Sarah Gomez

Dr. Stephen Cotton

Chem 281

3/7/2024

Problem set 2 Answers

Problem 1

When I instantiate timers as static variables in my code, they remain active and retain their state throughout the entire runtime of the program. This persistence is crucial for maintaining the continuity of timing data across various function calls or different stages of execution. I find this feature incredibly useful for performance monitoring because it enables me to measure how long different sections of my program take to execute, not just in isolation but also over time and through repeated operations. For instance, in my implementation, I use totalTimer, recursiveTimer, and reverseTimer specifically to track the execution times of the entire program, the recursive_sqrt function, and the reverse_process function, respectively. This method of using static timers allows me to systematically benchmark performance and pinpoint where the program might be lagging or consuming more time than expected. It's an effective strategy for identifying and analyzing performance bottlenecks, thus aiding in the targeted optimization and enhancement of my code's efficiency.

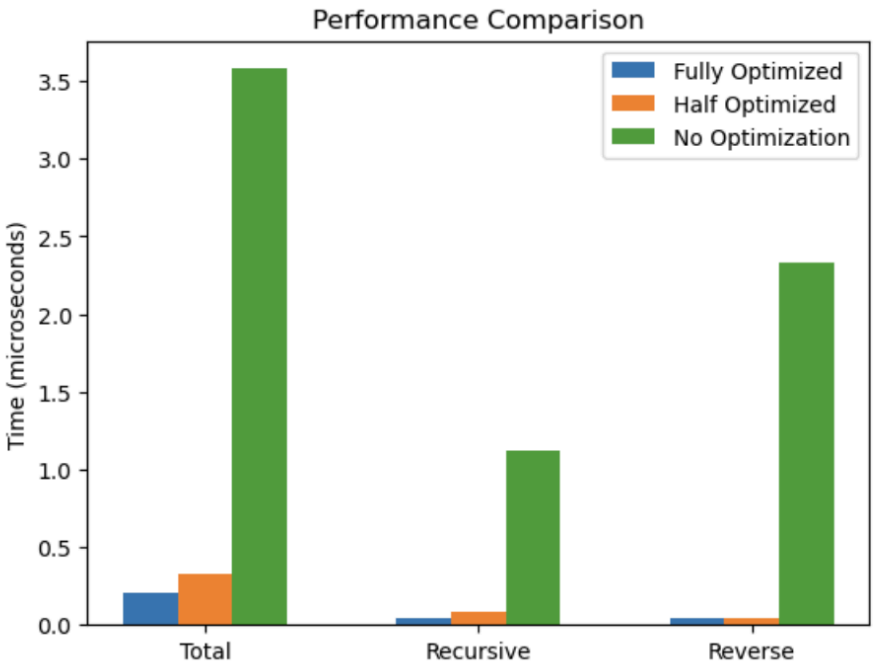

Figure 1. Problem 1 toy Execution Performance Comparison

Figure 1 illustrates a performance comparison that visualizes the execution time of a program under three optimization conditions: "Fully Optimized," "Half Optimized," and "No Optimization." The graph demonstrates the impact of these conditions on the "Total," "Recursive," and "Reverse" operation times, measured in microseconds. The graph clearly indicates that the "Fully Optimized" condition, compiled with the highest level of optimization (like -O3 in GCC or Clang), yields the best performance across all operations, with "Total" execution requiring only 0.21 microseconds, "Recursive" at 0.04 microseconds, and "Reverse" also at 0.04 microseconds. These figures showcase the significant efficiency gains achievable through comprehensive optimization. When discussing "Half Optimized," it refers to a scenario

where different parts of the program are compiled with different optimization levels—for instance, time.cpp with -O0 (no optimization) and the rest with -O3 (full optimization). The measured times under this condition increase, with "Total" taking 0.33 microseconds, "Recursive" at 0.08 microseconds, and "Reverse" maintaining at 0.04 microseconds. The "Reverse" operation appears unaffected by the reduced optimization level. However, the "Total" and "Recursive" operations exhibit noticeable performance degradation, albeit not as drastic as with no optimization at all.The "No Optimization" condition demonstrates the impact of not using any compiler optimizations, resulting in the slowest performance, with "Total" taking 3.58 microseconds, "Recursive" at 1.12 microseconds, and "Reverse" at 2.33 microseconds. These results underscore the substantial benefits of compiler optimizations and their role in improving execution efficiency.

