Práctica 3

Arquitectura de ordenadores

Memoria caché y rendimiento

Autores:

David García Fernández
david.garcia03@estudiante.uam.es

Santiago González-Carvajal Centenera
santiago.gonzalez-carvajal@estudiante.uam.es

Pareja 33, Grupo 1301

25 de noviembre de 2017

Profesor: Gustavo Sutter

Índice

1. Introducción

El objetivo principal de esta práctica es observar la relación existente entre la memoria caché y sus diferentes configuraciones y la calidad de los distintos algoritmos con los que podemos ejecutar una tarea determinada.

En particular vamos a trabajar con matrices de distintos tamaños. En los primeros apartados tomaremos tiempos y número de fallos de caché para suma de matrices con dos maneras distintas de guardado en memoria. Para los siguientes las medidas se toman para multiplicación de matrices (cuya ejecución es considerablemente más costosa).

A la vista de los resultados determinaremos como influyen distintas configuraciones de tamaños de cache y distintas formas de tratar los datos alamacenados dependiendo de las características de la memoria que dispone el equipo.

2. Datos a tener en cuenta

2.1. Intercalado de ejecuciones

El enunciado de la práctica pide que se intercale la ejecución de dos programas para distintos tamaños de matrices. En todos los apartados realizamos n repeticiones para un rango de tamaños determinado, el propio enunciado nos indica que no realicemos las repeticiones de un mismo tamaño para un mismo programa seguidas, por lo que nosotros hemos decidido intercalar de la siguiente manera:

```
./programa1 tamaño1
./programa2 tamaño1
./programa1 tamaño2
./programa2 tamaño2
./programa1 tamaño3
./programa2 tamaño3
./programa1 tamaño1
./programa2 tamaño1
...
```

Conseguimos así que no se ejecute repetidas veces exactamente el mismo programa, asegurándonos de que en cada ejecución los datos que necesitamos en caché no sean los mismos que los que había anteriormente.

2.2. Constante P

Para la selección del rango de tamaños de matrices que utilizaremos para realizar las medidas hemos utilizado la siguiente fórmula:

$$P = (Grupo + Pareja) \mod 3$$

En nuestro caso resulta:

$$P = (1301 + 33) \mod 3 = 2$$

En el enunciado el resultado debe reducirse módulo 10, pero por indicaciones del profesor hemos reducido a módulo 3 para que poder ejecutar en un tiempo razonable la multiplicación de matrices.

3. Ejercicio 0: Caché del sistema

Hemos obtenido los siguientes resultados para el comando `getconf -a | grep cache -i`:

```
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
```

Podemos observar tres niveles de caché:

- **Nivel 1:** Es el único nivel que diferencia caché de instrucciones y de datos. Ambas tienen una capacidad de 32 KB, con un tamaño de bloque de 64 bytes y por tanto 512 bloques. Además posee 8 vías, que se reparten por igual la capacidad de la memoria. Por tanto cada vía accede a 512/8 bloques, es decir 64 bloques, todos ellos de 64 bytes.
- **Nivel 2:** En este caso se comparte memoria de datos e instrucciones, su capacidad total es de 256 KB. El tamaño de bloque es de 64 bytes, por lo que hay 2^{12} bloques. Ahora tenemos 4 vías, cada una accede a 2^{10} bloques de 64 bytes.
- **Nivel 3:** El último nivel de caché comparte de nuevo memoria para datos e instrucciones. El tamaño total es de 6291456 bytes, 64 bytes por bloque y por tanto 98304 bloques. Disponemos de 12 vías que manejan 8192 bloques de 64 bytes cada una.

Los resultados son los esperados de una jerarquía de memoria (de tres niveles) como las que hemos visto en teoría. A medida que descendemos a nivel mayores el tamaño de la memoria es mayor, pero el tiempo de acceso a cada una se resiente. En cuanto al tipo de memoria, esta sigue la estructura de una memoria asociativa de 4 vías, por los motivos que hemos explicado anteriormente.

4. Ejercicio 1: Caché y rendimiento

4.1. Objetivo

En este ejercicio debemos tomar tiempos de ejecución de suma de matrices a partir de la ejecución de dos programas, `slow` y `fast`, que suman todos los elementos el primero por columnas y el segundo por filas. El tamaño de las matrices viene determinado por el número P, en nuestro caso 2.

