INFORME PRÁCTICA 3

Alejandro Santorum Varela - alejandro.santorum@estudiante.uam.es
David Cabornero Pascual - david.cabornero@estudiante.uam.es
Prácticas Arquitecturas de Ordenadores - Pareja PM19
Universidad Autónoma de Madrid

19-11-2018

Contents

1 Introducción

El objetivo de esta práctica es experimentar el efecto de las memorias caché en el rendimiento de un programa de usuario. Para ello utilizaremos el lenguaje de programación C, scripting utilizando bash y un poco de Python. Además, se usará el framework *Valgrind* (más en concreto una herramienta llamada *Cachegrind*) para analizar el programa variando algunos parámetros de la memoria caché.

2 Ejercicio 0: Información de la caché del sistema

En primer lugar vamos a obtener la información relativa a la memoria caché de la máquina sobre la que estamos trabajando. Para ello ejecutaremos el sencillo comando siguiente: `cat /proc/cpuinfo`. Como la salida era de gran extensión, se adjuntan dos archivos txt en el material de esta práctica con las salidas de este comando, uno es obtenido con los ordenadores del laboratorio (cpuinfo_lab.txt) y otro con nuestro ordenador personal (cpuinfo_per.txt), donde se han ejecutado los siguientes ejercicios de esta práctica.

```
e356974@12-3-66-172:~$ touch cpuinfo.txt
e356974@12-3-66-172:~$ cat /proc/cpuinfo >> cpuinfo.txt
```

La información más resaltable que podemos analizar es que los ordenadores de los laboratorios cuentan con procesadores de cuatro núcleos, cada uno de ellos modelo i5-7500 a 3.4 GHz con un tamaño caché de 6144kB.

A continuación hemos profundizado más en la obtención de datos de las memorias caché con el comando `getconf -a | grep -i cache`:

```
e356974@12-3-66-172:~$ getconf -a | grep -i cache
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  6291456
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

No es muy difícil analizar esta información: los ordenadores de los laboratorio cuentan con dos memorias caché de primer nivel (una de datos y otra de instrucciones) de 32768 bytes (32kB) asociativas de 8 vías y con un tamaño de línea de 8 bytes cada una. Además, tienen dos memorias caché más: una de segundo nivel, y otra de tercer nivel. Se recoge la información de la memoria caché de cuarto nivel, pero se ve que su tamaño es nulo.

La memoria de segundo nivel es de 256kB asociativa de 4 vías y tamaño de línea de 8 bytes. Por último, la de tercer nivel es de 6MB, asociativa de 12 líneas con un tamaño de línea de 8 bytes.

Para finalizar este ejercicio, hemos ejecutado el comando `lstopo`:

```
e356974@12-3-66-172:~$ lstopo
Machine (7399MB)
  Package L#0 + L3 L#0 (6144KB)
    L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
    L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
    L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#2)
    L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#3)
  HostBridge L#0
    PCI 8086:5912
      GPU L#0 "renderD128"
      GPU L#1 "card0"
      GPU L#2 "controlD64"
    PCI 8086:a282
      Block(Disk) L#3 "sda"
    PCIBridge
      PCI 10ec:8168
        Net L#4 "enp1s0"
```

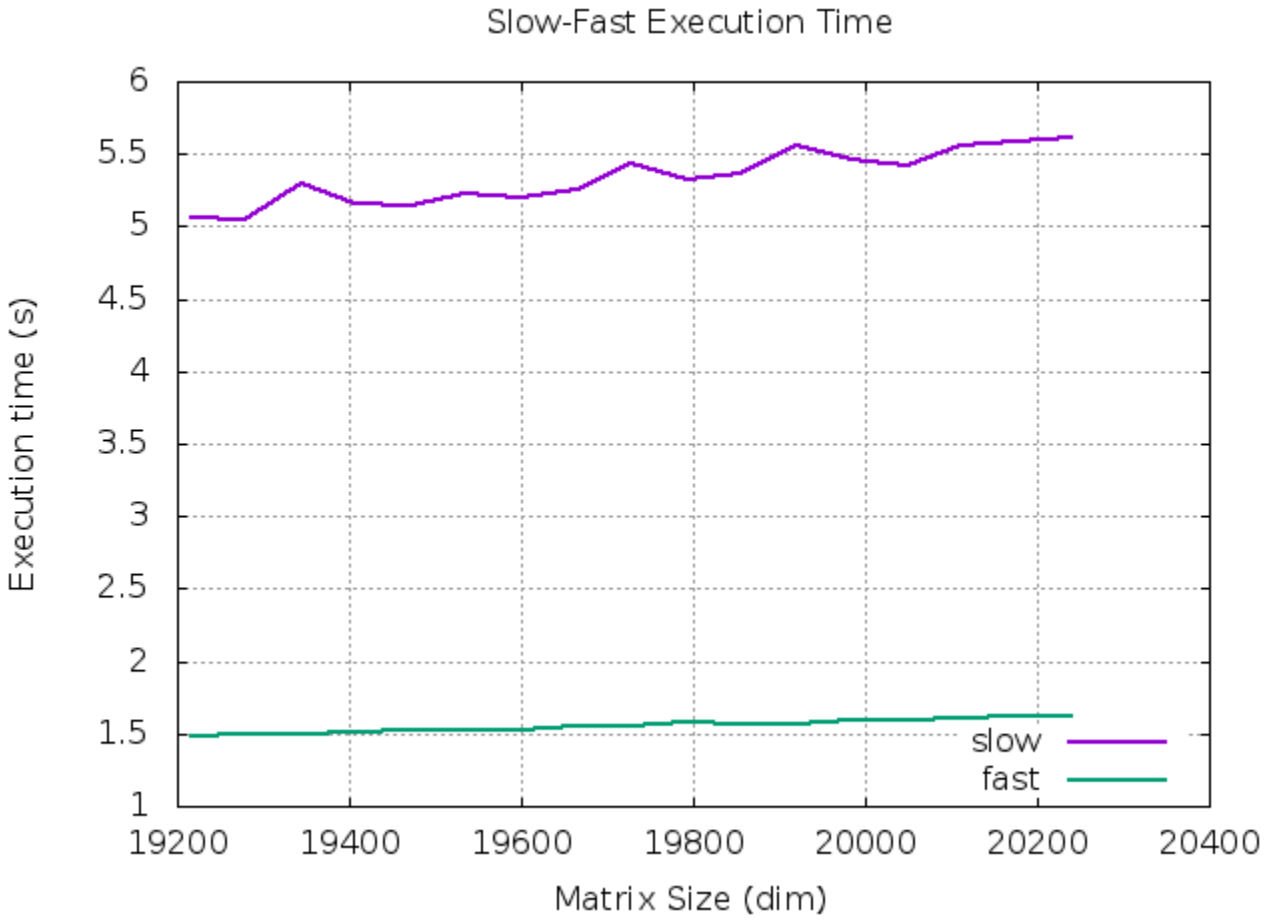
Nos repite alguna información ya obtenida con el comando anterior. Hay algún dato que no sabemos aún cómo interpretar, como el apartado PCIBridge, HostBridge , etc.

