



Instituto Politécnico Nacional. Escuela Superior de Cómputo (ESCOM).

Desarrollo de Sistemas Distribuidos.

Tarea 2. “Uso eficiente de la memoria cache”.

**Profesor Carlos Pineda
Alumno: Sánchez Martínez Eli
Grupo: 4CM5**

07/10/2020

Objetivo: Uso de la memoria cache a través de dos programas con diferente implementación, pero la misma salida, observando como hacer uso eficiente de la memoria cache.

Datos:

- Marca: AMD
- Modelo: E2-1800 APU with Radeon
- Tamaño de la cache: 32k
- Tamaño de la RAM: 4GB

Código:

MultiplicaMatriz.java

Esta operación funciona correctamente sin embargo es ineficiente ya que cada vez que se accede a un elemento de la matriz se necesita transferir una línea completa de cache.

```
1. //MultiplicaMatriz 1
2. class MultiplicaMatriz
3. {
4.     static int N = 1000;
5.     static int[][] A = new int[N][N];
6.     static int[][] B = new int[N][N];
7.     static int[][] C = new int[N][N];
8.
9.     public static void main(String[] args)
10.    {
11.        long t1 = System.currentTimeMillis();
12.
13.        // inicializa las matrices A y B
14.
15.        for (int i = 0; i < N; i++)
16.            for (int j = 0; j < N; j++)
17.            {
18.                A[i][j] = 2 * i - j;
19.                B[i][j] = i + 2 * j;
20.                C[i][j] = 0;
21.            }
22.
23.        // multiplica la matriz A y la matriz B, el resultado queda en la matriz C
24.
25.        for (int i = 0; i < N; i++)
26.            for (int j = 0; j < N; j++)
27.                for (int k = 0; k < N; k++)
28.                    C[i][j] += A[i][k] * B[k][j];
29.
30.        long t2 = System.currentTimeMillis();
31.        System.out.println("Tiempo: " + (t2 - t1) + "ms");
32.    }
```

33. }

Multiplicación_2.java

Este programa es mas eficaz ya que los elementos de una de las matrices se puede leer de forma secuencial, lo cual nos ayuda a los datos temporales y las localidades para acceder a estos.

```
1. //UpGraded Multiplicacion
2. public class MultiplicaMatriz_2 {
3.     static int N = 1000;
4.     static int[][] A = new int[N][N];
5.     static int[][] B = new int[N][N];
6.     static int[][] C = new int[N][N];
7.
8.     public static void main(String[] args)
9.     {
10.         long t1 = System.currentTimeMillis();
11.
12.         // inicializa las matrices A y B
13.
14.         for (int i = 0; i < N; i++)
15.             for (int j = 0; j < N; j++)
16.             {
17.                 A[i][j] = 2 * i - j;
18.                 B[i][j] = i + 2 * j;
19.                 C[i][j] = 0;
20.             }
21.
22.         // transpone la matriz B, la matriz traspuesta queda en B
23.
24.         for (int i = 0; i < N; i++)
25.             for (int j = 0; j < i; j++)
26.             {
27.                 int x = B[i][j];
28.                 B[i][j] = B[j][i];
29.                 B[j][i] = x;
30.             }
31.
32.         // multiplica la matriz A y la matriz B, el resultado queda en la matriz C
33.         // notar que los indices de la matriz B se han intercambiado
34.
35.         for (int i = 0; i < N; i++)
36.             for (int j = 0; j < N; j++)
37.                 for (int k = 0; k < N; k++)
38.                     C[i][j] += A[i][k] * B[j][k];
39.
40.         long t2 = System.currentTimeMillis();
41.         System.out.println("Tiempo: " + (t2 - t1) + "ms");
42.     }
43. }
```

Comparación:

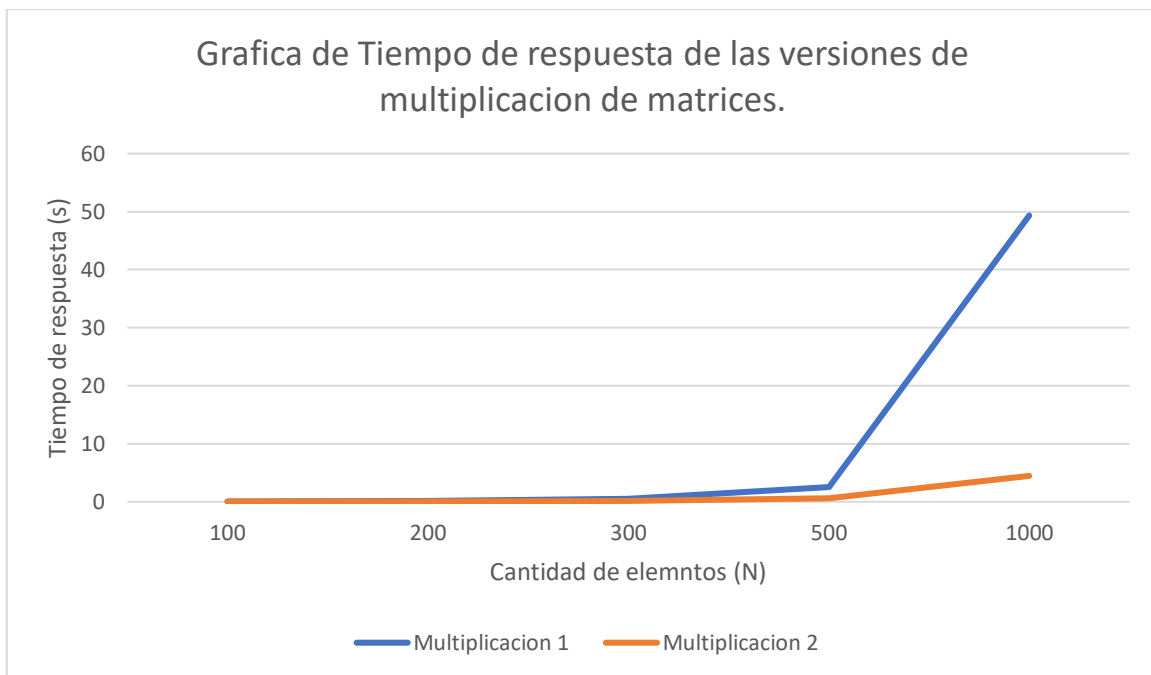
Para comparar entre el primer programa y el segundo, se le fueron dando valores distintos al tamaño de nuestras matrices:

| N | Matriz Multiplica | Matriz Multiplica 2 |
|------|-------------------|---------------------|
| 100 | 42ms =0.042s | 46ms = 0.046s |
| 200 | 140ms=0.140s | 75ms=0.075ms |
| 300 | 552ms=0.552s | 177ms=0.177s |
| 500 | 2560ms=2.560s | 583ms=0.583s |
| 1000 | 49337ms=49.337s | 4461ms=4.461s |

Podemos observar que el primer código tiene una complejidad en el peor de los casos de $O(n^3)$ mientras que el segundo programa se podría mantener con una complejidad de $O(n^2)$ pero en el peor de los casos puede ser de $O(n^3)$.

Esto se debe a como se accede a la memoria cache para sacar los datos, por ejemplo el primer programa va resolviendo la matriz renglón por renglón debido a esto hace ineficiente este programa, pero el segundo para darle solución a este problema es transponer la matriz y esto nos da una mayor flexibilidad tanto para realizar las operaciones como la memoria cache, para que tenga menor numero de operaciones a realizar.

Grafica:



Pruebas:

```
+ parte_2 top x TAREA_2: java TAREA_2: javac
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz
Tiempo: 42ms
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz
Tiempo: 140ms
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz
Tiempo: 522ms
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz
Tiempo: 2560ms
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz
Tiempo: 49337ms
```

1. Tiempo de respuesta del primer programa con N de tamaño.

```
+ x TAREA_2: java TAREA_2: javac
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz_2
Tiempo: 46ms
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz_2
Tiempo: 75ms
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz_2
Tiempo: 177ms
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz_2
Tiempo: 583ms
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ java MultiplicaMatriz_2
Tiempo: 4461ms
```

2. Tiempo de respuesta del segundo programa con N tamaño.

```
+ x TAREA_2: lscpu TAREA_2 sistemas-distribuidos
eliasma@ELIASHP:~/Documents/sistemas-distribuidos/DISTRIBUIDOS/TAREA_2$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                 2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):              1
NUMA node(s):          1
Vendor ID:              AuthenticAMD
CPU family:             20
Model:                  2
Model name:             AMD E2-1800 APU with Radeon(tm) HD Graphics
Stepping:               0
CPU MHz:                1266.355
CPU max MHz:            1700.0000
CPU min MHz:            850.0000
BogoMIPS:               3394.16
Virtualization:         AMD-V
L1d cache:              32K
L1i cache:              32K
L2 cache:               512K
NUMA node0 CPU(s):     0,1
Flags:                   fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid aperfmperf pni monitor ssse3 cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch ibt skinit wdt hw_pstate umwait arat
```

3. Información de la computadora en la cual se realizó la prueba.

```
sudo dmidecode --type memory

# dmidecode 3.1
Getting SMBIOS data from sysfs.
SMBIOS 2.7 present.

Handle 0x001B, DMI type 16, 23 bytes
Physical Memory Array
    Location: System Board Or Motherboard
    Use: System Memory
    Error Correction Type: None
    Maximum Capacity: 4 GB
    Error Information Handle: Not Provided
    Number Of Devices: 2
```

4. Información de la RAM