4.2. Gráficas

En este apartado hemos obtenido unos resultados irregulares al ejecutar el script en los ordenadores del laboratorio de la EPS, por lo que, de manera extraordinaria hemos decidido obtener medidas para distintos valores de P y para nuestros ordenadores personales, con el fin de entender qué está ocurriendo.

4.3. Ordenadores del laboratorio

Para estas medidas hemos utilizado un total de 40 repeticiones, pero tras distintos intentos hemos notado que con 20 obtenemos los mismos resultados (las repeticiones sirven para tomar la media de tiempos y descartar ralentizaciones de la ejecución por otras tareas que ejecute el sistema operativo).

4.3.1. P = 2

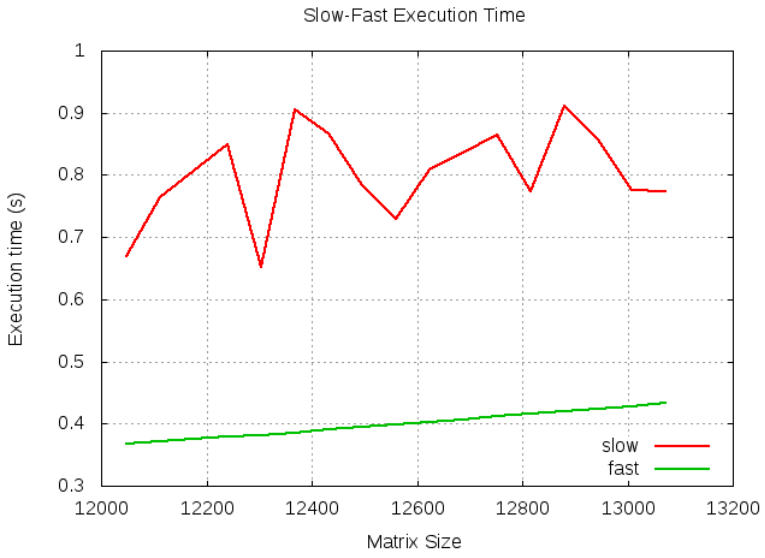

Figura 1: Tiempo de ejecución para P = 2

4.3.2. P = 1

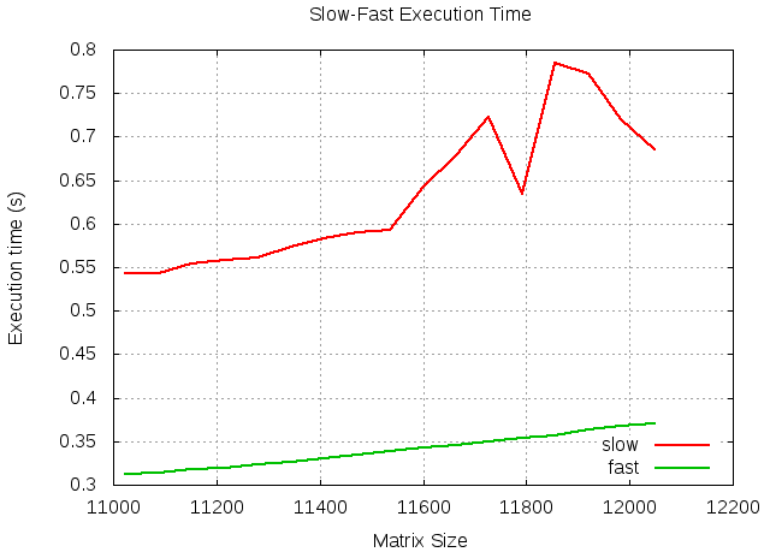

Figura 2: Tiempo de ejecución para P = 1

4.3.3. P = 0

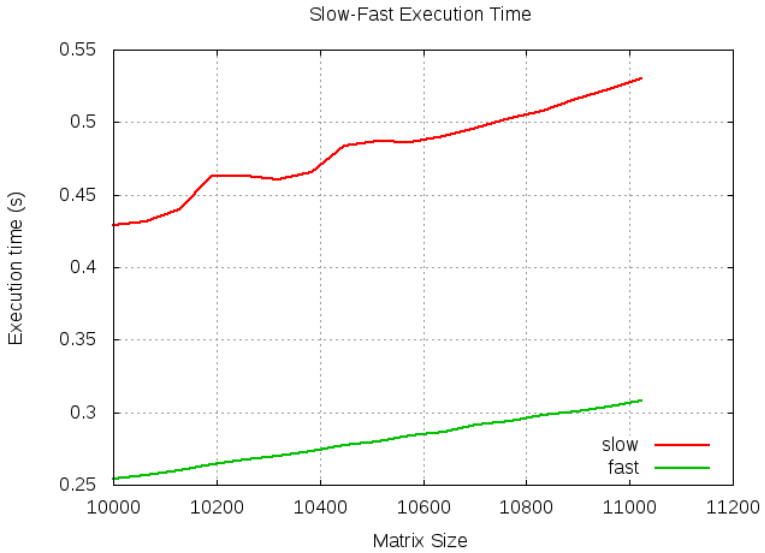

Figura 3: Tiempo de ejecución para P = 0

4.3.4. Ordenador personal

En este caso hemos tomado medidas para P = 2 con 20 repeticiones.

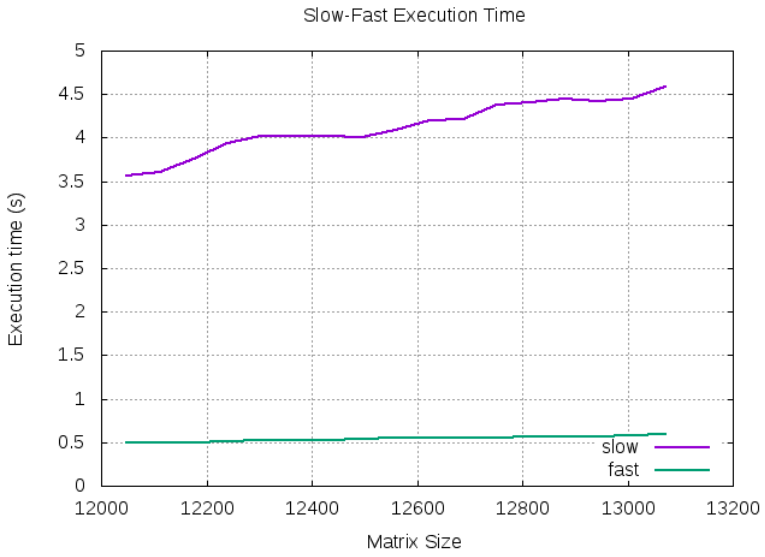

Figura 4: Tiempo de ejecución para P = 2

4.4. Método

El método seguido ha sido simple. Mediante un script bash hemos realizado varias repeticiones de los programas `fast` y `slow`. En un fichero auxiliar hemos ido grabando los resultados obtenidos para cada uno de los tamaños para después, mediante un script de awk, realizar la media para cada uno de los tamaños en función del número de repeticiones realizadas. Después hemos creado las gráficas utilizando este fichero final con la media de las repeticiones.

