Arquitectura de Ordenadores

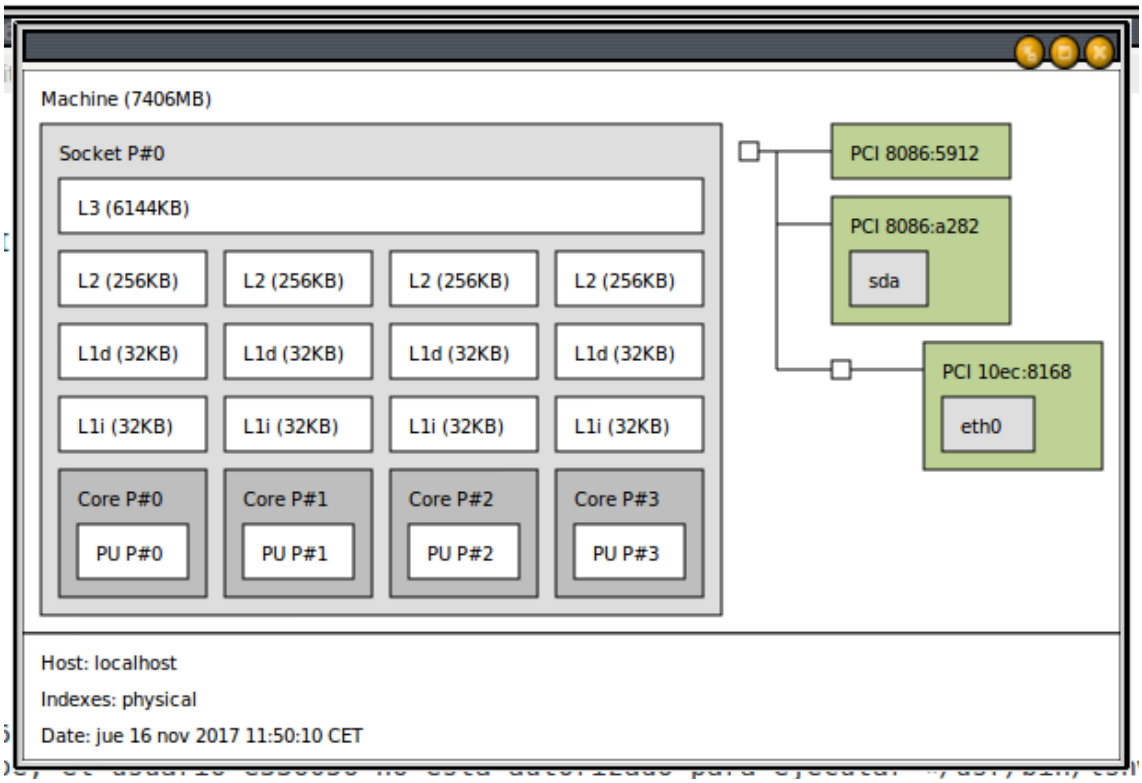
Práctica 3: Memoria caché y rendimiento

UAM 2017-2018

Óscar Gómez Borzdynski
José Ignacio Gómez García

Ejercicio 0:

Para este ejercicio necesitamos conocer la estructura de caché de nuestro sistema. Para ello utilizamos el comando *lstopo*, que proporciona una representación gráfica de la caché del sistema. Para obtener más información, como la asociatividad de cada nivel, utilizaremos *getconf -a* y buscaremos el término cache.



Vemos que nuestra máquina tiene 4 núcleos con una caché a 3 niveles:

Nivel 1: Tenemos una cache de 256 KB asociativa de 8 vías. Está dividida en 8 bloques de 32 KB, 2 por núcleo, con separación entre datos e instrucciones. Dados estos datos podemos suponer que el procesador tiene una arquitectura tipo Harvard.

Nivel 2: Tenemos una cache de 1 MB asociativa de 4 vías. Está dividida en 4 bloques de 256 KB, uno por núcleo, sin separación entre datos e instrucciones.

Nivel 3: Tenemos una cache de 6MB asociativa de 12 vías. En este caso no se encuentra dividida en bloques por núcleo, sino que es compartida entre todos ellos.

