# Informe de Tercera Práctica de Arquitectura de Computadores

Máximo García Aroca

23 de noviembre, 2023

## 1 Referencias a caché

El siguiente ejemplo permite comprender mejor dónde se encuentran los fallos de memoria caché cuando se usa la versión Naive del producto de matrices. Sea el producto de dos matrices de tamaño 3x3, donde la matriz A se encuentra almacenada a partir de la dirección 1000h, la matriz B de la dirección 2000h y la matriz C de 3000h. Además, el sistema de memoria donde se ejecuta el código tiene una memoria cache para datos, con 4 bloques de 2 palabras cada uno de ellos, y organización asociativa por conjuntos de 2 vías. Contesta las siguientes preguntas:

### 1.1 a)

Obtén la traza de referencias de memoria generadas al ejecutar el código usando el Listado 2, considerando que los accesos de escritura se comportan igual que los de lectura, y completa la siguiente tabla con la evolución de la memoria caché (añade todas las líneas que sean necesarias):

**Respuesta:** Analizando el código de *dgemm.c* y siguiendo el curso del ejecutable con un debugger relleno la tabla que me piden con el orden de arriba a abajo cronológicamente de las referencias a memoria tanto de lectura como de escritura (load and store) que se hacen a las distintas partes de las matrices.

| Referencia | Dirección | Conjunto | Vía | ¿Acierto? |
|---|---|---|---|---|
| C0 | 3000h | 0 | 0 | N |
| A0 | 1000h | 0 | 1 | N |
| B0 | 2000h | 0 | 0 | N |
| A1 | 1004 | 0 | 1 | S |
| B3 | 200C | 1 | 0 | N |
| A2 | 1008 | 1 | 1 | N |
| B6 | 2018 | 1 | 0 | N |
| C0 | 3000 | 0 | 0 | N |
| C1 | 3004 | 0 | 0 | S |
| A0 | 1000 | 0 | 1 | S |
| B1 | 2004 | 0 | 0 | N |
| A1 | 1004 | 0 | 1 | S |
| B4 | 2010 | 0 | 0 | N |
| A2 | 1008 | 1 | 1 | S |
| B7 | 201C | 1 | 0 | S |
| C1 | 3004 | 0 | 1 | N |
| C2 | 3008 | 1 | 1 | N |
| A0 | 1000 | 0 | 0 | N |
| B2 | 2008 | 1 | 0 | N |
| A1 | 1004 | 0 | 0 | S |
| B5 | 2014 | 0 | 1 | N |
| A2 | 1008 | 1 | 1 | N |
| B8 | 2020 | 0 | 0 | N |
| C2 | 3008 | 1 | 0 | N |
| C3 | 300C | 1 | 0 | S |
| A4 | 100C | 0 | 1 | S |
| B0 | 2000 | 0 | 1 | N |
| A4 | 1010 | 0 | 0 | N |
| B3 | 200C | 1 | 0 | N |
| A5 | 1014 | 0 | 0 | S |
| B6 | 2018 | 1 | 1 | N |
| C3 | 300C | 1 | 0 | N |
| C4 | 3010 | 0 | 1 | N |
| A3 | 100C | 1 | 1 | N |
| B1 | 2004 | 0 | 0 | N |
| A4 | 1010 | 0 | 1 | N |
| B4 | 2010 | 0 | 0 | N |
| A5 | 1014 | 0 | 1 | S |
| B7 | 201C | 1 | 0 | N |

