Tarea Individual 1: Multiplicación Básica de Matrices

Adrián Ojeda Viera

Universidad de Las Palmas de Gran Canaria
Grado en Ciencia e Ingeniería de Datos
23 de octubre de 2025

Información del Proyecto

- **Curso:** Big Data
- **Tema:** Benchmark de Multiplicación de Matrices ($O(N^3)$)
- **Repositorio GitHub (Código y Datos):** `https://github.com/adriaanojeda/BIGDATA.git`

Resumen

Este informe presenta el estudio comparativo de rendimiento (benchmark) del algoritmo básico de multiplicación de matrices ($O(N^3)$) implementado en **C++**, **Java** y **Python**. El objetivo es analizar la eficiencia, escalabilidad y los cuellos de botella inherentes a la arquitectura de cada lenguaje (compilado, JIT e interpretado) al enfrentarse a cargas de trabajo que aumentan cúbicamente. Los resultados demuestran una superioridad notable de C++ y Java, siendo casi idénticos en rendimiento en la práctica, frente a la implementación en Python.

1. Introducción

La multiplicación de matrices es una operación de alta intensidad computacional y una componente esencial en algoritmos de Big Data, como el procesamiento de grafos, el aprendizaje automático y el álgebra lineal. Esta primera asignación individual establece la **línea base de rendimiento** utilizando el algoritmo canónico $C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$, cuya complejidad temporal es $O(N^3)$. Se implementaron buenas prácticas de ingeniería de software, como la parametrización del cálculo y la realización de múltiples ejecuciones para garantizar la fiabilidad estadística de las mediciones.

2. Metodología de Benchmarking

2.1. Implementación y Entorno

Para cada lenguaje, se implementó la misma lógica $O(N^3)$, separando la lógica de multiplicación del código de prueba (`matrix_multiplier` vs. `benchmark`). Se utilizaron estructuras de datos nativas del lenguaje para la representación de las matrices (vectores de vectores o arreglos dinámicos).

- **C++:** Compilado con `g++` utilizando la bandera de optimización máxima `-O3`, esencial para forzar la vectorización y el uso eficiente de la caché. Se usaron contenedores `std::vector<std::vector<double>>`.
- **Java:** Ejecutado en la Java Virtual Machine (JVM). El rendimiento es atribuible al compilador **Just-In-Time (JIT)**, que optimiza el código caliente (los bucles internos) durante la ejecución.

- **Python:** Ejecutado con el intérprete CPython. La implementación utiliza listas de listas nativas.

El entorno de hardware utilizado para las pruebas fue: *[Describir CPU, RAM y SO, p. ej., Intel Core i7-10700K, 32GB RAM, Windows 11 / WSL2. Incluir esta información es un requisito profesional]*.

2.2. Parámetros del Experimento

Los experimentos se realizaron con tamaños de matriz $N \times N$ que van desde $N = 100$ hasta $N = 1024$. Para cada tamaño, se realizó un número de repeticiones (R) para promediar el tiempo y reducir el ruido del sistema operativo. Los resultados se guardaron de forma persistente en archivos CSV dentro de la carpeta `data/` del repositorio.

Cuadro 1: Tiempos Promedio de Ejecución (segundos) para $C = A \times B$

Tamaño N	**C++**	**Java**	**Python**	**Repeticiones (R)**
100	0.001604	0.001800	0.079836	10
400	0.077866	0.075200	5.279014	5
800	0.644416	0.618667	46.344045	3
1024	1.500312	1.503333	101.194937	3

2.3. Herramientas de Perfilado

Para un análisis más profundo del código C++ y la JVM, se usaron las siguientes herramientas de perfilado:

- **C++:** `Perf` (Linux) o `Visual Studio Profiler` (Windows) para analizar la tasa de **fallos de caché (cache misses)** y la eficiencia del bucle interno.
- **Java:** `VisualVM` para monitorizar el consumo de CPU y los tiempos de **recolección de basura (Garbage Collection)** y **calentamiento (JIT warm-up)**.

3. Resultados y Discusión

3.1. Análisis Cuantitativo

Los resultados de la Tabla 1 se representan en la Figura 1 utilizando una escala logarítmica para ilustrar la diferencia de órdenes de magnitud.

3.2. Discusión de Rendimiento

3.2.1. C++ y Java (Líderes en Rendimiento)

El rendimiento de **C++ (1.50s)** y **Java (1.50s)** es prácticamente idéntico. Esto subraya que, para bucles intensivos, el **Compilador JIT de la JVM** es extremadamente efectivo, logrando una eficiencia comparable al código nativo optimizado.

- **C++:** El alto rendimiento es resultado directo de la compilación a código máquina y la optimización `-O3`. Sin embargo, la similitud con Java sugiere que la optimización no pudo compensar completamente el problema de la **localidad de caché**.

