Laboratory 2 - Report

Group Members:
Javier Almario, NIP: 962449
Alvaro Provencio, NIP: 960625

1 Exercise 1

We have done again 3 measurements per test and showing the mean of those times. The function *clock*() tells the number of cycles it is taking the CPU to execute that particular process (user + system time), that is why in this laboratory we are only making the comparison with the Real time that the *time* command shows.

1. Clock()				
	Dimension	***Real Time***	***Clock() Time***	***Real Time - clock() Time***
Standard	100	0.0090	0.0029	0.0061
	500	0.1633	0.1575	0.0059
	750	0.3777	0.3720	0.0057
	1000	1.0420	1.0346	0.0074
	1250	2.7297	2.7213	0.0083
	1500	4.9100	4.9020	0.0080
	1750	10.1850	10.1742	0.0108
	2000	17.5820	17.5642	0.0178
Eigen	100	0.0093	0.0027	0.0066
	500	0.1423	0.1360	0.0063
	750	0.2023	0.1951	0.0073
	1000	0.3470	0.3400	0.0070
	1250	0.5433	0.5354	0.0079
	1500	0.8280	0.8199	0.0081
	1750	1.1935	1.1950	0.0088
	2000	1.7203	1.7099	0.0104

Table 1: Comparison of Real Time and clock() Time for Standard and Eigen methods.

Here we can appreciate that both times are very similar, so *clock*() provides a good intuition about the time spent in the execution. However, it is worth noting that the Real time is always slightly higher than the value obtained by the function. This happens because Real time also includes the time spent executing the *clock*() function itself, computing the time calculation, the *cout* call, and the time spent initializing and ending the program. This proves that using time-measurement functions increases the duration of the program, though it is small as seen in the 'difference' column.

Lastly, it is worth to mention that the time showed in the 'difference' column is more or less the same on all executions, but seems to increase slightly proportional to the number of dimensions. This growth could be related again to the memory management even though the *delete*() function is executed before the *clock*() call (in the standard algorithm), because the OS may still be handling the cleanup after the measurement.

2 Exercise 2

We did the same set of tests with the function $gettimeofday()$, which, on the other hand, shows the UTC time from 1970, so taking measurements with this command contains the elapsed time from one call to the other. Because the measurement is during execution, as on the previous case, the behavior expected comparing it to Real time is the same as before.

Since in this test we are timing the matrix multiplication block apart from the rest of the actions (declaration and, memory allocation and deallocation), we can confirm looking at Table 2 that the time spent in the program is mostly affected by the matrix multiplication. Hence, the relative overhead of the initialization code block decreases as the number of dimension increases, although they affect it to grow slightly.

2. GetTimeOfDay()					
	Dimension	***Real Time***	***Declaration Time***	***Matrix mult. time***	**Real Time - declaration time - matrix mult. time**
Standard	100	0.010	0.001	0.002	0.0070
	500	0.190	0.020	0.164	0.0060
	750	0.314	0.024	0.284	0.0060
	1000	1.064	0.079	0.978	0.0070
	1250	2.786	0.063	2.717	0.0060
	1500	4.985	0.062	4.830	0.0090
	1750	10.257	0.111	10.136	0.0100
	2000	17.406	0.158	17.236	0.0119
Eigen	100	0.009	0.001	0.002	0.0068
	500	0.040	0.009	0.025	0.0058
	750	0.106	0.024	0.076	0.0060
	1000	0.228	0.031	0.190	0.0070
	1250	0.424	0.038	0.379	0.0070
	1500	0.712	0.068	0.635	0.0090
	1750	1.122	0.103	1.009	0.0094
	2000	1.638	0.122	1.505	0.0113

Table 2: Comparison of Real Time, Declaration Time, and Matrix Multiplication Time using gettimeofday() for Standard and Eigen.

Talking about the returned values on both functions, with $gettimeofday()$ we are just observing the moment in time when it is called, but $clock()$ tells only the execution time, so it is expected the $clock()$ time to be lower than the other. Because this program is not complex we appreciate that both times are very similar (considering also that there is variance on each test), but this effect would be noticeable if they were more processes requesting the CPU at the same time or some action that would have the execution waiting.

3 Exercise 3

In this part, we execute our programs for $N = 2000$ with the command `strace -c ./matrix`. *Strace* lets us see all those interactions in real time, basically what the program is doing at the lower level[1]. We selected the most relevant system calls; for this purpose we based our selection on three categories:

- **Memory Management:** mmap, munmap, mprotect, brk.
- **File / Library Operations:** openat, close, fstat, read, pread64
- **Program Setup / Execution:** execve, arch_prctl, set_tid_address, set_robust_list, rseq

The results for both executions are shown in the table 3, where we selected 5 system calls that give us a significant data to compare between them.

As a first overview on the table we can notice that for the standard matrix all the percent of the time is used by munmap, in constrast, the time used by Eigen is divided into different Syscalls. It tells us that the heavy computations for standard matrix is happening inside user-space code not inside system calls, since $strace - c$ only measures time spent in system calls, strace attributes 100% of the time to the munmap call.

	standard		Eigen	
Syscall	**%time**	**calls**	**%time**	**calls**
munmap	100.00	2	72.12	6
mmap	0.00	25	9.27	27
execve	0.00	1	11.11	1
openat	0.00	5	1.78	5
mprotect	0.00	6	1.24	6

Table 3: Results when the programs are executed with the command strace

The difference in the number of calls (2 vs 6) and the time proportion suggests very different memory allocation and deallocation patterns between the two implementations. Standard implementations allocate arrays contiguously on the heap (new[]), which may result in fewer mmap/-munmap calls. Eigen may allocate temporary aligned blocks for vectorized operations.These aligned blocks are often allocated using mmap, hence more frequent mapping/unmapping calls. For library operation Both programs open the same 5 system libraries at startup, but Eigen spends more time inside those calls due to heavier template showing how loads shared libraries; using more dynamic linking. Finally, the difference in time for *excecve* tells us that Eigen's binary is larger and dependency setup are more complex, so loading it into memory takes longer.

4 Exercise 4

Metric	**Standard**	**Eigen**
Task-clock (ms)	17,255	1,691
IPC	0.54	2.90
Instructions ($\times 10^9$)	36.76	18.91
Branches ($\times 10^9$)	4.10	0.18

Table 4: Comparison of key `perf` metrics between the standard matrix and Eigen implementations.

We executed both programs with the command *perf* and picked up 4 metrics, that were the most relevant because they reveal meaningful architectural or algorithmic differences between two programs. From the table 4 we can see an overall performance with Task-clock where Eigen finishes in $\approx 10\times$ less CPU time, confirming the same speedup seen in the *gettimeofday* results. Eigen achieves over $\approx 5\times$ higher IPC, meaning it keeps the CPU pipelines much busier, the possible cause is because of vectorized (SIMD) execution, this tell us how well the CPU executes instructions per cycle. From instrunction we can say that Eigen executes $2\times$ fewer instructions, this metric shows how much work the CPU had to do. Eigen accomplishes the same matrix multiplication with about half the instructions, meaning the compiler-generated code is much more compact and optimized. Finally when the code uses many nested loops, the CPU has to check the loop condition for every single iteration, each of those checks is called a branch [2]. Eigen uses $\approx 23\times$ less tight loops with minimal branching, reducing pipeline stalls.

References

[1] strace(1) — Linux manual page, `https://man7.org/linux/man-pages/man1/strace.1.html`, 2025.

[2] Branch predictor, `https://en.wikipedia.org/wiki/Branch_predictor?utm_source=chatgpt.com`, 2025.