| | | | | |
|---|---|---|---|---|
| C4 | 3010 | 0 | 0 | N |
| C5 | 3014 | 0 | 0 | S |
| A3 | 100C | 1 | 1 | S |
| B2 | 2008 | 1 | 0 | N |
| A4 | 1010 | 0 | 1 | S |
| B5 | 2014 | 0 | 0 | N |
| A5 | 1014 | 0 | 1 | S |
| B8 | 2020 | 0 | 0 | N |
| C5 | 3014 | 0 | 1 | N |
| C6 | 3018 | 1 | 1 | N |
| A6 | 1018 | 1 | 0 | N |
| B0 | 2000 | 0 | 0 | N |
| A7 | 101C | 1 | 0 | S |
| B3 | 200C | 1 | 1 | N |
| A8 | 1020 | 0 | 1 | N |
| B6 | 2018 | 1 | 0 | N |
| C6 | 3018 | 1 | 1 | N |
| C7 | 301C | 1 | 1 | S |
| A6 | 1018 | 1 | 0 | N |
| B1 | 2004 | 0 | 0 | S |
| A7 | 101C | 1 | 1 | S |
| B4 | 2010 | 0 | 0 | N |
| A8 | 1020 | 0 | 0 | N |
| B7 | 201C | 1 | 1 | N |
| C7 | 301C | 1 | 1 | N |
| C8 | 3020 | 0 | 0 | N |
| A6 | 1018 | 1 | 1 | N |
| B2 | 2008 | 1 | 1 | N |
| A7 | 101C | 1 | 1 | S |
| B5 | 2014 | 0 | 0 | N |
| A8 | 1020 | 0 | 0 | N |
| B8 | 2020 | 0 | 0 | N |
| C8 | 3020 | 0 | 0 | N |

### 1.2 b)

$MissRate(A,B,C) = \frac{Misses}{IC} = \frac{49}{72} = 0.68$
$MissRate(A) = \frac{11}{27} = 0.41$
$MissRate(B) = \frac{24}{27} = 0.89$
$MissRate(C) = \frac{14}{18} = 0.78$

Por cómo se ejecuta el código, el acceso a valores de filas aprovecha mejor la localidad espacial lo que produce menos errores en los valores accedidos para la matriz A ante las matrices B y C.

## 2 Comparación

Compila el código incluido en las carpetas Naive y Blocking, con el compilador icx y la opción –O3 dentro del Makefile para que el compilador optimice el código al máximo. A continuación, usa perf para rellenar la tabla de eventos hardware y calcula los factores de mejora (dividiendo cada evento en la versión Naive por su correspondiente evento en la versión Blocking). Justifica a qué se deben los cambios observados. Anota también el tiempo que tarda cada versión y el factor de mejora (aceleración o speedup).

**Respuesta:**

He hecho las comprobaciones con *"perf stat"* para valores de matriz 512x512 y estos son los resultados:

| Matriz 512x512 | Naive | Blocking | Factor |
|---|---|---|---|
| *branches* | 148.548.700 | 22.975.516 | 647% |
| *branch-misses* | 302.844 | 71.914 | 421% |
| *bus-cycles* | 33.318.963 | 9.068.129 | 367% |
| *cache-misses* | 20.495 | 5.443 | 377% |
| *cache-references* | 134.875.010 | 1.630.705 | 8271% |
| *cpu-cycles* | 1.027.798.080 | 248.691.222 | 413% |
| *instructions* | 1.002.059.688 | 580.393.547 | 173% |
| *tiempo* | 5,70E-03 | 7,24E-02 | 8% |

Se puede intuir que para valores de matriz más pequeños esta mejora de optimización con cache-blocking pierde eficiencia. Por lo que es recomendable usarse para matrices con muchos datos.

| Matriz 128x128 | Naive | Blocking | Factor |
|---|---|---|---|
| *branches* | 2.099.775 | 1.635.402 | 128% |
| *branch-misses* | 12.418 | 13.111 | 95% |
| *bus-cycles* | 343.404 | 271.825 | 126% |
| *cache-misses* | 972 | 1.057 | 92% |
| *cache-references* | 31.594 | 32.223 | 98% |
| *cpu-cycles* | 7.840.822 | 9.366.293 | 84% |
| *instructions* | 11.289.185 | 15.244.564 | 74% |
| *tiempo* | 4,70E-04 | 1,00E-03 | 47% |

