Informe - Practico 1

Mateo Quiller, Leonardo Pombo, Joaquín Mezquita

Grupo 8

30 de noviembre de 2025

Índice

Para este trabajo se utilizo una computadora con los siguientes componentes: CPU: i5-10400, Cache L1 64 KB, Cache L2 256 KB, Cache L3 12 MB, Ram: 16 GB 3200 Mhz.

1. Ejercicio 1

Para tener un arreglo de `Char` de 100 MB, usamos la siguiente linea:

```
constexpr size_t SIZE = 100 * 1024 * 1024; // 100MB
```

Y para crear un arreglo con indices inicializados previamente tenemos el siguiente código (el cual sirve para el secuencia y aleatorio):

```
void initializeIndex(vector<size_t>& index, bool random)
    for (size_t i = 0; i < index.size(); ++i) index[i] = i;
    if (random)
        random_device rd;
        mt19937 g(rd());
        shuffle(index.begin(), index.end(), g)
```

1.1. Parte 1: Recorrido Secuencial

El código realiza lo siguiente:

- Se reserva un arreglo de tipo `Char` con tamaño 100 MB.
- Se inicializa un arreglo de índices en orden secuencial (0, 1, 2, ..., 100MB-1).
- Se recorre el arreglo accediendo a cada índice según el arreglo de índices secuenciales.
- Finalmente, se mide el tiempo de ejecución utilizando la biblioteca chrono (`std::chrono::high_resolution_clock`).

El tiempo de acceso secuencial fue registrado varias veces, con los siguientes resultados: 98.114 ms, 94.501 ms, 91.188 ms, 91.624 ms y 91.413 ms. Siendo el promedio de **93.368 ms**.

1.2. Parte 2: Recorrido Aleatorio

Para medir el tiempo de acceso aleatorio, el código realiza los siguientes pasos:

- Se inicializa el arreglo de índices con números secuenciales (0, 1, 2, ..., 100MB-1) y el arreglo de `Char` vacío.
- Se baraja el arreglo de índices para simular un acceso aleatorio.
- Se recorre el arreglo accediendo a los índices indicados por el arreglo de indices (generando cierta aleatoriedad de acceso).

- Se mide el tiempo de ejecución de este recorrido aleatorio.

Los resultados obtenidos para el tiempo de acceso aleatorio fueron 1135.98 ms, 1122.65 ms, 1122.8 ms, 1117.65 ms y 1189.42 ms. Siendo el promedio de **1137.7 ms**.

1.3. Reflexión sobre los resultados

Se observa una diferencia significativa en los tiempos de acceso entre el recorrido secuencial y el aleatorio. El tiempo de acceso secuencial es notablemente menor que el de acceso aleatorio. Esto se debe a que la cantidad de cache miss aumenta significativamente cuando se accede de manera aleatoria al arreglo de `Char`.

El acceso secuencial es más eficiente porque la memoria cache puede predecir con mayor certeza qué datos se necesitan a continuación, haciendo uso de la localidad espacial los datos consecutivos son cargados en la cache, lo que reduce el tiempo de acceso a los datos pues se encuentran niveles de memoria de mayor jerarquía.

En cambio, el acceso aleatorio implica que los índices de memoria no se siguen de forma continua, lo que dificulta la carga de los datos en la cache de manera que estamos forzando a propósito que el programa no cumpla con la heuristica que usan las caches llamada localidad temporal. Como resultado, el tiempo de acceso aleatorio es mucho más alto.

2. Ejercicio 2

Introducción. Se utilizarán matrices cuadradas para la resolución de todo el ejercicio, al igual que en el ejemplo provisto en la letra del practico.

2.1. Parte 1: Versión sin bloques

Un pseudocódigo para la multiplicación de matrices sin técnicas de optimización sería:

- Primero se recorre la matriz C para asignar ceros a todas sus posiciones.
- Luego, para cada fila i de A, se realiza el producto escalar con la columna k de B, obteniendo así el resultado C[i][k].

