Laboratory 2. Profiling: Instrumented and Event-based Techniques

Beatriz Rosell Cortés
María Peribáñez Tafalla

October 2024

1 Time measurements using `gettimeofday()`

The `gettimeofday()` function captures system time with microsecond precision, allowing for detailed benchmarking of specific code sections. While the `time` command provides a comprehensive view of overall execution time (including real, user, and system time), `gettimeofday()` can measure individual parts of a program, like matrix multiplication, offering greater granularity.

In this exercise, we measured the execution time of standard matrix multiplication and the Eigen library's optimized version for matrices of varying sizes. We conducted five runs per matrix size to mitigate the effect of outliers, calculating mean values and standard deviations. As shown in Table 1, the time for memory allocation is minimal compared to the time spent on computation, particularly for larger matrices.

For example, for a 2000x2000 matrix, the allocation time in standard multiplication is approximately 0.146 seconds, while the multiplication time is 156.7 seconds. This indicates that most of the time is spent on computation rather than on memory allocation. The allocation time for Eigen is slightly lower (0.106 seconds), but the multiplication time is significantly reduced to 2.5 seconds, highlighting the efficiency of the Eigen library in both computation and memory management.

	Standard Multiplication				Eigen Math Library			
	Allocation		Multiplication		Allocation		Multiplication	
Matrix size	Mean (s)	Std. Dev (s)	Mean (s)	Std. Dev (s)	Mean (s)	Std. Dev (s)	Mean (s)	Std. Dev (s)
2000	0.1459	0.0010	156.716	10.1697	0.1063	0.0011	2.5153	0.0052
1000	0.0385	0.0030	17.0417	0.1073	0.0263	0.0004	0.3195	0.0044
500	0.0091	0.0002	1.9334	0.0067	0.0070	0	0.0424	0.0010

Table 1: Mean and Standard Deviation of Execution Times for the gettimeofday() function.

The relative overhead of initialization decreases as matrix size increases. This is because, while allocation time remains relatively low, the multiplication time grows significantly, leading to computation dominating overall execution time.

In comparing the system, user, and real times from the `time` command, we find that:

- **User Time**: Mainly reflects matrix multiplication computation, aligning closely with `gettimeofday()` results. This time represents the CPU time spent in user-mode code (outside the kernel).
- **System Time**: Related to memory allocation and system calls. Allocation time measured with `gettimeofday()` is linked to this. This time represents the CPU time spent in kernel-mode code (inside the kernel).
- **Real Time**: Includes both user and system time, closely mirroring the total times captured by `gettimeofday()`. This is the wall-clock time from start to finish of the command.

When comparing the user time from the 'time' command with the times obtained using gettimeofday(), the differences are less pronounced but still noticeable. For standard multiplication of a 2000x2000 matrix, the user time averages 66.2 seconds, while with gettimeofday() it iss 156.7 seconds. In the case of Eigen, both methods are more consistent, with 3.8 seconds for user time and 2.5 seconds for gettimeofday(). This suggests that the overhead in standard multiplication is captured differently by each method, but Eigen shows closer results, indicating better optimization and CPU usage. The variation in the standard multiplication results could also be due to the optimizations applied during compilation with the -O2 flag, which improves code performance by eliminating redundant code and optimizing loops, potentially reducing execution time.

The results clearly demonstrate the efficiency of the Eigen library in handling large matrix multiplications. The significant reduction in multiplication time with Eigen, compared to standard multiplication, underscores the importance of optimized libraries in high-performance computing tasks. The minimal allocation time further emphasizes that the primary bottleneck in matrix operations is the computation itself, not the memory management.

2 System Call Analysis using `strace`

Using `strace` with the `-c` option, we analyzed the system calls made by both the standard matrix multiplication program and the Eigen-based implementation. The outputs from `strace` reveal differences in system interaction, particularly regarding memory management.

System Call	Standard Multiplication	Eigen
`brk`	207 calls, 0.000294 sec	3 calls, 0.000002 sec
`munmap`	1 call, 0.000006 sec	6 calls, 0.000803 sec
`openat`	29 calls, 0.000155 sec	29 calls, 0.000038 sec
`mmap`	14 calls, 0.000064 sec	19 calls, 0.000077 sec
`mprotect`	10 calls, 0.000050 sec	10 calls, 0.000050 sec

Table 2: Comparison of key system calls between standard and Eigen implementations.