4.5. Análisis de resultados

A grandes rasgos los resultados son los esperados. Es de esperar que el programa `fast` que suma las entradas de la matriz por filas sea mucho más eficiente que el que lo hace por columnas, ya que cada vez que accedemos a una nueva fila esta se carga por completo en memoria. Si sumamos por columnas cada vez que accedemos a un nuevo campo de la columna su fila correspondiente se carga por completo en memoria (por tanto hay que cargar n filas para sumar una columna), sin embargo sumando por filas solo hay que cargar en memoria una vez la fila. Este hecho se hace más visible cuanto más grande es la matriz, ya que más filas se tienen que cargar.

Otro resultado esperado es que a mayor tamaño de matriz mayor es el tiempo que tardamos en ejecutar la suma. Si nos fijamos en el código observamos que tenemos un doble bucle anidado, por lo que el número de operaciones a ejecutar crece de manera cuadrática con las dimensiones de la matriz.

El hecho que nos ha resultado sorprendente son los dientes de sierra que presenta la gráfica de la *figura 1*. Los resultados que esperabamos eran los que obtenemos en nuestros ordenadores personales (*figura 4*), es decir, un crecimiento constante del tiempo de ejecución respecto del tamaño de la matriz.

Nos hemos planteado que este efecto puede deberse a que no realizabamos un número suficiente de repeticiones, pero tanto para 20 como para 40 hemos obtenido exactamente el mismo resultado, por lo que no parece ser fruto de la casualidad y hemos deshechado esta explicación.

