Assignment 2

Profiling: Instrumented and Event-based Techniques.

David Padilla Orenga
Ignacio Pastore Benaim

1 Introduction

The goal of this assignment is to evaluate and compare the performance of two matrix multiplication implementations: one using a standard 2D array approach in C++ and another using the Eigen library. Performance metrics such as execution time are analyzed using both the `clock()` function and the `gettimeofday()` system call for different matrix sizes and compared to the metrics of the `time` command. The results help to draw conclusions about the efficiency of the two approaches, particularly in how they handle matrix initialization, memory allocation, and multiplication.

2 Methodology

The matrix multiplication implementations were evaluated by varying the matrix size ($N = 100, 250, 500, 600, 700, 800, 900, 1000, 1200, 1500$) for both approaches. Each matrix size was tested 10 times to ensure statistical accuracy. We used the `-O2` optimization level, as it is widely employed in the industry and demonstrated excellent performance in the previous assignment without being as aggressive as `-O3`. The tests were conducted on Ubuntu 20.04 LTS, running on an Intel Core i7-6700HQ with x86_64 architecture.

3 Results and Discussion

The `clock()` function measures CPU time used by the program, offering insights into the computational cost incurred by the CPU during the entire execution of the program, including matrix initialization, memory allocation, and matrix multiplication. On the other hand, `gettimeofday()` offers wall-clock time, which includes the real elapsed time of program execution. We have instrumented the code to capture the initialization and multiplication phases separately using `gettimeofday()`, allowing us to break down the time taken by each part. The results of the matrix multiplication timing tests for various matrix sizes are summarized in Figure 1.

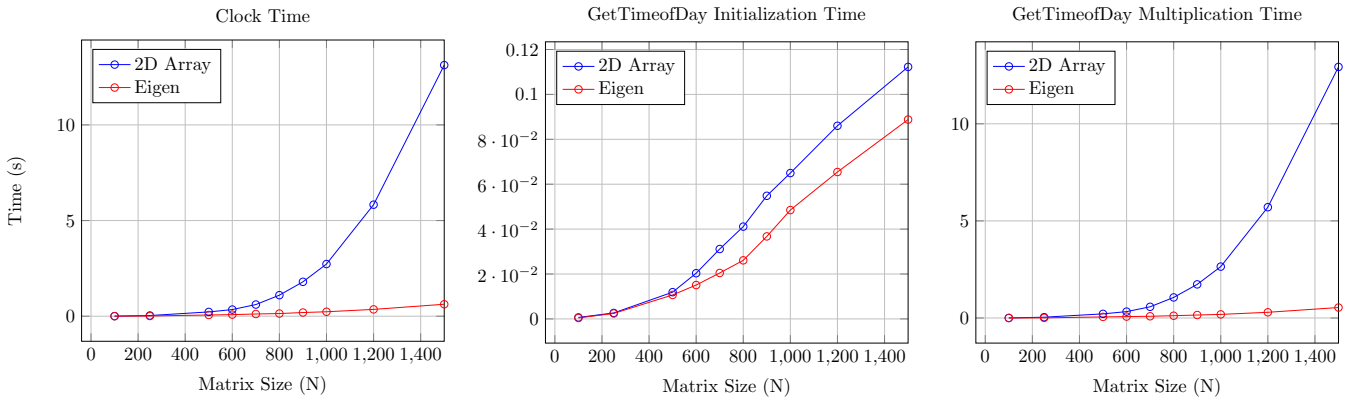

Figure 1: Comparison of clock, initialization, and multiplication times for both implementations.

3.1 Clock and Time Command Comparison

Figure 1 shows the clock time registered by the tests, with the Eigen-based implementation significantly outperforming the 2D array approach, consistent with the results observed in the previous lab. Although theoretically, following our clock implementation, the `clock()` function is expected to measure exclusively the CPU time used by the process,

experiments have shown that its values are closer to the real time than to the user time, as shown in Table 1. This may be due to the fact that `clock()` includes not only user time, but also external factors such as I/O locks or waiting for resources in multitasking systems. Another possible explanation for the discrepancy is because it has lower accuracy, as it measures time in CPU ticks, the granularity of which varies by operating system. In contrast, the `system` time remains relatively low across all matrix sizes, indicating minimal involvement of kernel-level operations such as memory management or I/O during matrix multiplication. The standard deviation for the `clock()` times remains small, indicating consistent CPU usage across runs.