3 Ejercicio 1: Memoria caché y rendimiento

En este ejercicio vamos a comprobar de manera empírica como una buena técnica a la hora de acceder a los datos, aprovechando correctamente las memorias caché, puede mejorar el rendimiento de un programa.

Para realizar este ejercicio se han utilizado tamaños de matriz desde 19216 (10000 + 1024*9) hasta 20240 (10000 + 1024*10) con saltos de 64 en 64, intercalando los programas *slow* y *fast* y los diferentes tamaños de matrices. El intercalamiento es importante porque sino el sistema operativo puede tener ya mapeados en memoria algunos datos de la ejecución anterior que no ha liberado por falta de necesidad, por esa razón se le obliga a realizar un experimento lo más puro posible mezclando los diferentes programas y diferentes tamaños de matriz. Además, como estamos analizando tiempos de ejecución hemos repetido el experimento 20 veces para calcular una buena estimación.

El resultado obtenido es el siguiente:



Se puede observar que el programa destinado a ser más lento, es efectivamente, más lento que el programa que aprovecha mejor la memoria caché.

La pregunta ahora es, ¿por qué aprovechar mejor la memoria caché el programa *fast* que el *slow*? Esto es porque va percorriendo los elementos de la matriz de la misma forma que se encuentran juntos en la memoria caché:

```
tipo compute(tipo **matrix,int n){
    tipo sum=0;
    int i,j;

    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            sum += matrix[j][i];
        }
    }
    return sum;
}
```

Esta es la función compute(.) en el programa *slow*, donde el bucle más interno (índice j) va iterando los elementos de una misma columna, pero los elementos de la matriz están colocados en la memoria caché por filas.

```
tipo compute(tipo **matrix,int n){
    tipo sum=0;
    int i,j;

    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            sum += matrix[i][j];
        }
    }
    return sum;
}
```

Esta es la función compute(.) en el programa *fast*. Se ve que el bucle interno (índice j) itera sobre los elementos de una misma fila, y como hemos dicho, son los que se encuentran contiguamente en la memoria caché, por lo que su tiempo de ejecución se reduce drásticamente.

Para la realización de este ejercicio se ha programado un script en bash llamado **ejercicio1.sh**, que se encarga de ejecutar los programas *slow* y *fast* y recoger sus respectivos tiempos de ejecutación para posteriormente guardarlos en un fichero.

Una vez que todas las repiticiones se hayan completado, el script ejecuta un programa escrito en python llamado **cal_mean_slow_fast.py** que recibe como parámetros el fichero donde se han escrito todos los tiempos de ejecución y el número de diferentes tamaños de matriz con los que se han ejecutado previamente *slow* y *fast* y calcula la media de los tiempos para cada tamaño de matriz.

Para finalizar, el script ejecuta unos comandos de GNUPlot para crear la gráfica expuesta previamente.