## 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 desciendan 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:

```
arqcom016@aloe:~/Descargas/MultMat/Naive$ perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores,L1-dcache-store-misses,L1-dcache-prefetches,L1-dcache-prefetch-mis
ses,L1-icache-loads,L1-icache-load-misses,L1-icache-prefetches,L1-icache-prefetch-misses,LLC-loads,LLC-load-misses,LLC-stores,LLC-store-misses,LLC-prefetches,LLC-prefetch-misses,dTLB-loads,dTLB-load-misses,
dTLB-stores,dTLB-store-misses,dTLB-prefetches,dTLB-prefetch-misses,iTLB-loads,iTLB-load-misses,branch-loads,branch-load-misses,node-loads,node-load-misses,node-stores,node-store-misses,node-prefetches,node-
prefetch-misses ./benchmark 512
Description:    A naive C version on matrices of size 512 x 512

Time 3.128642e-01 s

 Performance counter stats for './benchmark 512':

       134.570.439      cache-references                                              (11,84%)
             5.833      cache-misses              #    0,004 % of all cache refs      (16,61%)
       290.258.569      L1-dcache-loads                                               (21,39%)
       122.244.285      L1-dcache-load-misses     #   42,12% of all L1-dcache accesses  (26,16%)
        27.485.010      L1-dcache-stores                                              (30,93%)
   <not supported>      L1-dcache-store-misses
   <not supported>      L1-dcache-prefetches
   <not supported>      L1-dcache-prefetch-misses
   <not supported>      L1-icache-loads
            64.632      L1-icache-load-misses                                         (35,71%)
   <not supported>      L1-icache-prefetches
   <not supported>      L1-icache-prefetch-misses
       120.279.041      LLC-loads                                                     (36,89%)
             5.519      LLC-load-misses           #    0,00% of all LL-cache accesses  (38,08%)
            20.382      LLC-stores                                                    (9,55%)
               230      LLC-store-misses                                              (9,55%)
   <not supported>      LLC-prefetches
   <not supported>      LLC-prefetch-misses
       281.663.996      dTLB-loads                                                    (14,32%)
               790      dTLB-load-misses          #    0,00% of all dTLB cache accesses  (19,09%)
           367.413      dTLB-stores                                                   (23,87%)
                 0      dTLB-store-misses                                             (28,64%)
   <not supported>      dTLB-prefetches
   <not supported>      dTLB-prefetch-misses
                 2      iTLB-loads                                                    (33,42%)
                86      iTLB-load-misses          # 4300,00% of all iTLB cache accesses  (38,19%)
       142.827.254      branch-loads                                                  (38,19%)
           285.011      branch-load-misses                                            (38,05%)
               895      node-loads                                                    (36,86%)
               286      node-load-misses                                              (35,66%)
               488      node-stores                                                   (7,16%)
               125      node-store-misses                                             (7,16%)
   <not supported>      node-prefetches
   <not supported>      node-prefetch-misses

     0,336248356 seconds time elapsed

     0,331603000 seconds user
     0,000043000 seconds sys
```

```
arqcom016@aloe:~/Descargas/MultMat/Naive$ perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores,L1-dcache-store-misses,L1-dcache-prefetches,L1-dcache-prefetch-mis
ses,L1-icache-loads,L1-icache-load-misses,L1-icache-prefetches,L1-icache-prefetch-misses,LLC-loads,LLC-load-misses,LLC-stores,LLC-store-misses,LLC-prefetches,LLC-prefetch-misses,dTLB-loads,dTLB-load-misses,
dTLB-stores,dTLB-store-misses,dTLB-prefetches,dTLB-prefetch-misses,iTLB-loads,iTLB-load-misses,branch-loads,branch-load-misses,node-loads,node-load-misses,node-stores,node-store-misses,node-prefetches,node-
prefetch-misses ../Blocking/benchmark 512
Description:    dgemm using cache blocking on matrices of size 512 x 512

Time 8.047952e-02 s

 Performance counter stats for '../Blocking/benchmark 512':

         1.687.689      cache-references                                              (14,47%)
             3.541      cache-misses              #    0,210 % of all cache refs      (22,25%)
       184.502.284      L1-dcache-loads                                               (30,04%)
        12.894.313      L1-dcache-load-misses     #    6,99% of all L1-dcache accesses  (37,73%)
        28.106.508      L1-dcache-stores                                              (41,62%)
   <not supported>      L1-dcache-store-misses
   <not supported>      L1-dcache-prefetches
   <not supported>      L1-dcache-prefetch-misses
   <not supported>      L1-icache-loads
            48.170      L1-icache-load-misses                                         (45,49%)
   <not supported>      L1-icache-prefetches
   <not supported>      L1-icache-prefetch-misses
         1.287.478      LLC-loads                                                     (45,49%)
             4.088      LLC-load-misses           #    0,32% of all LL-cache accesses  (45,49%)
               500      LLC-stores                                                    (7,79%)
               308      LLC-store-misses                                              (7,79%)
   <not supported>      LLC-prefetches
   <not supported>      LLC-prefetch-misses
       202.453.922      dTLB-loads                                                    (11,68%)
             2.504      dTLB-load-misses          #    0,00% of all dTLB cache accesses  (15,57%)
        11.824.879      dTLB-stores                                                   (19,47%)
               231      dTLB-store-misses                                             (23,36%)
   <not supported>      dTLB-prefetches
   <not supported>      dTLB-prefetch-misses
                 0      iTLB-loads                                                    (27,26%)
                25      iTLB-load-misses          #    0,00% of all iTLB cache accesses  (31,15%)
        10.681.480      branch-loads                                                  (31,15%)
            41.077      branch-load-misses                                            (31,15%)
                 0      node-loads                                                    (31,15%)
               199      node-load-misses                                              (31,15%)
               102      node-stores                                                   (7,79%)
                12      node-store-misses                                             (7,79%)
   <not supported>      node-prefetches
   <not supported>      node-prefetch-misses

     0,104126834 seconds time elapsed

     0,103256000 seconds user
     0,000000000 seconds sys
```

