MEMORIA
PRÁCTICA 3 ARQO

Memoria caché y rendimiento

Jesús Daniel Franco López

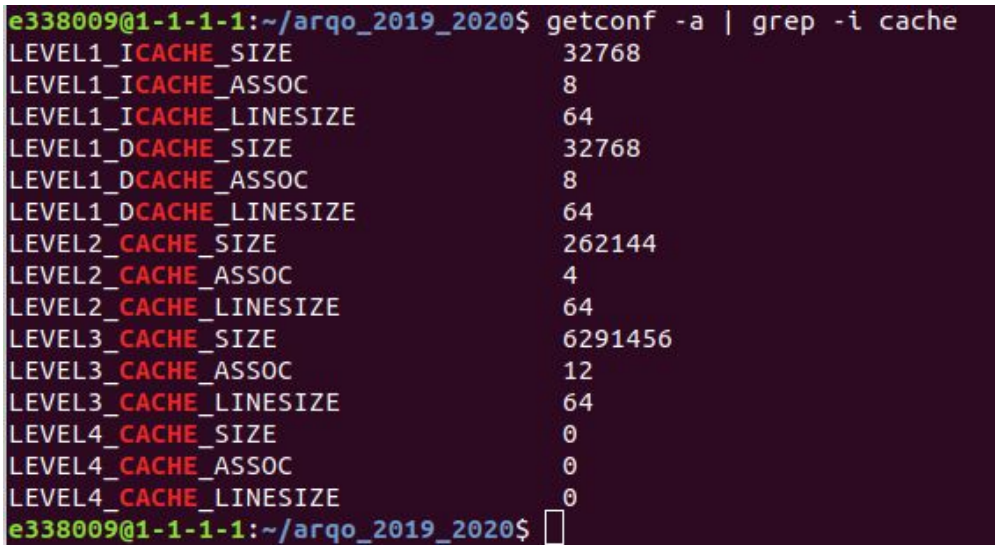
INDICE

EJERCICIO 0

Información sobre la caché del sistema

Vamos a obtener un análisis de la memoria caché, de los niveles que la componen y de las utilidades de cada uno de ellos mediante el comando

> getconf -a | grep -i cache



Este comando lo he ejecutado en un ordenador i5 de los laboratorios de la EPS. Podemos observar en la imagen que la memoria caché se compone de 4 niveles, de cuáles sólamente 3 son útiles. Comentaremos cada uno de los niveles por separado en profundidad.

Nivel 1:

Es el único en el que se separan la caché de datos y la caché de instrucciones. El tamaño de estas dos cachés es el mismo, 32768 bytes. Afirmamos además que las dos son asociativas con 8 vías y con tamaño de línea 64 bytes.

Este será el primer nivel de la memoria caché al que acceda la CPU en caso de requerir información de la misma. Debido a esto el nivel 1 de la caché es el que almacena la información que se busca con mayor frecuencia.

Nivel 2:

Vemos que este nivel, a diferencia del primero, no separa las cachés de datos e instrucciones. Esta memoria tiene un tamaño de 262144 bytes, es asociativa con 4 vías y tiene un tamaño de línea de 64 bytes.

Nivel 3:

El nivel 3 de la caché tiene un tamaño de 6291456. Al igual que los niveles anteriores es asociativa, esta vez con 12 vías y con un tamaño de palabra de 64 bytes.