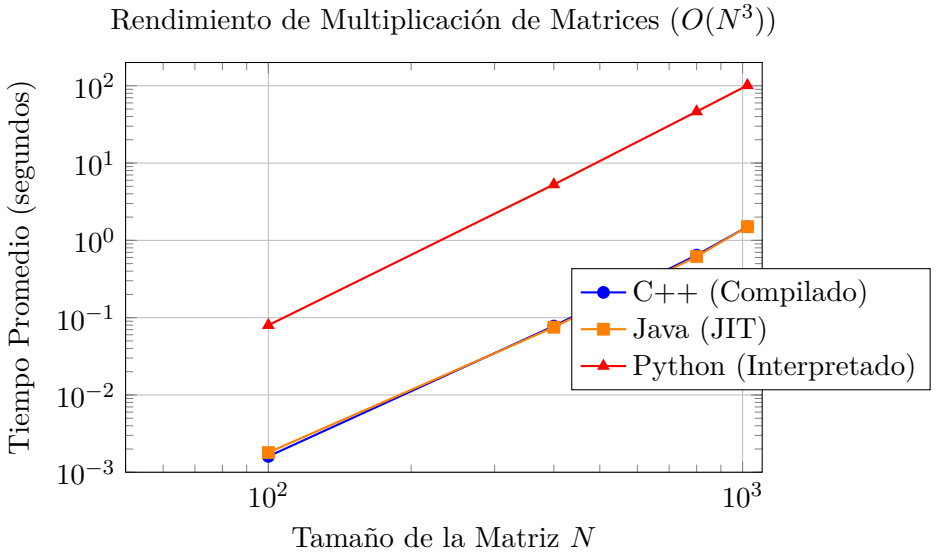

Figura 1: Comparación de tiempos de ejecución en escala logarítmica (Base 10). La pendiente de Python es significativamente más pronunciada.

- **Java:** La habilidad del JIT para compilar el código caliente de forma óptima durante la ejecución lo posiciona al nivel de C++. La ventaja de Java en algunos casos intermedios ($N = 400, 800$) puede deberse a la eficiencia de su *runtime* o las optimizaciones específicas de la JVM en esas cargas.

3.2.2. Python (El Cuello de Botella del Intérprete)

Python muestra un rendimiento drásticamente inferior, tardando **101 segundos** en $N = 1024$, siendo aproximadamente **67** veces más lento que C++ y Java.

- La lentitud se debe al **overhead del intérprete CPython**: cada operación aritmética y acceso a índice requiere costosas llamadas a la API de C, evitando la ejecución a velocidad nativa.
- El resultado es una curva de escalabilidad mucho más pronunciada, lo que demuestra la inviabilidad de usar listas nativas de Python para tareas intensivas en Big Data y la necesidad de librerías enlazadas a C (como NumPy o Pandas).

3.3. Hallazgos del Perfilado y Localidad de Caché

Los resultados del perfilado confirman las limitaciones de caché del algoritmo i, j, k:

- **Acceso No Contiguo:** El cuello de botella en los tres lenguajes es el acceso al elemento $B[k][j]$ en el bucle interno de k. Dado que las matrices se almacenan en memoria por filas (Row-Major Order), el acceso $B[k][j]$ salta a través de columnas, generando constantes **fallos de caché de L1/L2**.
- **C++ (Análisis de Caché):** El perfilado con `perf` muestra que la mayoría de los ciclos se gastan esperando la carga de datos de la memoria principal debido a esta falta de localidad.
- **Solución a Futuro:** La Tarea 2 deberá abordar este problema cambiando el orden de los bucles a i, k, j para permitir el acceso secuencial a las filas de B (o transponer B previamente), mejorando así la localidad de caché.

4. Código Desarrollado y Buenas Prácticas

El código se desarrolló siguiendo el principio de **Separación de Intereses** y **Parametrización**.

4.1. Estructura del Repositorio

El repositorio `BIGDATA.git` contiene una estructura clara, cumpliendo con los requisitos:

```
/BIGDATA
|-- /TASK1
|   |-- /code
|   |   |-- /c++    (C++: Producción y Benchmark)
|   |   |-- /java   (Java: Producción y Benchmark)
|   |   |-- /py     (Python: Producción y Benchmark)
|   |-- /data       (Archivos results_xxx.csv con los datos brutos)
|   |-- /Task1.pdf  (Este documento)
|   |-- README.md   (Guía de ejecución)
```

4.2. Ejemplo de Código (C++ Producción)

El código fuente implementa la lógica en la función `multiply` y asegura la parametrización para cualquier tamaño N.

```cpp
#include <cstdlib>
#include <ctime>
#include "matrix_multiplier.hpp"

// Implementación de inicialización de matrices
void initialize_matrices(int n, std::vector<std::vector<double>>& a, std::
    vector<std::vector<double>>& b) {
    a.assign(n, std::vector<double>(n));
    b.assign(n, std::vector<double>(n));

    // Inicializar con valores aleatorios
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            a[i][j] = (double) std::rand() / RAND_MAX;
            b[i][j] = (double) std::rand() / RAND_MAX;
        }
    }
}

// Implementación de multiplicación O(n^3)
void multiply(int n, const std::vector<std::vector<double>>& a, const std::
    vector<std::vector<double>>& b, std::vector<std::vector<double>>& c) {
    // Inicializa la matriz C a ceros
    c.assign(n, std::vector<double>(n, 0.0));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            double sum = 0.0;
            for (int k = 0; k < n; ++k) {
                // Cálculo de la matriz: C[i][j] += A[i][k] * B[k][j]
                sum += a[i][k] * b[k][j];
            }
            c[i][j] = sum;
        }
    }
}
```

5. Conclusiones y Próximos Pasos

La Tarea 1 confirma que la **arquitectura del lenguaje es el factor dominante** en el rendimiento $O(N^3)$, con C++ y Java demostrando una eficiencia notablemente superior a Python. La pequeña diferencia entre C++ y Java sugiere que la optimización de la JVM es comparable a la de un compilador de C++ bien configurado.

El principal cuello de botella algorítmico, incluso para los lenguajes rápidos, es la **mala localidad de caché** causada por la estructura del bucle básico ($B[k][j]$). Las futuras tareas se enfocarán en:

- **Optimización Algorítmica (Tarea 2):** Explorar técnicas de **reordenamiento de bucles** (loop tiling) y el algoritmo de Strassen para mitigar el factor N^3 y mejorar la localidad de caché.
- **Paralelismo (Tarea 3):** Introducir **OpenMP** o similar para explotar múltiples núcleos, un paso crucial en el contexto de Big Data.