Práctico 1

Nicolás Alejandro Núñez Hosco
Facundo Diaz
Grupo 17
Profesores: Martín Pedemonte, Pablo Ezzatti y Ernesto Dufrechou

GPU

Abril 2024

Índice

1. Introducción

Este informe aborda un análisis experimental y teórico sobre el impacto del uso optimizado de la memoria caché en el rendimiento. En otras palabras, se examina cómo la optimización de la memoria caché afecta el desempeño general.

En la primera parte, se detallan las especificaciones de la jerarquía de memoria de los procesadores utilizados. Se realizan mediciones de los tiempos de acceso promedio en cada nivel de caché. Además, se analizan los efectos técnicas de prefetching, tanto a nivel de hardware como de software, para esto se diseñan programas con el objetivo de que la única diferencia de rendimiento entre ellos provenga de la realización de prefetch.

La segunda parte se centra en la evaluación de diferentes estrategias de reordenamiento de bucles en la multiplicación de matrices. El objetivo es optimizar el rendimiento en términos de la eficacia de la caché. Se realizan cálculos de hit rates para distintos reordenamientos de bucles, junto con mediciones de tiempo de ejecución y Gflops para evaluar el rendimiento de cada estrategia de manera experimental.

Finalmente, se estudia el impacto de la aplicación de la técnica de blocking para mejorar el rendimiento en la multiplicación de matrices. Se realizan pruebas con diferentes tamaños de bloques y matriz y se comparan los tiempos de ejecución con y sin esta técnica.

2. Ejercicio 1

2.1. Parte A

Para este trabajo se utilizan dos computadoras con procesadores modelo i7-7700HQ y i7-9750H.

Ambos procesadores tienen 3 niveles de cache, el primer nivel (L1) se divide en L1I para instrucciones, y L1D para datos. La memoria L2 es exclusiva y L3 inclusiva. Cada núcleo del CPU tiene una cache L1 y L2, mientras que el nivel L3 es compartido entre todos los núcleos. El tamaño del nivel L1 es de 32KB, el de L2 es de 256KB, y el de L3 de 6MB (12 MB en i7 9750h). Los 3 niveles utilizan correspondencia por conjuntos. El nivel L1 utiliza conjuntos de 8 lineas, L2 de 4 lineas, y L3 de 12 lineas en el procesador i7-7700HQ y 16 lineas en i7-9750H, en ambos casos el largo de línea es 64bytes. Por último, todos los niveles de ambos procesadores utilizan la política de escritura write-back, lo que implica que, en caso de escribir sobre un byte que se encuentra en la cache, solo se traslada esta información a memoria principal una vez se elimine de la caché.

Para calcular la velocidad máxima de acceso en cada nivel de cache, creamos un programa que sigue la siguiente estructura:

Se utilizan 3 arrays:

- `data` de 8KB el cual contiene datos que moveremos entre los diferentes niveles de cache y sobre el cual se medirá el tiempo de lectura.
- `rellenoL1` de 32KB, que usaremos con el objetivo de llenar la caché L1 de forma que data pase a estar almacenado en L2.
- `rellenoL2` que de forma análoga a rellenoL1 se utiliza para llenar la caché L2. Los 3 arrays contienen datos de tipo `char`.

El programa ejecuta un bucle de 16384 iteraciones. En primer lugar se realizan lecturas sobre la totalidad del arreglo `data`, con el objetivo de que sus datos sean almacenados en L1, posteriormente se lee sobre una entrada pseudoaleatoria dentro del array, y se registra la cantidad de ciclos de CPU requerida para esto. A continuación se accede a todos los datos de `rellenoL1`, de esta forma se trasladan los contenidos de `data` a L2 y, nuevamente, se registra la cantidad de ciclos de reloj requerida para acceder a una entrada arbitraria del array. Finalmente, y de forma análoga a los casos anteriores, se realiza una lectura completa del arreglo `RellenoL2` con el objetivo de determinar tiempo de acceso a una entrada de `data`, que en este punto se encuentra almacenada en L3.

