

# Informe de Cuarta Práctica de Arquitectura de Computadores

Máximo García Aroca

December 2, 2023

Trataremos la estrategia de *Loop unrolling*.

## 1 Comparación de eventos

Compila el código incluido en las carpetas *Naive* y *ManualUnrolling*, con el compilador *icx* y la opción *-O0* dentro del *Makefile* para evitar que el compilador optimice el código. A continuación usa *perf* para rellenar la siguiente tabla de eventos hardware y calcula el factor de mejora dividiendo cada evento en la versión *Naive* por su correspondiente evento en la versión *ManualUnrolling*. Justifica a qué se deben los cambios observados. Anota también el tiempo que tarda cada versión y su aceleración o "*speedup*".

**Respuesta:**

Evento	Naive	Manual Unrolling	Factor
branches	38.147.171	12.837.187	2.9716
branch-misses	84.027	35.182	2.3884
bus-cycles	10.250.485	5.931.576	1.7281
cache-misses	1.321	2.388	0.5494
cache-references	8.270.860	2.828.464	2.9242
cpu-cycles	241.231.220	136.426.270	1.7682
instructions	407.432.221	268.627.130	1.5167
Tiempo	0.0948	0.0526	1.8023

Ambas anotaciones han sido conseguidas con matrices **256x256**.

El factor de mejora ronda entre un 1.5 y un 3 en la mayoría de eventos. Para valores de matrices más pequeños el factor se reduce y para más grandes el factor se respeta entre este rango que he remarcado. Esto se debe a que la estrategia de Manual Unrolling procura reducir los ciclos perdidos por riesgos de control evitando ejecutar más saltos que el código *Naive*.

## 2 Cálculo de branches

Calcula de forma manual cuantos branches tiene el código y compáralo con el valor obtenido con `perf`, ¿coinciden los valores?. 12 Justifica tu respuesta.

**Respuesta:**

Para esto voy a mirar el código de `ManualUnrolling`. Este es su código:

```
#include <stdio.h>

const char* dgemm_desc = "dgemm using loop unrolling"; // Aquí puedes dar una pequeña descripción de tu programa

void square_dgemm (int n, float* A, float* B, float* C)
{
    /* For each row i of A */
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j+=4)
        {
            register float aki, cij0, cij1, cij2, cij3;

            cij0 = C[i*n+j+0];
            cij1 = C[i*n+j+1];
            cij2 = C[i*n+j+2];
            cij3 = C[i*n+j+3];

            for (int k = 0; k < n; k++)
            {
                aki = A[k*i*n];
                cij0 += aki * B[j+k*n+0];
                cij1 += aki * B[j+k*n+1];
                cij2 += aki * B[j+k*n+2];
                cij3 += aki * B[j+k*n+3];
            }

            C[i*n+j+0] = cij0;
            C[i*n+j+1] = cij1;
            C[i*n+j+2] = cij2;
            C[i*n+j+3] = cij3;
        }
}
```

Se puede apreciar que hay tres bucles y que el segundo avanza de 4 en 4, por lo cual si divido el tamaño de matriz entre 4 correspondiente a este bucle ya podría proceder en calcular los branches como si los tres bucles avanzaran de uno en uno, como en el código *Naive*. Para esto elevo al cubo (porque son tres bucles) el tamaño de la matriz, que es **256**.

$$N^{\circ} \text{ branches} = \frac{256^3}{4} = 4.194.304$$

Este valor difiere de los 12.837.187 *branches* receptados por *perf stat*. Para comprobar por qué puede darse esto he decidido por hacer otra prueba con matrices más grandes, como por ejemplo de 1024:

```
arqucom016@aloe:~/Descargas/MultMat/ManualUnrolling$ perf stat -e branches ./benchmark 1024
Description:      dgemm using loop unrolling on matrices of size 1024 x 1024

Time 3.438504e+00 s

Performance counter stats for './benchmark 1024':

      598.603.004      branches

      3,527012027 seconds time elapsed

      3,525467000 seconds user
      0,000000000 seconds sys
```

En este caso también difiere del cálculo manual:

$$N^{\circ} \text{ branches} = \frac{1024^3}{4} = 268.435.456$$

También difiere de los 596.603.004 *branches* que da *perf stat*. El motivo de esta diferencia no la he podido concluir pero también puedo apuntar que en caso de hacerlo con *Naive* para el caso de matriz **256x256** el valor de *perf* es 38.147.171 y el cálculo manual es  $256^3 = 16.777.216$ , valor que también difiere considerablemente.

