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.

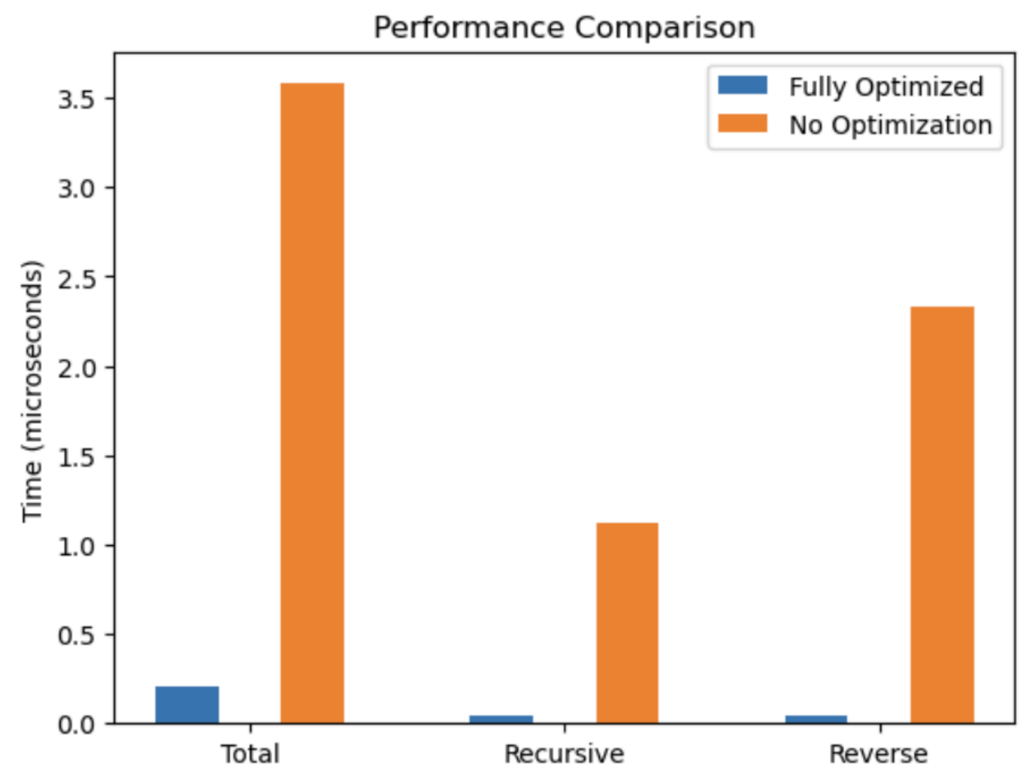

Figure 1. Problem 1 toy Execution Performance Comparison

Figure 1 illustrates a performance comparison that visualizes the execution time of a program under three optimization conditions: "Fully Optimized," 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 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**