Hemos tomado medidas para otros tamaños utilizando P= 0 y P = 1 (*figuras 2 y 3*). En ellas observamos que para tamaños más pequeños hay cada vez menos picos en las gráficas, pero

vemos que en general a pesar de los picos los tiempos crecen (asintóticamente hablando). Para ser exactos en la primera gráfica los tamaños en los que la gráfica ha decrecido y alcanza un mínimo local inesperadamente son los siguientes: 12304, 12560, 12816, 13072.

Aprovechamos para recordar que nuestra caché tiene un tamaño de bloque de 64 bytes. Para una fila completa tenenos un tamaño de $n \times 8$ bytes. En los tamaños que acabamos de citar significa un tamaño de fila respectivamente de: 98432, 100480, 102528, 104576 bytes. Estos tamaños no coinciden con el tamaño exacto del bloque (lo que podría significar que hubiese una mejora en el rendimiento del programa). A parte de este hecho también podemos notar que los picos no se dan de una ejecución para otra, es decir, que su tendencia creciente o decreciente se da durante varias repeticiones consecutivas, por lo que descartamos que se deba a esta coincidencia de tamaño de cache respecto de la fila de la matriz.

Lo único que podemos suponer es que el sistema operativo tiene una manera específica de gestionar los distintos niveles de caché para una cantidad fija de datos. Es decir, si nuestro programa trabaja con una cantidad de datos que sobrepasa un cierto umbral el sistema operativo se encarga de gestionar la memoria caché de una manera que pueda resultar más eficiente para ese orden de datos.

5. Ejercicio 2: Tamaño de caché y rendimiento

5.1. Objetivo

En este caso hemos experimentado con distintos tamaños de caché (utilizando la herramienta cachegrind) para comprobar el número de fallos de memoria obtenidos en la ejecución de ambos programas.

5.2. Gráficas

Estas gráficas se han obtenido con P = 2 y un total de 20 repeticiones, aunque en este caso con solo una repetición basta, pues la herramienta valgrind se encarga de que en la simulación se obtengan los mismos resultados siempre, por lo que es inútil realizar una media de las ejecuciones.