### 3 Eventos importantes

¿Cuáles son los eventos más representativos de la mejora de tiempo, es decir, qué eventos son los que han tenido más impacto en la mejora y por qué?

#### Respuesta:

A esta pregunta se puede responder sobre los datos aportados en las figuras del apartado anterior (2). Los datos nos arrojan una importante fuente de información donde el apartado del factor nos indica que eventos como *branches* que tienen un factor de mejora de un 650% en matrices 512x512, por ejemplo, o cache-references con un 8271%. Así como los fallos de caché el cual es muy representativo porque con cache-blocking se reduce mucho los fallos en la caché, por lo cual se interpreta que aumentan los hits al eliminar fallos por conflicto por ejemplo, y esto hace que los accesos a memoria principal descendan en número, lo que implica que el tiempo se ve reducido por usar la caché que es mucho más, rápida, etc.

Pero, como concluí en el apartado anterior esto depende del tamaño de la matriz que usemos, puesto que en ejemplos como la matriz 128x128 en la mayoría de eventos la estrategia Blocking es peor, lo que la hace más lenta.

### 4 Eventos de caché

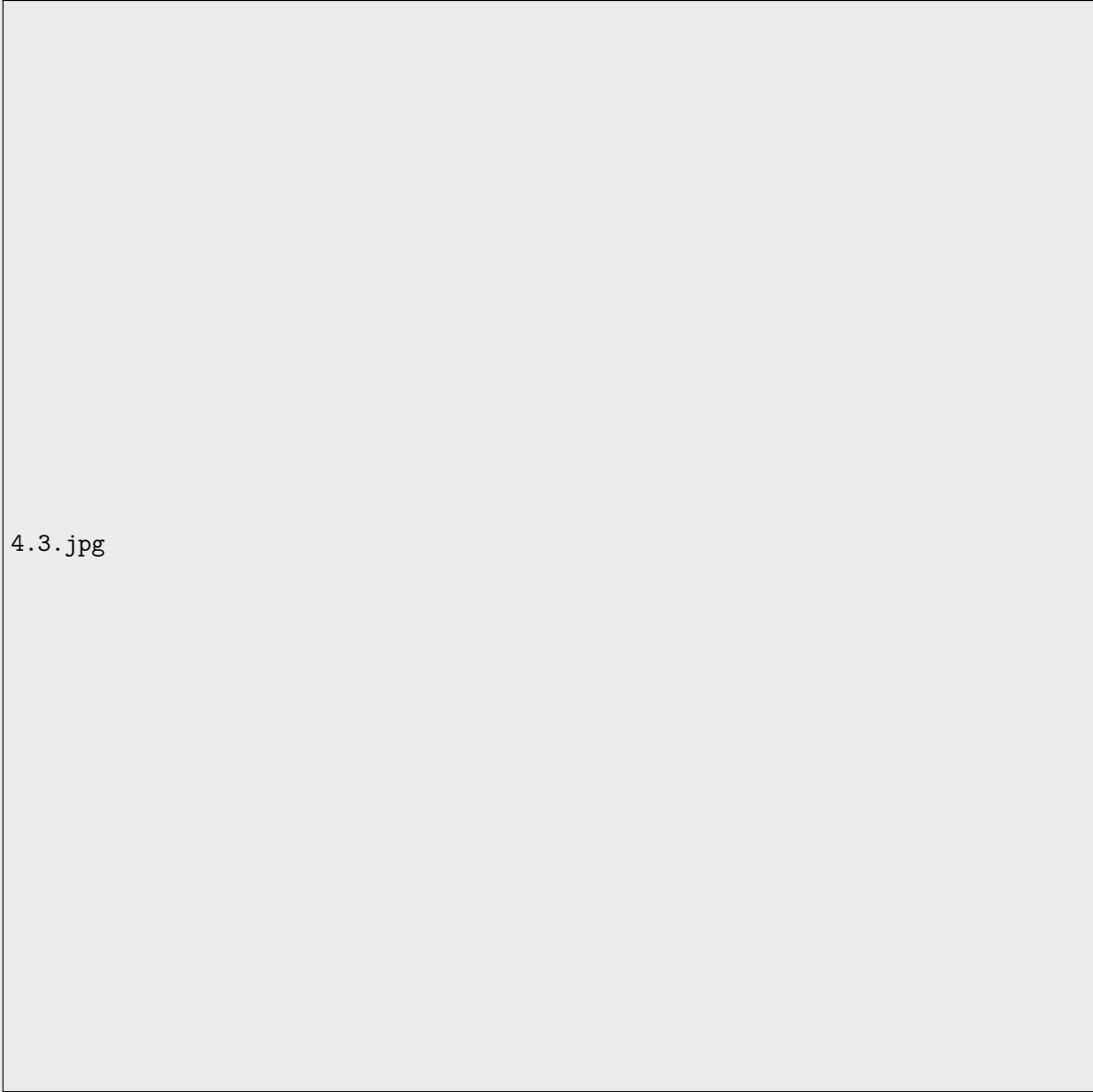
Aparte de los eventos anteriores, hay otros relacionados con la caché como por ejemplo L1-dcache-loads, LLC-loads, dTLB-load-misses, etc. Estos eventos permiten determinar con más exactitud que está ocurriendo en los diferentes niveles del sistema de memoria. Similarmente al ejercicio 2, anota en una tabla los valores de todos los eventos relacionados con la caché de datos y calcula los factores de mejora, para las versiones Naive y Blocking, anotando también el tiempo (y speedup). Justifica razonadamente la variación de los eventos y cómo afectan al tiempo de ejecución.