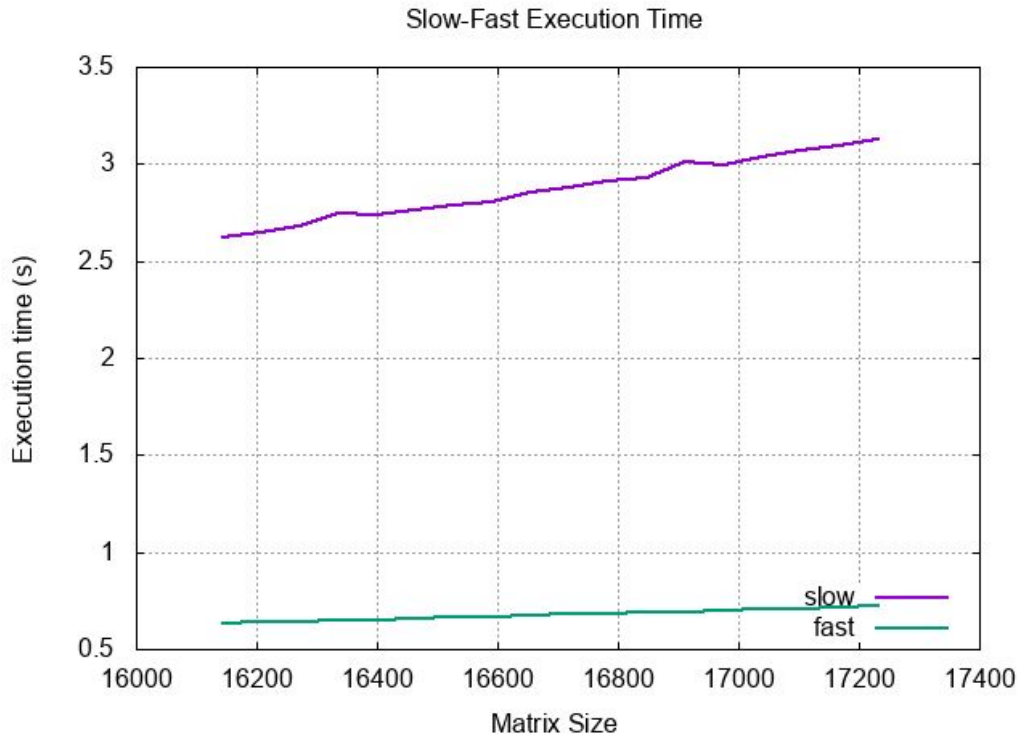
Nivel 4:

Este nivel tiene todo los valores a cero, por lo que asumimos que no está siendo utilizado y no lo contamos a la hora de describir la memoria caché.

EJERCICIO 1

Tiempo de ejecución

En la gráfica siguiente se muestra el tiempo de ejecución de los programas slow y fast para unas matrices cuyo tamaño varía de 16144 a 17168:



Los tamaños de las matrices iterarán bajo un NPaso de 64. Para resolver este ejercicio he creado un script ejercicio1.sh en el cual vamos calculando los tiempos de ejecución de los programas fast y slow para cada uno de los tamaños de las matrices sobre los que trabajamos. Para calcular estos valores, ejecutaremos ambos programas de manera alterna. Es decir, en vez de ejecutar:

slow N1, fastN1, slowN2, fast N2

ejecutaremos

slow N1, slow N2, fast N1, fast N2

El bucle para los tamaños de las matrices es entonces:

for((N = NInicio; N <= NFinal; N+= 2*NPaso))

A ejecutar los programas guardamos las salidas en variables auxiliares pasándole un grep 'time' | awk '{print $3}' para obtener el tiepo de ejecución. Esto lo vamos sumando para cada una de las repeticiones con el objetivo de hacer la media al finalizar el script. El número de repeticiones que he usado para este ejercicio ha sido 15.

