Ejercicio práctico I

Trabajo realizado por: Antonio Gómez Giménez

Ejercicios:

Las matrices se han ajustado a matrices de 512*512 ya que las matrices anteriores superan el espacio en la pila.

a) Medir tiempo de ejecución de la multiplicación de las dos matrices tal y como se planteó en clase. Orden de los bucles: i, j, k.

Primer experimento:
4.777035s

b) Medir tiempo de ejecución de la multiplicación de las dos matrices tal y como se planteó en clase. Orden de los bucles: i, k, j.

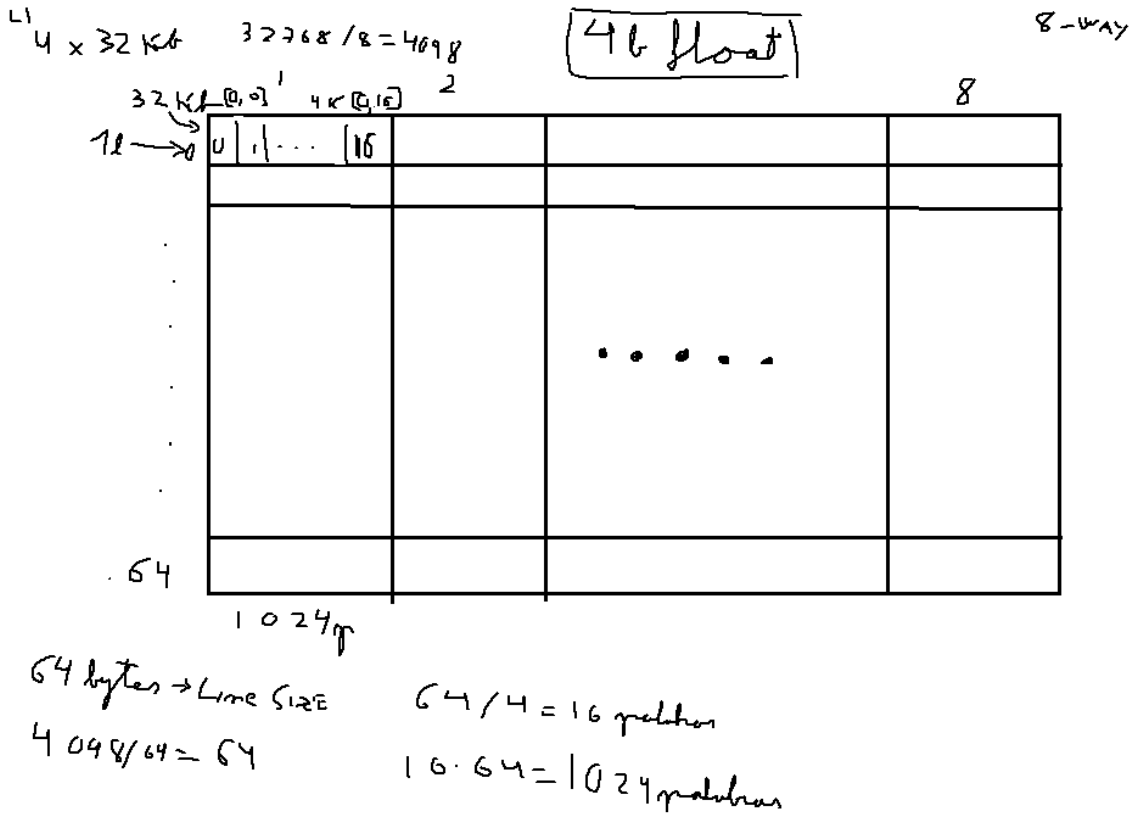
Segundo experimento:
0.748840s

c) Justificar los resultados obtenidos. Destacando el principio de localidad espacial y las características hardware del equipo donde se han realizado las pruebas.

