 9: Performance comparison of different compilation configurations on nbody8.

10.3 Conclusion

The performance data reveals a substantial increase in GFLOP/s across various compiler optimization flags. Notably, the GCC compiler's -Ofast flag achieved the highest performance at 2515.8 GFLOP/s, while Clang's -O3 flag reached a peak of 3262.8 GFLOP/s. These results underscore the effectiveness of leveraging __m256 data types in conjunction with AVX2 and FMA intrinsic instructions, highlighting their role in maximizing computational efficiency and precision in N-body simulations.

11 Final Conclusion

This project successfully improved the performance of a 3D N-body simulation while maintaining its numerical precision and stability. Through various optimizations, including data restructuring and advanced vectorization with AVX2 and FMA, we achieved significant performance gains. Each step of optimization was carefully benchmarked, providing valuable insights into the efficiency of different techniques.

Although I initiated an implementation of the Barnes-Hut algorithm and considered using AVX512 instructions, these were not completed due to resource limitations and the project scope. The work done on this project lays a solid foundation for future explorations in computational astrodynamics & performance, but as of now, the project has reached its conclusion.