| Matriz 512x512 | Naive | Blocking | Factor |
|---|---|---|---|
| *cache-references* | 134.570.439 | 1.687.689 | 7974% |
| *cache-misses* | 5.833 | 3.541 | 165% |
| *L1-dcache-loads* | 290.258.569 | 184.502.284 | 157% |
| *L1-dcache-load-misses* | 122.244.285 | 12.894.313 | 948% |
| *L1-dcache-stores* | 27.485.010 | 28.106.508 | 98% |
| *L1-icache-loadmisses* | 64.632 | 48.170 | 134% |
| *LLC-loads* | 120.279.041 | 1.287.478 | 9342% |
| *LLC-load-misses* | 5.519 | 4088 | 135% |
| *LLC-stores* | 20.382 | 500 | 4076% |
| *LLC-store-misses* | 230 | 308 | 75% |
| *dTLB-loads* | 281.663.996 | 202.453.922 | 139% |
| *dTLB-load-misses* | 790 | 2.504 | 32% |
| *dTLB-stores* | 367.413 | 11.824.879 | 3% |
| *dLTB-store-misses* | 0 | 231 | 0% |
| *iTLB-loads* | 2 | 0 | #¡DIV/0! |
| *iTLB-load-misses* | 86 | 25 | 344% |
| *branch-loads* | 142.827.254 | 10.681.480 | 1337% |
| *branch-load-misses* | 285.011 | 41077 | 694% |
| *node-loads* | 895 | 0 | #¡DIV/0! |
| *node-load-misses* | 286 | 199 | 144% |
| *node-stores* | 488 | 102 | 478% |
| *node-stores-misses* | 125 | 12 | 1042% |

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:**

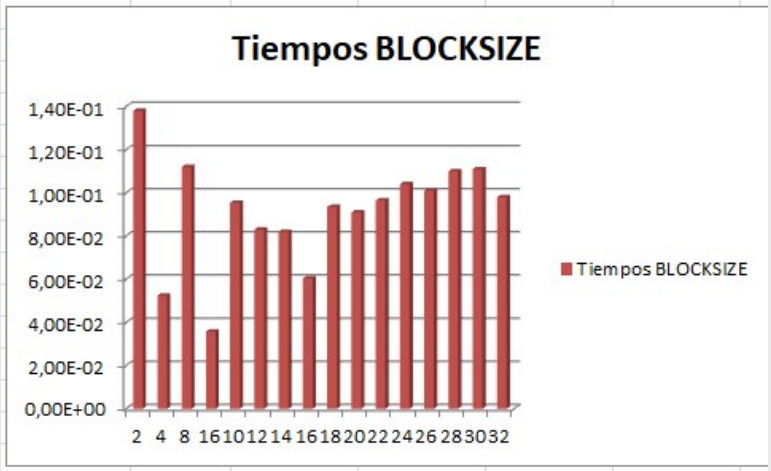
Se ha hecho el estudio con los tiempos de ejecución cambiando los valores de *BLOCKSIZE* por múltiplos de dos en el rango de 2 a 32 que corresponde con los valores de:
$R = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32$.