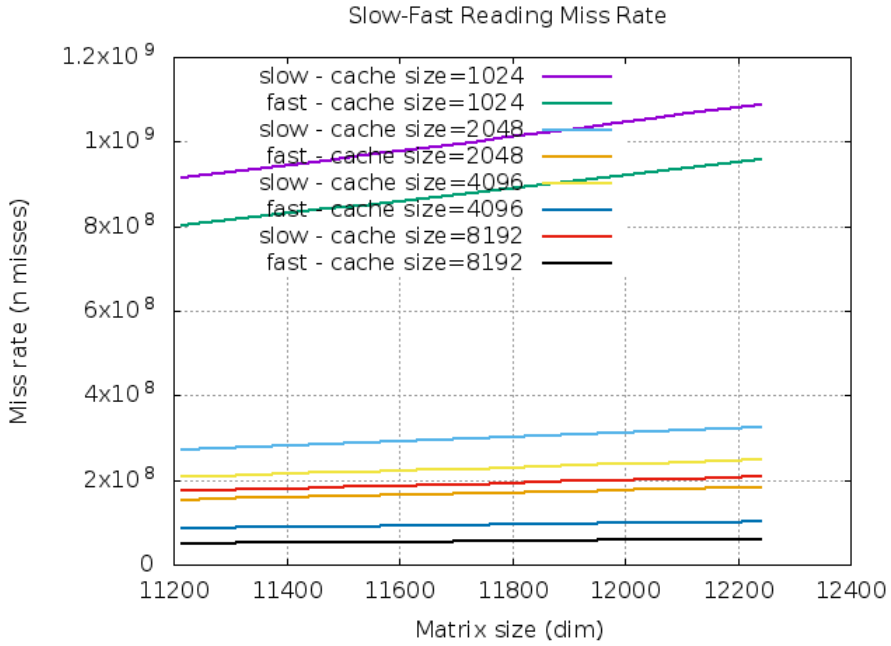
4 Ejercicio 2: Tamaño de la caché y rendimiento

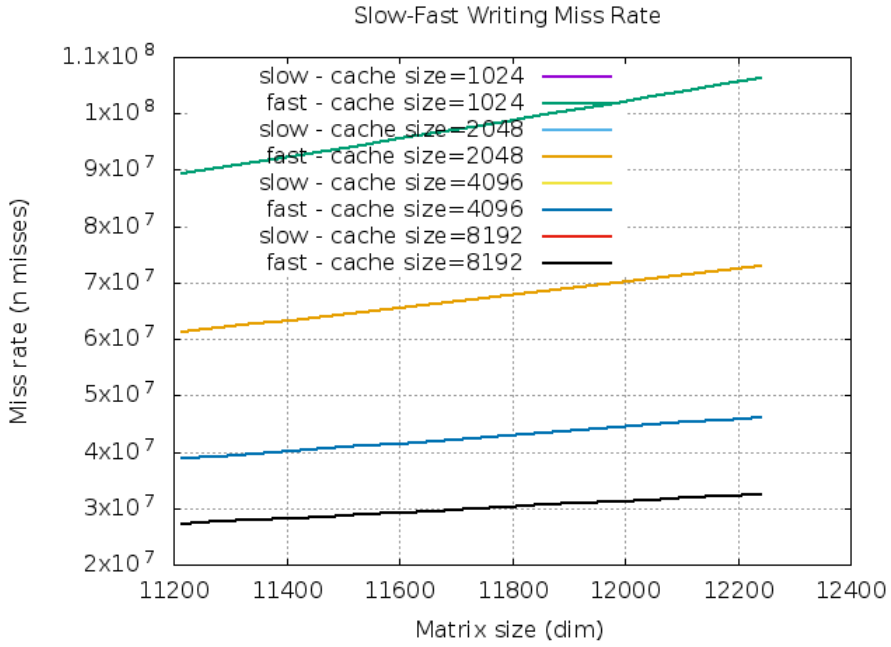
En este ejercicio vamos a utilizar la herramienta *Cachegrind* del framework Valgrind para emular diferentes tamaños y configuraciones de las memoria caché y, posteriormente, analizar el número de fallos de memoria que se preducen en función del tamaño caché.

Para la realización de este ejercicio se ha implementado un script llamado **ejercicio2.sh**. Este script para cada tamaño caché deseado (1024, 2048, 4096 y 8192 bytes) itera sobre diferentes tamaños de matriz (en el rango 11216-12240 de saltos de 64) y ejecuta los programas *slow* y *fast* para posteriormente, ayudado de la herramienta *Cachegrind*, recoge los fallos en escritura y lectura de la memoria caché y los guarda en un fichero de texto (un fichero de texto para cada tamaño de la memoria caché).

Como aquí solo estamos contando el número de fallos producidos por la memoria caché, no necesitamos repetir el experimento.

Por lo tanto, una vez que el doble bucle haya finalizado, procederemos a analizar los datos con GNUPlot, que creará dos gráficas: una para los fallos en lectura y otro para los fallos en escritura. A continuación podemos ver un ejemplo:





Las gráficas de por si ya son muy intuitivas. Podemos observar primero en la gráfica de fallos de lectura que cuanto menor es el tamaño caché, mas fallos se producen (como cabía esperar). Adicionalmente, el programa *slow* genera una mayor cantidad de fallos que el programa *fast*, lo cual tiene sentido ya que el segundo es más rápido que el primero, y eso

se debe en parte por el correcto aprovechamiento de la memoria caché al percorrer la matriz por filas y no por columnas.

En la segunda gráfica, fallos de escritura, podemos ver que usando el mismo tamaño caché los dos programas producen el mismo número de fallos debido a que escriben en el mismo sitio y el mismo número de veces(las gráficas pueden estar superpuestas y solo ser visible un color de las dos). Además, como podíamos haber predicho antes de ejecutar el programa, cuanta mayor sea el tamaño de la memoria caché, menos fallos se producen.

