

Informe de Prácticas

Sesión 2

Computación de Altas Prestaciones

Máster en Ingeniería Informática



Universidad de Oviedo

Autor:

Rubén Martínez Ginzo, UO282651@uniovi.es

Abril 2025

Índice

1. Introducción	4
1.1. <i>Desarrollo</i>	4
1.2. <i>Benchmarking</i>	4
2. Expectativas y análisis inicial	5
2.1. Fase 1	5
2.2. Fase 2	5
2.3. Fase 3	5
3. Fase 1	6
4. Fases 2 y 3	8
5. Conclusiones	9

Índice de figuras

1. Tiempos de ejecución de *Row-Major order* y *Column-Major order* en *Python* 6
2. Tiempos de ejecución de *Z-Order* en *Python* ($N = 512$ y $N = 1024$) 7

Índice de tablas

1. Tiempos de ejecución de *Row-Major order* y *Column-Major order* en *Python* 6
2. Tiempos de ejecución de los tres algoritmos en *Python* 7

1. Introducción

En esta sesión de prácticas de laboratorio se evalúa la hibridación de lenguajes para la implementación de la multiplicación de matrices, combinando específicamente *C* y *Python*. La experimentación se divide en tres fases:

- **Fase 1:** Implementación de la multiplicación de matrices en *Python* y toma de tiempos.
- **Fase 2:** Creación y manejo de matrices con *Python*, multiplicación de matrices en *C* y toma de tiempos.
- **Fase 3:** Creación y manejo de matrices con *C*, multiplicación de matrices en *C* y toma de tiempos.

Utilizando la librería *ctypes* se lleva a cabo la hibridación de ambos lenguajes, con el objetivo de evaluar el rendimiento de cada una de las fases y comparar los resultados obtenidos en cada uno de los escenarios. La multiplicación de matrices se realizará siguiendo los diferentes esquemas de acceso a memoria evaluados en la sesión anterior: *Row-Major order*, *Column-Major order* y *Z-Order*.

1.1. Desarrollo

Para el desarrollo de esta práctica, se han seguido las indicaciones recogidas en el guion de la sesión correspondiente. Se han implementado las tres fases del experimento, y se han llevado a cabo diversas pruebas para medir su rendimiento. Con el fin de comparar los tiempos de ejecución, se ha desarrollado un programa en *Python* con fines de *benchmarking*, tal y como se detalla en el capítulo siguiente. Todo el código fuente se encuentra disponible públicamente en el siguiente repositorio de GitHub, así como en el archivo *zip* asociado a esta entrega.

1.2. Benchmarking

Como parte complementaria al desarrollo de la sesión de laboratorio, se ha implementado un programa en *Python* que automatiza la medición de los tiempos de ejecución correspondientes a cada una de las fases del experimento.

Este programa, ubicado en el subdirectorio *benchmark*, permite especificar diferentes tamaños de matriz y un número fijo de iteraciones para cada prueba. Una vez configurado, ejecuta secuencialmente cada una de las fases, registrando el tiempo medio de ejecución para la multiplicación de matrices en cada caso. Para facilitar esta tarea, en cada uno de los programas desarrollados en esta práctica (*multiply_matrices.py*, *multiply_matrices_hybrid.py* y *multiply_matrices_hybrid_pro.py*) se han definido las funciones *run_phase_X_row_col* y *run_phase_X_zorder*, donde *X* indica el número de fase. Estas funciones llevan a cabo la toma de tiempos de ejecución de cada producto en la fase correspondiente con los esquemas de acceso a memoria *Row-Major order*, *Column-Major order* y *Z-Order*.

Una vez completadas todas las iteraciones, el programa genera un *log* en formato CSV que recoge los resultados obtenidos. Dichos tiempos de ejecución constituyen la base de los análisis y comparativas que se presentan en los capítulos siguientes.

2. Expectativas y análisis inicial

Antes de comenzar con el análisis de los resultados obtenidos en las diferentes fases del experimento, se establecen las expectativas iniciales en cuanto a los tiempos de ejecución y rendimiento de cada una de ellas.