Executable	**N**	**t_clock (s)**	**t_real (s)**	**t_user (s)**	**t_sys (s)**
Eigen mult	500	0.063 ± 0.007	0.062 ± 0.007	0.052 ± 0.010	0.005 ± 0.005
Eigen mult	700	0.115 ± 0.011	0.113 ± 0.014	0.101 ± 0.009	0.007 ± 0.006
Eigen mult	1000	0.231 ± 0.007	0.228 ± 0.007	0.196 ± 0.009	0.029 ± 0.009
Eigen mult	1200	0.355 ± 0.017	0.356 ± 0.017	0.320 ± 0.022	0.030 ± 0.009
Eigen mult	1500	0.624 ± 0.005	0.625 ± 0.007	0.569 ± 0.015	0.048 ± 0.013
2D matrix	500	0.222 ± 0.002	0.221 ± 0.003	0.217 ± 0.004	0.000 ± 0.000
2D matrix	700	0.608 ± 0.034	0.606 ± 0.035	0.598 ± 0.035	0.002 ± 0.004
2D matrix	1000	2.726 ± 0.076	2.726 ± 0.076	2.703 ± 0.077	0.016 ± 0.008
2D matrix	1200	5.828 ± 0.039	5.831 ± 0.040	5.791 ± 0.038	0.034 ± 0.006
2D matrix	1500	13.125 ± 0.207	13.129 ± 0.207	13.076 ± 0.205	0.045 ± 0.008

Table 1: Resume for the clock tests for both implementations including the mean and standard deviation.

3.2 Gettimeofday Analysis

In this section, we examine the performance of the matrix multiplication implementations by separately measuring the time spent on initialization and multiplication using `gettimeofday()`. Figure 1 displays the initialization and multiplication times for the Eigen-based and 2D array implementations.

As seen in Figure 1, the initialization time remains relatively small compared to the multiplication time for both implementations (two orders), even as the matrix size increases. For small matrix sizes ($N < 600$), both methods are comparable, but as N increases, the implementation with Eigen scales more efficiently. Regarding the multiplication time, the implementation with Eigen is considerably more efficient resulting in significantly lower execution time, indicating that advanced optimizations and better memory management are leveraged compared to the 2D array in C++, as seen in lab 1. It can be observed for high N values, how multiplication times for the 2D array version are increasing exponentially, spending most of the execution time in multiplication, not initialization.

Executable	**N**	**t_init (s)**	**t_mult (s)**	**t_real (s)**	**t_user (s)**	**t_sys (s)**
Eigen mult	500	0.010 ± 0.000	0.056 ± 0.002	0.064 ± 0.005	0.053 ± 0.004	0.007 ± 0.004
Eigen mult	700	0.020 ± 0.002	0.088 ± 0.006	0.108 ± 0.004	0.091 ± 0.012	0.013 ± 0.009
Eigen mult	1000	0.048 ± 0.003	0.184 ± 0.010	0.231 ± 0.011	0.197 ± 0.014	0.031 ± 0.008
Eigen mult	1200	0.065 ± 0.004	0.293 ± 0.013	0.359 ± 0.017	0.317 ± 0.017	0.034 ± 0.009
Eigen mult	1500	0.088 ± 0.001	0.538 ± 0.004	0.627 ± 0.006	0.572 ± 0.013	0.050 ± 0.014
2D matrix	500	0.011 ± 0.002	0.212 ± 0.004	0.222 ± 0.006	0.218 ± 0.006	0.000 ± 0.000
2D matrix	700	0.031 ± 0.001	0.575 ± 0.031	0.604 ± 0.030	0.594 ± 0.031	0.002 ± 0.004
2D matrix	1000	0.064 ± 0.004	2.645 ± 0.074	2.710 ± 0.075	2.682 ± 0.076	0.020 ± 0.008
2D matrix	1200	0.086 ± 0.001	5.704 ± 0.050	5.791 ± 0.049	5.749 ± 0.051	0.038 ± 0.007
2D matrix	1500	0.112 ± 0.002	12.928 ± 0.227	13.043 ± 0.226	12.989 ± 0.226	0.043 ± 0.009

Table 2: Resume for the gettimeofday tests for both implementations including the mean and standard deviation.