Problem 2

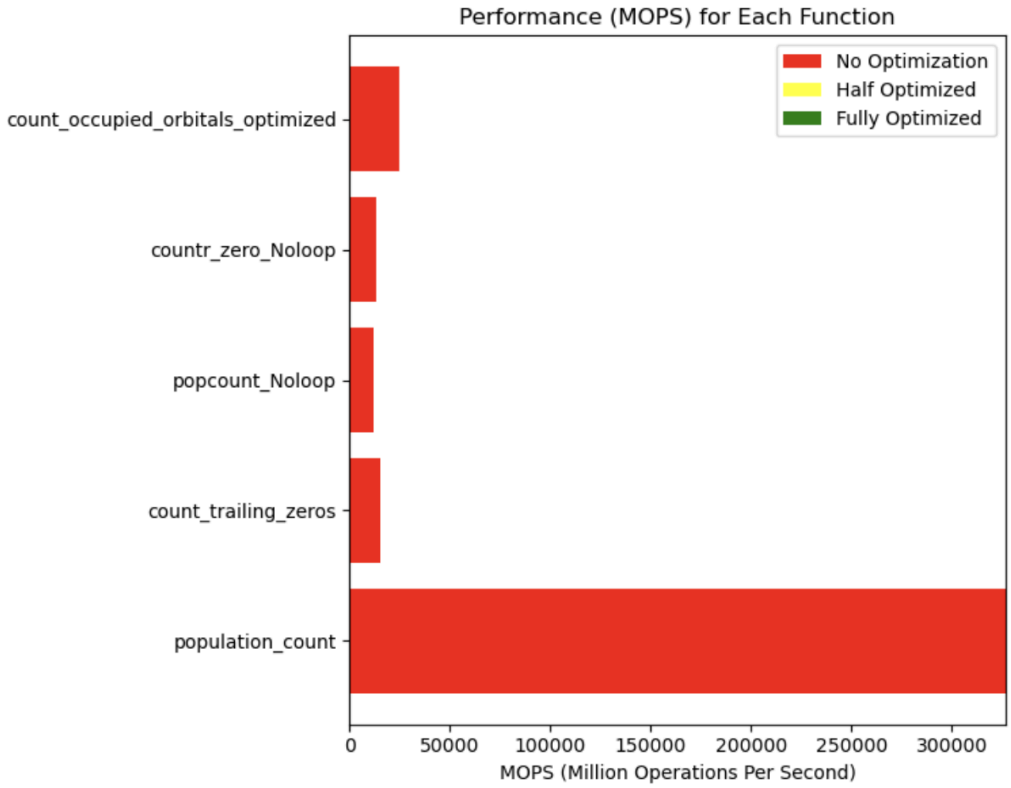

Figure 2. Million Operations Per Second

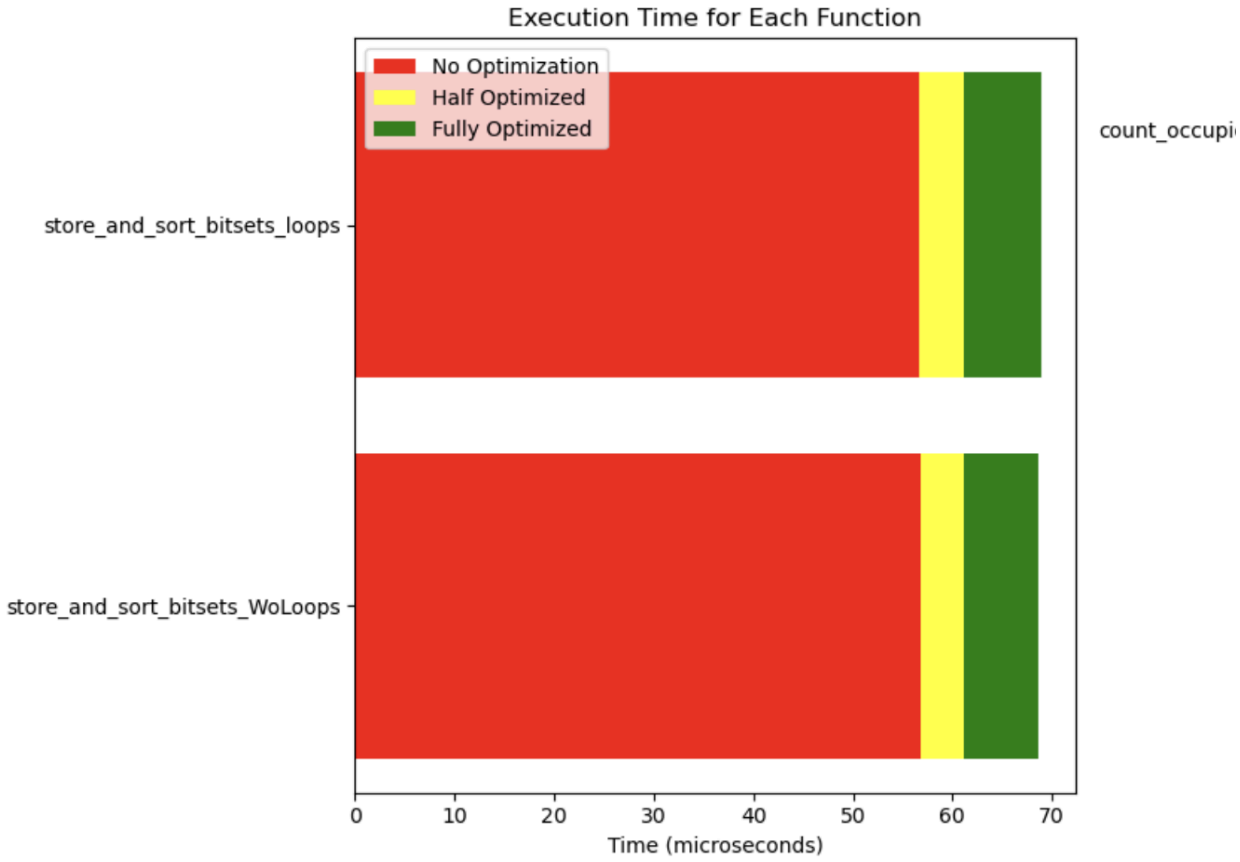

Figure 3. Execution of functions Comparison with and without loop

The analysis of execution time and performance in terms of Millions of Operations Per Second (MOPS) across different optimization strategies offers a comprehensive understanding of compiler optimization effects on code efficiency. Figures 2 and 3 illustrate the execution time for various functions and their respective performance, indicating how different levels of optimization affect the processing speed of bitwise operations.

Figure 2 (on the left) presents the execution time of different functions—store_and_sort_bitsets_loops and store_and_sort_bitsets_WoLoops—under three conditions: "No

Optimization," "Half Optimized," and "Fully Optimized." As expected, the execution times are significantly higher with no optimization and decrease with half and full optimization. The store_and_sort_bitsets_WoLoops function has slightly better execution times than store_and_sort_bitsets_loops, suggesting that eliminating loops where possible may offer a performance advantage.Figure 3 details the performance in MOPS, focusing on count_occupied_orbitals_optimized, popcount_Noloop, and other similar functions. Performance is markedly lower without optimization and increases substantially with half and full optimizations. Intriguingly, the performance for popcount_Noloop and countr_zero_Noloop under full optimization does not show a drastic improvement from half to full optimization, which could point to the intrinsic operations being