La suma parcial de la cantidad de ciclos necesarios se almacena en las variables `ciclosL1`, `ciclosL2` y `ciclosL3`, y al finalizar las iteraciones, se dividen estos valores entre 16.384 para obtener el promedio de ciclos necesarios para cada nivel. Es necesario considerar que accedemos a variables aleatorias en data para evitar que el prefetch mueva los datos a los que accedemos al nivel L1.

Se obtienen los siguientes resultados:

- Ciclos promedio por lectura en L1: 17 ciclos.

- Ciclos promedio por lectura en L2: 22 ciclos.
- Ciclos promedio por lectura en L3: 37 ciclos.

En base a la información presentada se deduce:

- Tiempo promedio por lectura en L1: 4.046 ns
- Tiempo promedio por lectura en L2: 5.236 ns
- Tiempo promedio por lectura en L3: 8.806 ns

Estos valores son del orden esperado. En el caso de la memoria caché L1, observamos que el tiempo promedio por lectura, con un valor de 4.046 ns, supera lo que normalmente se considera estándar para el acceso a la caché L1. Es posible que esta discrepancia se deba a latencias adicionales introducidas por la metodología de medición utilizada.

2.2. Parte B

La técnica de caché prefetching puede ser implementada a nivel de hardware, algunas CPUs tienen mecanismos de hardware dedicados a reconocer patrones en las peticiones de datos o instrucciones realizadas por un programa, permitiendo el prefetch de estos datos, o a nivel de software mediante la inclusión de instrucciones de prefetch ya sea durante la compilación o escritura del programa. El análisis presentado se divide en 3 casos: La CPU hace prefetch sobre secciones de memoria que son efectivamente utilizadas, el programa realiza lecturas sobre secciones de memoria no fácilmente predecibles, por lo que no se aprovechan las capacidades de prefetch de la CPU y, finalmente, se utilizan mediante instrucciones de prefetch explicitas con el objetivo de almacenar en L1 los datos que el programa leerá en un futuro próximo.

Prefetch por hardware

Para este caso, se define un programa que accede a un array (de tipo `char`) de tamaño arbitrario

En primer lugar se recorre este arreglo dividiéndolo en bloques de 64bytes (tamaño de línea de cache). El programa itera de forma lineal realizando lecturas sobre las entradas del arreglo contenidas en los bloques en posiciones pares (entradas 0 a 63, 128 a 191, etc.) y posteriormente aquellas en bloques impares (entradas 64 a 127, 192 a 255, etc). Luego se recorre al mismo arreglo de forma puramente lineal.

Dado que cada bloque contiene 64 bytes de información, se espera que el primer acceso a cada bloque cause que este sea trasladado a la caché, por lo que las siguientes 63 lecturas serán hits. En el caso de la recorrida lineal, sin embargo, la CPU puede anticipar y cargar en caché bloques adicionales mediante prefetching, lo que aumenta la proporción de hits y mejora el rendimiento en comparación con el acceso alternado entre bloques pares e impares.

Para este experimento, se emplea un array de 4GB con el propósito de ampliar la escala de las medidas de tiempo y obtener resultados más significativos. Los resultados obtenidos fueron los siguientes:

- Duración de acceso no lineal: 8.900ms
- Duración de acceso lineal: 4.300ms

Lo cual implica una ganancia de aproximadamente 50 % en tiempo, se observa por lo tanto observar que el aprovechamiento del prefetch por hardware logra bajo estas condiciones una mejora significativa en el rendimiento del acceso a la memoria caché.

Prefetch por software

Para medir los efectos del prefetch por hardware, se define una matriz cuyas filas se dividen en bloques de 64 bytes. La misma es recorrida siguiendo un patrón no lineal, donde se seleccionan pares de bloques de datos alternando entre filas y columnas de la matriz (ver Figura 1), una vez seleccionados ambos son recorridos linealmente. Notar que al recorrer de esta forma se minimizan los efectos del prefetch por hardware.

Este procedimiento se repite dos veces, en una de ellas se utilizan instrucciones de prefetch explícitas (particularmente `_mm_prefetch`) con el fin de reducir el tiempo del acceso. Por ejemplo, antes de acceder a los bloques 3 y 4, se indica al CPU que adelante la carga de los bloques 5 y 6. Esto posibilita anticipar los datos necesarios y potencialmente reducir los tiempos de espera del procesador.