#### Respuesta:

Aquí van unas medidas de varios eventos asociados a la caché. Algunos eventos no estaban incorporados a la herramienta por alguna razón pero esto es lo que me reportó *perf*:

4.1.jpg

4.2.jpg



4.3.jpg

Se puede observar una mejora de tiempo con la estrategia Blocking de 0.3 s a 0.08 aproximadamente y se justifica con la importante diferencia que hay en todas las medidas de eventos tomadas, principalmente en las de acceso a la memoria caché de nivel 1 (L1) en los fallos (misses) donde se han reducido impresionantemente la cantidad de fallos alcanzando mejoras de 950% por ejemplo en *L1-dcache-load-misses*. La mejora del tiempo también es significativa, es casi 4 veces más rápido.

## 5 Cambio de *BLOCKSIZE*

Cambia el factor de *BLOCKSIZE* en la versión Blocking y vuelve a compilar y ejecutar el programa. Prueba con todas las potencias de 2 entre 1 y 32 y representa gráficamente el factor de mejora del tiempo de ejecución (o la aceleración -speedup- cuando se compara con la versión Naive) para todos los valores de *BLOCKSIZE*. Explica razonadamente qué ocurre.

**Respuesta:**

Se ha hecho el estudio con los tiempos de ejecución cambiando los valores de *BLOCKSIZE* por múltiplos de dos en el rango de 2 a 32 que corresponde con los valores de:  
 $R = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32$ .

Se ha hecho de la siguiente forma:



5.1.jpg

Tomando los valores y haciendo la media con cada valor de *BLOCKSIZE*.