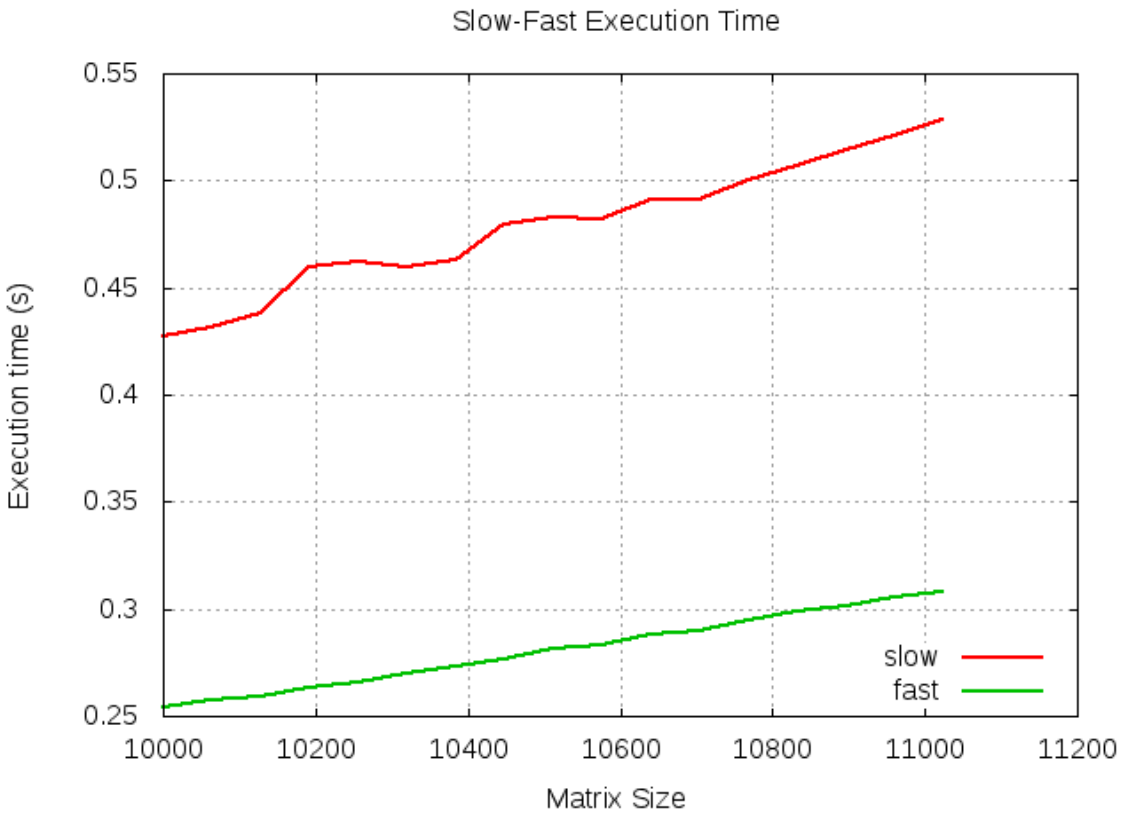
Ejercicio 1:

Nótese que a partir de este ejercicio utilizaremos P=0, en vez de P=6, por indicación del profesor.

Para este apartado utilizaremos un script que realizará un número de iteraciones determinados. Para definir este dato, se deberá modificar la variable Iterations. En nuestro caso haremos 10 iteraciones. Esto es para suavizar los posibles picos de nuestra gráfica, que pueden deberse a procesos que nuestro sistema ejecute mientras tiene lugar la simulación.

Además de ello utilizaremos matrices de 10000 a 11024 en saltos de 64. Escribiremos los tiempos de ejecución de los programas slow y fast en un fichero temporal. Obtendremos sus medias y luego los ordenaremos para proceder a realizar la gráfica solicitada.

La manera de obtener los datos de la manera más veraz es intercalar los programas. Para ello, ejecutaremos siempre un slow seguido de un fast para cada tamaño de matriz.



En la gráfica se puede apreciar una diferencia notable en el tiempo de ejecución entre slow y fast. Debido a que el programa hace básicamente lo mismo, creemos que esta diferencia se debe a los fallos de caché. La diferencia en el orden de acceso a los datos es lo que provocará esta diferencia.

Entrando más en detalle podemos observar que la diferencia de tiempo para matrices de tamaño 10000 es 0.173s, mientras que para matrices de tamaño 11024 de 0.219s. Pensamos que según aumentamos los tamaños de las matrices, el número de datos accedidos aumenta.