2.1. Fase 1

En esta primera fase se implementa la multiplicación de matrices en *Python* así como la creación y manejo de las mismas. Ya que este lenguaje de programación no está optimizado para operaciones de bajo nivel, tal como es el producto de matrices, se espera que los tiempos de ejecución sean considerablemente más altos en comparación a los obtenidos en las fases posteriores.

Será interesante analizar cómo se comportan las diferentes configuraciones de acceso a memoria (*Row-Major order*, *Column-Major order* y *Z-Order*) en este contexto, y si existe alguna diferencia significativa respecto a las fases 2 y 3.

2.2. Fase 2

En la segunda fase del experimento se implementa la multiplicación de matrices en *C* mientras que la creación y manejo de las mismas se realiza en *Python*. Teniendo esto en cuenta, se espera que los tiempos de ejecución sean mucho más bajos que en la fase 1, ya que *C* es un lenguaje de programación de bajo nivel y está optimizado para este tipo de operaciones.

Sin embargo, la creación y manejo de matrices en *Python* puede introducir cierta sobrecarga, lo que podría afectar los tiempos de ejecución.

2.3. Fase 3

En esta última fase del experimento, tanto la creación y manejo, como la multiplicación de matrices se realizan en *C*. Se espera que los tiempos de ejecución sean los más bajos de todas las fases, ya que se aprovechan al máximo las ventajas de *C* en términos de eficiencia y optimización.

En este contexto, sería interesante comparar estos tiempos de ejecución con los obtenidos en la sesión anterior, donde la implementación y toma de tiempos se realizó exclusivamente en *C*. Debe tenerse en cuenta que, en este caso, la toma de tiempos se realiza en *Python*, lo que podría introducir cierta variabilidad en los resultados.

3. Fase 1

En esta sección se presenta el análisis de los resultados obtenidos en la Fase 1 del experimento. Para ello, se han tomado los tiempos de ejecución de la multiplicación de matrices en *Python* utilizando los diferentes esquemas de acceso a memoria: *Row-Major order*, *Column-Major order* y *Z-Order*. Se han utilizado los siguientes tamaños de matrices: $N = 2$, $N = 4$, $N = 8$, $N = 16$, $N = 32$, $N = 64$, $N = 128$, $N = 256$, $N = 512$ y $N = 1024$. En el caso del algoritmo de *Z-Order*, se ha ejecutado con todos los tamaños de bloque posibles para cada tamaño de matriz. Todos los productos se han repetido un total de 8 iteraciones, y se ha tomado el tiempo medio de ejecución. En la Tabla 1 se presentan los resultados obtenidos para *Row-Major order* y *Column-Major order*.

Matrix size	<i>Row-Major order</i> (s)	<i>Column-Major order</i> (s)
[...]	[...]	[...]
64×64	0,046619	0,046878
128×128	0,363829	0,369213
256×256	2,933288	2,948093
512×512	27,198958	25,990449
1024×1024	200,909463	192,439038

Tabla 1: Tiempos de ejecución de *Row-Major order* y *Column-Major order* en *Python*

A simple vista, se trata de resultados con un rendimiento mucho peor a la multiplicación de matrices en *C* de la cual se tienen resultados de la sesión anterior. En la Figura 1 se grafican los tiempos de ejecución recogidos en la Tabla 1.

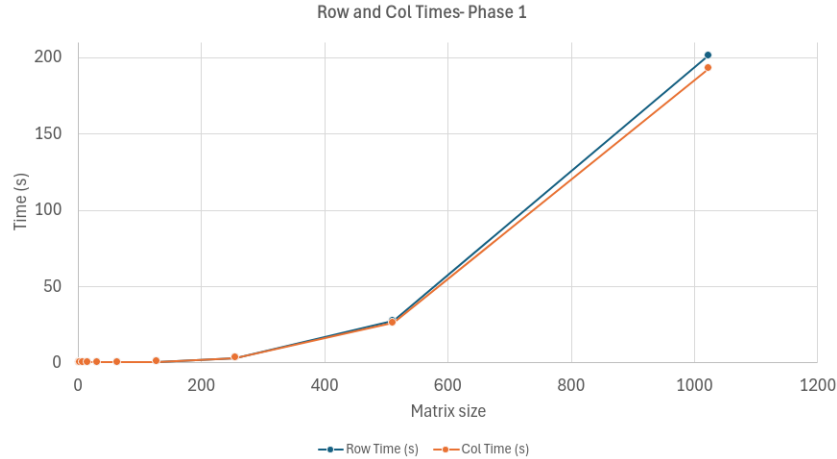


Figura 1: Tiempos de ejecución de *Row-Major order* y *Column-Major order* en *Python*