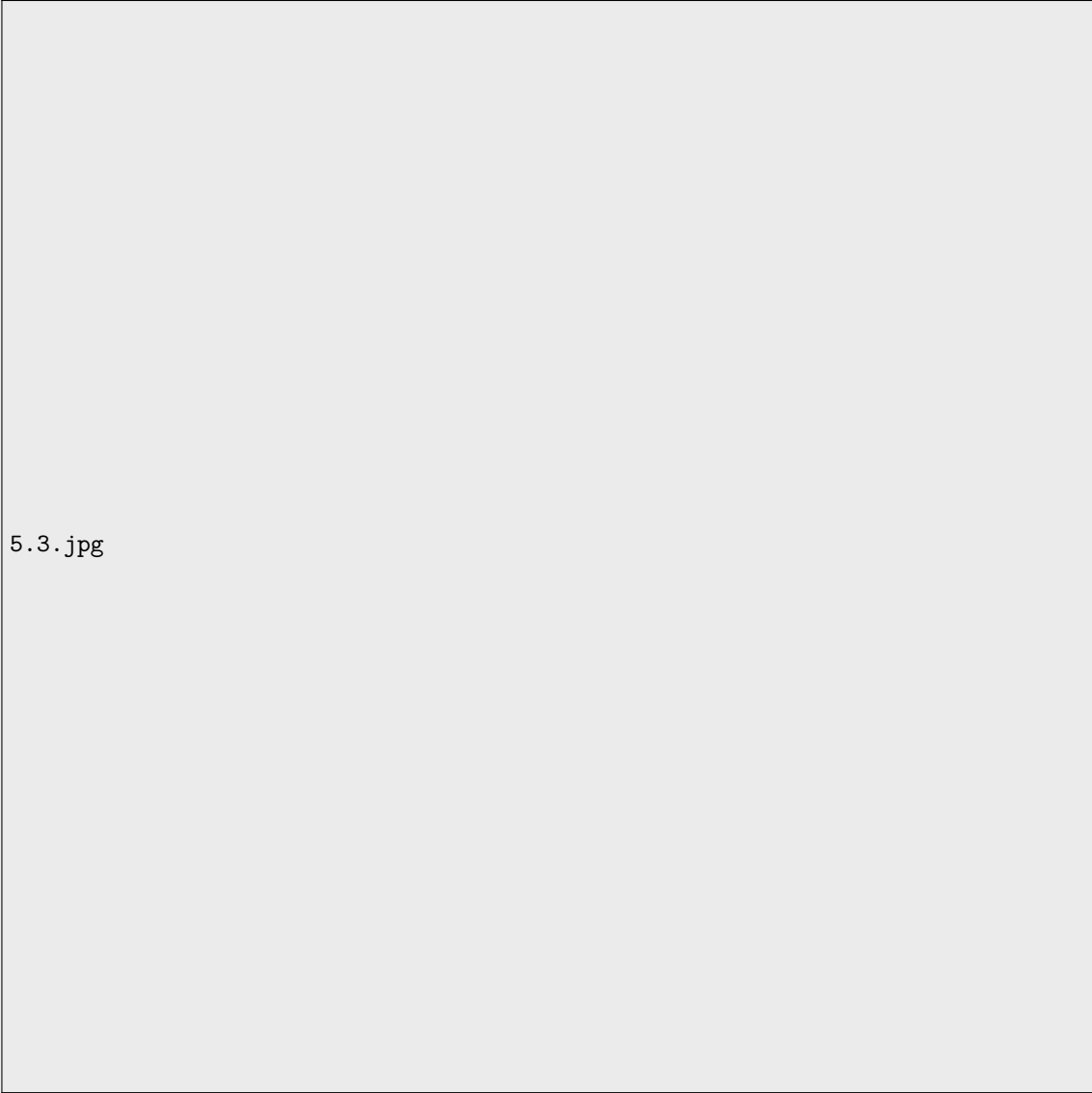


5.2.jpg

Estos son los datos obtenidos para los valores de *BLOCKSIZE* en el eje de la X y como se puede comprobar son valores muy parecidos y que no tienden a aumentar o disminuir, solo distan mínimamente para un tamaño de matriz 512x512.

Sin embargo si la matriz es 256x256:





5.3.jpg

Como se puede comprobar para los valores de *BLOCKSIZE* 4 y 8 hay una diferencia en cuanto a mejora de tiempo de ejecución evidente. Pero para 16 y 32 vuelve a subir. Esto puede darse porque para estos valores más grandes de tamaño de bloque se deben de estar produciendo fallos de caché por capacidad, donde el tamaño de bloque es tal que se producen fallos porque no todos los bloques con los datos de las matrices pueden almacenarse en la caché por falta de espacio/capacidad. Una tendencia parecida se puede intuir en la primera gráfica donde los valores entre 4 y 16 tienden a descender con respecto a cuando el tamaño de bloque es de 2 pero una vez pasamos a tamaño de bloque 18 en adelante empieza a subir por esto que acabo de comentar.