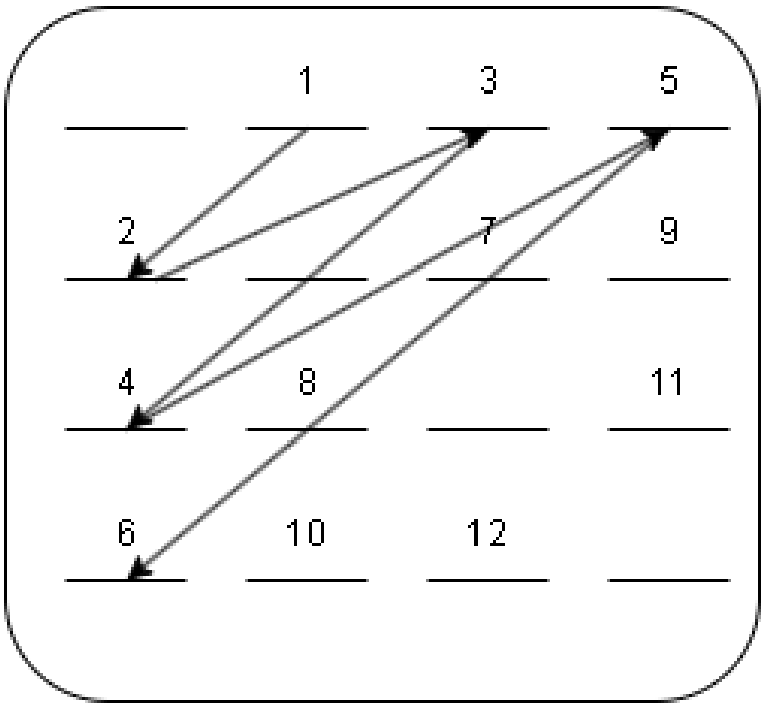

Figura 1: Ejemplo del patrón de acceso

Usando una matriz de 64MB, obtenemos los siguientes resultados:

- Duración de acceso sin indicar prefetch: 9.100ms
- Duración de acceso indicando prefetch: 6.800ms

Se observa que el uso de prefetch por software genera una mejora de aproximadamente 25 % en este contexto.

2.3. Parte C

Para mostrar el efecto del acceso desalineado a memoria, se diseña un programa donde se definen dos estructuras: `estrcutura_desalineada` y `estructura_alineada`, ambas de 48 bytes, compuestas de 12 floats. En la definición de `estructura_alineada` se utiliza el especificador `alignas(64)` que provoca que cada instancia de `estructura_alineada` se almacene en memoria desde una dirección divisible entre 64, (notar que este es el tamaño de línea utilizado por las caches de los procesadores empleados) de esta manera se logra que instancias de esta estructura se encuentren almacenadas en un único bloque de memoria principal y, como consecuencia, que en el caso de ser guardadas en caché esto se haga de forma alineada. Esto logra un comportamiento equivalente al que se obtiene con el uso de `aligned_malloc()` pero en este caso para cualquier instancia del tipo definido.

Es importante notar que, tanto la definición de los tipos, como de las funciones para recorrerlos es exactamente la misma, con la única diferencia del operador `alignas`.

Una vez definidas, se instancia un array de cada tipo con tamaños arbitrarios (en este caso, 128, lo que se corresponde a 6KB), y se determina el tiempo requerido para recorrer linealmente ambos arrays 10.000.000 de veces, leyendo cada uno de sus datos.

Al ejecutar este programa utilizando el procesador i7-7700HQ, a 2800MHz se obtiene:

- Duración al usar una estructura desalineada: 16.905ms
- Duración al usar una estructura alineada: 12.412ms

Por lo que vemos una ganancia de aproximadamente un 25 %.

Si agrandamos los valores del array a 2.048 elementos, lo que se corresponde a 96KB, obtenemos los siguientes resultados:

- Duración al usar una estructura desalineada: 263.200ms
- Duración al usar una estructura alineada: 193.500ms

Nuevamente, vemos una ganancia de aproximadamente un 25 %.