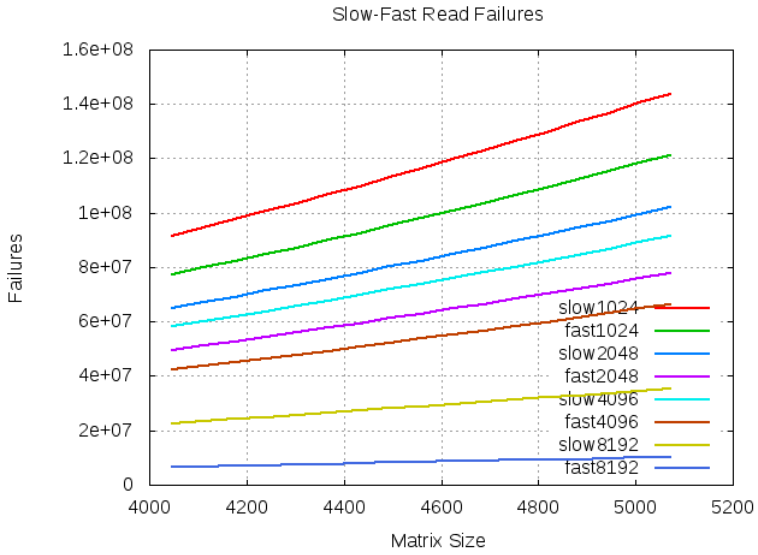

Figura 5: Fallos de memoria en lectura

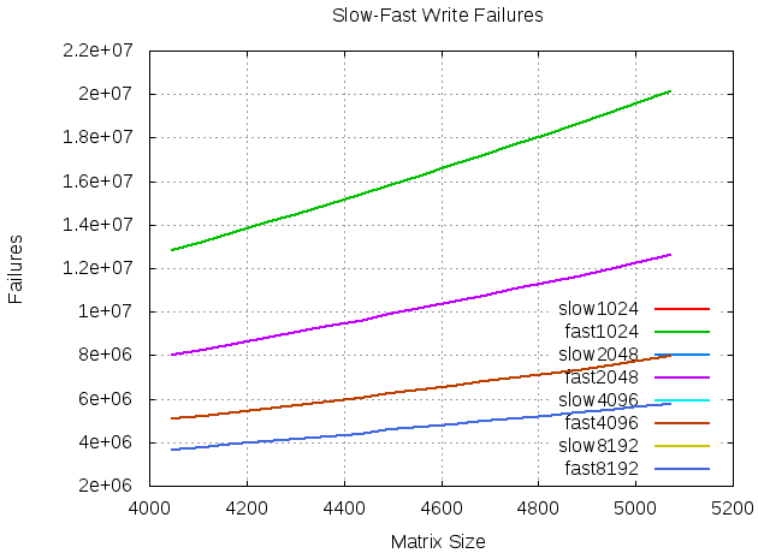

Figura 6: Fallos de memoria en escritura

5.3. Análisis de resultados

En la gráfica de fallos de lectura hemos obtenido los resultados esperados, es decir, cuanto menor es el tamaño de la caché más fallos se producen (ya que caben menos datos en caché). Y dentro del mismo tamaño, se producen más fallos en slow que en fast, ya que slow suma por columnas y fast por filas (mientras que los datos siempre se cargan a caché por filas siempre).

Mientras que en la gráfica de fallos de escritura, podemos observar que solo son visibles cuatro "líneas", esto es debido a que los fallos producidos por slow y por fast para un mismo tamaño de caché coinciden (ya que ambos programas guardan los datos igual en memoria, la única diferencia es como los cargan luego para usarlos al operar).

6. Ejercicio 3: Caché y multiplicación de matrices

6.1. Objetivo

En este ejercicio conjuntamos los tests realizados en los apartados 1 y 2 para medir la eficiencia de la multiplicación de matrices. Disponemos de un programa que las multiplica con el algoritmo usual (filas por columnas) y otro que traspone la segunda matriz y las multiplica filas por filas (con la consiguiente mejora explicada anteriormente).

Nota: En un principio el enunciado de la práctica nos indujo a error y pensamos que el número de fallos de memoria calculado por valgrind variaba para cada ejecución. De esta manera realizamos este ejercicio ejecutando con cachegrind cada una de las llamadas a los programas de multiplicación de matrices. De esta manera una sola repetición para P = 2 tardaba más de una hora, por lo que se nos hizo imposible realizar más de dos repeticiones. Sin embargo recientemente descubrimos que no hacía falta llamar a callgrind repetidas veces y decidimos realizar las medidas de tiempos utilizando el script que habíamos implementado para el apartado 1. De esta manera hemos obtenido dos gráficas distintas, una de ellas ejecutando cachegrind y la otra simplemente midiendo tiempos.

6.2. Gráficas

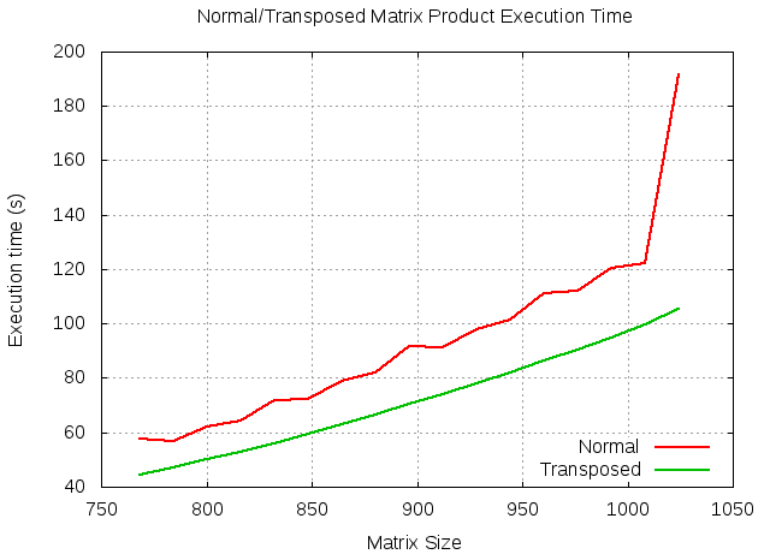

Figura 7: Tiempo de ejecución multiplicación de matrices (con cachegrind)

Figura 8: Tiempo de ejecución multiplicación de matrices (sin cachegrind)