## 6 Speed-up con diferente *BLOCKSIZE*

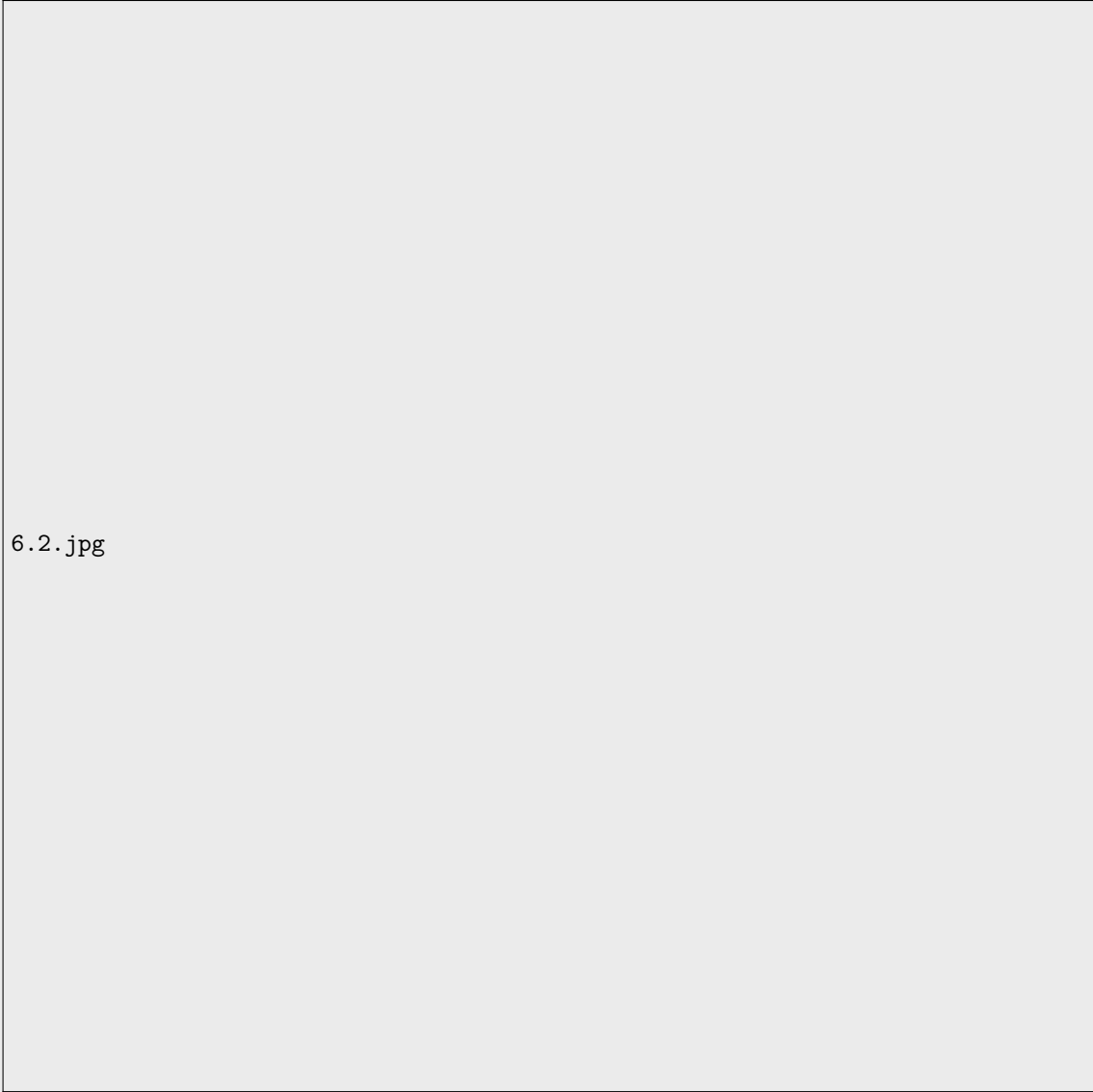
Ejecuta ahora cada versión (Naive y Blocking) con diferentes tamaños de matrices y representa gráficamente el factor de mejora del tiempo (aceleración) obtenida para cada valor de *BLOCKSIZE* (entre 1 y 32, como en el ejercicio 5). Puedes probar con tamaños de matrices de 64, 128, 256, 512 y 1024. Para cada tamaño

de matriz, explica cómo afecta al rendimiento el tamaño de BLOCKSIZE y qué ocurre si cambia el tamaño de la matriz.

**Respuesta:**



Aquí se ven los valores que toma la estrategia de *cache blocking* para los distintos valores de tamaño de bloque *BLOCKSIZE* y diferentes tamaños de matriz.



6.2.jpg

Y aquí se puede ver el speed-up que genera la estrategia de *cache-blocking* para los distintos tamaños de matrices a lo largo de el rango de valores que se le da al tamaño de bloque en caché (B1=BLOCKSIZE 1, B8=BLOCKSIZE 8, etc).