Se ha hecho de la siguiente forma:

```
arqcom016@aloe:~/Descargas/MultMat/Blocking$ ./benchmark 256
Description:    dgemm using cache blocking on matrices of size 256 x 25

Time 1.554989e-02 s
arqcom016@aloe:~/Descargas/MultMat/Blocking$ ./benchmark
Description:    dgemm using cache blocking on matrices of size 512 x 51

Time 1.236678e-01 s
arqcom016@aloe:~/Descargas/MultMat/Blocking$ ./benchmark
Description:    dgemm using cache blocking on matrices of size 512 x 51

Time 1.427393e-01 s
arqcom016@aloe:~/Descargas/MultMat/Blocking$ ./benchmark
Description:    dgemm using cache blocking on matrices of size 512 x 51

Time 1.261194e-01 s
arqcom016@aloe:~/Descargas/MultMat/Blocking$ ./benchmark
Description:    dgemm using cache blocking on matrices of size 512 x 51

Time 1.381449e-01 s
arqcom016@aloe:~/Descargas/MultMat/Blocking$
```

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

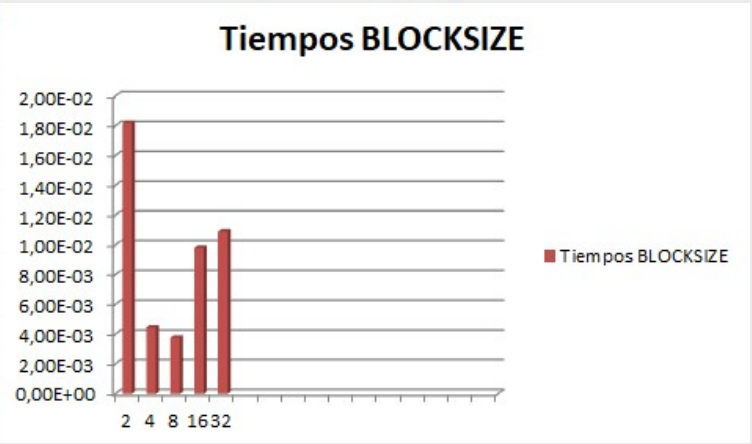
| EjeY | Tiempos BLOCKSIZE |
|---|---|
| 2 | 1,38E-01 |
| 4 | 5,24E-02 |
| 8 | 1,12E-01 |
| 16 | 3,58E-02 |
| 10 | 9,54E-02 |
| 12 | 8,30E-02 |
| 14 | 8,20E-02 |
| 16 | 6,04E-02 |
| 18 | 9,35E-02 |
| 20 | 9,10E-02 |
| 22 | 9,65E-02 |
| 24 | 1,04E-01 |
| 26 | 1,01E-01 |
| 28 | 1,10E-01 |
| 30 | 1,11E-01 |
| 32 | 9,80E-02 |




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:

| EjeX | Tiempos BLOCK |
|---|---|
| 2 | 1,82E-02 |
| 4 | 4,45E-03 |
| 8 | 3,76E-03 |
| 16 | 9,80E-03 |
| 32 | 1,09E-02 |



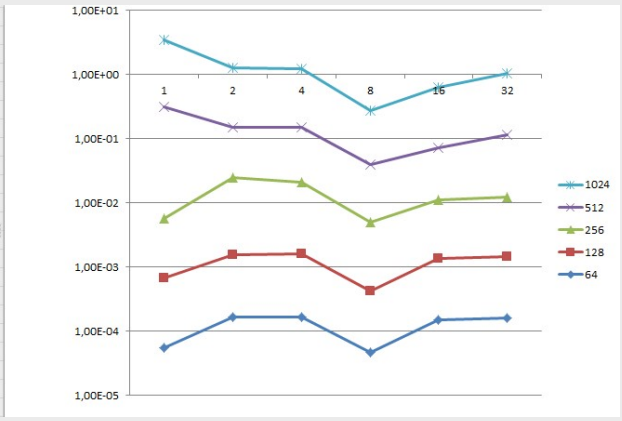

Como se puede comprobar para los valores de *BLOCKSIZE* 4 y 8 hay una diferencia en cuanto a mejora de tiempo de ejecución evidente. Pero para 16 y 32 vuelve a subir. Esto puede darse porque para estos valores más grandes de tamaño de bloque se deben de estar produciendo fallos de caché por capacidad, 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 entr e4 y 16 tieden 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.