Por ello, debido a que el programa slow tiene una forma de acceso menos eficiente, la diferencia en los fallos de caché entre ambas aumenta.

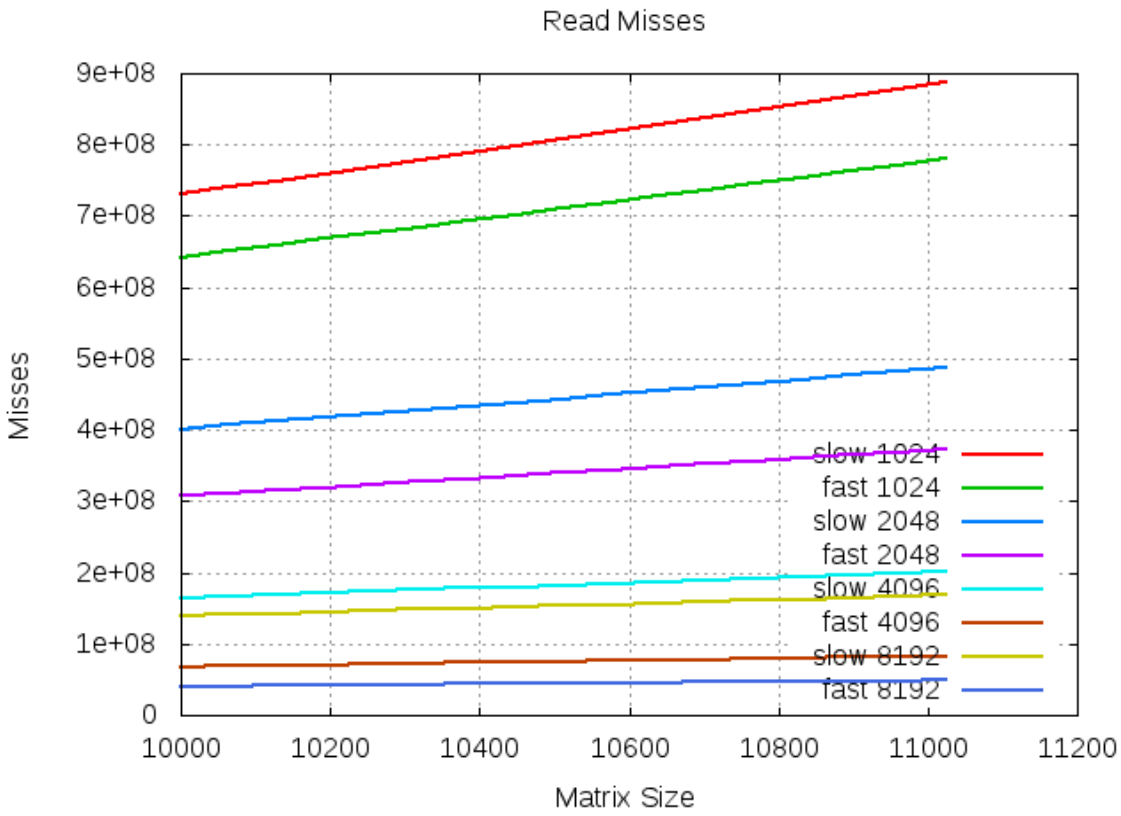
Para comprobar por qué sucede esto, procedemos a analizar el código proporcionado. Observamos que la rutina que crea la matriz mediante un array de dimensión “n x n”. Posteriormente reasigna las direcciones a un array de n punteros, dividiendo dicho array en segmentos de n elementos. Por ello, vemos que los datos se almacenan por filas. Siendo todos los datos de una fila contiguos en memoria.

Vemos que el programa fast suma los elementos por filas, siendo estos contiguos en memoria. Por ello, al cargar el primer dato, se carga el bloque entero, disponiendo del resto de datos del bloque en la cache y evitando fallos posteriores. En el caso del programa slow, accede por columnas a los datos, provocando fallos en casi todos los accesos a memoria.