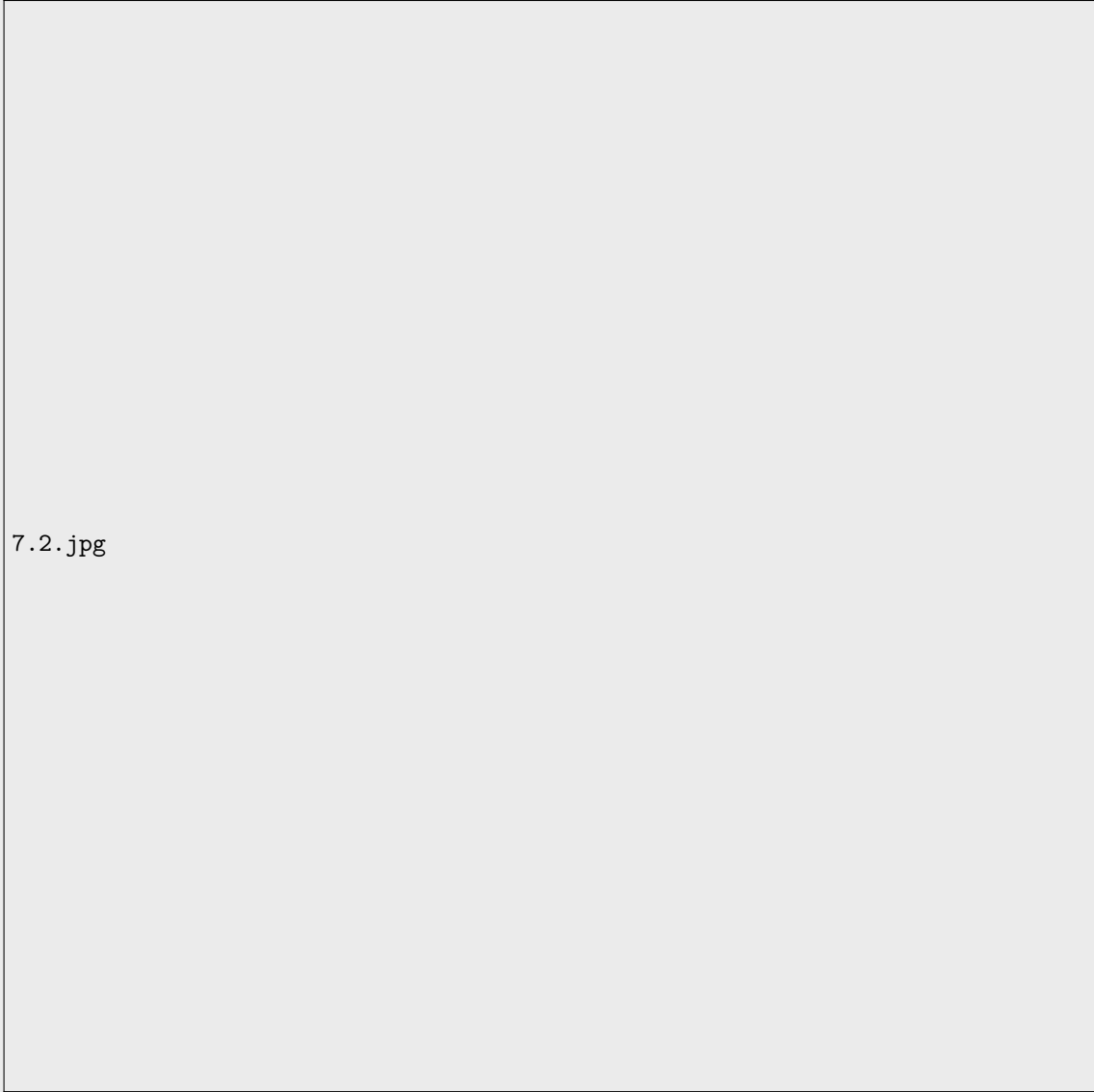
Lo que se puede apreciar de esta gráfica es que cuand el tamaño de bloque es de 8, para las matrices más grandes, como 512 o 1024 se aprecia un incremento de aceleración muy alta en comparación con el resto. Cuando se supera ese tamaño de bloque sigue mejor que con valores más bajos pero reduce mucho su mejora posiblemente por los fallos en caché producidos por capacidad, como dije anteriormente.

## 7 Valores de eventos para diferentes *BLOCKSIZE*

Para los distintos tamaños de matrices y valor de BLOCKSIZE, anota los eventos hardware relacionados con la caché, comprueba como varían y justifica cómo impactan en el tiempo cuándo varía BLOCKSIZE para un tamaño dado, y cuando varía el tamaño de la matriz.

**Repuesta:**

7.1.jpg



7.2.jpg

Se puede apreciar con los datos comparando entre diferentes casos de *BLOCKSIZE* con el mismo tamaño de matriz que para valores de 8-16 se reduce en gran medida el número de fallos de caché sobre todo en matrices de tamaño más grande como las 512x512 o las 1024x1024.