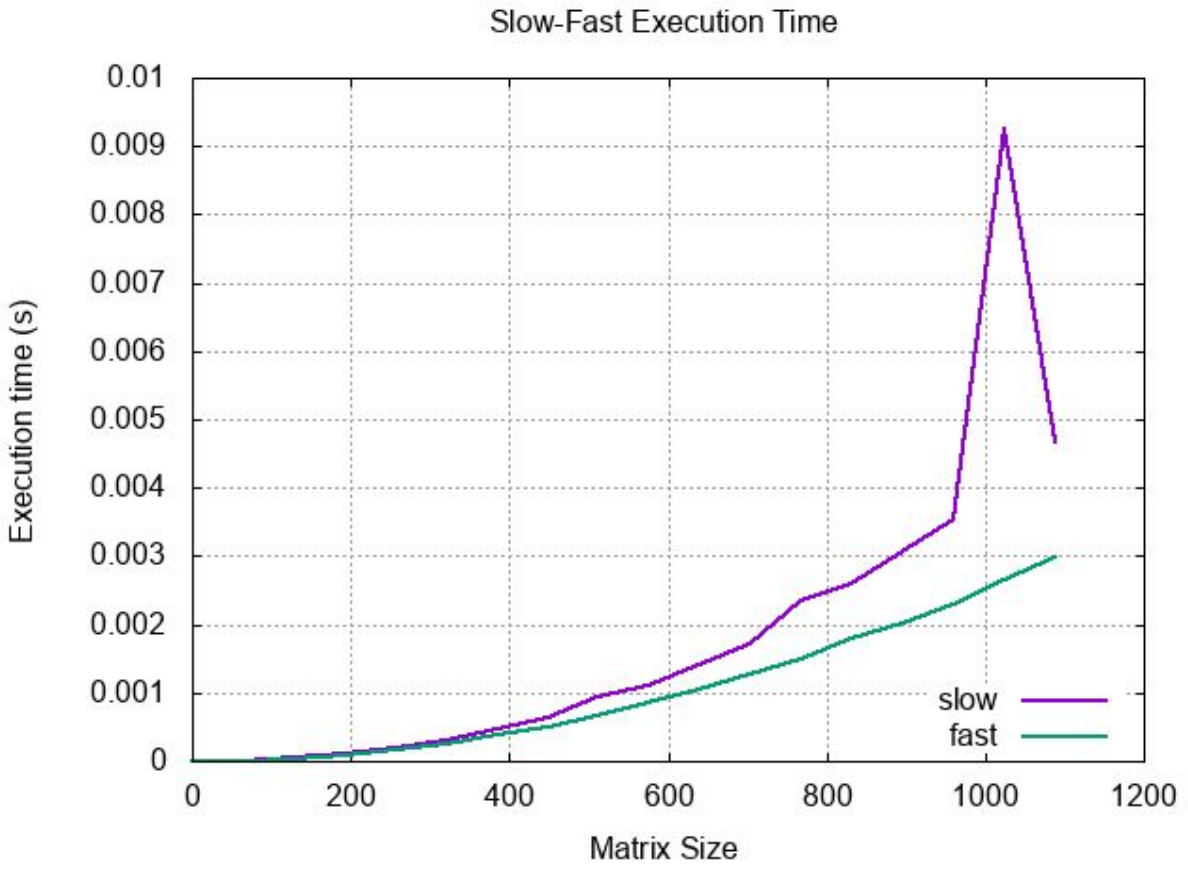
Los valores de las variables auxiliares las vamos guardamos en un fichero .dat a partir del cual generaremos la gráfica mediante la herramienta GNUPlot.

La diferencia de tiempos obtenida entre ambos programas se debe a la manera en que se suman las matrices. En el programa slow las matrices se suman por columnas y en fast, por filas. Como la matriz está guardada por filas en la memoria caché, los elementos de una fila están guardados en direcciones contiguas. Si el tamaño de la matriz es muy grande puede que las direcciones se guarden en distintos bloques.

Si accedemos a la matriz por filas, antes de saltar a un bloque distinto se lee completamente el actual. Si accedemos por columnas vamos a tener que saltar de bloque con mucha más frecuencia, lo que hace que el acceso a memoria sea más lento.

Nuestro programa empieza en matrices muy grandes, por lo que esta diferencia de tiempos se nota claramente. Si las matrices fueran muy pequeñas el acceso a memoria caché sería más parecido, menos saltos entre bloques, por lo que el tiempo de ejecución es casi igual.