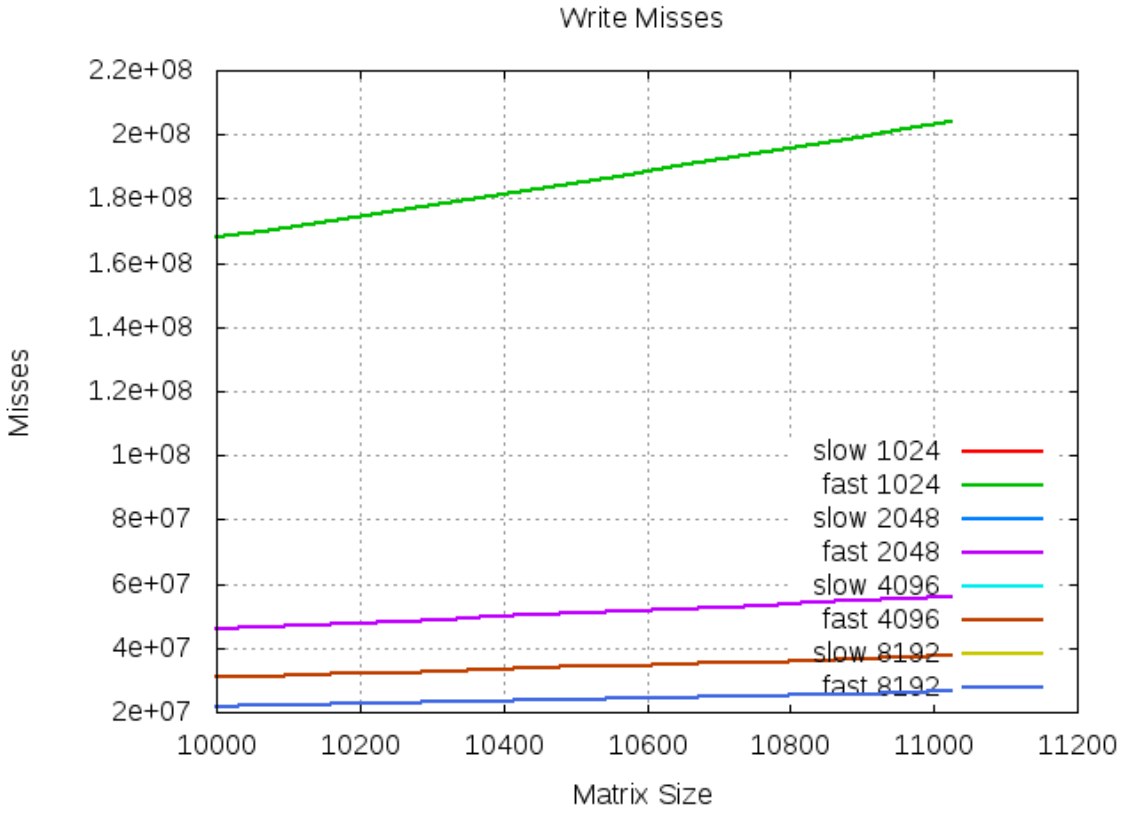
Ejercicio 2:

En este apartado vamos a utilizar el framework *Valgrind* para analizar los fallos de caché tanto en el programa *slow* como en el *fast.* Para que los tamaños de matrices coincidiesen con los del ejercicio 1 y así poder comparar resultados, elegimos matrices de tamaño entre 10000 y 11024. También fuimos variando los tamaños de caché para analizar la repercusión de este cambio en nuestros programas. Empezamos con un tamaño de caché de 1024 Bytes y terminamos con 8192 Bytes.

Por indicación del enunciado, usamos la herramienta *cachegrind*, simulando memorias de una única vía y con palabras de 64 bits. Los resultados fueron los siguientes:



Como se puede apreciar, al aumentar el tamaño de la caché disminuye el número de fallos de ejecución, siendo la simulación con más fallos la de 1024 Bytes con el programa *slow*, y la más eficiente la de 8192 Bytes, con el programa *fast.* De hecho, se puede apreciar una gran diferencia entre los programas *fast* y *slow*, ya que la ejecución ***slow en 8192 Bytes*** tiene incluso más fallos que la ***fast en 4096 Bytes.*** Por ello, no apreciamos cambios de tendencia al variar tamaños de caché, aunque sí se aprecia un aumento de la pendiente al pasar del programa *fast* al *slow*.