Conclusion: The overhead of the initialization phase remains minimal for both implementations, though it becomes more significant in the standard matrix implementation as matrix size increases. The Eigen-based implementation consistently demonstrates lower multiplication times, thanks to its optimized memory access and matrix operations. The separation of initialization and multiplication times helps identify where optimizations are most effective, with the multiplication phase dominating the overall execution time. The `gettimeofday()` function measures time with high precision in microseconds, recording the elapsed time based on the system clock. By adding the initialization time and the multiplication time, we obtain a measure very close to real time (*real time*), since `gettimeofday()` captures both the CPU time used and any other system activity that affects the total execution time. This makes it an accurate and reliable measure for evaluating program performance.

3.2.1 strace System Call Analysis

We used `strace` to analyze the system call behavior of both implementations. Table 3 shows the most frequent system calls during the execution of both the Eigen-based and standard implementations with a matrix size of $N = 1500$. The system calls analyzed include memory management-related calls (e.g., `mmap`, `brk`), I/O calls (e.g., `read`, `write`), and process management calls (e.g., `execve`, `mprotect`).

Executable	System Call	Calls	Time (s)	Percentage (%)
Eigen-based (N=1500)	`munmap`	7	0.005472	85.85%
Eigen-based (N=1500)	`mmap`	28	0.000429	6.73%
Eigen-based (N=1500)	`mprotect`	7	0.000152	2.38%
Standard (N=1500)	`brk`	392	0.001172	99.49%
Standard (N=1500)	`write`	2	0.000004	0.34%
Standard (N=1500)	`fstat`	6	0.000002	0.17%

Table 3: System calls and execution time percentages for Eigen-based and standard implementations using `strace`.

Most of the time (99.49%) is spent in the brk call, which is used to adjust the heap size. This indicates that the standard implementation relies heavily on dynamic memory allocation. Calls to other functions such as write, fstat, and mmap are negligible in terms of total time. A total of 454 system calls were reported, reflecting a moderate use of system resources. About the Eigen usage, the most used call is `munmap`, which frees previously allocated memory, occupies 85.85% of the total time. This suggests that the implementation with Eigen performs more intensive memory management, especially in freeing resources. Also noteworthy is the frequent use of `mmap` (6.73%) and `mprotect` (2.38%), indicating more dynamic memory management compared to the standard implementation. In total, there were 77 system calls, considerably fewer than the 454 calls in the standard implementation. This suggests that Eigen optimizes the number of interactions with the system, even though the time cost per call is higher.

Conclusion: The analysis suggests that the standard implementation makes more system calls, especially for memory allocation, while Eigen reduces the total number of calls, but spends more time on memory release and management. These differences reflect specific optimizations in the Eigen implementation, which improves performance at the expense of higher cost in individual memory management operations.

3.2.2 perf Hardware Performance Metrics

To better understand the hardware-level efficiency of the implementations, we used `perf` to measure key performance metrics, including CPU cycles, instructions per cycle, and branch misses. Table 4 provides a comparison of these metrics for both the Eigen-based and standard matrix implementations with $N = 1500$.

Metric	Eigen-based (N=1500)	Std Dev	Standard (N=1500)	Std Dev
Task Clock (ms)	657.17	1.72%	13,561.47	2.64%
CPU Cycles	2,066,738,370	0.38%	33,811,730,234	1.68%
Instructions	6,234,227,228	0.00%	30,810,076,435	0.00%
Instructions per Cycle	3.00	0.00%	0.88	0.00%
Branch Misses (%)	0.24%	25.12%	0.07%	0.94%

Table 4: Comparison of hardware performance metrics using `perf`.

The Eigen-based implementation executes fewer CPU cycles compared to the standard implementation, highlighting its optimized algorithmic efficiency. The instruction-per-cycle (IPC) metric shows that the Eigen-based implementation makes better use of CPU resources, with 3 instructions executed per cycle, compared to only 0.88 instructions per cycle for the standard implementation. The Eigen-based implementation incurs a higher percentage of branch misses, but thisis likely due to the higher complexity of its optimizations, which involve more branching. The standard matrix implementation consumes significantly more CPU time, as indicated by the task clock and CPU cycle counts. This correlates with its longer real time and user time observed in the previous sections.

Conclusion: The Eigen method demonstrates higher efficiency at the hardware level, with fewer cycles and higher IPC, making it more suitable for larger matrix sizes. However, the standard implementation, while simpler, incurs significant overhead due to its less optimized use of CPU resources. As conclusion, the advantages of using the Eigen library for matrix multiplication, especially for larger matrix sizes are clear, outperforming the standard matrix implementation across all metrics, including CPU time, real time, and hardware efficiency.