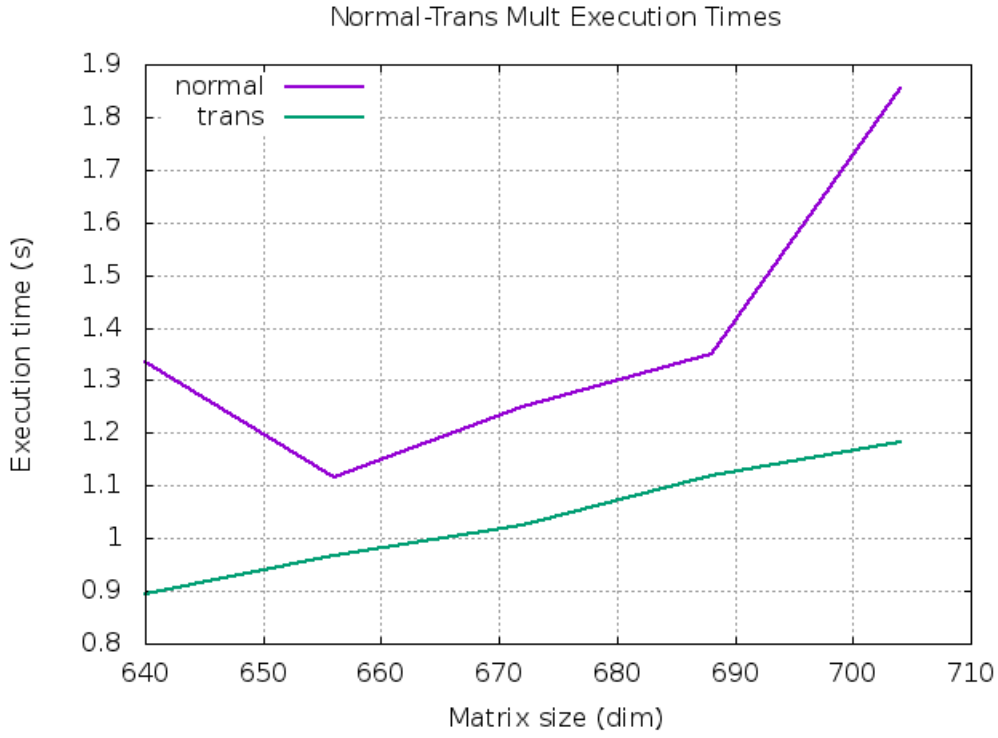
5 Ejercicio 3: Caché y multiplicación de matrices

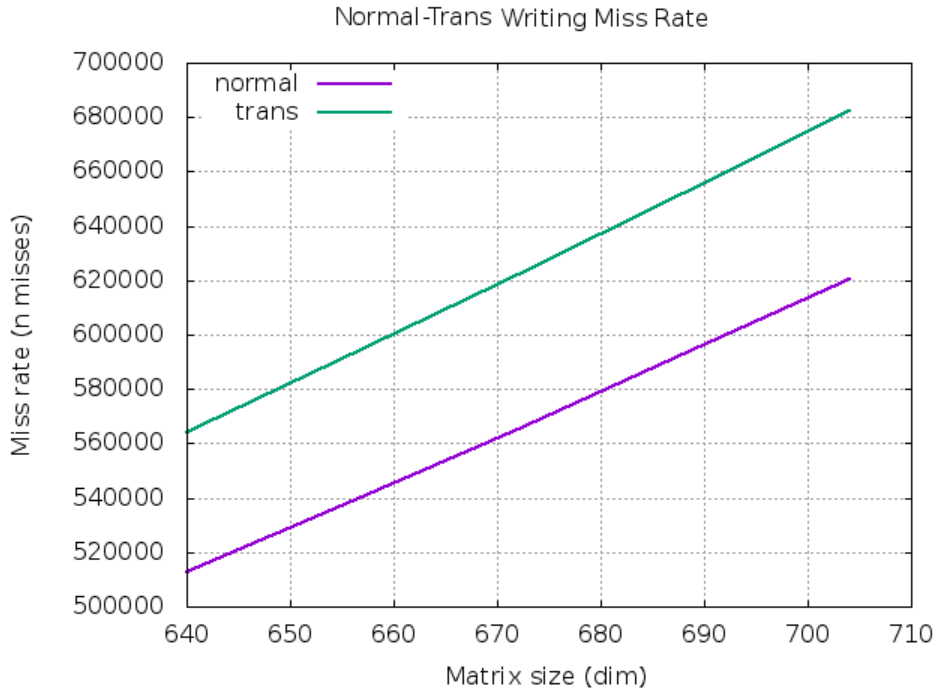
Este ejercicio tiene como objetivo implementar dos programas que lleven a cabo la multiplicación de matrices cuadradas del mismo tamaño, uno que realice la multiplicación de forma estándar y otro que trasponga la matriz previamente para luego percorrerla por filas a la hora de realizar la operación. Ambos programas generan el mismo resultado, pero no con el mismo tiempo de ejecución ni con los mismos fallos de caché. Nuestro objetivo será analizar cual de los dos programas es más ventajoso a la hora de realizar dicha operación.