3. Ejercicio 2

3.1. Parte A

Para el análisis realizan las siguientes suposiciones:

- Solo los accesos a A, B y C generan cambios en las caches (ignoramos escrituras en SUM o variables de iteradores).
- Los tipos de variables a los que accedemos son floats de 4 bytes, por lo que, dentro de cada linea de 64 bytes de la cache, entran 16 elementos.
- N, P y M tienen valores lo suficientemente grandes como para que, si al finalizar de recorrer una fila o columna de principio a fin, al regresar al inicio, estos valores ya no se encuentran en cache.

Reordenamiento row-col-it

En este reordenamiento, se fija una fila de A, y se recorre esta fila junto con cada columna de B.

Acceso a $A_{0,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $A_{0,0}$ hasta $A_{0,15}$
Acceso a $B_{0,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $B_{0,0}$ hasta $B_{0,15}$
Acceso a $A_{0,1} \rightarrow$ Hit
Acceso a $B_{1,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $B_{1,0}$ hasta $B_{1,15}$
Acceso a $A_{0,2} \rightarrow$ Hit
Acceso a $B_{2,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $B_{2,0}$ hasta $B_{2,15}$
.
.
.
Acceso a $A_{0,15} \rightarrow$ Hit
Acceso a $B_{15,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $B_{15,0}$ hasta $B_{15,15}$

Luego de esto, se vuelve a repetir los mismos accesos, con un miss en A, seguido de 15 hits, y 16 misses en B. Podemos calcular que hay 15 hits cada 32 accesos, por lo que obtenemos un hit rate de 15/32.

Reordenamiento col-row-it

En este reordenamiento, se fija una columna de B, y se recorre cada fila de A junto con esta columna, antes de pasar a la siguiente. Considerando lo supuesto, una vez que se recorre la columna fijada de B, y se vuelve a comenzar, no habrá hit (por el tamaño de p). Podemos concluir que nuevamente tendremos un hit-rate de 15/32

Reordenamiento row-it-col

En este reordenamiento, se fija un ELEMENTO de A, y se recorre cada fila de B antes de pasar al siguiente elemento de A (en la misma fila), y a la siguiente fila en B

Acceso a $A_{0,0}$ → Miss → Se trae a la cache desde $A_{0,0}$ hasta $A_{0,15}$
Acceso a $B_{0,0}$ → Miss → Se trae a la cache desde $B_{0,0}$ hasta $B_{0,15}$
Acceso a $C_{0,0}$ → Miss → Se trae a la cache desde $C_{0,0}$ hasta $C_{0,15}$
Acceso a $A_{0,0}$ → Hit
Acceso a $B_{0,1}$ → Hit
Acceso a $C_{0,1}$ → Hit
Acceso a $A_{0,0}$ → Hit
Acceso a $B_{0,2}$ → Hit
Acceso a $C_{0,2}$ → Hit
.
.
.
Acceso a $A_{0,0}$ → Hit
Acceso a $B_{0,15}$ → Hit
Acceso a $C_{0,15}$ → Hit

Luego de esto, se vuelve a repetir los mismos accesos, un miss en cada matriz, seguido de 15 hits en cada una de ellas, podemos concluir que tendremos un hit rate de 45/48.

Reordenamiento col-it-row

En este reordenamiento, se fija un elemento de B, y se recorre una columna de A, multiplicando sus elementos por el elemento fijado de B. Una vez terminada la columna, se fija como elemento el siguiente en la misma columna que el anterior.

Acceso a $A_{0,0}$ → Miss → Se trae a la cache desde $A_{0,0}$ hasta $A_{0,63}$
Acceso a $B_{0,0}$ → Miss → Se trae a la cache desde $B_{0,0}$ hasta $B_{0,63}$
Acceso (lectura y escritura) a $C_{0,0}$ → Miss → Se trae a la cache desde $C_{0,0}$ hasta $C_{0,63}$
Acceso a $A_{1,0}$ → Miss → Se trae a la cache desde $A_{1,0}$ hasta $A_{1,63}$
Acceso a $B_{0,0}$ → Hit
Acceso (lectura y escritura) a $C_{1,0}$ → Miss → Se trae a la cache desde $C_{1,0}$ hasta $C_{1,63}$
Acceso a $A_{2,0}$ → Miss → Se trae a la cache desde $A_{1,0}$ hasta $A_{2,63}$
Acceso a $B_{0,0}$ → Hit
Acceso (lectura y escritura) a $C_{2,0}$ → Miss → Se trae a la cache desde $C_{2,0}$ hasta $C_{2,63}$
.
.
.
Acceso a $A_{15,0}$ → Miss → Se trae a la cache desde $A_{1,0}$ hasta $A_{15,63}$
Acceso a $B_{0,0}$ → Hit
Acceso (lectura y escritura) a $C_{15,0}$ → Miss → Se trae a la cache desde $C_{15,0}$ hasta $C_{15,63}$