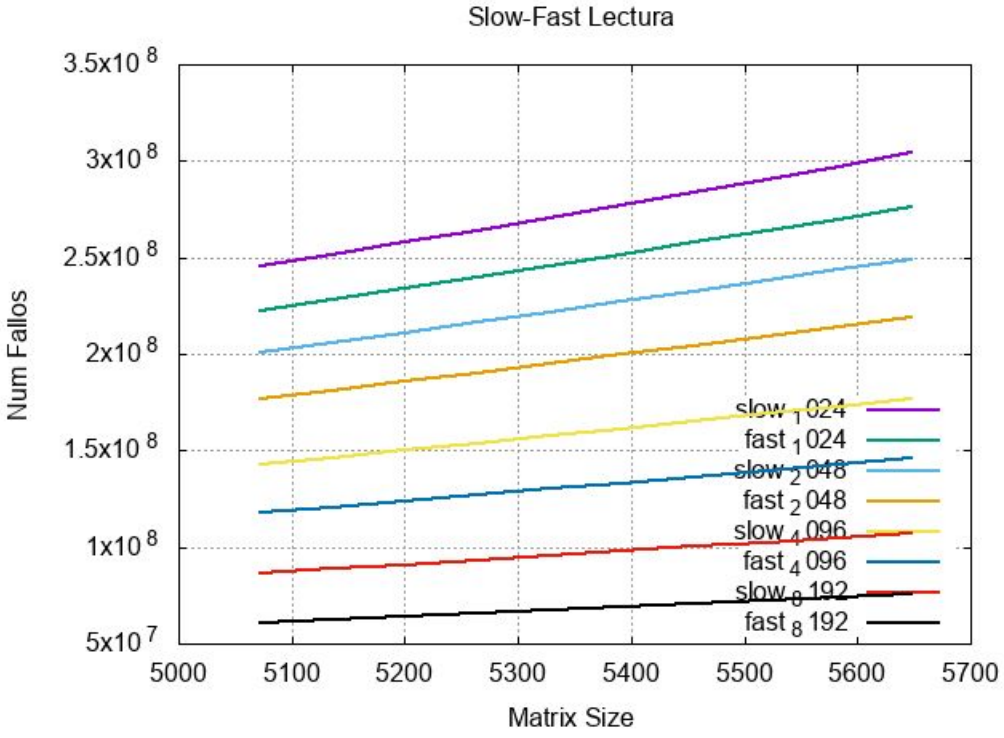
Para ver qué pasaría con matrices muy pequeñas he implementado el script ejercicio1_pequenas.sh, de donde sacamos:



EJERCICIO 2

Veremos el número de fallos en lectura y escritura para cachés de diferentes tamaños

Fallos en lectura



Los fallos de lectura se producen cuando se quiere leer un dato que no se encuentra en la caché, de forma que se lleva a cabo su búsqueda en niveles superiores y su actualización en la caché de nivel 1 con todo el bloque al que pertenece.

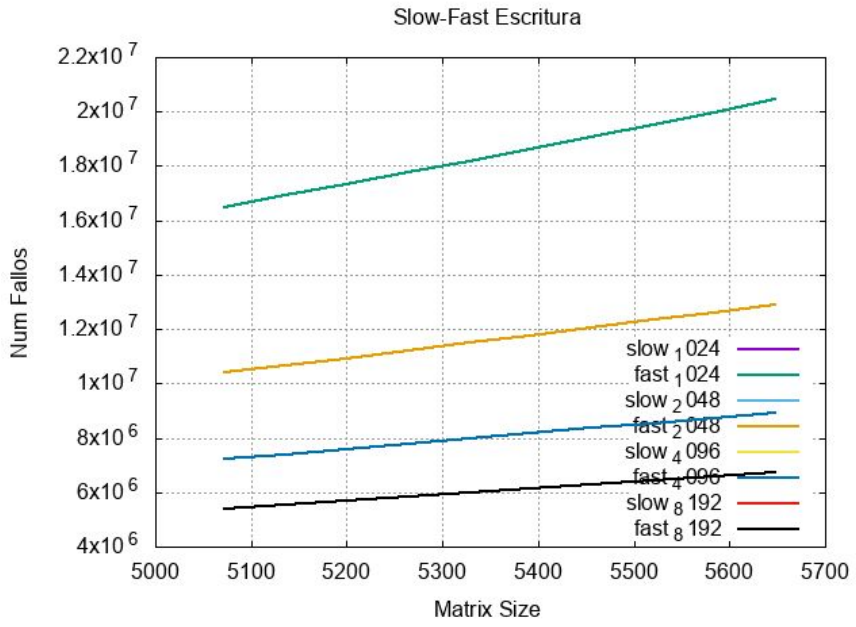
El número de fallos es mayor según aumenta el tamaño de las matrices, ya que hay un mayor número de accesos a memoria.

