Algebra Lineal Numérica

Laboratorio 1

Matrices y Errores

Integrantes

Nombre	CI	Correo
Renzo Gambone	5.155.486-4	renzo.gambone@fing.edu.uy
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy

Índice

1. Introducción

En el presente informe se detallan los puntos pedidos en la consigna de la **Tarea 1**. A continuación, se destacan ciertos aspectos a tener en cuenta del análisis y de los archivos entregables:

1. La implementación de los algoritmos se realizó empleando el lenguaje **C**. Dentro de la entrega se proporciona un archivo `Makefile` para compilar los mismos.

2. El código fue compilado con la flag `O3` y ejecutando en un **CPU AMD Ryzen 5 3600**, cuyas especificaciones pueden ser consultadas en el siguiente enlace. De las mismas se destaca lo siguiente:
   - **Cores:** 6 (12 threads)
   - **CPU max MHz:** 3600 MHz
   - **L1i Cache:** 192 KB (6 x 32 KB)
   - **L1d Cache:** 192 KB (6 x 32 KB)
   - **L2 Cache:** 3 MB (6 x 512 KB)
   - **L3 Cache:** 32 MB (2 x 16 MB)
   - **Cache Latency:** 4 (L1); 12 (L2); 40 (L3)

3. Los datos manipulados fueron matrices compuestos por números decimales representados en punto flotante de precisión simple (*float* de **C**).

4. Los tiempos registrados al realizar distintas ejecuciones de un mismo script no fueron siempre constantes, sin embargo su variación osciló con un radio lo suficientemente bajo como para que las observaciones y conclusiones presentadas se preserven.

2. Desarrollo

2.1. Ejercicio 1 - Errores de Representación

Con el objetivo de visualizar los ejemplos de forma sencilla, se propone una implementación de punto flotante más simple, la cual cumple con la siguiente estructura:

$$\pm\ d_0.d_1d_2\ \times\ 10^e$$

Donde los parámetros cumplen las siguientes restricciones:

- $e \in [-4, 4]$ (se acota el rango del exponente)
- $d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (los dígitos son decimales)
- $d_0 \neq 0$ (la representación está normalizada con el dígito no decimal distinto a 0)

De esta manera, sin tener en cuenta el signo, el rango de números representables tiene como cotas superior e inferior a los números:

- $f_{min} = 1{,}00\ \times\ 10^{-4}\ =\ 0{,}0001$
- $f_{max} = 9{,}99\ \times\ 10^{4}\ =\ 99900$

Para el caso del *underflow*, basta con intentar representar un número menor al menor número representable, como por ejemplo $x = 0{,}000099$. Al intentar utilizar los parámetros anteriormente mencionados, sucede que $x = 0{,}99\ \times\ 10^{-4}$, no cumpliendo con la restricción de normalización. Si la representación utilizada admitiera números desnormalizados (como en el caso de la IEEE 754), se puede tomar como ejemplo el número $x = 0{,}0000009 = 0{,}009\ \times\ 10^{-4}$ y por tanto no representable dentro del rango de dígitos d_i presentado.

Para el caso de *cancelación catastrófica*, se plantea una situación similar, donde dos números representables en el mismo signo se restan, y el resultado es un número tan pequeño que no es representable dentro del rango preestablecido. Tomando:

$$x = 99{,}99 \sim 9{,}99\ \times\ 10^1$$
$$y = 99{,}98 \sim 9{,}99\ \times\ 10^1$$

Al restarlos sin tener en cuenta la representación, se obtiene:

$$x - y = 99{,}99 - 99{,}98 = 0{,}01 \sim 0{,}01\ \times\ 10^0$$

No obstante, dentro de la representación son números iguales y al restarlos se obtiene 0, lo cual es una pérdida importante de información (sobretodo al escalar la cantidad de dígitos representables), y puede desencadenar errores importantes (Como por ejemplo en el divisor a la hora de computar una derivada)

2.2. Ejercicio 2 - Recorrida de Matrices

2.2.1. Código implementado

En el archivo `benchmark_matrix_iteration.c`, se proporcionan las siguientes funciones que iteran a través de una matriz:

- `row_sum`: Suma y almacena los elementos en una matriz A recorriendola por filas.
- `column_sum`: Suma y almacena los elementos en una matriz A recorriendola por columnas.

2.2.2. Experimentación y resultados obtenidos

A modo de experimentación, se realizaron 100 recorridas de matrices de 8192×8192, capturando tiempo de ejecución promedio para cada función a lo largo de las corridas.

Algoritmo	Tiempo (ms)
Por filas	48.723484
Por Columnas	314.303484

2.2.3. Observaciones

Teniendo en cuenta los resultados capturados, se pueden realizar las siguientes observaciones:

- El algoritmo que recorrió la matriz por filas ejecutó en un tiempo 6,5 veces menor al que recorrió por columnas.
- Pese al gran tamaño de la matriz (65545216 elementos), ambos algoritmos lograron procesar la misma en un tiempo razonablemente pequeño (La implementación más lenta tomando menos de 1/3 de segundo).

De los datos observados se pueden extraer las siguientes interpretaciones y conclusiones:

Los resultados obtenidos fueron dentro de todo los esperados. Dado que C almacena las matrices en memoria de forma tal que los elementos en una fila se encuentran contiguos, esto permite que cuando se va a leer un elemento desde memoria, el resto de elementos contiguos en su fila también se cargan en el caché (facilitando lecturas más rápidas para los mismos).