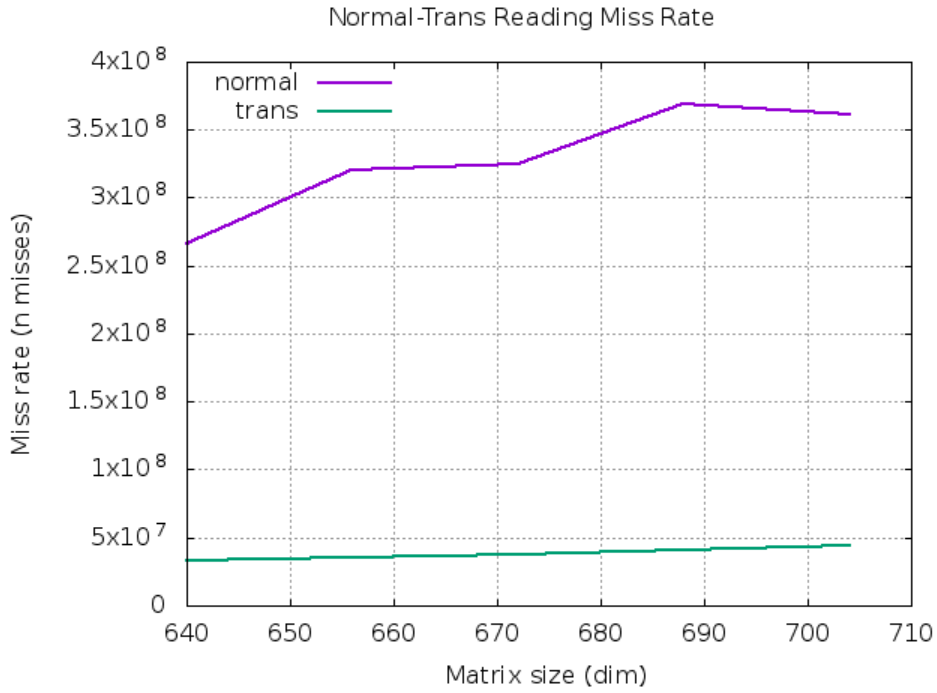
Para ello se ha programado un script en bash llamado **ejercicio3.sh** que primero recogerá datos de los tiempos de ejecución de ambos programas para diferentes tamaños de matriz (en nuestro caso desde 64+64*9=640 hasta 64+64*10=704 en saltos de 8). Estos datos se guardarán en un fichero, y se utilizará como parámetro de entrada de un programa escrito en python llamado **cal_mean_matrix_mult.py**, que calcula la media de los tiempos.

A continuación, se recogen datos sobre los fallos caché (tanto escritura como lectura) utilizando la herramienta *Cachegrind* y se guardan en otro fichero de texto. El programa en python ya mencionado, aparte de calcular el promedia de los tiempos de ejecución, también colocará todos los datos en un buen formato para ser utilizado próximamente por GNUPlot.

Por último, GNUPlot se encarga de graficar todos los datos obtenidos. Justo debajo se muestran las gráficas obtenidas (fíjese que se ha separado la gráfica de fallos de caché en dos porque el número de fallos en escritura son mucho menores a los fallos en lectura, por lo que en una misma gráfica no se pueden apreciar correctamente).


Normal-Trans Mult Execution Times
normal
trans
Execution time (s)
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
640
650
660
670
680
690
700
710
Matrix size (dim)


Normal-Trans Writing Miss Rate
normal
trans
Miss rate (n misses)
700000
680000
660000
640000
620000
600000
580000
560000
540000
520000
500000
640
650
660
670
680
690
700
710
Matrix size (dim)



La primera gráfica, adjuntada con el nombre **mult_time.png**, recoge un ejemplo de los posibles tiempos de ejecución que tendrían los programas de multiplicación de matrices. Es fácil ver que la multiplicación que sigue el algoritmo estándar necesita más tiempo (aparte de que resulta ser más irregular) para su ejecución que el algoritmo que traspone primero la segunda matriz y después la percorre por filas aprovechando así que los datos de una fila de una matriz se colocan contiguamente en la memoria caché.

La segunda gráfica, adjuntada con el nombre **mult_cache_escritura.png**, corrosponde con el número de fallos de caché en escritura para ambos programas. Se puede apreciar que la tendencia de fallos en escritura de ambos programas es linealmente dependiente (una es un múltiplo de la otra), y que el número que fallos del programa que se ayuda de la trasposición es mayor que el que realiza la multiplicación estándar. Esto se debe a que el primero mencionado genera dos matrices aleatoriamente (M_1 y M_2), y la segunda M_2 la traspone guardando los datos en una matriz vacía B tal que $B = M_2^T$. Como en el programa que hace la multiplicación estándar no tiene que hacer esto, produce menos fallos en escritura. En el caso de que se implementase la trasposición *in-place* sobre M_2, es decir, sin necesitar de otra matriz vacía auxiliar para guardar la traspuesta, el número de fallos en escritura del programa apodado como `normal` tendría igual o más fallos de caché en escritura que el programa denominado como `trans`.