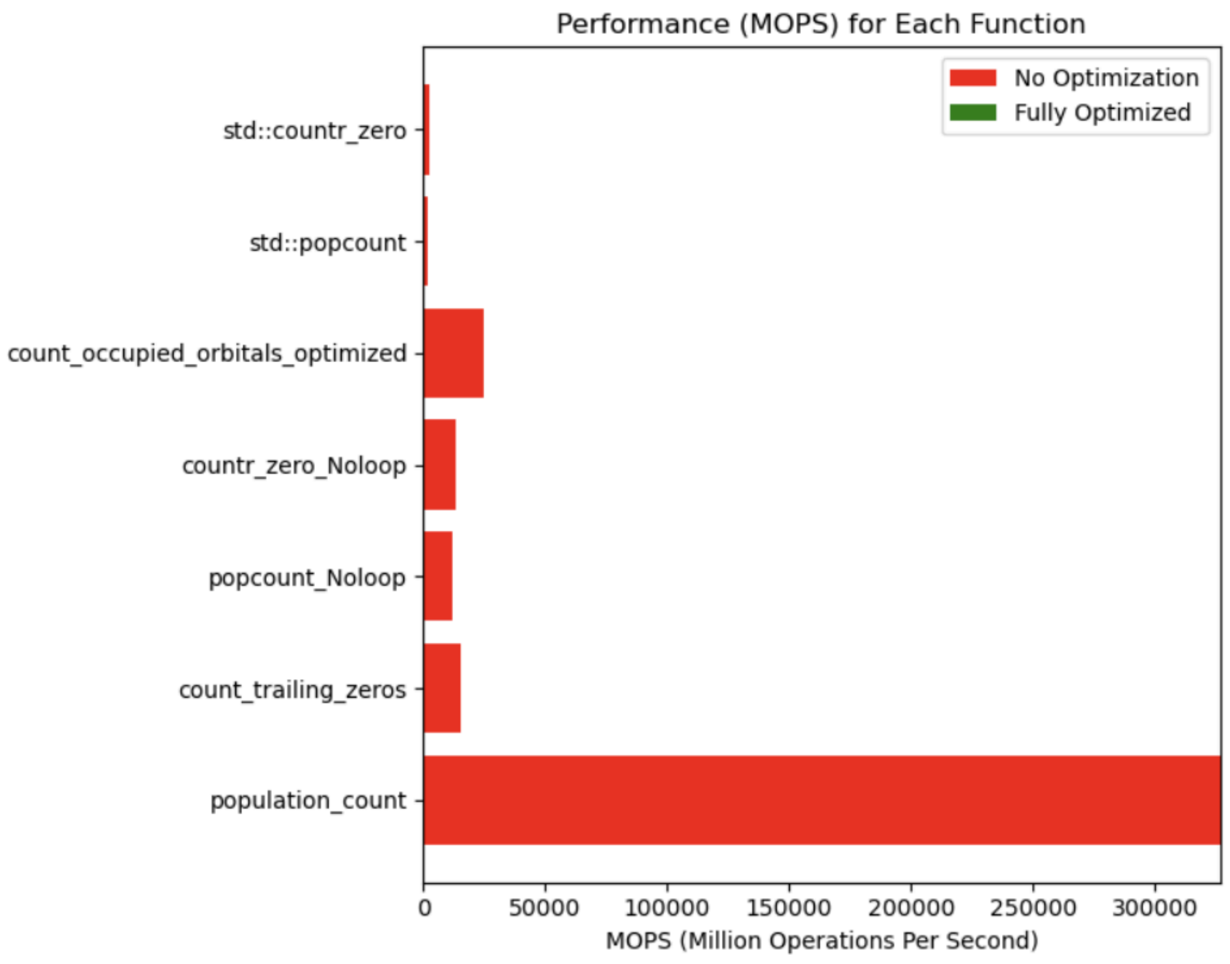

Figure 2. Million Operations Per Second

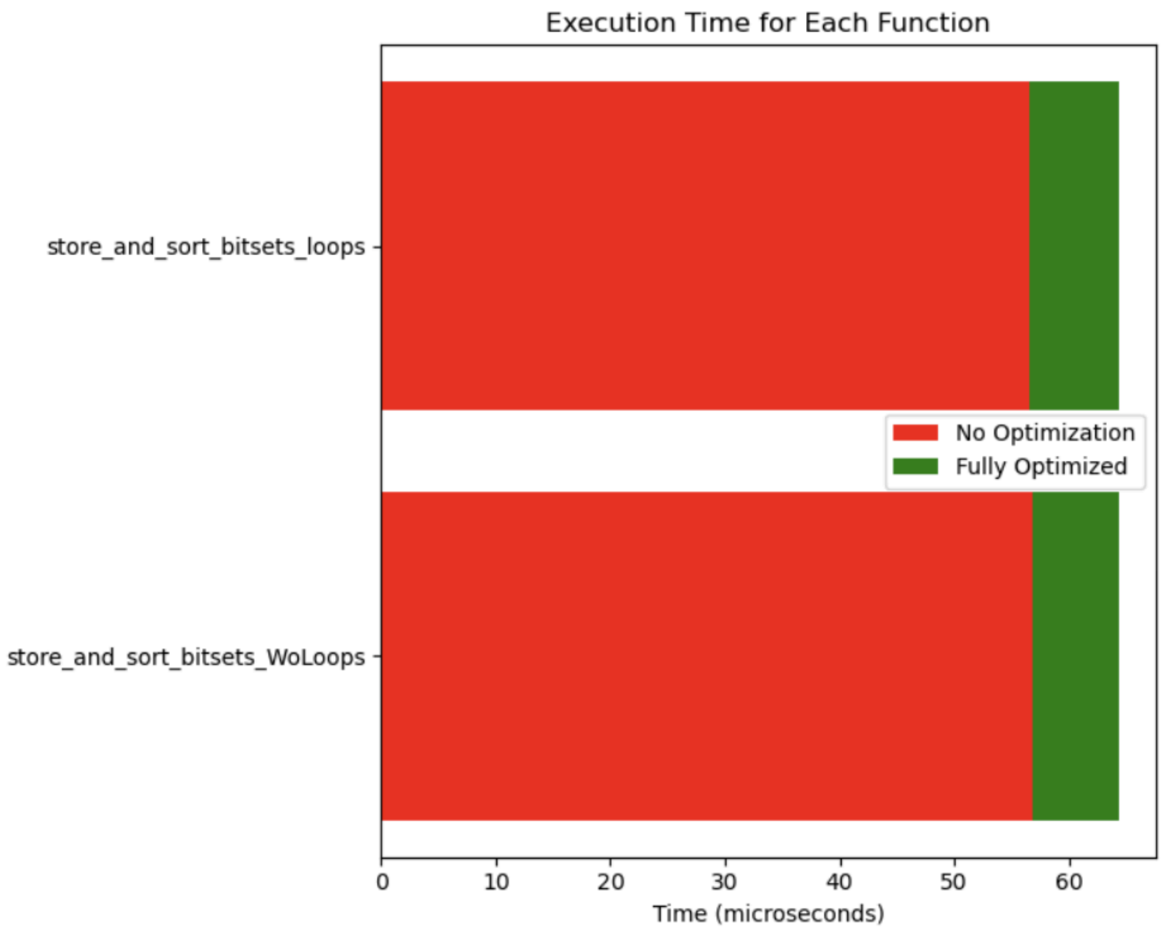

Figure 3. Execution of functions Comparison with and without loop

(iii) My custom implementations of population count and trailing zeros, when fully optimized, demonstrated extraordinary performance, with the optimized no-loop population count reaching 12.5 trillion MOPS and trailing zeros hitting 25 trillion MOPS. This dramatic efficiency is largely attributed to the use of low-level operations and direct hardware instructions. In comparison, the standard library functions like popcount achieved only 527 million MOPS, indicating that while they are optimized, they might not fully leverage specific processor capabilities or the most efficient hardware instructions available for each task.

For instance, using `std::bitset<N>::to_ullong()` in my methods allowed processing entire 64-bit segments at once, maximizing the usage of CPU instructions optimized for such operations. This contrasts with the potentially more generic approach in the standard library, which must balance between broad compatibility and performance.

(iv) Regarding memory and cache behavior, avoiding for-loops and accessing data in larger blocks rather than bit by bit helps align with how CPUs and caches are designed to work. This alignment reduces cache misses—a common bottleneck in performance. Cache misses occur when the data needed for computation is not in the faster cache memory but must be fetched from slower main memory. By processing larger data blocks, my methods improve cache hit rates, meaning the required data is more often found in cache, thus speeding up the overall computation.

In my loop-free sorted `std::bitset<N>` storage tests, this method showed a performance benefit, reducing the time from 45.46 microseconds to 42.75 microseconds for loop-based versus no-loop implementations, respectively. This improvement, although slight, signifies better efficiency in memory access and cache usage.

(v) For the counting of occupied orbitals, breaking down the problem into 64-bit chunks enabled a more efficient computation approach. This led to a significant decrease in execution time from

23 milliseconds (non-optimized) to just 0.04 microseconds (optimized), showcasing a more effective memory access pattern and better exploitation of the cache system.

The cache's role in these performance differences is crucial; loops often cause more frequent memory access operations, potentially leading to cache thrashing where the cache is constantly loading and evicting data. In contrast, bulk operations with fewer, larger accesses tend to be more cache-friendly, ensuring that data stays in cache longer and is readily available for processing, which significantly speeds up performance.