Luego, se repite el ciclo, con un miss en A y un miss en B y otro en C, seguido de 15 misses en A, 15 hits en B y 15 misses en C. Concluimos que este reordenamiento tiene un hit-rate de $14/48 \sim 1/3$

Reordenamiento it-row-col

En este reordenamiento, se fija un elemento de A, y se recorre una fila de B multiplicando sus elementos por el fijado de A. Una vez terminada la fila, se fija el siguiente elemento en la misma columna de A.

Acceso a $A_{0,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $A_{0,0}$ hasta $A_{0,15}$
Acceso a $B_{0,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $B_{0,0}$ hasta $B_{0,15}$
Acceso (lectura y escritura) a $C_{0,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $C_{0,0}$ hasta $C_{0,15}$
Acceso a $A_{0,0} \rightarrow$ Hit
Acceso a $B_{0,1} \rightarrow$ Hit
Acceso (lectura y escritura) a $C_{0,1} \rightarrow$ Hit
Acceso a $A_{0,0} \rightarrow$ Hit
Acceso a $B_{0,2} \rightarrow$ Hit
Acceso (lectura y escritura) a $C_{0,2} \rightarrow$ Hit
.
.
.
Acceso a $A_{0,0} \rightarrow$ Hit
Acceso a $B_{0,15} \rightarrow$ Hit
Acceso (lectura y escritura) a $C_{0,15} \rightarrow$ Hit

Luego de esto, se vuelve a repetir los mismos accesos, con misses en A, B y C, seguidos de 15 hits en A, B y C , podemos concluir que tendremos un hit rate de 45/48.

Reordenamiento it-col-row

En este reordenamiento, se fija un elemento de B, mientras se recorre una columna de A (y de C). Cuando se acaba la columna, se pasa al siguiente elemento de B en la misma fila. Cuando se acaban todos los elementos de B, se comienza desde el incio, y se pasa a la siguiente columna en A.

Acceso a $A_{0,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $A_{0,0}$ hasta $A_{0,15}$
Acceso a $B_{0,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $B_{0,0}$ hasta $B_{0,15}$
Acceso (lectura y escritura) a $C_{0,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $C_{0,0}$ hasta $C_{0,15}$
Acceso a $A_{1,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $A_{1,0}$ hasta $A_{1,15}$
Acceso a $B_{0,0} \rightarrow$ Hit
Acceso (lectura y escritura) a $C_{1,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $C_{1,0}$ hasta $C_{1,15}$
Acceso a $A_{2,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $A_{2,0}$ hasta $A_{2,15}$
Acceso a $B_{0,0} \rightarrow$ Hit
Acceso (lectura y escritura) a $C_{2,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $C_{2,0}$ hasta $C_{2,15}$

.
.
.
Acceso a $A_{15,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $A_{15,0}$ hasta $A_{15,15}$
Acceso a $B_{0,15} \rightarrow$ Hit
Acceso (lectura y escritura) a $C_{15,0} \rightarrow$ Miss $\rightarrow$ Se trae a la cache desde $C_{15,0}$ hasta $C_{15,15}$

Luego de esto, se vuelve a repetir los mismos accesos, con 16 misses en A, 1 miss en B, 16 misses en C, y 15 hits en B. Podemos concluir que tendremos un hit rate de aproximadamente 15/48

Se logra determinar que, las dos mejores formas de ordenar los bulces presentes en el programa en relación a hit-rate son it-row-col y row-it-col con casi un 95 %, row-col-it y col-row-it con aproximadamente el 50 %, y por último, los dos con menor hit-rate son it-col-row y col-it-row con aproximadamente 33 %.