Se observa un crecimiento exponencial, tal como se esperaba y como sucederá en las siguientes fases, pero con unos tiempos de ejecución muchísimo más altos. *Column-Major order* es ligeramente más rápido que *Row-Major order*, pero la diferencia es mínima.

Respecto al algoritmo de *Z-Order*, en la Figura 2 se muestran los resultados obtenidos para los tamaños de matriz 512×512 y 1024×1024 , los cuales se consideran los más representativos.

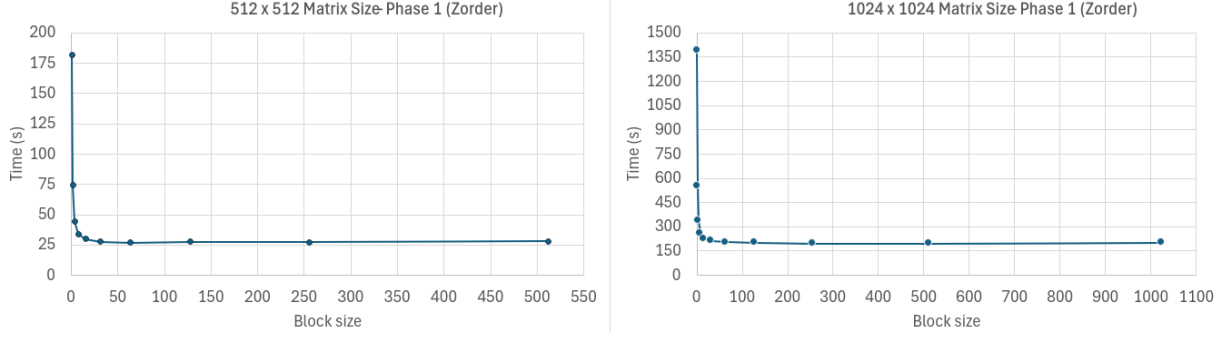


Figura 2: Tiempos de ejecución de *Z-Order* en *Python* ($N = 512$ y $N = 1024$)

Se observa una tendencia decreciente en ambos casos cuando el tamaño de bloque aumenta, si bien es cierto que resulta complicado determinar a simple vista un tamaño de bloque óptimo. De hecho, el uso de tamaños de bloque muy grandes (cerca de $N/2$ e incluso N) produce un rendimiento muy similar al de tamaños de bloque pequeños (4, 8, 16, 32...), lo que sugiere que no se está aprovechando la idea del propio algoritmo. Teniendo en cuenta que en *Python* no se accede a memoria de la forma más eficiente, esta conclusión resulta bastante lógica.

Respecto a su comparación con *Row-Major order* y *Column-Major order*, teniendo en cuenta los mejores resultados de *Z-Order* para cada tamaño de matriz, apenas mejora o iguala a los otros dos algoritmos. Quizá habiendo experimentado con tamaños de matriz aun mayores, se podrían haber obtenido resultados más favorables, pero en este caso no se ha considerado necesario debido a los tiempos de ejecución tan elevados.

En la Tabla 2 se presenta la comparativa de los tiempos de ejecución de los tres algoritmos, teniendo en cuenta el resultado más óptimo de *Z-Order* para cada tamaño de matriz.

Matrix size	<i>Row-Major order</i> (s)	<i>Column-Major order</i> (s)	Best <i>Z-Order</i> (s)
[...]	[...]	[...]	[...]
64×64	0,046619	0,046878	0,047778
128×128	0,363829	0,369213	0,362896
256×256	2,933288	2,948093	3,007014
512×512	27,198958	25,990449	26,586173
1024×1024	200,909463	192,439038	193,267807

Tabla 2: Tiempos de ejecución de los tres algoritmos en *Python*

4. Fases 2 y 3

5. Conclusiones