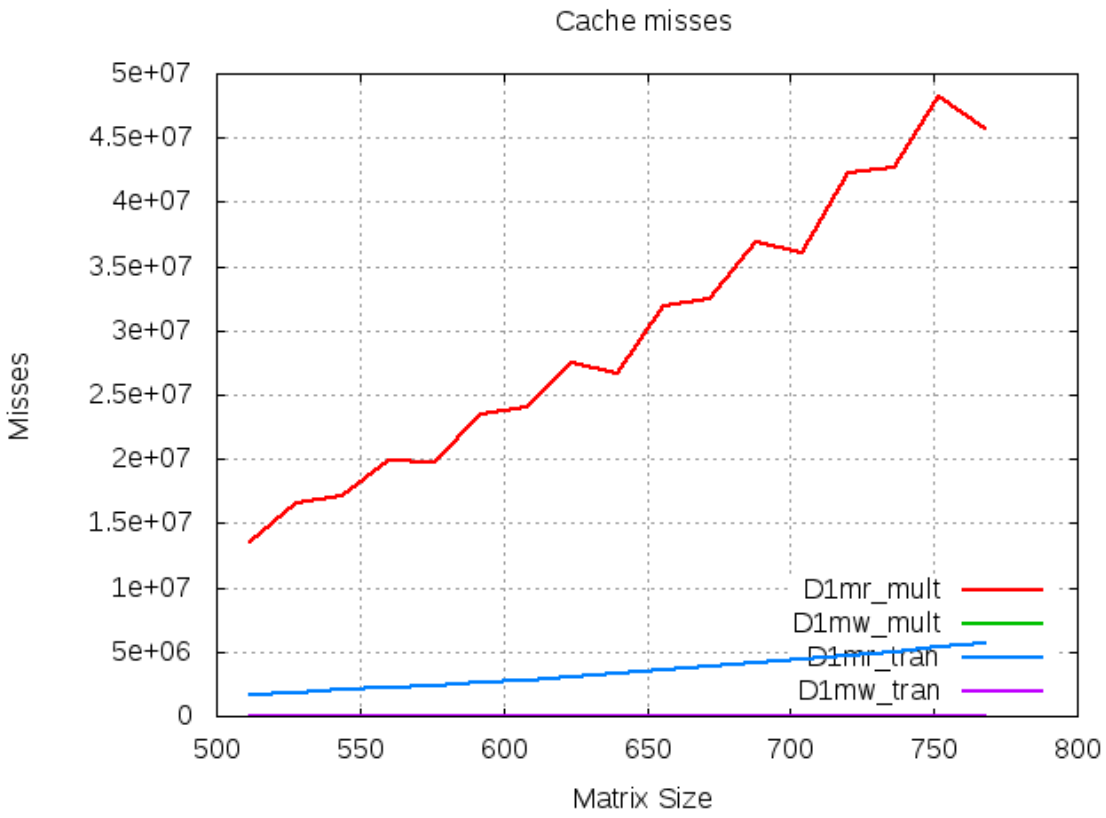


En este caso, el número de fallos de escritura es el mismo en *fast* y *slow*. Esto se debe a que la escritura incluye la creación de la matriz (que es la misma en los dos programas) y la actualización de la variable *sum*, que también es igual en ambos casos. Como ya hemos dicho, la única diferencia entre los dos es la forma en que acceden a los campos de la matriz como lectura, por lo que no se reflejan diferencias en la escritura. Cabe destacar que, tal y como se espera, al aumentar el tamaño de caché disminuye el número de fallos e, igual que antes, se aprecia un aumento de la pendiente en los tamaños menores.

Ejercicio 3:

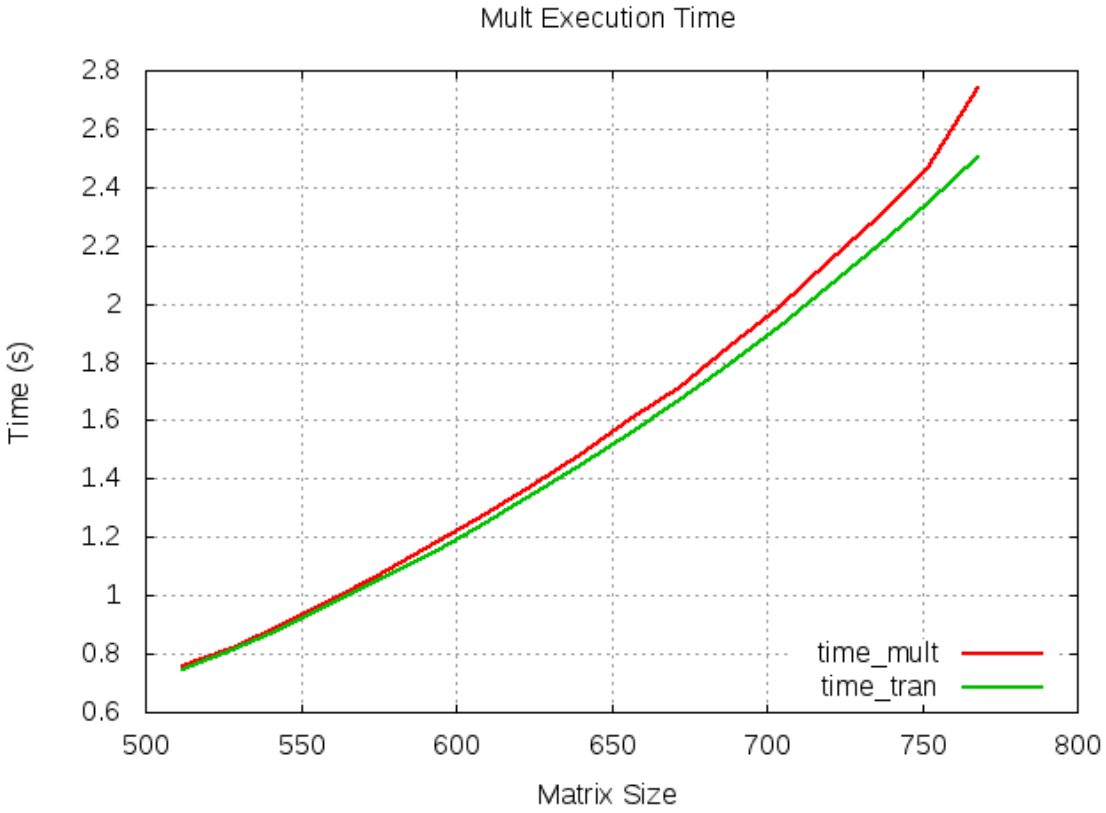
En este apartado analizaremos la diferencia entre dos programas: uno para multiplicar matrices, y otro similar que trabaja con traspuestas. A priori, nos damos cuenta de que va a haber diferencias en la eficiencia de cada programa, debido a la forma de acceder a las matrices.

Para este ejercicio hemos usado matrices de tamaño 512-768 y, de nuevo, hemos utilizado la herramienta *cachegrind* para analizar los fallos de caché. En este caso, la memoria simulada es la que viene por defecto, que se corresponde con la de nuestro ordenador. Hemos creado tres scripts: *multiply_error.sh* (con el que obtenemos los errores de caché), *multiply_time.sh* (con el que obtenemos los tiempos de ejecución tras varias iteraciones, para tomar medias) y *merge_graphics.sh* (con el que juntamos los resultados para hacer las gráficas).



En esta primera gráfica se reflejan los fallos de caché de los programas de multiplicación. Como se esperaba, el programa *multiply* genera un mayor número de fallos, ya que accede a la matriz de forma similar al programa *slow* de los primeros apartados. Por otro lado, el programa *transposed* aprovecha que los datos de la matriz se almacenan por filas de modo que, al utilizar la matriz traspuesta, accede de una vez a todos los datos de una misma fila, por lo que es mucho más eficiente que el primero.

En nuestra gráfica no se aprecian los fallos de escritura, ya que son mucho menores que los de lectura.