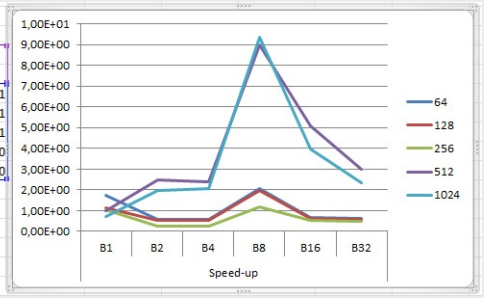
**Repuesta:**

| | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|
| **1** | 5,57E-05 | 6,25E-04 | 5,07E-03 | 3,03E-01 | 3,18E+00 |
| **2** | 1,67E-04 | 1,40E-03 | 2,28E-02 | 1,24E-01 | 1,12E+00 |
| **4** | 1,66E-04 | 1,41E-03 | 1,97E-02 | 1,28E-01 | 1,08E+00 |
| **8** | 4,70E-05 | 3,70E-04 | 4,57E-03 | 3,43E-02 | 2,36E-01 |
| **16** | 1,50E-04 | 1,20E-03 | 9,86E-03 | 6,06E-02 | 5,58E-01 |
| **32** | 1,61E-04 | 1,29E-03 | 1,08E-02 | 1,03E-01 | 9,36E-01 |




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.

| | | Speed-up | | | | | |
|---|---|---|---|---|---|---|---|
| | Naive | B1 | B2 | B4 | B8 | B16 | B32 |
| 64 | 9,75E-05 | 1,75E+00 | 5,84E-01 | 5,87E-01 | 2,07E+00 | 6,50E-01 | 6,06E-01 |
| 128 | 7,22E-04 | 1,16E+00 | 5,16E-01 | 5,12E-01 | 1,95E+00 | 6,02E-01 | 5,60E-01 |
| 256 | 5,31E-03 | 1,05E+00 | 2,33E-01 | 2,70E-01 | 1,16E+00 | 5,39E-01 | 4,92E-01 |
| 512 | 3,08E-01 | 1,02E+00 | 2,48E+00 | 2,41E+00 | 8,98E+00 | 5,08E+00 | 2,99E+00 |
| 1024 | 2,21E+00 | 6,95E-01 | 1,97E+00 | 2,05E+00 | 9,36E+00 | 3,96E+00 | 2,36E+00 |