La diferencia de tiempos obtenida entre ambos programas se debe a la misma razón que el ejercicio anterior (a la manera de guardar y acceder a las matrices dese la caché)

Por último, se observa que claramente que el número de fallos se reduce a medida que aumenta el tamaño de la caché, ya que la matriz ocupa menos filas en caché y se producen menos saltos

Fallos en ecritura

Los dos programas escribirán en memoria un avez por cada elemento, por lo que cabría esperar que el número de fallos sea el mismo en los dos casos.



Según aumenta el tamaño de las matrices, aumentan los fallos, por las razones antes explicadas.

Si comparamos las dos gráficas obtenidas, nos damos cuenta de que el número de fallos en lectura es mayor que el número de fallos en escritura para el mismo tamaño de caché y de matriz en todos los casos. Tiene sentido, ya que en escritura sólo se accede una vez, mientras que en lectura puede ser que tengamos que acceder varias veces.

EJERCICIO 3

Funcionamiento de multNormal y multTrasp

Vamos a comprobar antes de empezar que el funcionamiento de los dos programas implementados es correcto:

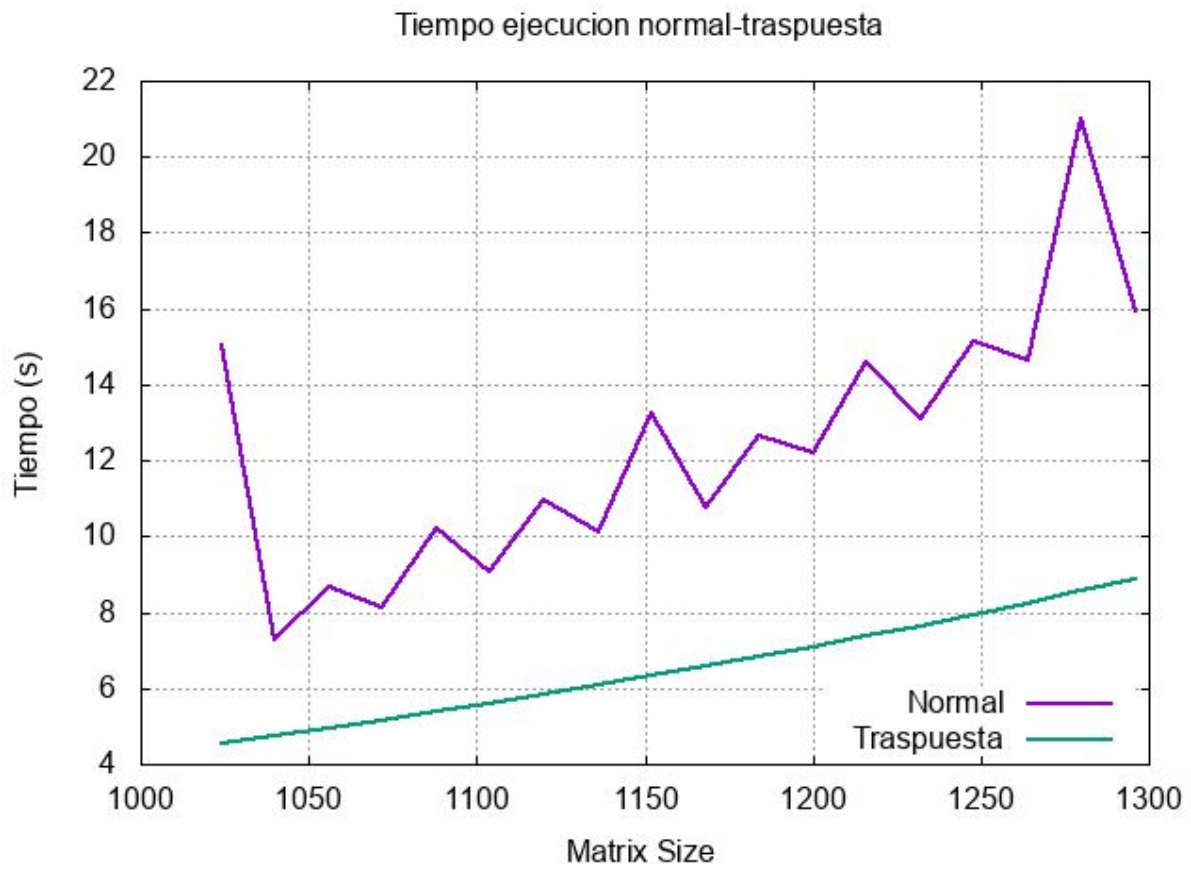
```
jesus@odin:~/Documentos/repositorios git/arqo_2019_2020/p3/ejercicio3$ ./multNormal 3
        --- MATRIZ A ---
8.401877        3.943829        7.830992
7.984400        9.116474        1.975514
3.352228        7.682296        2.777747
        --- MATRIZ B ---
8.401877        3.943829        7.830992
7.984400        9.116474        1.975514
3.352228        7.682296        2.777747
        --- MATRIZ RESULTADO ---
128.331921      129.249385      95.338640
146.495898      129.775684      86.022973
98.815172       104.595537      49.143628
Execution time: 0.000067