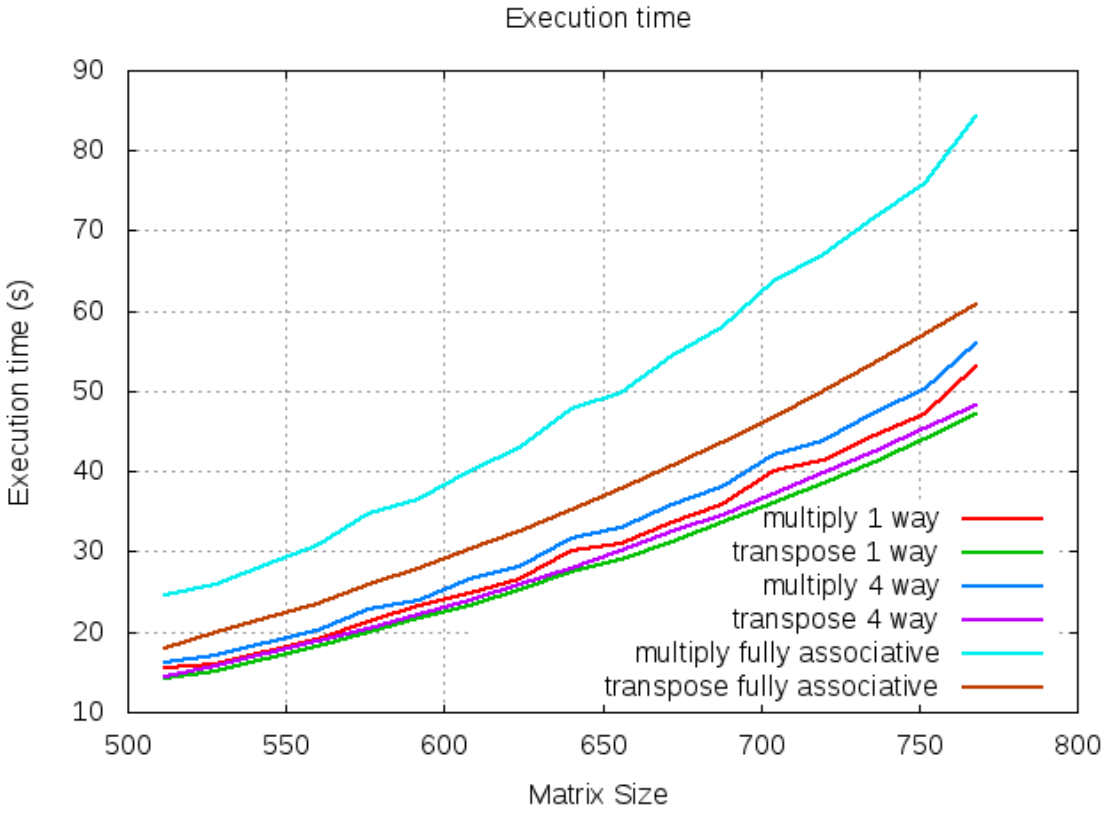


Como hemos tomado tamaños no muy grandes de matrices, la diferencia entre los tiempos de ejecución es poco representativa. Sin embargo, vemos que el programa *multiply* es más lento que el *transposed*, pese a que este último también incluye el tiempo de la transposición. Como además los tiempos aumentan de forma exponencial, conforme tomemos matrices de mayor tamaño obtendremos una diferencia de tiempos mucho más significativa.

Ejercicio 4:

En este último ejercicio hemos probado a cambiar la asociatividad de las memorias para obtener información complementaria acerca de su funcionamiento. Como ya hemos variado los tamaños en ejercicios anteriores, nos vamos a basar en esos resultados en lugar de repetir el proceso.

Ya habíamos llegado a las conclusiones de que, al aumentar el tamaño de las cachés, disminuía el número de fallos del programa. Si nos fijamos ahora en la asociatividad, podemos apreciar en la gráfica que, para tamaños de matriz de entre 512 y 768, la de una vía provoca más fallos que la de 4 vías o la completamente asociativa. En este caso, apenas se aprecia diferencia entre la de 4 vías y la completamente asociativa. Como Valgrind no admite la posibilidad de simular una memoria completamente asociativa, hemos “forzado” esta posibilidad simulando tantas vías como entradas hay en la memoria. Sin embargo, a la luz de los resultados, no podemos conocer cómo se comporta Valgrind ante esta situación. Puede que haya simulado una memoria de 12 vías, o de 8 (por defecto, al no admitir tantas vías). Lo que sí es destacable es que este aumento de asociatividad no aporta gran beneficio, ya que llega un punto en el que nos va a dar igual tener más vías, porque no vamos a terminar de llenar las que ya tenemos.



En el caso de la gráfica de tiempos, podemos ver que a mayor número de vías aumentan los tiempos de ejecución. Esto se debe a que Valgrind realiza las comparaciones por software, por lo que, al aumentar el número de comparaciones a realizar, va a aumentar el tiempo considerablemente.