Por último, la tercera imagen mostrada (**mult_cache_lectura.png**) enseña gráficamente la relación entre el tamaño de la matriz y el número de fallos de caché en lectura de ambos programas. Como cabía esperar, el número de fallos del programa que realiza la multipli-

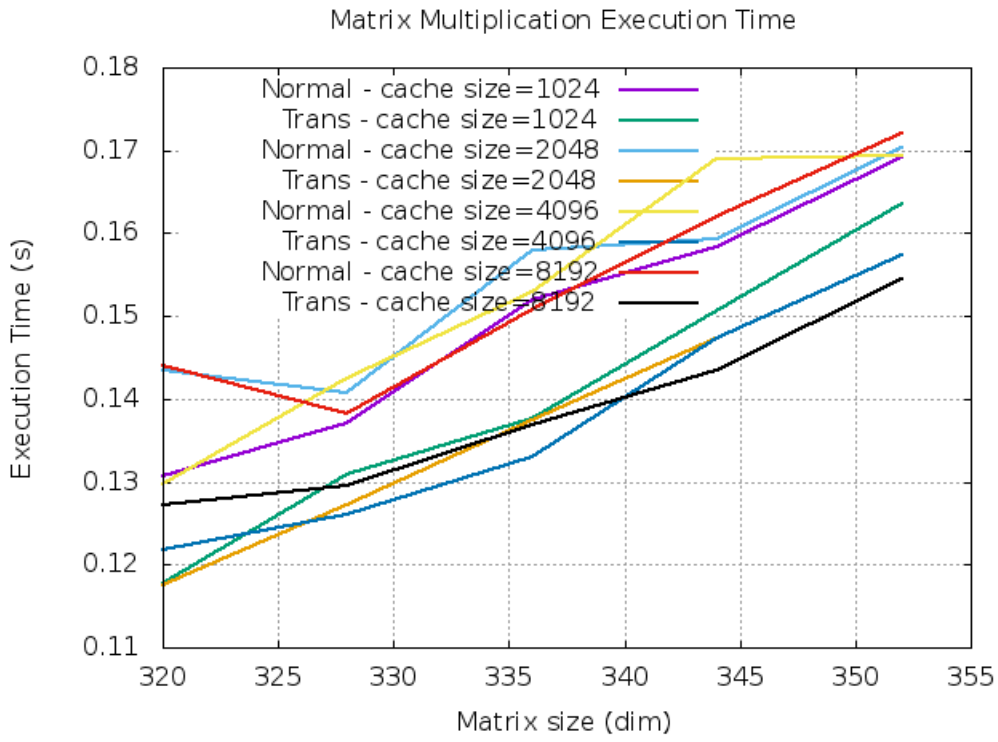
cación de matrices estándar es mucho mayor a la que realiza la multiplicación ayudándose previamente de la trasposición de la matriz para luego percorrerla por filas, que es como están los datos colocados en la memoria caché, y de ahí que se produzcan muchos menos fallos de lectura.

6 Ejercicio 4 (Opcional): Configuraciones de Caché en la multiplicación de matrices

Llegamos finalmente al último ejercicio de esta práctica, que consistía en hacer un pequeño estudio de como las diferentes configuraciones podía afectar al comportamiento de los programas de multiplicación de matrices.

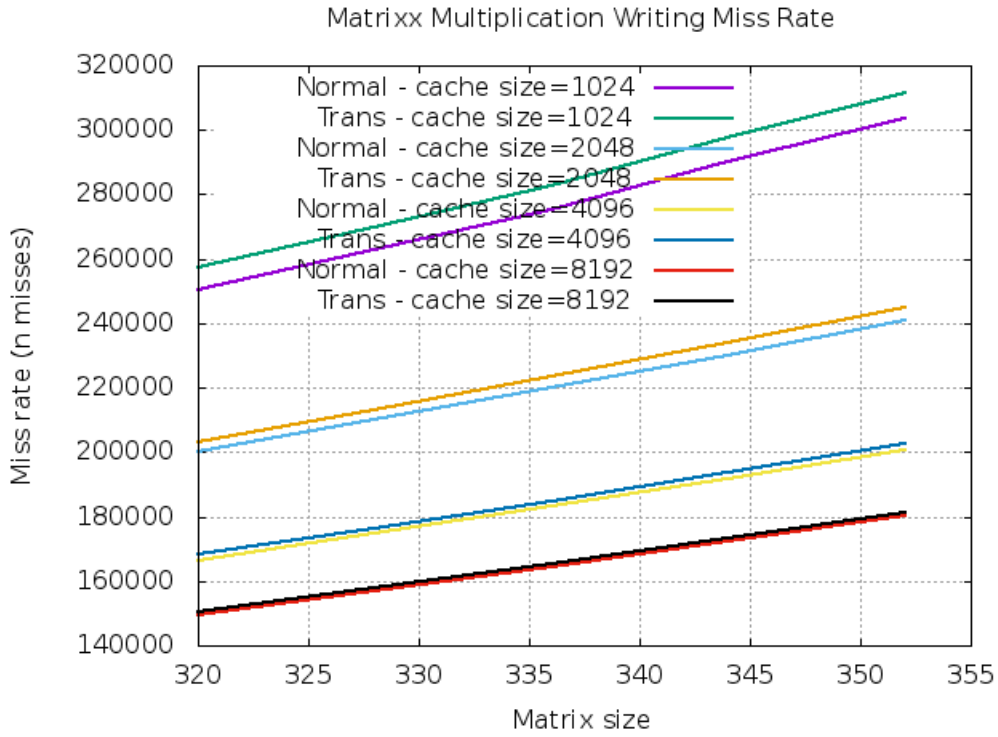
En el enunciado no se especificaba que parámetros se deseaban medir/analizar, pero se recomendaba fijarse en los tiempos de ejecución y en los fallos de caché. Así pues, hemos realizado un script llamado **ejercicio4.sh** que para cada tamaño caché en la siguiente lista [1024, 2048, 4096 y 8192] calcula el tiempo de ejecución y los fallos de caché, guardando todos estos datos en un fichero de texto (un fichero para cada tamaño de caché). Un programa escrito en python llamado **cal_mean_mult_cache.py** será el encargado de calcular los valores medios y escribirlos en un fichero de salida (uno para cada tamaño de caché también) que será usado por GNUPlot para crear las gráficas.

A continuación se muestran las gráficas originadas al ejecutar dicho script.

