

# **Arquitectura y Organización de Computadores**

## **Práctica 02. Memoria Caché**

Fermín Sola y Eduardo Ezponda

1. a)

Se arranca la computadora con Linux

b)

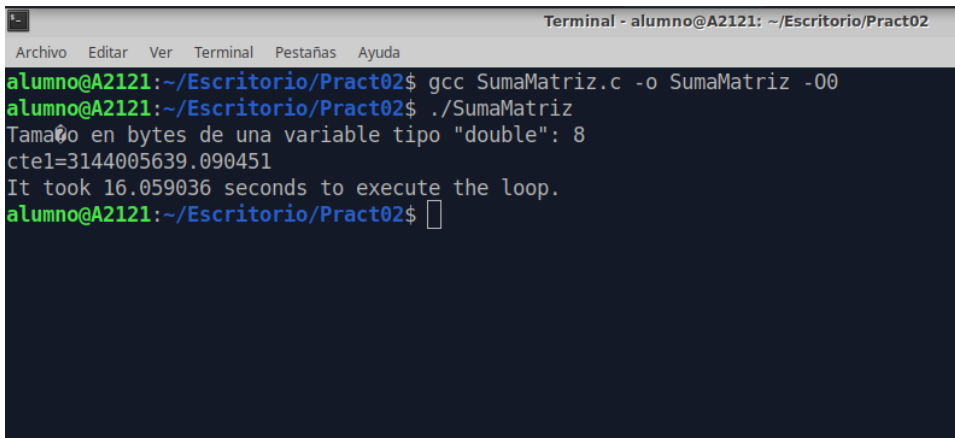
Podemos observar que el acceso se hace por filas, por la posición de los “for”. El primer “for” recorre las filas con la variable *i* que representa la primera posición de la matriz A, y el segundo “for con la variable *j* las columnas. Es decir, para cada fila, se recorren todas sus celdas (columnas), y hasta que éstas no se han terminado de recorrer, no se pasa a la siguiente fila.

c)

Para hacer que el recorrido sea por columnas, hemos cambiado el orden de los “for”, de manera que hasta que no se han recorrido las celdas de una columna, no se pasa a la siguiente. De la misma manera, se podría haber únicamente reemplazado la primera variable *i*, con la segunda variable *j* de la matriz.

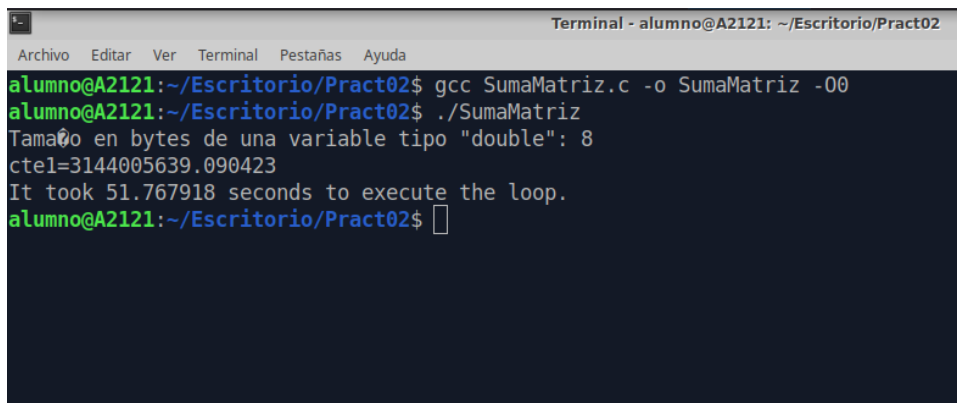
e)

Tiempo matriz por filas: 16.059036 segundos



```
Terminal - alumno@A2121: ~/Escritorio/Pract02
Archivo  Editar  Ver  Terminal  Pestañas  Ayuda
alumno@A2121:~/Escritorio/Pract02$ gcc SumaMatriz.c -o SumaMatriz -O0
alumno@A2121:~/Escritorio/Pract02$ ./SumaMatriz
Tamaño en bytes de una variable tipo "double": 8
ctel=3144005639.090451
It took 16.059036 seconds to execute the loop.
alumno@A2121:~/Escritorio/Pract02$
```