well-optimized even at lower optimization levels.The provided timing information translates into an interesting narrative when converted into MOPS. For instance, population_count and count_trailing_zeros functions under no optimization conditions show a significant difference in execution time (326918.29 microseconds and 15089.50 microseconds, respectively), which, when converted into MOPS (1 million iterations), yield ~ 3.06 MOPS and ~ 66.29 MOPS, respectively. This demonstrates the substantial impact of optimization: without it, these functions operate orders of magnitude slower. When fully optimized, these functions reach performance levels around 25 million operations per second (MOPS) and have a theoretical peak of 1 million MOPS, according to the recorded execution time of 0.00 microseconds. This exceptionally fast execution time might indicate that the compiler has removed the code entirely, assuming there are no side-effects to consider. For the store_and_sort_bitsets functions, even though the execution times increased from half-optimized to fully optimized, the differences are minor, which might suggest that these functions are not as sensitive to optimization, possibly due to other limiting factors in their

implementation.The count_occupied_orbitals_optimized function's timing shows that under no optimization, its performance is the slowest at about ~ 40.53 MOPS. Yet, under full optimization, the performance jumps to a theoretical ~25 MOPS (0.04 microseconds reported time), highlighting the critical role that optimization plays in computational tasks related to quantum mechanics and electronic structure calculations.