jesus@odin:~/Documentos/repositorios git/arqo_2019_2020/p3/ejercicio3$ ./multTraspuesta 3
        --- MATRIZ A ---
8.401877        3.943829        7.830992
7.984400        9.116474        1.975514
3.352228        7.682296        2.777747
        --- MATRIZ B ---
8.401877        3.943829        7.830992
7.984400        9.116474        1.975514
3.352228        7.682296        2.777747
        --- MATRIZ RESULTADO ---
128.331921      129.249385      95.338640
146.495898      129.775684      86.022973
98.815172       104.595537      49.143628
Execution time: 0.000065
jesus@odin:~/Documentos/repositorios git/arqo_2019_2020/p3/ejercicio3$
```

Fallos de lectura y escritura

Los fallos en escritura coinciden, por la razón que ya expliqué para los programas fast y slow. Se accede a la matriz una vez por cada elemento a escribir.

Los fallos en lectura sí que varían, pero es obvio teniendo en cuenta lo ya expuesto. El programa multNormal accede a las matrices por columnas en vez de por fias, como hace multTrasp, por lo que el acceso será más costoso. Tendremos que cambiar más tiempo entre las cachés, ya que estas almacenan las matrices por filas.

Tiempo de ejecucion



De la gráfica anterior podemos sacar varias conclusiones:

- La multiplicación traspuesta es más rápida, debido al tiempo de acceso a memoria
- El tiempo de ejecución va aumentando según lo hace el tamaño de la matriz, ya que el número de operaciones que hay que relizar aumenta.
- Observamos varios picos en la gráfica, ¿por qué? porque las matrices se generan de manera aleatoria, lo que deriva en posibles valores numéricos más difíciles de multiplicar que otros. Si pruebo el programa con más repeticiones la gráfica se mantiene igual, por lo que estoy bastante seguro de que lo anterior es cierto