Tiempo matriz por columnas: 51.767918 segundos



```
Terminal - alumno@A2121: ~/Escritorio/Pract02
Archivo  Editar  Ver  Terminal  Pestañas  Ayuda
alumno@A2121:~/Escritorio/Pract02$ gcc SumaMatriz.c -o SumaMatriz -O0
alumno@A2121:~/Escritorio/Pract02$ ./SumaMatriz
Tamaño en bytes de una variable tipo "double": 8
cte1=3144005639.090423
It took 51.767918 seconds to execute the loop.
alumno@A2121:~/Escritorio/Pract02$
```

Rendimiento por Filas / Rendimiento por Columnas =

= Tiempo por Columnas / Tiempo por Filas =  $1 + n / 100 \rightarrow$

$\rightarrow n = ((\text{Tiempo por Columnas} - \text{Tiempo por Filas}) * 100) / \text{Tiempo por Filas} =$

$= ((51.767918 - 16.059036) * 100) / 16.059036 = 222.36\% = n$

Por lo tanto, la mejora de rendimiento del primer programa recorriendo la matriz por filas, respecto al segundo programa por columnas es de un 222.36%.

## 2. a)

8 bytes/celda \* 1024\*64 celdas/fila = 524288 bytes/fila =  $2^{19}$  bytes/fila

A[1][0] está en la dirección de memoria  $2^{19}$

A[2][0] está en la dirección de memoria  $2 * 2^{19} = 2^{20}$

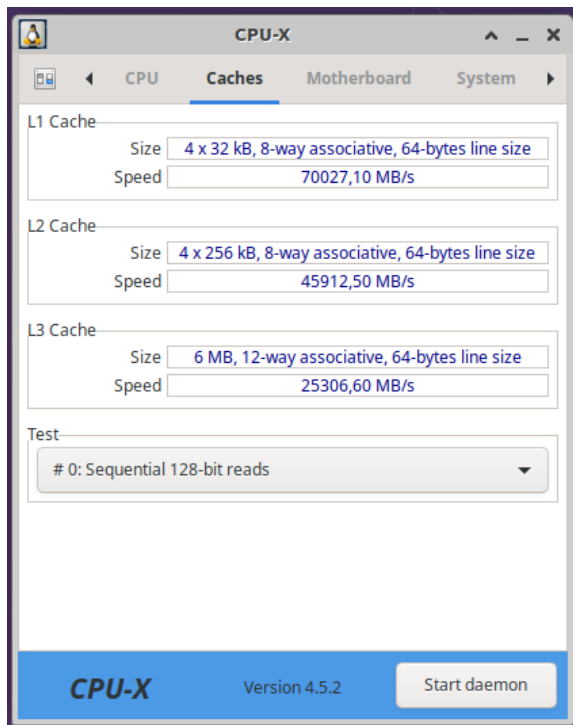
A[i][0] está en la dirección de memoria  $i * 2^{19}$

Para este caso, hay que tener en cuenta que las direcciones de memoria siempre comienzan por el 0. Por lo tanto, si hay N elementos en una fila, irán del 0 al N - 1.

## b)

Representando mediante 32 bits las direcciones de memoria anteriores:

Captura de la cache L3:



#### Datos extraídos de la Cache L3:

- La caché tiene 6 MB de memoria
- Las líneas son de 64 bytes
- Los conjuntos son de 12 vías (líneas)

Como las líneas son de 64 bytes ( $2^6$  bytes), se necesitan 6 bits para direccionar cada byte. Esto implica que quedan  $32-6=26$  bits para direccionar las líneas. Por lo tanto, sabiendo la dirección de memoria de un dato, para conocer su línea, hay que fijarse en los primeros 26 bits.