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:**

| BLOCKSIZE | EVENTO | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| 1 | *cache-references* | 28.696 | 32.755 | 302.513 | 143.019.050 | 1.146.092.993 |
| | *cache-misses* | 635 | 878 | 1.985 | 5.744 | 130.041 |
| | *L1-dcache-loads* | 1.409.229 | 3.132.151 | 13.659.116 | 285.994.031 | 2.215.322.206 |
| | *L1-dcache-load-misses* | 31.831 | 199.126 | 1.234.335 | 144.221.024 | 1.175.574.590 |
| | *L1-dcache-stores* | 814.994 | 1.271.305 | 3.075.381 | 143.809.770 | 1.110.514.539 |
| | *L1-icache-load-misses* | 17.977 | 19.410 | 17.990 | 29.084 | 87.205 |
| | *LLC-loads* | 11.450 | 13.842 | 280.503 | 142.919.276 | 1.143.658.221 |
| | *LLC-load-misses* | 247 | 313 | 699 | 2.208 | 120.263 |
| 2 | *cache-references* | 28.506 | 31.756 | 372.215 | 143.031.368 | 1.146.540.225 |
| | *cache-misses* | 654 | 618 | 1.743 | 14.943 | 170.177 |
| | *L1-dcache-loads* | 1.412.998 | 3.129.952 | 13.662.703 | 286.027.153 | 2.215.326.611 |
| | *L1-dcache-load-misses* | 32.574 | 197.086 | 1.233.770 | 144.316.148 | 1.180.203.627 |
| | *L1-dcache-stores* | 817.380 | 1.269.882 | 3.076.932 | 143.833.163 | 1.110.511.552 |
| | *L1-icache-load-misses* | 15.960 | 16.126 | 18.724 | 29.702 | 83.661 |
| | *LLC-loads* | 11.275 | 13.116 | 351.024 | 142.919.926 | 1.143.761.640 |
| | *LLC-load-misses* | 269 | 248 | 576 | 12.219 | 158.851 |
| 4 | *cache-references* | 28.003 | 32.885 | 522.200 | 143.036.657 | 1.146.106.943 |
| | *cache-misses* | 574 | 896 | 1.725 | 6.672 | 73.736 |
| | *L1-dcache-loads* | 1.408.173 | 3.129.174 | 13.647.866 | 285.996.251 | 2.215.272.668 |
| | *L1-dcache-load-misses* | 31.812 | 198.089 | 1.230.200 | 144.254.464 | 1.176.934.228 |
| | *L1-dcache-stores* | 813.525 | 1.269.318 | 3.068.725 | 143.811.394 | 1.110.470.229 |
| | *L1-icache-load-misses* | 17.780 | 19.449 | 20.860 | 30.189 | 71.106 |
| | *LLC-loads* | 11.493 | 13.950 | 501.441 | 142.923.856 | 1.143.646.047 |
| | *LLC-load-misses* | 205 | 349 | 552 | 3.043 | 61.651 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | *cache-references* | 27.467 | 31.583 | 404.411 | 143.012.789 | 1.146.085.880 |
| | *cache-misses* | 599 | 1.089 | 835 | 4.810 | 106.949 |
| | *L1-dcache-loads* | 1.408.242 | 3.128.122 | 13.660.833 | 285.983.558 | 2.215.202.341 |
| | *L1-dcache-load-misses* | 32.477 | 198.433 | 1.233.273 | 144.299.852 | 1.176.919.512 |
| | *L1-dcache-stores* | 814.030 | 1.269.237 | 3.076.436 | 143.802.933 | 1.110.464.298 |
| | *L1-icache-load-misses* | 14.690 | 14.766 | 20.275 | 23.913 | 66.895 |
| | *LLC-loads* | 11.100 | 12.869 | 382.854 | 142.917.663 | 1.143.619.415 |
| | *LLC-load-misses* | 228 | 435 | 322 | 1.700 | 94.219 |
| 16 | *cache-references* | 28.888 | 31.050 | 362.904 | 143.027.677 | 1.146.485.765 |
| | *cache-misses* | 1.404 | 1.052 | 1.718 | 21.127 | 82.821 |
| | *L1-dcache-loads* | 1.412.367 | 3.127.620 | 13.652.562 | 286.014.275 | 2.215.300.974 |
| | *L1-dcache-load-misses* | 29.896 | 198.741 | 1.232.521 | 144.190.818 | 1.180.000.443 |
| | *L1-dcache-stores* | 816.139 | 1.268.703 | 3.071.784 | 143.820.937 | 1.110.489.803 |
| | *L1-icache-load-misses* | 13.934 | 15.365 | 18.342 | 30.706 | 72.437 |
| | *LLC-loads* | 11.058 | 12.761 | 341.560 | 142.921.655 | 1.143.728.570 |
| | *LLC-load-misses* | 621 | 385 | 495 | 17.052 | 70.906 |
| 32 | *cache-references* | 27.985 | 30.790 | 292.813 | 143.026.700 | 1.146.089.963 |
| | *cache-misses* | 642 | 813 | 1.736 | 7.307 | 84.559 |
| | *L1-dcache-loads* | 1.415.960 | 3.131.384 | 13.653.149 | 285.996.564 | 2.215.263.648 |
| | *L1-dcache-load-misses* | 32.155 | 196.093 | 1.231.840 | 144.306.424 | 1.175.688.460 |
| | *L1-dcache-stores* | 818.246 | 1.270.506 | 3.072.387 | 143.811.709 | 1.110.472.476 |
| | *L1-icache-load-misses* | 17.551 | 16.555 | 20.330 | 28.439 | 66.697 |
| | *LLC-loads* | 11.469 | 12.741 | 271.112 | 142.920.386 | 1.143.666.361 |
| | *LLC-load-misses* | 254 | 256 | 559 | 5.170 | 72.499 |

Se puede apreciar con los datos comparando entre diferntes 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.