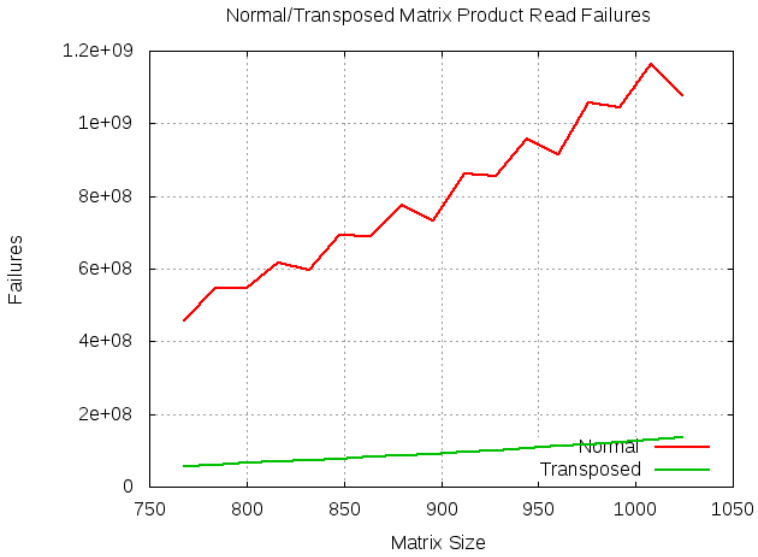

Figura 9: Fallos de memoria en lectura

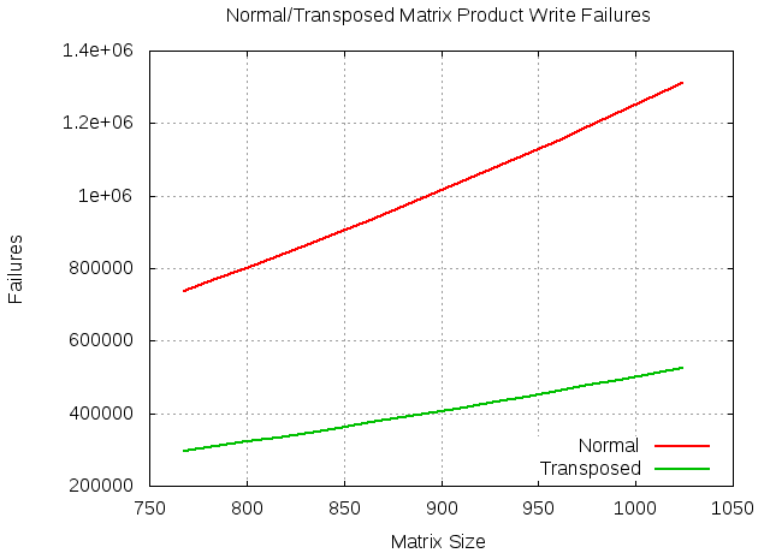

Figura 10: Fallos de memoria en escritura

Nota: Para la gráfica sin valgrind hemos realizado 10 repeticiones, para la gráfica con valgrind solamente 2.

6.3. Análisis de resultados

En este caso el coste de la multiplicación de matrices si solo nos fijamos en eficiencia del algoritmo es el mismo para ambos tipos, siendo por el mismo motivo de antes (cargar las filas en memoria) la multiplicación traspuesta más eficiente. El númeor de operaciones crece de manera cúbica con el tamaño de las matrices. Por otro lado, aunque no aparezca reflejado en las gráficas realizadas, para tamaños más pequeños de matriz (del orden de cien filas y columnas) el tiempo de ejecución de la multiplicación traspuesta es mayor que el de la multiplicación normal, ya que tenemos en cuenta también el tiempo que tardamos en trasponer la matriz.

Hay una gran diferencia entre la multiplicar utilizando cachegrind o de manera normal. Observamos que en ambas gráficas el patrón (relación tiempo/tamaño para estándar y traspuesta) es idéntico, pero al usar cachegrind el tiempo de ejecución queda mutiplicado por un factor de al menos 10, quedando una pendiente mucho más pronunciada. En ambos casos observamos que para el tamaño de matriz 1000 el coste temporal se dispara.