A [1][0] está en la dirección de memoria  $2^{19} \rightarrow$  **0000 0000 0000 1000 0000 0000 0000 0000** está en la línea **0000 0000 0000 1000 0000 0000 00** que es la línea  $2^{13}$ .

A [2][0] está en la dirección de memoria  $2 \cdot 2^{19} = 2^{20} \rightarrow$  **0000 0000 0001 0000 0000 0000 0000 0000** está en la línea  $2^{14}$ .

A [i][0] estará en la línea  $i \cdot 2^{19} / 2^6 = i \cdot 2^{13}$ .

c)

Como la caché tiene una capacidad de 6 MB ( $3 \cdot 2^{21}$  bytes) y hay 64 ( $2^6$ ) bytes por línea. El número total de líneas de la caché es  $3 \cdot 2^{21} / 2^6 = 98304$  líneas.

Los conjuntos son de 12 vías, por lo que habrá 12 líneas por conjunto. Por lo tanto, el total de conjuntos es de  $98304 / 12 = 8192$  conjuntos.

**d)**

Tenemos 8192 ( $2^{13}$ ) conjuntos, por lo que se necesitarán 13 bits para direccionar los conjuntos.

Es decir, de la dirección de memoria, los primeros 13 bits son para el tag, los 13 siguientes para el conjunto, y los últimos 6 para el byte.

<b>TAG</b>	<b>CONJUNTO</b>	<b>BYTE</b>
13 bits	13 bits	6 bits

A [0][0] está en la dirección 0000 0000 0000 **0000 0000 0000 0000** 0000 y pertenece al conjunto 0

A [1][0] está en la dirección 0000 0000 0000 **1000 0000 0000 0000** 0000 y pertenece al conjunto 0

A [2][0] está en la dirección 0000 0000 0001 **0000 0000 0000 0000** 0000 y pertenece al conjunto 0

A [i][0] también pertenecerá al conjunto 0

**e)**

Como las líneas son de 64 bytes, y los datos de la matriz son de 8 bytes, cada línea contiene

$64 / 8 = 8$  datos. Los elementos serían los siguientes:

A [i][0], A [i][1], A [i][2], A [i][3], A [i][4], A [i][5], A [i][6] y A [i][7]

**f)**

Si la matriz tiene 64 filas, para leer una columna se han de traer 64 líneas.

Al leer por primera vez la primera columna, se producen 64 fallos, porque ninguna de las líneas que se están leyendo están guardadas en la caché.

Todas las líneas van al mismo conjunto, y cada conjunto tiene 12 líneas. Las primeras 12 líneas se guardan sin problemas en el conjunto, pero cuando empiezan a llegar más, ya no caben en el conjunto 0 y el algoritmo LRU comienza a eliminar las líneas que llevan más tiempo sin utilizarse. Por lo tanto, tras leer la primera columna, solo se quedan las últimas 12 líneas leídas.

g)

Como hemos comentado anteriormente, tras leer la primera columna, solo se quedan las 12 últimas líneas (de las 12 últimas filas), por lo que la línea de la fila 0 no estaría guardada en la caché. Al leer el dato A [0][1] se producirá un fallo en la caché.

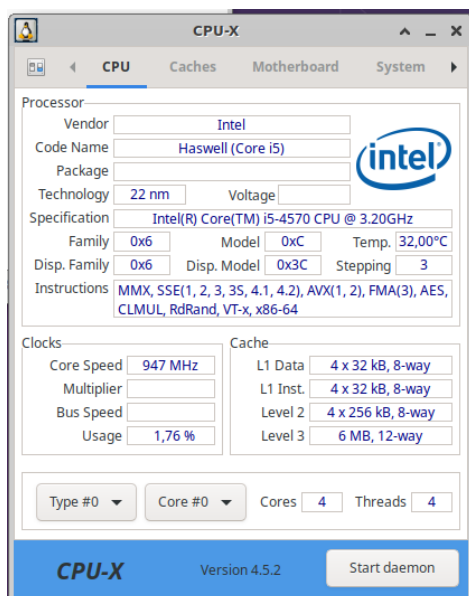
Como a medida que avanzamos, se van quitando las líneas que llevan más tiempo sin usarse, al leer la segunda columna todos los intentos de lectura de dato producirían error. Es decir, que habría 64 fallos otra vez. Lo mismo pasaría con la tercera, y todas las demás.