El algoritmo por columnas, sin embargo, necesita acceder a los elementos contiguos a lo largo de la columna, los cuales, salvo que la matriz sea muy pequeña (o el tamaño de la caché muy grande) no estarán contenidos en la misma. Por lo tanto, ocurrirá un caché miss y será necesario invalidar la caché para traer nuevos elementos desde la memoria (en los que ocurrirá lo mismo).

Esto es claramente menos eficiente que usar todos los elementos que fueron cargados (contiguos por fila) en la caché antes de realizar otra lectura en memoria.

2.3. Ejercicio 3 - Multiplicación de Matrices

2.3.1. Código implementado

En el archivo `benchmark_matrix_multiplication.c`, se proporcionan las siguientes funciones que resuelven el problema de la multiplicación de matrices:

- `normal_mult`: Ejecuta el algoritmo tradicional de multiplicación de matrices $C = A \times B$, recorriendo A a lo largo de sus filas y B a lo largo de sus columnas.
- `row_mult`: Ejecuta el algoritmo tradicional de multiplicación de matrices $C = A \times B$, recorriendo A y B a lo largo de sus filas.
- `col_mult`: Ejecuta el algoritmo tradicional de multiplicación de matrices $C = A \times B$, recorriendo A y B a lo largo de sus columnas.
- `strassen_mult`: Implementa el algoritmo de Strassen, empleando recursión y las funciones auxiliares `add` y `subtract` (Que reciben dos matrices A, B, su tamaño y realizan $A + B$ y $A - B$ respectivamente). Se destaca de esta implementación de Strassen que solo es funcional para matrices cuadradas cuyo tamaño sea potencia de 2.

2.3.2. Experimentación y resultados obtenidos

Experimentando con el algoritmo de Strassen, se encontró un problema importante al trabajar con grandes datos y un algoritmo recursivo Divide & Conquer. Debido a que a lo largo de la recursión el algoritmo crea múltiples matrices, (las cuales no pueden ser liberadas hasta que se terminen de usar para operar con ellas), la sobrecarga de memoria reservada en el stack crece de forma exponencial respecto al tamaño original de la matriz.

Es a causa de esto que el tamaño más grande de matrices con el que se pudo experimentar con el algoritmo de Strassen fue de 512×512, dado que para tamaños mayores el programa llenaba los $16GB$ de RAM disponible en el equipo en donde se realizó la experimentación y el sistema operativo automáticamente finalizaba la ejecución del proceso. Los resultados obtenidos fueron los siguientes:

Algoritmo	Tiempo (ms)
Por filas/columnas	100.420532
Por columnas/columnas	198.516968
Por filas/filas	36.021118
Strassen	6883.022583

Para el resto de algoritmos se realizó una experimentación por separado, trabajando con matrices de tamaño 4096×4096. Los resultados obtenidos de la misma fueron los siguientes:

Algoritmo	Tiempo (ms)
Por filas/columnas	220154.662109
Por columnas/columnas	501091.119141
Por filas/filas	20739.329102

2.3.3. Observaciones

Teniendo en cuenta los resultados capturados, se pueden realizar las siguientes observaciones:

- Para el caso donde fue posible medirlo, el algoritmo de Strassen demoró sustancialmente más que el resto de los otros algoritmos.
- El algoritmo con menor tiempo de ejecución siempre fue el que accedia a ambas matrices por sus filas.
- El algoritmo que accedia a la matriz A por sus filas y a B por sus columnas tuvo un menor tiempo de ejecución al que accedia a ambas matrices por sus columnas.
- Para las matrices de tamaño 512×512 el algoritmo por filas/filas demoró casi 3 veces menos que el de filas/columnas. Para las matrices de tamaño 4096×4096 el algoritmo filas/filas demoró más de 10 veces menos que el de filas/columnas.

Se pueden extraer múltiples interpretaciones y conclusiones de los datos anteriormente observados:

- Para los algoritmos de multiplicación simple los patrones de acceso por filas siempre dieron un mejor tiempo de ejecución que accediendo por columnas (debido al fenómeno anteriormente discutido respecto a como C almacena las matrices en memoria y por ende a que estos algoritmos en C hacen un uso más eficiente de la caché).
- Se puede ver como un patrón de acceso por columnas no solo es menos performante, sinó que también posee menor escalabilidad debido al tamaño de las caché y cantidad de caché misses ocurridos.
- El algoritmo de Strassen demoró un tiempo extremadamente mayor respecto al resto (pese a que nivel algoritmico el mismo tenga $O(n^{2,808})$, mientras que el resto tienen $O(n^3)$.
- Una posible interpretación de porque el algoritmo de Strassen fue tan lento es debido a tener en cuenta la alta carga de inicialización y destrucción de submatrices auxiliares, teniendo que a lo largo de la ejecución crear y destruir un total de 19 matrices auxiliares. Se cree que una implementación más óptima de este algoritmo, que en lugar de hacer uso de exclusivamente estructuras auxiliares operase en subsecciones de la matriz original (por medio de desplazamientos) podría obtenier un tiempo sustancialmente mejor.