2.1 Memory Management Insights

The standard implementation makes significantly more calls to `brk` (207 calls), indicating frequent dynamic memory allocation. In contrast, Eigen's optimized memory management reduces this to only 3 calls, minimizing system overhead. Eigen achieves this efficiency by internally managing memory, reducing the need for system calls.

However, the Eigen-based implementation performs more `munmap` calls (6 compared to 1), suggesting more frequent memory deallocation. Despite this, the time spent on `munmap` (0.000803 sec) is still minimal. The `mmap` and `mprotect` calls remain similar in both implementations, suggesting comparable levels of memory protection.

Overall, the total time spent on system calls is minimal in both cases. For standard multiplication, system calls took 0.000714 seconds, while Eigen's system call time was slightly higher at 0.001007 seconds. This small overhead indicates that the bulk of the execution time is spent on computation rather than system interactions.

The difference in system calls underscores the improved memory efficiency of Eigen, which allocates and deallocates memory more strategically, leading to faster execution times and reduced system load. By reducing the number of `brk` calls, Eigen minimizes the overhead associated with frequent memory allocation, which can be a significant bottleneck in high-performance computing tasks. Additionally, the slight increase in `munmap` calls in Eigen suggests a more aggressive memory deallocation strategy, which helps in maintaining a lower memory footprint during execution.

In summary, the system call analysis using `strace` highlights the superior memory management of the Eigen library, contributing to its overall performance gains. The reduced number of system calls and the efficient handling of memory operations are key factors in the enhanced computational efficiency observed with Eigen.

3 Hardware Profiling with `perf`

The `perf` tool was used to capture hardware-level performance data. We compared the execution of standard matrix multiplication with the Eigen-based implementation for a large matrix (1000x1000).

Metric	Standard Multiplication	Eigen
Task Clock (ms)	16,927.17	347.71
Instructions	122,322,911,710	3,355,077,283
Cycles	55,731,243,223	1,139,855,153
Instructions per Cycle (IPC)	2.19	2.94
Branch Misses	1,331,438	297,550

Table 3: Key `perf` metrics comparing standard and Eigen implementations.

3.1 Performance Insights

Using perf, we examined hardware-level performance metrics to understand how Eigen's optimizations translate to actual CPU efficiency. The results show a dramatic reduction in execution time and resource consumption when comparing the Eigen-based implementation to the standard matrix multiplication.

- **Task Clock:** Eigen completed the task in just 347.71 ms, whereas the standard method took 16,927.17 ms, a difference of nearly 50x. This shows how Eigen minimizes unnecessary operations, focusing purely on essential computation.
- **Instructions and Cycles:** Eigen executed 3.35 billion instructions compared to 122.32 billion in the standard case, while using only 1.13 billion CPU cycles versus 55.73 billion. Fewer instructions and cycles directly correlate to reduced task duration, highlighting Eigen's streamlined code execution.
- **Instructions per Cycle (IPC):** The IPC metric is crucial for understanding CPU efficiency. The Eigen implementation achieved an IPC of 2.94, meaning it executed nearly 3 instructions per CPU cycle, compared to the standard multiplication's 2.19. This difference reflects a more optimized use of CPU resources in Eigen, reducing the time the CPU spends waiting for instructions or data.
- **Branch Misses:** With only 297,550 branch misses compared to 1.33 million in the standard multiplication, Eigen significantly reduces mispredictions, which cause the CPU to waste cycles correcting them. This means the Eigen-based implementation is better at keeping the CPU pipeline full and running efficiently, without unnecessary stalls or interruptions.

The Eigen implementation demonstrates a significant improvement in performance, executing in approximately 2% of the time required by the standard multiplication method. The increase in IPC from 2.19 to 2.94 indicates more efficient use of CPU cycles, while the reduction in task clock time and branch misses shows better overall optimization.

The lower number of cycles for Eigen also correlates with reduced stalled frontend cycles, meaning the CPU spent less time waiting for instructions to be fetched. This is critical for performance in computationally intensive tasks like matrix multiplication.

4 Conclusion

Through this exercise, we have observed the substantial performance improvements offered by the Eigen library over standard matrix multiplication. Using `gettimeofday()`, we quantified the execution times, showing that Eigen significantly reduces computation time. System call analysis with `strace` revealed that Eigen's optimized memory management reduces the overhead of dynamic memory operations. Finally, `perf` profiling highlighted Eigen's superior CPU utilization, with higher instructions per cycle and fewer stalls, leading to faster execution times. Together, these profiling techniques provide a comprehensive understanding of performance bottlenecks and optimizations.