Como al recorrer la matriz, ningún dato buscado está en la caché, el número de fallos es igual al número de datos, que es  $64 \times 65536 = 4194304 = 2^{22}$  fallos (100% de fallos).

Teniendo en cuenta que la suma se realiza 1500 veces, la cantidad de fallos es de  $1500 \times 2^{22}$ .

h)

Captura CPU:



Tiempo Matriz por Filas = 16.059036 segundos

Tiempo Matriz por Columnas = 51.767918 segundos

Ambas matrices tienen  $2^{22}$  datos (número de filas \* número de columnas), por lo que, por media, para acceder a cada dato se tarda

Acceso a Dato Matriz por Filas =  $16.059036 / 2^{22} = 3.828 \times 10^{-6}$  segundos

Acceso a Dato Matriz por Columnas =  $51.767918 / 2^{22} = 12.342 \cdot 10^{-6}$  segundos

Suponemos que, para el acceso al dato de la matriz por filas, no se produce fallo y que en el acceso a la matriz por columnas siempre se produce fallo. Por lo tanto, la diferencia de acceso a los datos de cada matriz nos daría como resultado el tiempo de penalización aproximado por cada fallo.

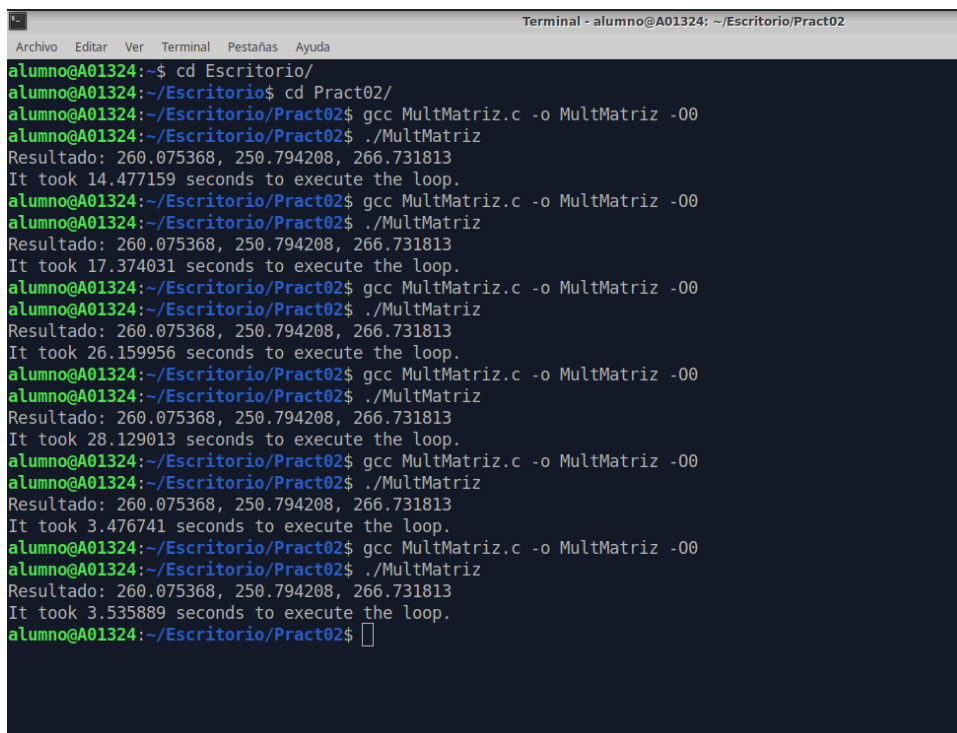
$12.342 \cdot 10^{-6} - 3.828 \cdot 10^{-6} = 8.514 \cdot 10^{-6}$  segundos de penalización por fallo

