

Pruebas sobre el comportamiento de la memoria caché

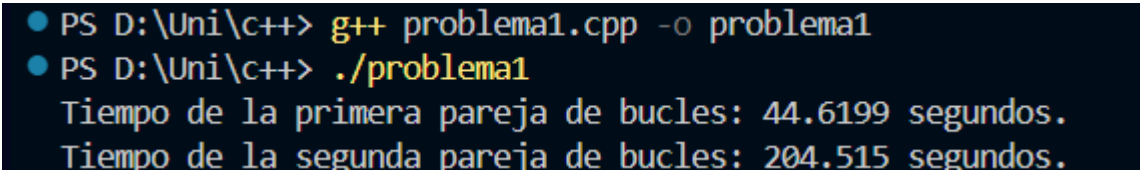
Laboratorio CPD

Alumno: Apaza Andaluz Diego Francisco,

- (a) Implementar y comparar los 2-bucles anidados FOR presentados en el cap. 2 del libro, pag 22.

```
double A[MAX][MAX], x[MAX], y[MAX];  
/* Initialize A and x, assign y = 0 */  
/* First pair of loops */  
for (i = 0; i < MAX; i++)  
    for (j = 0; j < MAX; j++)  
        y[i] += A[i][j]*x[j];  
/* Assign y = 0 */  
/* Second pair of loops */  
for (j = 0; j < MAX; j++)  
    for (i = 0; i < MAX; i++)  
        y[i] += A[i][j]*x[j];
```

Ejemplo con 50000 elementos



```
PS D:\Uni\c++> g++ problema1.cpp -o problema1  
PS D:\Uni\c++> ./problema1  
Tiempo de la primera pareja de bucles: 44.6199 segundos.  
Tiempo de la segunda pareja de bucles: 204.515 segundos.
```

La primera pareja de bucles es más eficiente debido a la forma en que las matrices se almacenan en la memoria (filas). La primera versión se beneficia más de la cache del sistema porque accede a los elementos en la misma fila, que suelen estar almacenados consecutivamente en la memoria. La segunda versión, sin embargo, accede a la matriz por columnas, lo que puede resultar en una demora considerable.

- (b) Implementar en C/C++ la multiplicación de matrices clásica, la versión de tres bucles anidados y evaluar su desempeño considerando diferentes tamaños de matriz.
- (c) Implementar la versión por bloques (investigar en internet), seis bucles anidados, evaluar su desempeño y compararlo con la multiplicación de matrices clásica.
- (d) Ejecutar ambos algoritmos paso a paso, y analizar el movimiento de datos entre la memoria principal y la memoria cache. Hacer una evaluación de acuerdo a la complejidad algorítmica.
- (e) Ejecutar ambos algoritmos utilizando las herramientas valgrind y kcachegrind para obtener una evaluación mas precisa de su desempeño en términos de

- Cálculo de la capacidad de la caché

```
admin@nuevo:~$ lscpu|grep L2
L2 cache: 2 MiB (4 instances)
admin@nuevo:~$
```

Cálculo de la capacidad de la caché L2

Dado que el tamaño de la caché L2 de mi procesador es de 2048 KiB (2 MiB), queremos calcular cuántos elementos de una matriz pueden caber en dicha caché. Asumimos que estamos trabajando con enteros de 32 bits, donde cada entero ocupa 4 bytes en memoria.

Tamaño de la caché L2 en bytes

Sabemos que 1 KiB (kibibyte) equivale a 1024 bytes, por lo tanto:

$$2048 \text{ KiB} \times 1024 \text{ bytes por KiB} = 2,097,152 \text{ bytes}$$

Tamaño de un elemento de la matriz

Un entero de 32 bits ocupa 4 bytes en memoria:

$$\text{Tamaño de cada elemento} = 4 \text{ bytes}$$

Número de elementos que caben en la caché L2

El número total de elementos que pueden almacenarse en la caché L2 se obtiene dividiendo el tamaño total de la caché entre el tamaño de cada elemento:

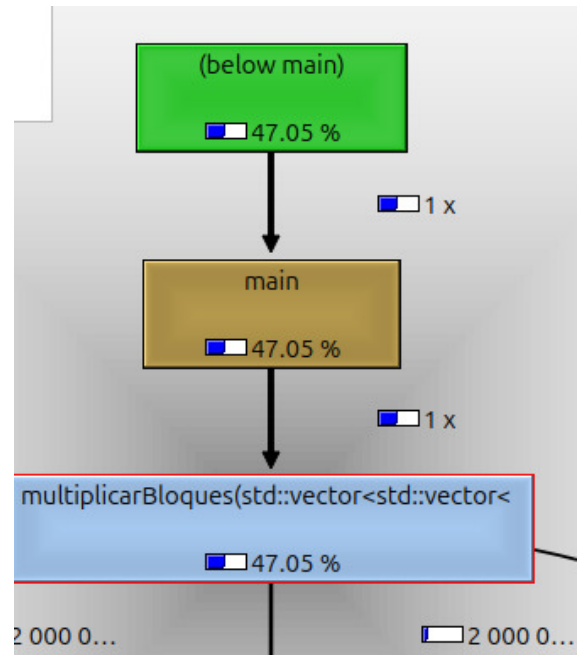
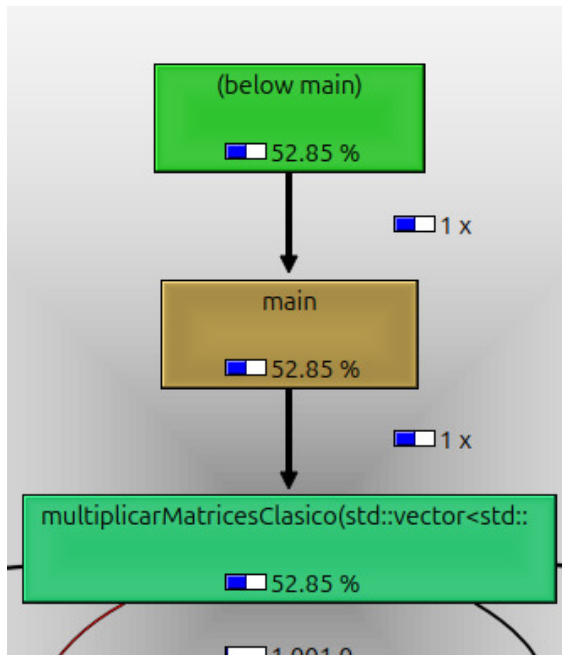
$$\frac{2,097,152 \text{ bytes}}{4 \text{ bytes por elemento}} = 524,288 \text{ elementos}$$

Por lo tanto, la caché L2 puede almacenar un máximo de 524,288 elementos de una matriz de enteros de 32 bits.

- Resultados:

Tamaño de Matrices	Multiplicación Clásica (ms)	Multiplicación por Bloques (ms)
1000	420,726	403,168
500	56,543	49,773

Table 1: Comparación de tiempos de ejecución entre la multiplicación clásica y por bloques (Aclarar que se usó un tamaño de 50 bloques)



Análisis de las funciones

multiplicarMatricesClasica:

- **Incl.:** 52.85%, lo que significa que esta función consume el 52.85% del tiempo total de ejecución, incluyendo las llamadas a otras funciones.
- **Self:** 19.14%, lo que significa que esta función usa 19.14% del tiempo total por sí sola, sin contar las funciones a las que llama.
- **Called:** Se llamó a esta función 19 veces durante la ejecución.

multiplicarBloques:

- **Incl.:** 47.04%, indicando que esta función consume el 47.04% del tiempo total de ejecución.
- **Self:** 18.30%, lo que significa que la función por sí sola consume un 18.30% del tiempo total, lo que es muy cercano a **multiplicarMatricesClasica**.
- **Called:** Se llamó 18 veces durante la ejecución.

Interpretación:

- Las dos funciones principales a las que te estás refiriendo, **multiplicarMatricesClasica** y **multiplicarBloques**, están consumiendo cantidades de tiempo muy similares (alrededor del 52% y 47%, respectivamente). Esto sugiere que ambas implementaciones tienen un rendimiento parecido en este caso particular.
- La asignación de memoria y el uso de `std::vector` también contribuyen al consumo de CPU. Aunque su impacto es menor, es importante tenerlo en cuenta en aplicaciones que trabajen con grandes cantidades de datos, como en la multiplicación de matrices.
- Las operaciones de generación de números aleatorios (`rand`) también aparecen, pero con una contribución pequeña al tiempo total.

(f) **Repositorio de GitHub**

El código fuente del proyecto está disponible en el siguiente enlace: [Repositorio en GitHub](#).