Mi equipo consta de las siguientes características:

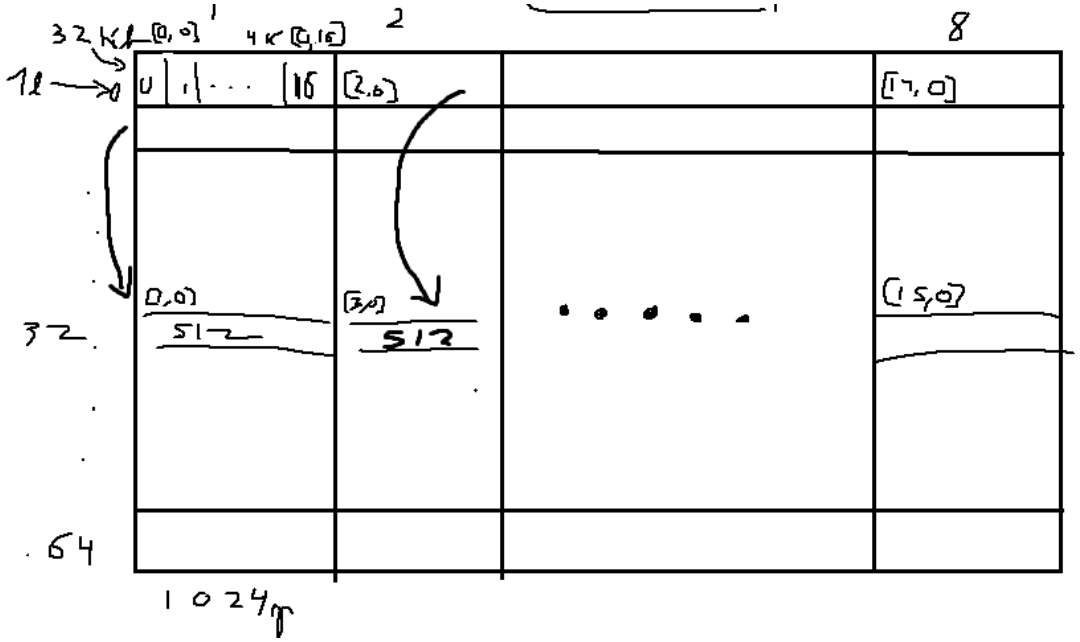
L1:
4 x 32 KB 8-way
L2:
2 x 1 MB 16-way
No hay L3
Tamaño de palabra:
Line size= 64 bytes
Data width = 64 bits = 8 bytes(no se usa en esta práctica, usamos float que son 4 bytes)
8-way

Imagen de la estructura de mi L1:



La mejora de tiempo entre el primer experimento y el segundo experimento es de (4.777035/0.748840-1)*100=**537.92%**, por tanto se realiza una mejora de **538%**, del segundo experimento respecto al primero.

Esto ocurre porque las matrices tienen tamaño de 512 x 512 por tanto al almacenar los datos en la caché sería de la siguiente forma:



No se aprovecha toda la caché en total, se aprovecha únicamente 8 x 2 = 16 bloques de cache donde caben 16 palabras como en una vía caben 1024 palabra, cada vez que se almacenan toda una línea de la matriz A[0],[0], se almacenarán hasta A[0][511], por ello en una sola vía, cabe A[0][0]y A[1][0]. Una vez se llene la caché se escogerá otra vía para evitar sobreescribir, como mucho sucederá 8x2 veces, por tanto el máximo de número de primer índice que podremos almacenará será de A[0][x] hasta A[15][x], siendo x hasta 512.
Cabe destacar que por cada bloque almacenaremos 16 palabras por tanto en el caso del primer bloque será A[0][0] hasta A[0][15], el siguiente A[0][16] hasta A[0][31] y así hasta 512 por tanto al escribir el siguiente bloque, este caerá en la línea 32, ya que 512/16=32.

Esto explica como funciona la caché, ahora vamos a explicar por qué un código es más eficiente que otro:
Si nos fijamos en la parte más problemática a la hora de multiplicar las matrices, la matriz B[k][j], si nos fijamos en los órdenes de los bucles podremos observar por qué son diferente:

-Bucles ijk: en ese caso el último bucle es el k, siendo el que se va a iterar más veces, por tanto, cuando cogemos de memoria principal los datos de la matriz B[k][j], se cogen los datos de B[0][0] (línea 0) hasta B[0][15], posteriormente de B[0][16] (línea 32) hasta B[0][31], todo esto como lo vimos anteriormente y hasta que se llene la memoria, cuando esta se llene se borrarán datos que se encontraban almacenados en caché.
Por consiguiente, si cogemos los datos de esta forma: B[0][0], B[1][0], B[2][0],... ,B[511][0], nunca vamos a tener los datos en caché ya que continuamente se están borrando por errores de caché.