La frecuencia de ciclo es de 3.2 GHz, por lo que el periodo es de  $1 / 3.2 \cdot 10^9 = 3.125 \cdot 10^{-10}$  segundos el ciclo

El tiempo de penalización en ciclos es de  $8.514 \cdot 10^{-6} / 3.125 \cdot 10^{-10} = 27245$  ciclos de reloj de CPU

### 3. a)

Diferentes valores de tiempo para cada combinación:



```
Terminal - alumno@A01324: ~/Escritorio/Pract02
Archivo Editar Ver Terminal Pestañas Ayuda
alumno@A01324:~$ cd Escritorio/
alumno@A01324:~/Escritorio$ cd Pract02/
alumno@A01324:~/Escritorio/Pract02$ gcc MultMatriz.c -o MultMatriz -O0
alumno@A01324:~/Escritorio/Pract02$ ./MultMatriz
Resultado: 260.075368, 250.794208, 266.731813
It took 14.477159 seconds to execute the loop.
alumno@A01324:~/Escritorio/Pract02$ gcc MultMatriz.c -o MultMatriz -O0
alumno@A01324:~/Escritorio/Pract02$ ./MultMatriz
Resultado: 260.075368, 250.794208, 266.731813
It took 17.374031 seconds to execute the loop.
alumno@A01324:~/Escritorio/Pract02$ gcc MultMatriz.c -o MultMatriz -O0
alumno@A01324:~/Escritorio/Pract02$ ./MultMatriz
Resultado: 260.075368, 250.794208, 266.731813
It took 26.159956 seconds to execute the loop.
alumno@A01324:~/Escritorio/Pract02$ gcc MultMatriz.c -o MultMatriz -O0
alumno@A01324:~/Escritorio/Pract02$ ./MultMatriz
Resultado: 260.075368, 250.794208, 266.731813
It took 28.129013 seconds to execute the loop.
alumno@A01324:~/Escritorio/Pract02$ gcc MultMatriz.c -o MultMatriz -O0
alumno@A01324:~/Escritorio/Pract02$ ./MultMatriz
Resultado: 260.075368, 250.794208, 266.731813
It took 3.476741 seconds to execute the loop.
alumno@A01324:~/Escritorio/Pract02$ gcc MultMatriz.c -o MultMatriz -O0
alumno@A01324:~/Escritorio/Pract02$ ./MultMatriz
Resultado: 260.075368, 250.794208, 266.731813
It took 3.535889 seconds to execute the loop.
alumno@A01324:~/Escritorio/Pract02$
```

### b)

La mejor combinación de bucles es la siguiente:

for (i=0; i<N; i++)

for (k=0; k<N; k++)

for (j=0; j<N; j++)

C[i][j]=C[i][j]+A[i][k]\*B[k][j];

que tarda 3.476741 segundos

Razonamiento:

Tenemos 3 tipos de accesos porque hay 3 tipos de matrices:  $C[i][j]$ ,  $A[i][k]$  y  $B[k][j]$ .

Para reducir el tiempo total, lo que hay que intentar es hacer es recorrer las 3 matrices por filas en vez de por columnas. Como tenemos el "for" de  $i$  antes que el de  $j$ , la matriz  $C$  ( $C[i][j]$ ) la vamos a recorrer por filas (rápido). Como el "for" de  $k$  esta también por encima de  $j$ , la matriz  $B$  también vamos a recorrerla por filas. La matriz  $A$ , sin embargo, se recorre por columnas, puesto que el "for" de  $k$  está por debajo del "for" de  $i$ . En resumen, recorreremos tres matrices, dos de ellas por filas, por lo que reducimos al máximo el tiempo tardado al reducir el número de fallos.