Laboratory 1 - Report

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

1 Computational complexity

We measured the time using two programs, the first one without any libraries and the second one with the Eigen library. For the first one, we performed the multiplication of the two $n \times n$ matrices with three nested loops. The algorithm is based on the following equation:

$$C[i][j] = \sum_{k=1}^{n} A[i][k] * B[k][j] \tag{1}$$

C requires n multiplications and has n^2 inputs, so it requires n^3 multiplications. For count additions, to add n numbers we need only $n-1$ addition, but C has n^2 entries, so the number of additions is $n^2 \cdot (n-1) = n^3 - n^2$, as a rule the total time complexity is the largest time complexity, thus its computational complexity is $O(n^3)$ [1]. The second one uses Eigen library, it maps the $A * B$ expression to a single GEMM (General matrix multiplication) primitive, so the library can merge the work and avoid temporaries. Inside that GEMM, Eigen packs panels of A and B into contiguous buffers and computes in cache-sized blocks, then feeds each block pair to a small micro-kernel, they are architecture-tuned and vectorized [2].

Both programs have the same complexity $O(n^3)$, but Eigen reuses data from fast cache and executes wide vector ops for better CPU utilization.

2 Memory allocation

The tests have been done in an Intel Core i5-8300H CPU @ 2.30GHz and for this CPU the stack size is limited to 8Mb so it will not be able to allocate 3 matrices for dimensions greater than 600x600, approximately. For this reason, we have decided to define the matrices dynamically, to be able to test greater matrices and to have a comparison with the eigen library, that also allocates the matrices in the heap.

3 Time command on Linux

We have tested both programs with 7 different matrix dimensions (100, 500, 750, 1000, 1250, 1500, 2000) to have a better understanding of their performance. Each dimension has been tested 3 times per program to compute the mean time consumed in each case.

Taking a look at the mean times table, we can see that the execution time practically depends on the user time. This is because the system mode mainly takes care of the memory allocation at the beginning of the program, which does not take much time, but it is worth to notice a growth proportional to the matrices dimensions.

4 Analysis of Mean, Variance, and Run Consistency

Based on the graphic, we can confirm the exponential behavior expected, and comparing the two algorithms we see that for low dimensions, both programs have more or less similar execution times, but when increasing the dimensions from 750, the standard approach tends to take more time than the used in the Eigen library. This behavior reflects the difference between the standard algorithm with its 3 iterations per dimension and the eigen library with its optimization functions, that allows it to have that huge improvement in performance.

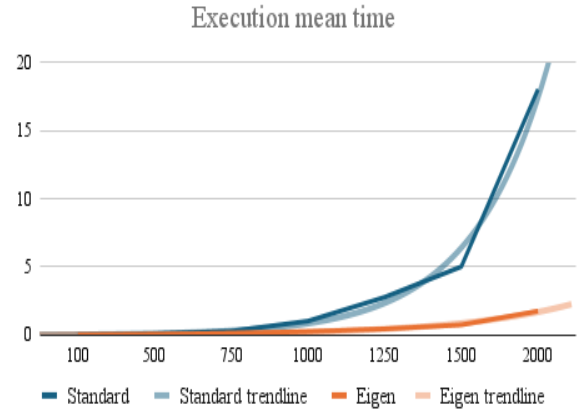

Figure 1: Execution mean time

	Dimension	Execution time	User time	System time	CV exec time
Standard	100	0,013	0,007	0,007	8,660
	500	0,107	0,103	0,005	3,370
	750	0,295	0,278	0,017	4,234
	1000	1,031	1,007	0,023	12,186
	1250	2,781	2,754	0,027	1,461
	1500	5,019	4,982	0,034	0,243
	2000	18,002	17,945	0,052	2,004
eigen	100	0,013	0,006	0,007	8,660
	500	0,061	0,051	0,011	21,311
	750	0,136	0,126	0,011	12,215
	1000	0,264	0,252	0,013	3,864
	1250	0,464	0,447	0,017	2,245
	1500	0,756	0,733	0,024	2,752
	2000	1,762	1,734	0,028	7,795

Figure 2: Mean times table

In addition, we computed the determination coefficient from a regression $R^2 \approx 0.98$ for normal matrix and $R^2 \approx 0.99$ for Eigen, which means that the model (time vs. n^3) explains about 98% and 99% of the variation respectively, it is a good fit on both, thus for the consistency we used coefficient of variation (CV). In matrix without Eigen consistency runs are stable $CV \leq 5\%$, Aside from $n = 1000$ it might be considered as noisy. The consistency for Eigen, mid-sizes (500–750) show variability; $n \geq 1000$ are very stable, indicating consistent performance in the heavier compute regime.

5 System Metrics and Benchmarks

From the graph we can get the number of FLoating-point Operations (FLOP) of the program over execution time, at $n = 2000$, a real nxn multiply costs (FLOPs $\approx 2n^3$), where for large n we drop the lower-order term, $FLOP = 2 \cdot 2000^3 = 16,000,000 flop$, from the chart at this point time for the normal matrix $t \approx 18s$ and for Eigen $t \approx 1.8s$, then for the normal matrix: $16e^9/18 \approx 0.89 GFLOP/s$ and with Eigen: $16e^9/1.8 \approx 8.9 GFLOP/s$ we can see that $\sim$ 10x, Eigen is about ten times faster than the normal matrix implementation.

References

[1] Mark Allen Weiss, *Data Structures and Algorithm Analysis in C*, 2nd ed., Pearson, 2010, ch. 2.

[2] Eigen: Quick Reference Guide, `https://libeigen.gitlab.io/eigen/docs-nightly/group__QuickRefPage.html`, 2025.