-Bucles ikj: en este caso es bastante mejor, ya que para la matriz de datos B[k][j], si cogemos los datos de esta forma: B[0][0], B[0][1], B[0][2],... ,B[0][511], por tanto, si los bloques de datos los traemos de 512 en 512, por cada fallo de caché, obtenemos 15 aciertos, por tanto, ahorramos tener que traer los datos continuamente de memoria principal, en el caso anterior, siempre se tenía que acceder a memoria principal para traer los datos hasta la caché.

d) Realizar la compilación de las dos versiones anteriores con las opciones de optimización -O0 y -O2 y: Determinar, en base a los tiempos de ejecución, cuál es la opción por defecto; Analizar los resultados y realizar una justificación de los mismos.

Para la opción de optimización -o0 los resultados son:
ijk=4.905061s
ikj=0.764756s

Con esta opción de optimización, los resultados son prácticamente iguales, con esto, lo que podemos darnos cuenta es que mi compilador (gcc) compila por defecto con estas optimizaciones, la optimizaciones que aporta la opción de optimización -O0.

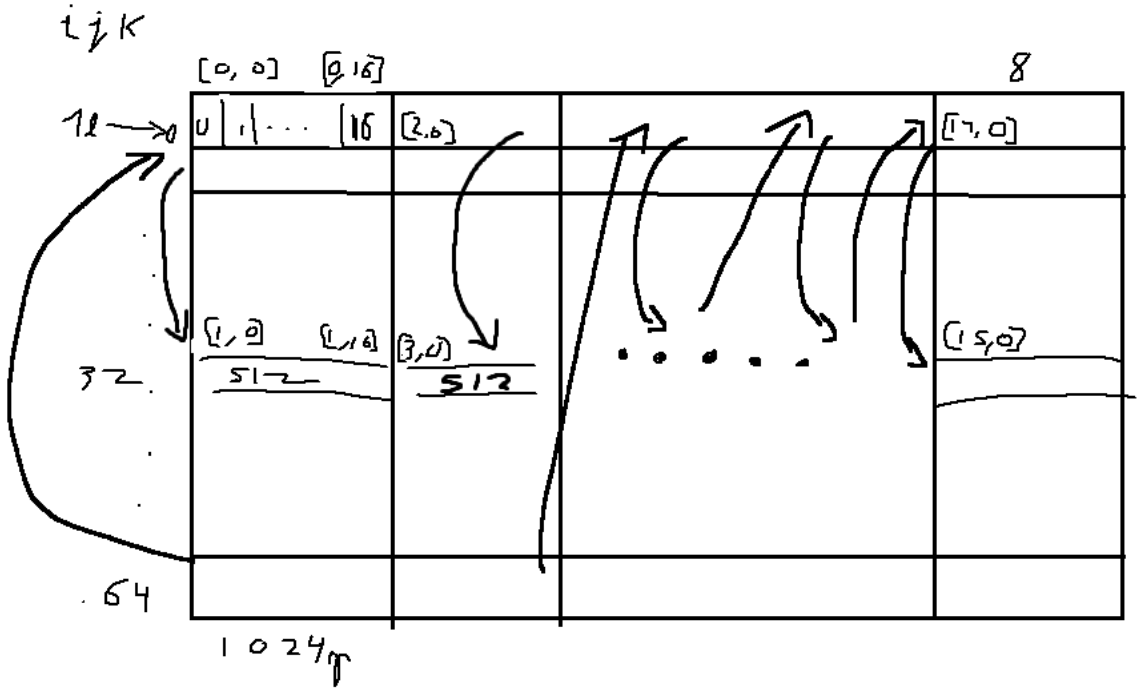
Para la opción de optimización -o2 los resultados son:
ijk=2.100558s
ikj=0.257102s