Para calcular el rendimiento en GFlops, que se corresponde a la cantidad (en miles de millones) de operaciones sobre punto flotante hechas en un segundo, usaremos la siguiente formula:

$$\frac{CantOper}{TiempoEjecucionEnSegundos \cdot 10^9}$$

Se observa que se realizan 2 operaciones de punto flotante dentro del bucle interior, por lo que calculamos $CantOper$ como $n \cdot m \cdot p \cdot 2$

A continuación se muestra el desempeño de cada reordenamiento, en tiempo y GFLops, para $n = m = p = 1700$,

Reordenamiento	**Tiempo de ejecucion**	**GFlops**
it-row-col	24.18945s	0.4062
row-it-col	24.417934s	0.4024
row-col-it	50.021590s	0.1964
col-row-it	42.058092s	0.2336
it-col-row	86.350548s	0.1137
col-it-row	93.441202s	0.1051

Los resultados experimentales presentados se encuentran en línea con las expectativas teóricas.

3.2. Parte B

Para implementar la técnica de blocking en la multiplicación de matrices, se segmentan las matrices originales en bloques bidimensionales de $n \times n$ elementos de tipo float. Luego, se

calcula el producto de cada par de bloques y se suman los resultados de manera apropiada para obtener el resultado final.

Se espera que esta división del algoritmo en productos matriciales de menor tamaño permita que los datos necesarios para cada uno de estos sub-productos puedan ser almacenados simultáneamente en cache, aumentando el hitrate durante la ejecución. Notar que la efectividad de esta técnica depende del valor seleccionado para el tamaño de bloque en relación al tamaño de linea y de cache.

Por ejemplo al calcular AB utilizando matrices de de tipo `float`, en una CPU cuya cache utiliza lienas de 64B y estableciendo `block_size = 16` se provoca que que cada fila de cada sub-matriz o bloque ocupe exactamente una linea de cache. Por este motivo cuando el programa recorre la primera columna de la sub-matriz de B es posible que la totalidad de la misma sea almacenada en L1 de forma alineada, además la fila seleccionada de la sub-matriz A podrá también ser cacheada. Bajo estas condiciones el hit-rate esperado es alto.

$$miss\_rate = \frac{2 \cdot block\_size}{2 \cdot block\_size^3} = \frac{1}{block\_size^2} = \frac{1}{16^2}$$

A continuación, se presentan comparaciones de los tiempos de ejecución para distintos tamaños de matrices (n, m, p) y tamaños de bloque, considerando los siguientes casos: el algoritmo row-col-it con una tasa de aciertos (hit rate) de 15/32, row-it-col que demostró mejores resultados, con una tasa de aciertos de 45/48, y finalmente, el método row-col-it implementado con la técnica de blocking.

Reordenamiento	m	p	n	block-size	Tiempo de ejecución	GFlops
row-col-it	1200	1200	1200	-	12.720625s	0.2716
row-it-col	1200	1200	1200	-	8.664842s	0.3988
row-col-it (con blocking)	1200	1200	1200	20	9.731915s	0.3551
row-col-it	1400	1900	700	-	14.144568s	0.2632
row-it-col	1400	1900	700	-	9.551054s	0.3899
row-col-it (con blocking)	1400	1900	700	20	10.903806s	0.3415
row-col-it	1700	1700	1700	-	51.169086s	0.1920
row-it-col	1700	1700	1700	-	24.615008s	0.3991
row-col-it (con blocking)	1700	1700	1700	20	27.941701s	0.3516
row-col-it (con blocking)	1700	1700	1700	425	33.273059ss	0.2953
row-col-it (con blocking)	1700	1700	1700	340	32.186386s	0.3052
row-col-it (con blocking)	1700	1700	1700	170	29.389045s	0.3343
row-col-it (con blocking)	1700	1700	1700	100	29.341215s	0.3348
row-col-it (con blocking)	1700	1700	1700	50	26.153402s	0.3757
row-col-it (con blocking)	1700	1700	1700	25	26.257978s	0.3742
row-col-it (con blocking)	1700	1700	1700	20	27.941701s	0.3516
row-col-it (con blocking)	1700	1700	1700	10	26.774632s	0.3669

Se observa que la ejecucion de row-it-col sin blocking sigue obteniendo mejores resultados.

