Práctico 01

Diseño y Análisis de Algoritmos

Adriel Reyes Suárez

Índice

Aspectos Previos

Antes de presentar los datos recopilados, sería deseable contextualizar los algoritmos llevados a cabo. Ambos algoritmos para la multiplicación de matrices se basan en 3 bucles anidados, lo que para matrices cuadradas de dimensión N resulta en complejidad en tiempo de $O(N^3)$. Por tanto, se espera que los resultados muestren una curva de crecimiento rápido (cúbico).

Arquitectura

Las pruebas se han llevado a cabo en un sistema con la siguiente configuración de hardware:

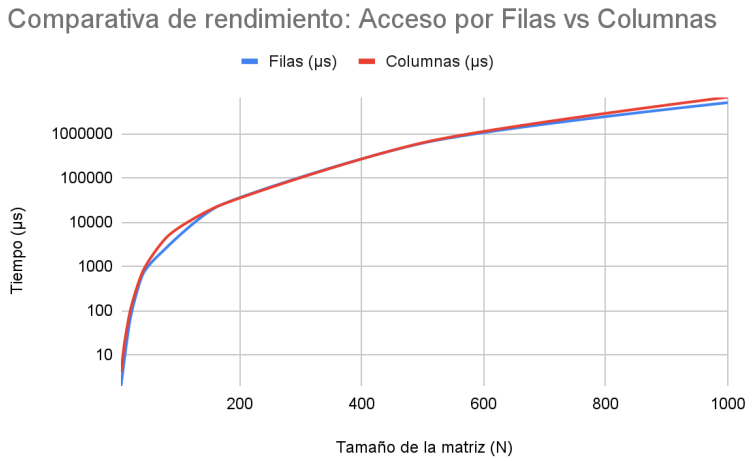
- **Modelo de CPU:** Intel® Core™ i7-1360P (13ª Generación).
- **Microarquitectura:** x86_64 (Little Endian).
- **Núcleos y Hilos:** 8 núcleos físicos con tecnología *Hyper-Threading* (16 hilos lógicos visibles por el sistema).
- **Entorno de Ejecución:** El sistema opera bajo una capa de virtualización en Windows, más específicamente WSL2.
- **Lenguaje de programación:** C++**.**

Tras el estudio de los ***Principios SOLID*** y el ***Patrón estrategia,*** se ha planteado el desarrollo de la práctica de la siguiente manera (componentes principales):

- **Clase `Matriz`:** se ha implementado esta clase con el fin de encapsular todo lo necesario para la representación de matrices.
- **Clases `Algoritmo`, `RowMultAlg` y `ColMultAlg`:** se ha creado esta jerarquía de clases basada en una clase abstracta (interfaz) que declara el método común que las clases concretas compartirán (siguiendo el ***Patrón estrategia*****).** Dichas clases nos permiten cambiar en el momento de ejecución el algoritmo que se emplea para realizar la multiplicación, usando un puntero de la clase abstracta (interfaz), aprovechando el polimorfismo.
- **Clase `ResolverMatriz`:** siguiendo de nuevo el ***Patrón estrategia***, se ha creado esta clase para que actúe como el contexto, la cual recibe una referencia a la interfaz y ejecuta de forma uniforme el código para cualquier algoritmo que se desee implementar.

Resultados Obtenidos

Tamaño Matriz ($N * N$)	Tiempo Algoritmo Filas (μs)	Tiempo Algoritmo Columnas (μs)
5 x 5	2	4
10 x 10	7	16
20 x 20	64	99
40 x 40	629	736
80 x 80	2700	4585
160 x 160	21450	21794
500 x 500	612036	624941
1000 x 1000	5014142	6663120



Conclusiones

Observando los resultados, podemos extraer la conclusión de que el procesamiento por filas es ligeramente más rápido que el procesamiento por columnas. Esto se debe a una característica de C++, la cual organiza las matrices en memoria de forma continua, fila por fila. Por tanto, al leerse los datos en el mismo orden en el que son almacenados, el procesador puede predecirlos y cargarlos en la caché de forma más eficiente.

A pesar de ello, el algoritmo se comporta como esperábamos, mostrando un crecimiento cúbico $O(N^3)$. El tiempo aumenta exponencialmente a medida que, a su vez, el tamaño de la matriz aumenta. Pero aquí es cuando se ha de plantear la siguiente pregunta: ¿es realmente óptimo este algoritmo? La respuesta es no.

Para instancias mucho mayores, cuyo N tiende a ser muy grande, su coste computacional también es cada vez mucho más mayor, llegando a colapsar. En este contexto, surgen alternativas como el Algoritmo de Strassen, el cual baja, aunque no mucho, la complejidad a $O(N^{2.87})$. Por tanto, se concluye que la implementación realizada, aunque correcta en su solución, resulta ser más cara y menos óptima en comparación a otras soluciones propuestas.

Referencias

Principios SOLID (1)
Principios SOLID (2)
Patrón estrategia