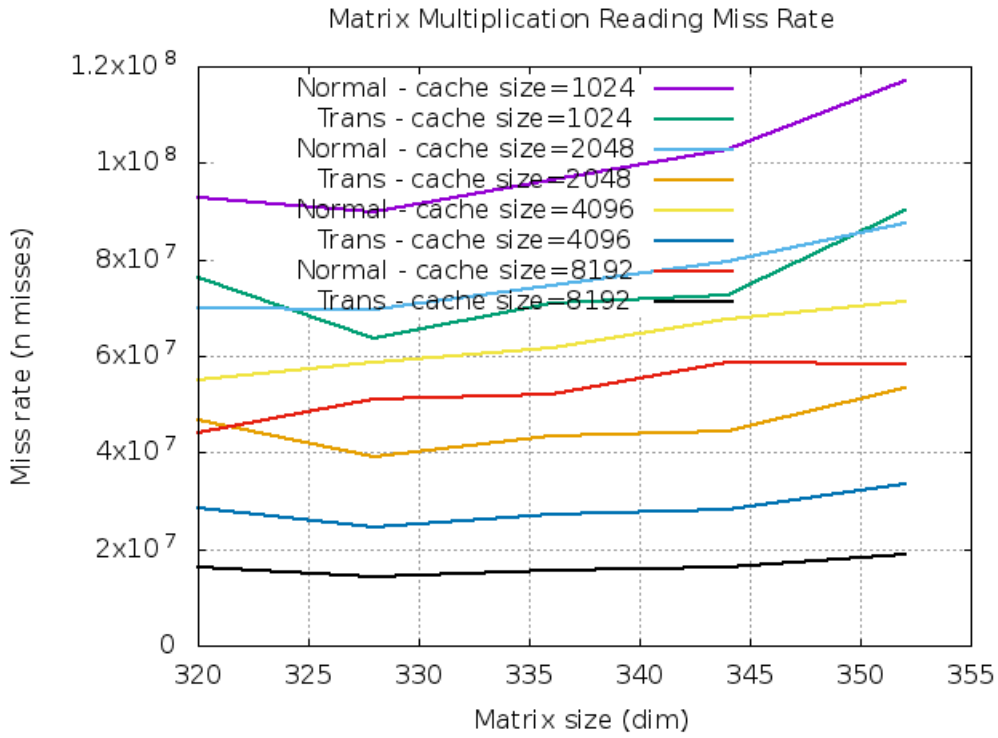


Esta primera gráfica la puede encontrar en el material entregado como **mult_cache_tiempos_ex4.png**. Recoge los tiempos de ejecución en función del tamaño de matriz, cada una de las líneas será una ejecución de los programas **matrix_mult.c** y **transMatrix_mult.c** con diferentes tamaños caché.

Como podemos ver, a grandes tamaños de matriz se estabiliza la situación y los tamaños de caché más grandes tienen un tiempo de ejecución menor. Adicionalmente el programa que realiza la trasposición es notoriamente más eficiente, que como ya hemos explicado, se debe a un mejor aprovechamiento del uso de la memoria caché.



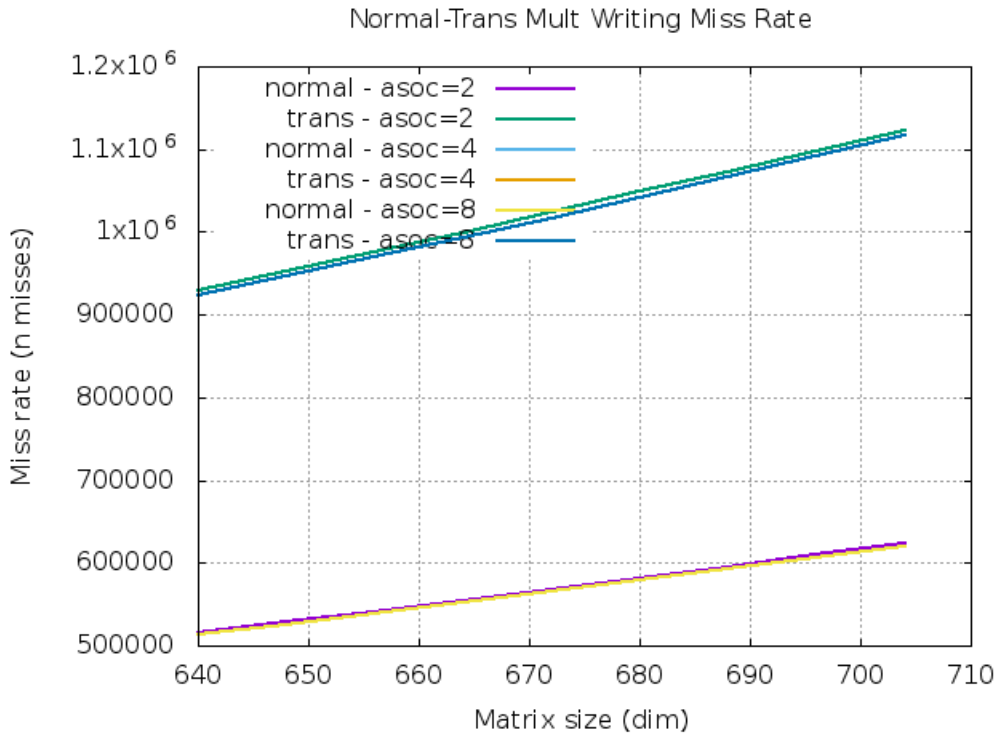
En esta segunda gráfica (en el material como **mult_cache_escritura_ex4.png**) podemos ver la suave gráfica de los fallos de caché en escritura. Como cabía esperar, se producen más fallos cuanto menor es el tamaño de caché. Por otro lado, el programa que realiza la trasposición previa a la multiplicación tiene ligeramente un número mayor de fallos. Ya se ha comentado a qué se puede deber esto en el ejercicio 3.