Al aplicar el método de blocking al reordenamiento original (row-col-it) observamos que esta técnica mejora de forma consistente el tiempo de ejecución del algoritmo. En particular, se observa que el rendimiento mejora al utilizar valores de `block-size` menores a 50 .

3.3. Parte C

Para determinar los tamaños de matriz y bloques pedidos se determina bajo que condiciones dos bloques de memoria son almacenados en el mismo set de cache. Los procesadores utilizados disponen de caches L1D de 32KB, con tamaño de linea de 64 bytes, y cada set de 8 lineas. Dada una dirección de memoria determinada, por lo tanto, los últimos 6 bits están dedicados a direccionar un byte dentro del bloque de memoria, y los siguientes 6 determinan el set correspondiente al bloque. En base a esto se deduce que, cada 64 bloques (esto es cada $4096 = 2^{12}$ direcciones), se repetirá el mismo set.

Tomando en cuenta que se utilizan elementos de tipo float (4 bytes), cada linea de cache puede contener hasta 16 elementos.

Asumiendo que la matriz B se almacena alineada con el comienzo de un set (es definida desde una dirección de memoria divisible entre 4096) las primeras 16 entradas son almacenadas en el primer set, los siguientes 16 irán al segundo set, y de forma sucesiva, hasta que, luego de 16*64 elementos, se repetirá el mismo set que los primeros 16 elementos.

Tomando esto en cuenta, se utiliza un tamaño de fila $m = 16 \cdot 64$ para la matriz B, de forma tal que las primeras 16 entradas de cada fila de B sean correspondientes al primer set de cache, las siguientes 16 al segundo set y así sucesivamente.

Para aumentar el grado de competencia por un set especifico, se implementa el algoritmo con blocking utilizando bloques rectangulares, para así definir bloques que, en B, estén formados por un numero elevado de filas, y pocas columnas (se detalla mas adelante), a fin de mantener un tamaño de bloque que no supere el de la cache.

Por este motivo, se establece `block_size_n = 2` y `block_size_p = 1024`, de forma que cada bloque contiene 8KB, esto provoca que al computar el producto de un par de bloques todas las entradas de la submatriz de B seleccionada sean direccionadas en el mismo set de cache. Luego, dado que el numero de filas por submatriz (1024) es mayor a la cantidad de lineas por set (8), se produce un gran numero de cache misses, debido a que cada acceso a una fila distinta de B a partir de la octava provocara que se sustituya una linea de cache en el set.

Para asegurarnos de esto, alineamos la matriz B con el tamaño de la cache, por lo que cada fila comienza en el set 0, y finaliza en el set 63

En resumen, usamos:

- m = 1024
- p = 1024
- n = 1024 (16*64)

- block-size-m = 64
- block-size-p = 2
- block-size-n = 1024

Para lo cual obtenemos los siguientes valores:

Reordenamiento	Tiempo de ejecución	GFlops
row-col-it	14.050611s	0.1528
row-col-it (con blocking, bloques rectangulares)	13.262110s	0.1619

Con estos datos, se observa como el rendimiento en GFlops del algoritmo al aplicar blocking se reduce de aproximadamente 0.35 (obenido anteriormente) a 0.1619, o sea, una perdida de aproximadamente el 50 %.

Si reducimos el tamaño de block-size-p a 128, y, por consiguiente, disminuye potencialmente la cantidad de misses provocados por la competencia por un set, obtenemos los siguientes resultados:

Reordenamiento	Tiempo de ejecución	GFlops
row-col-it	14.281526s	0.1503
row-col-it (con blocking, bloques rectangulares)	9.096079s	0.2362

En este caso se observa un aumento de rendimiento en GFlops, como era esperado, al reducir block-size-p, aumenta el rendimiento. Pese a esto, sigue existiendo competencia por sets, por lo que el algoritmo sigue desempeñándose peor que al utilizar bloques cuyo tamaño no provoque esta competencia.