Como era de esperar el método de multiplicación traspuesta no presenta apenas fallos de memoria (tanto en lectura como en escritura) en comparación con la multiplicación estándar, hecho que se ve reflejado en el tiempo de ejecución. En escritura tenemos una gráfica muy suave que no nos transmite demasiada información relevante, pero en lectura observamos cómo aparecen ciertos picos. Los pasos que estamos haciendo entre tamaños son de 16, si nos fijamos en la gráfica observamos que los picos de "subida" se dan en los pasos pares, mientras que los de bajada en los impares. Por tanto si el paso fuese de 32 en 32 obtendríamos una gráfica suave como la de escritura, esto puede deberse al tamaño de 64 bytes de cada bloque de la caché.

7. Ejercicio 4: Configuración de cache en multiplicación de matrices

7.1. Objetivo

En este ejercicio vamos a experimentar con el número de fallos de caché dependiendo del tamaño de esta, para multiplicación de matrices estándar y traspuesta.

Hemos seguido el mismo procedimiento que en el ejercicio 2, cambiando los parámetros de las memorias I1 y D1, dejando la LL como en el 2. Los tamaños que hemos utilizado son: 1024, 2048, 4096 y 8192.

En este caso hemos tomado medidas para P = 0 y para P = 1, con rango de tamaños igual al del ejercicio 3 e incremento de 64.