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < K; j++)
        float sum = 0.0f;
        for (int r = 0; r < M; r++)
            sum += A[i*M + r] * B[r*K + j];
        C[i*K + j] = sum;
```

Para medir el rendimiento, se toma el tiempo total t de ejecución (en segundos) y se calculan los MFLOPS como:

$$\text{MFLOPS} = 2 \times \frac{size^3}{t \times 10^{-6}}.$$

Tamaño de matriz	Bytes	Nivel de caché	MFLOPS
35	14,700	L1	2906.78
75	67,500	L2	2614.72
100	120,000	L2	2264.49
300	1,080,000	L3	2105.44
1500	27,000,000	No entra en caché	1453.28

Cuadro 1: La tabla compara el desempeño en MFLOPS para valores de *size* de distinto orden, comparables al tamaño de los distintos niveles de caché (L1, L2 y L3)

Como podemos ver en la tabla 2, a medida que la dimensión de las matrices aumenta, disminuyen los $MFLOPS$, ya que dejan de entrar en los niveles de cache mas rápidos, hasta el punto en que si las matrices no caben en ninguna cache y se trabaja directamente con la RAM la degradación es de casi el 50 % respecto a trabajar con caché L1.

2.2. Parte 2: Técnica de bloques para la multiplicación de matrices

La técnica explicada en la letra consiste en subdividir las matrices en regiones o bloques de tamaño `TAM_BL`, y realizar la multiplicación entre bloques correspondientes

de forma que los datos requeridos por las operaciones se mantengan en caché el mayor tiempo posible, o dicho de otra forma cuando ponemos un bloque en cache hago todas las operaciones que van a usar los datos de ese bloque para aprovechar la localidad temporal. Lo implementamos de la siguiente forma:

```
for (int i0 = 0; i0 < n; i0 += TAM_BL)
    for (int j0 = 0; j0 < n; j0 += TAM_BL)
        for (int k0 = 0; k0 < n; k0 += TAM_BL)
            for (int i = i0; i < min(i0 + TAM_BL, n); i++)
                for (int k = k0; k < min(k0 + TAM_BL, n); k++)
                    for (int j = j0; j < min(j0 + TAM_BL, n); j++)
                        C[i][j] += A[i][k] * B[k][j];
```

En cada iteración de la triple anidación principal (`i0, j0, k0`), se seleccionan bloques de **A**, **B** y la porción correspondiente de **C**. Estos bloques se multiplican y acumulan, reduciendo así el número de accesos a la memoria principal o a niveles de caché más lentos. La idea principal es reutilizar al máximo los datos que ya se han cargado en los niveles más cercanos (y rápidos) de la jerarquía de memoria, explotando la *localidad temporal* y, en la medida de lo posible, la *localidad espacial*.

2.3. Comparación con la técnica sin bloques

- **Matrices grandes:** Cuando las matrices son lo suficientemente grandes para no caber en la caché L1, el uso de *bloques* mejora drásticamente el desempeño (MFLOPS) en comparación con la versión sin bloques. Como ya explicamos esto se debe a que, gracias al uso de bloques, se evita que los mismos datos se reemplacen rápidamente en caché y se disminuye el número de accesos costosos a niveles inferiores de memoria.
- **Matrices pequeñas:** En cambio, si las matrices son tan pequeñas que caben por completo en la caché L1 no se percibe una mejora con la técnica de *bloques*. De hecho, la sobrecarga adicional introducida por los bucles y el cálculo de índices degrada ligeramente el rendimiento frente a la versión sin bloques.

En la siguiente tabla se presentan los resultados que obtuvimos usando bloques y se comparan con no usar bloques

Tamaño de matriz	**TAM_BL**	**MFLOPS (bloques)**	**MFLOPS (sin bloques)**
10	2	1111.11	2857.14
30	10	1942.45	3396.23
75	10	3118.07	2515.65
75	20	3455.16	2746.58
300	10	3877.33	2083.4
300	75	2671.61	2060.93
2000	10	3738.55	674.448
2000	75	2657.49	646.773
2000	300	2095.52	640.216
2000	1000	1614.57	656.187

Cuadro 2: La tabla compara el desempeño en MFLOPS para valores de *size* de distinto orden, comparables al tamaño de los distintos niveles de caché (L1, L2 y L3)

Se puede observar que los MFLOPS en la version con bloques dependen casi unicamente de el TAM_BL elegido ya que por ejemplo para un tamañio de matriz de 2000 y 35 cuando usamos un TAM_BL de 10 los MFLOPS obtenidos son casi iguales, en la version sin bloques a medida que aumenta el tamanio de la matriz los MFLOPS disminuyen alcanzando un minimo de aproximadamente 650