Respecto a esta opción de optimización podemos entender que, se realizan mejoras pero estas son más notorias para el bucle ikj, esto ocurre porque cuando se realiza la optimización para los bucles ijk se realiza la permutación de bucles mejorando el tiempo, pero al realizar estos cambios consume tiempo que se podría usar para realizar otros tipos de optimizaciones.
Por ello, en el caso de los bucles ikj, como se encuentran ya optimizados, pueden realizar otras mejoras ya que disponen de más tiempo que en el ijk.

e) Compilar por defecto y analizar los tiempos cambiando las dimensiones de las matrices a 1024x1040. Justificar el resultado. Nota: Hay que tener en cuenta que no cambia el número de iteraciones de los bucles.
Ajustandolo a mi problema, serían matrices de 512x528.

Los resultados obtenidos son, para la combinación de bucles, ijk= 1.373206s y para el bucle ikj=0.717652s.

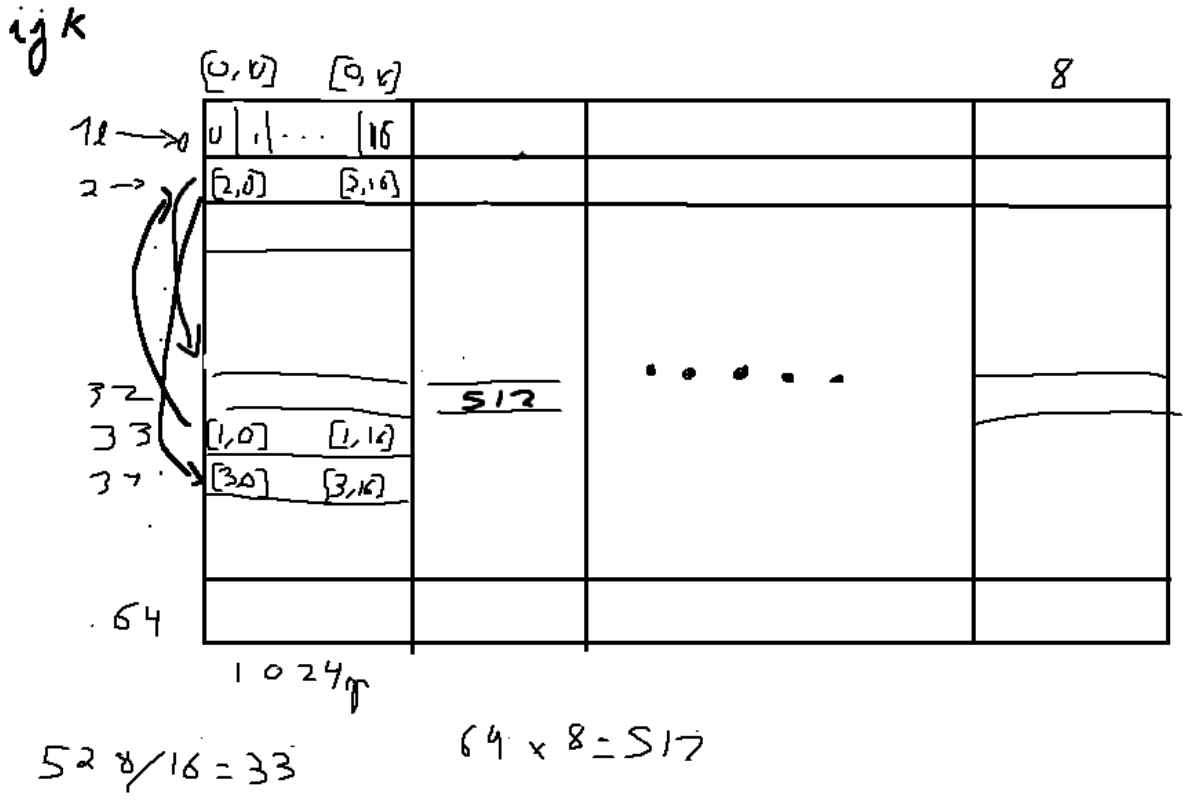
Como hemos explicado anteriormente, el funcionamiento de la cache para la matriz B[k][j] funciona de la siguiente forma:



Esto se debe a que las matrices son de 512 por 512, por tanto, si el tamaño de cada línea es de 64 y usamos float que ocupan 4 bytes, caben 16 palabras, por ello, para traer las 512 palabras es necesario usar 32 líneas (en la vía, caben 1024). Por eso mismo, cada vez que

traemos un bloque nuevo, caemos siempre en las mismas líneas y por consiguiente para no sobreescribirlas usamos los diferentes ways. Para concluir, nos damos cuenta que por lo tanto solo podemos almacenar 16 bloques distintos.

Este problema, al usar una matriz con un tamaño que no es una potencia de dos, mejora el rendimiento y esto se va a entender mejor con la siguiente imagen:



Como podemos observar, se está utilizando toda la caché, esto se debe a que la matriz es de tamaño 528, si caben 16 palabras por bloque, necesitaremos 33 líneas de caché, por tanto, cada vez que traigamos un bloque de memoria desde principal hasta cache iremos realizando saltos de 33 en 33 que permite evitar que siempre estemos utilizando constantemente las mismas vías. Esto se puede observar claramente en la imagen.
Esto nos permite utilizar completamente la caché permitiendo así 64 bloques por 8 vías lo que viene a ser 512 bloques, por lo tanto, en la caché podremos almacenar completamente B[0][0] a B[511][0].
Al usar la caché entera, evitamos que el sistema se sobrecaliente ya que no estará atacando al mismo punto constantemente y así el rendimiento no baja.

El funcionamiento para cualquiera de los dos bucles es el mismo pero se mantienen los conceptos explicados anteriormente.
El ijk que tenía más problemas, se ve beneficiado enormemente pero el ikj ya funciona mejor de por sí ya que seguirá teniendo menos fallos de caché, aun así, el tiempo mejora por lo explicado anteriormente, se usa toda la caché, de esta forma evitamos que se ataque constantemente al mismo punto, lo que produce un sobrecalentamiento y una bajada en el rendimiento.

Cabe destacar que cuando se llena la caché por primera vez, los datos almacenados van desde B[0][0]-B[0][15] hasta B[511][0]-B[511][15]. Por tanto, con esto se permite que para las iteraciones de 1 a 15 no haya fallos de caché ya que los datos se encuentran ya en la caché, este proceso se repetirá cada vez que hagamos 16 iteraciones. Por ello, para el bucle ijk se reducen la cantidad de fallos de caché y mejora la eficiencia.

Apuntes para mí:

$ulimit -n 100000
(Tpeor_caso/Tmejor_caso-1)*100=%mej
https://www.cpu-world.com/CPUs/Bulldozer/AMD-A10-Series%20A10-8700P.html
C[i][j]= A[i][k]*B[k][j];

Tiempos obtenidos:

```
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ gcc p1.c -o P1
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ ./P1
C[0][0]: 12788340736.000000
Total time taken by CPU: 4.777035
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ gcc p1.c -o P1 -O0
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ ./P1
C[0][0]: 13317256192.000000
Total time taken by CPU: 4.905061
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ gcc p1.c -o P1 -O2
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ ./P1
C[0][0]: 12352466944.000000
Total time taken by CPU: 2.100558
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ gcc p1_v2.c -o P1_v2
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ ./P1_v2
C[0][0]: 12549739520.000000
Total time taken by CPU: 0.748840
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ gcc p1_v2.c -o P1_v2 -O0
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ ./P1_v2
C[0][0]: 12666316800.000000
Total time taken by CPU: 0.764756
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ gcc p1_v2.c -o P1_v2 -O2
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ ./P1_v2
C[0][0]: 12924879872.000000
Total time taken by CPU: 0.257102
```

```
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ gcc p1.c -o P1
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ ./P1
C[0][0]: 13633398784.000000
Total time taken by CPU: 1.373206
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ gcc p1_v2.c -o P1_v2
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P1$ ./P1_v2
C[0][0]: 12188018688.000000
Total time taken by CPU: 0.717652
```