7.2. Gráficas

7.2.1. P = 0

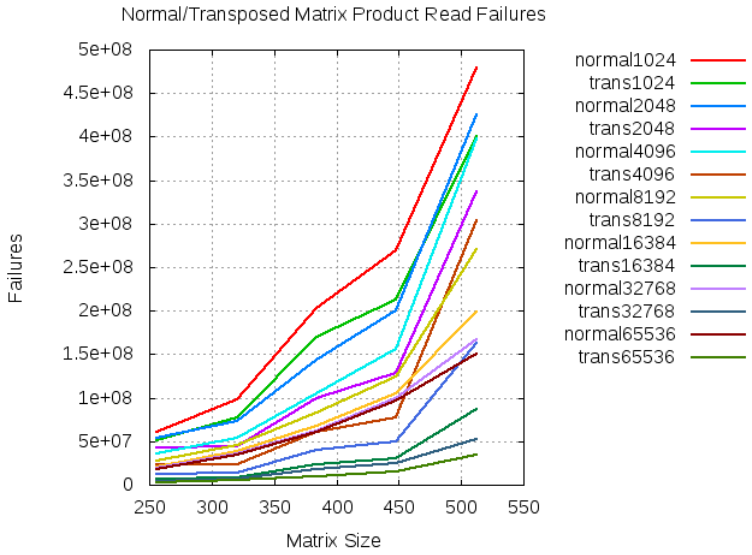

Figura 11: Fallos de memoria en lectura

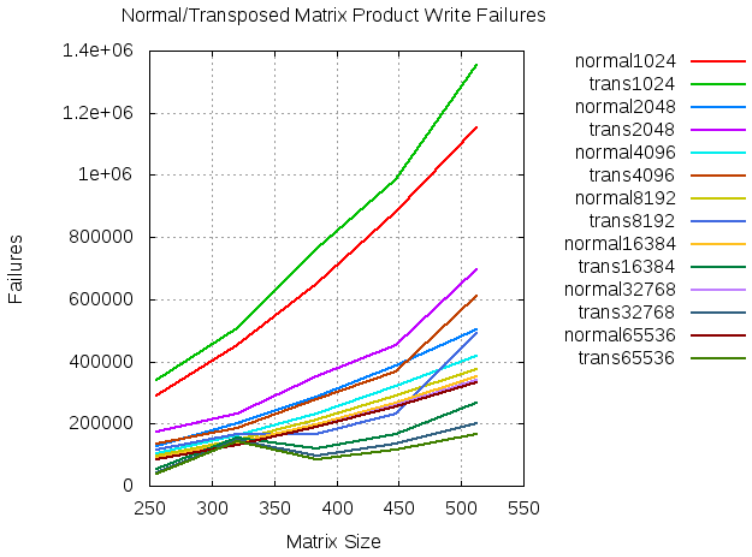

Figura 12: Fallos de memoria en escritura

7.2.2. P = 1

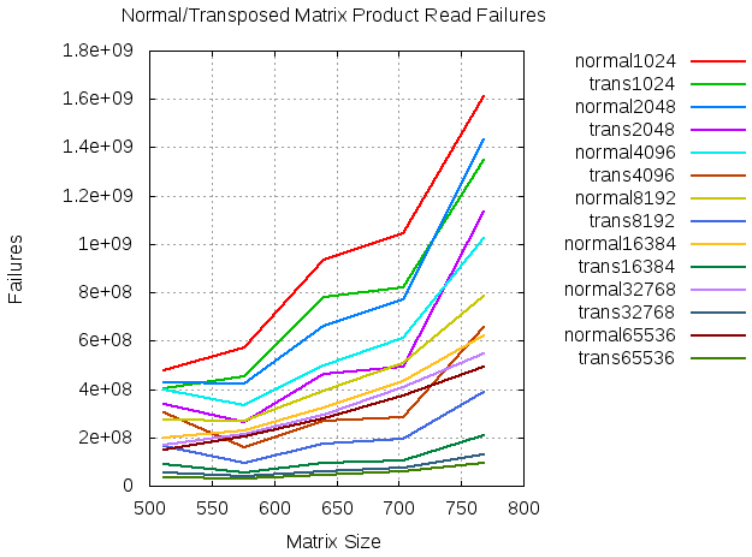

Figura 13: Fallos de memoria en lectura

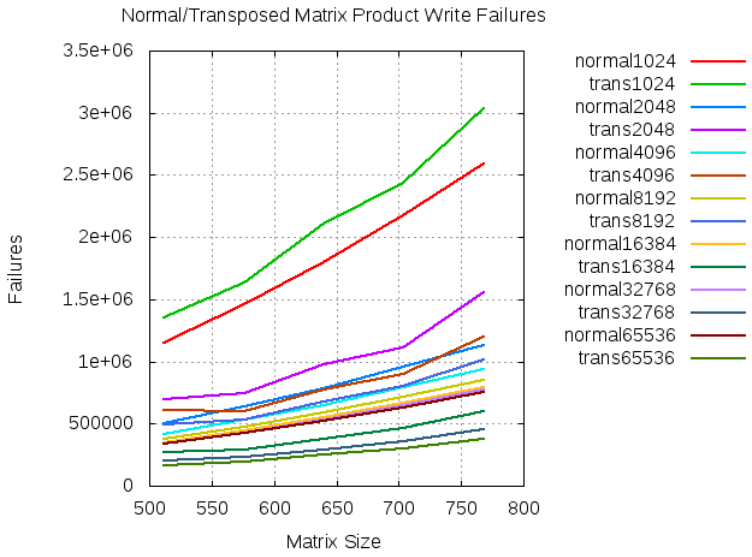

Figura 14: Fallos de memoria en escritura

7.3. Análisis de resultados

Los resultados obtenidos no son en absoluto sorprendentes, teniendo en cuenta los que habíamos obtenido en los anteriores apartados.

Para P = 0 obtenemos unos resultados muy parecidos para todos los tipos de ejecución y tamaños, sobre todo en escritura, en la que las gráficas de las multiplicaciones que utilizaron tamaño de memoria 1024 son las únicas que destacan por su gran número fallos. Para P = 1 esta diferencia es igual, pero el resto de gráficas se entrelazan menos veces. Como ya vimos en el ejercicio 2 el tipo de algoritmo utilizado no interfiere con el número de fallos en el caso de escritura, ya que lo que cambia es la forma de leer los operandos, por lo que sólo se nota en lectura.

En el caso de lectura obtenemos los resultados esperados. A mayor tamaño de caché menor número de fallos, y la multiplicación traspuesta es mucho mejor que la estándar para el mismo tamaño de caché. Lo que cabe destacar y es el resultado clave de esta práctica es que en muchos de los casos, tanto para P = 0, como para P = 1 es mejor el método de multiplicación traspuesta que el estándar con una memoria con el doble de capacidad.

8. Conclusiones

Esta práctica nos ha permitido observar la importancia de la gestión de la memoria caché para la eficiencia de la computadora. Como conlcusión general podríamos decir que tener grandes tamaños para la caché siempre mejora los tiempos de ejecución, pero no siempre es lo óptimo. Se pueden obtener mejores resultados si conocemos la estructura de la caché y programamos en función de esta, como hemos podido observar en el último ejercicio. Es decir, podemos ahorrar dinero en bytes de caché a favor de un código optimizado y obtener mejores resultados que, por ejemplo, doblando el tamaño de la memoria.