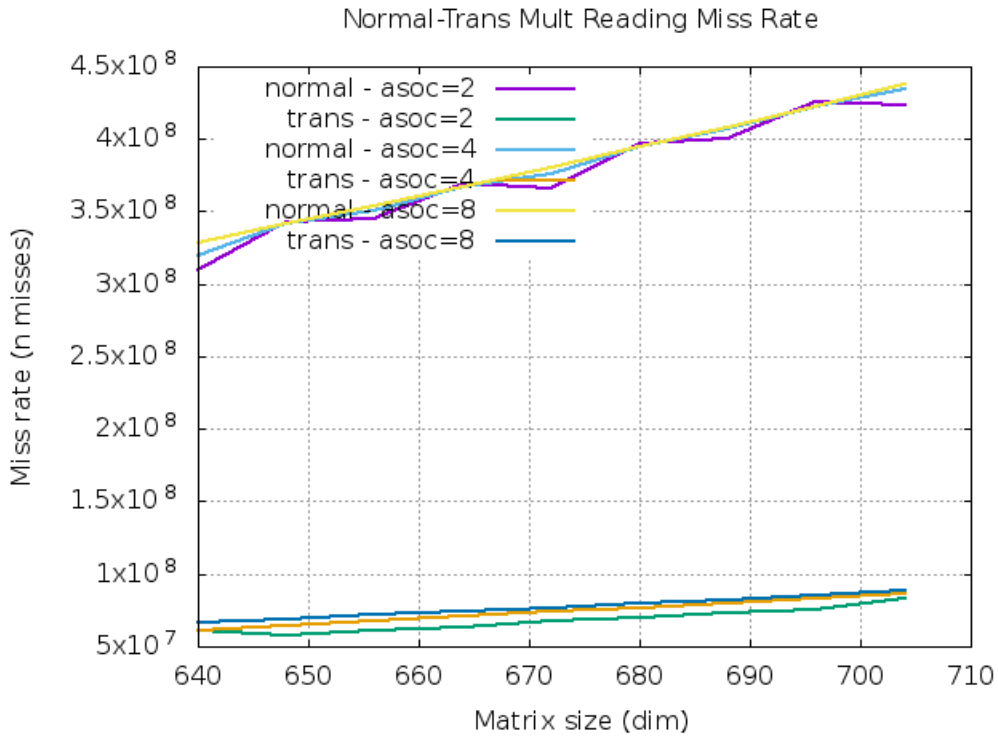


Por último, la tercera imagen (**mult_cache_lectura_ex4.png**), se analizan los fallos caché de lectura. Se obtiene el resultado esperado: cuanto mayor es el tamaño caché, menor es el número de fallos y, adicionalmente, el programa que traspone previamente la segunda matriz es significativamente más eficiente en cuanto a un menor número de fallos de caché. Como se ha dicho ya numerables veces, esto se debe a que en el programa **transMatrix_mult.c** las matrices se acceden por filas, que es la forma en que están disposicionados los datos en memoria.

Como trabajo extra, se ha programado un pequeño script **ejercicio4_2.c** que "juega" con la asociatividad de las cachés.

En este caso no se ha modificado el tamaño caché (se ha fijado a 8192), pero sí que se ha cambiado el número de vías de la memoria caché con el emulador *Cachegrind*. El script sigue la misma metodología que los anteriores, y la asociatividad analizada es para un número de vías igual a 2, 4 y 6.
Se puede ver un ejemplo de las gráficas a continuación:





En la primera imagen (**mult_cacheNvias_escritura_ex4.png**) se recogen los fallos de caché en escritura para las diferentes asociatividades analizadas. Podemos ver que la variación principal la marca el tipo de programa usado, más que el número de vías de la memoria caché.

Ligeramente, cuanto mayor es el número de vías, menor es el número de fallos de caché en escritura.

En la segunda imagen (**mult_cacheNvias_lectura_ex4.png**) pasa más de los mismo. La asociatividad no produce claras diferencias en el número de fallos de caché, sino que la diferencia de tendencias viene determinado por el programa ejecutado.

A pesar de que en cuanto a fallos de caché la asociatividad no juega un papel de gran importancia, puede que sí se note en los tiempos de ejecución, pero eso lo dejamos para futuros experimentos.

7 Observaciones importantes

· Todas las gráficas que ha visto se encuentran en el directorio `files/exercise`X donde X marca el número de ejercicio.

· Todas las gráficas expuestas han sido obtenidas a partir de unos datos numéricos guardados en ficheros de texto .dat, los cuales los podemos encontrar en el directorio correspondiente a la gráfica que generan.

· Si se ejecuta cualquier script mencionado anteriormente, las gráficas y ficheros de datos del ejercicio en cuestión serán borrados por el script, por lo que si desea comprobar/ejecutar cualquier script se le recomienda copiar el directorio `files` previamente (aunque siempre queda descargar la carpeta de la entrega de nuevo en Moodle).

· El código C se encuentra en el directorio `csource`. Deberá compilarlo previamente ejecutando **make** antes de ejecutar cualquier script.

· El código Python, por si desea revisarlo, se encuentra en el directorio `pythonCode`.

8 Conclusión

Llegado al final de la práctica podemos sostener que un correcto uso y conocimiento de las memorias caché pueden potenciar la eficiencia de un programa significativamente. Si esto pasa simplemente con la forma de percorrer una matriz, no queremos ni imaginar toda la utilizadad que puede tener esta lección en futuros trabajos.

Esto, junto con la programación en paralelo usando *threads* de la siguiente práctica, son herramientas muy útiles para los programadores que busquen exprimir el rendimiento del ordenador (en especial del microprocesador y memorias caché) al máximo.