Programación masivamente paralela en procesadores gráficos (GPUs)

Informe - Práctico

Grupo 26

Nombre	**CI**
Agustín Matías Martínez Acuña	5.074.743-0
Natalie Valentina Alaniz Ferreira	5.209.018-4

Índice

1. Introducción

Este es el informe del Práctico 1 del Grupo 26 del curso de Programación Masivamente Paralela en Procesadores Gráficos (GPUs) de la Facultad de Ingeniería de la Universidad de la República. El mismo consta de dos ejercicios.

El primero ejercicio se enfoca en obtener los datos de la jerarquía de memoria del equipo donde se ejecutarán los experimentos. Se obtienen datos pertinentes a cada nivel de caché, seguido por el desarrollo de un programa que mida la velocidad máxima de acceso de cada uno de estos niveles. Luego, se analiza el impacto que tiene el prefetch sobre el rendimiento de un programa. Finalmente, se realizan una serie de pruebas con acceso desalineado a la memoria y como la distribución de datos en dos líneas de caché afecta el desempeño de un programa.

En el segundo ejercicio se busca optimizar la eficiencia del uso de caché para un código dado, mediante el reordenamiento de bucles e implementación de la técnica de "blocking"para distintos tamaños de matriz y bloque. Por último se busca un tamaño de matriz y bloque que genere una alta tasa de caché misses debido a la competencia de los accesos específicos de la caché.

1.1. Especificación del entorno de prueba

- CPU: `12th Gen Intel(R) Core(TM) i7-12700KF   3.60 GHz`
  - Caché L1 Data (P-core): `48 KB (por núcleo)`
  - Caché L1 Data (E-core): `32 KB (por núcleo)`
  - Caché L1 Instrucciones (P-core): `32 KB (por núcleo)`
  - Caché L1 Instrucciones (E-core): `64 KB (por núcleo)`
  - Caché L2 (P-core): `1280 KB (por núcleo)`
  - Caché L2 (E-core): `2 MB (compartido por los E-core)`
  - Caché L3: `25 MB (compartida)`
- RAM: `32,0 GB. Ddr5 4800Mhz`
- OS: `Windows 11 Pro.`
  - Versión: `23H2.`
  - Compilación `22631.3296`
- GPU: `NDVIDIA GeForce RTX 3060 Ti`

1.2. Sobre el procesador

Las generación 12 de procesadores Intel implementa una *Arquitectura Híbrida de Desempeño* [1], esta integra dos tipos de núcleo: los Performance Core (P-Core) y los Efficent Core (E-Core). Describiéndolos *a grosso modo*, los primeros son núcleos de alto desempeño de tamaño más grande y creados para lograr una velocidad básica sin perder la eficiencia. Mientras que los segundos más pequeños que su contraparte y múltiples E-cores se ajustan al espacio físico del P-core. Dada la diferencia tanto en velocidad como tamaño de caché, surge la necesidad de averiguar en qué tipo de núcleo se ejecutarán los programas correspondientes a este práctico.

Para controlar en qué tipo de núcleo (P-Core o E-Core) se ejecutan los programas, es necesario identificar primero el identificador de cada núcleo. Utilizando CoreInfo, se pueden obtener detalles como las especificaciones de cada núcleo y sus identificadores. Como los tamaños de las caché por tipo de núcleo son datos conocidos, es posible deducir a qué tipo pertenece cada núcleo. Una vez identificado el identificador de un E-Core, podemos

asignar explícitamente la afinidad de CPU de nuestro programa para que se ejecute en este tipo de núcleo. Esta asignación se realiza al inicio de la ejecución del programa principal. El proceso para establecer una afinidad de núcleo varía según el sistema operativo, por esta razón se desarrollaron implementaciones tanto para Windows como para Linux.

1.3. Sobre el programa construido

El proyecto está escrito en el lenguaje C y cuenta con un archivo `Makefile` para facilitar la compilación. En el equipo donde se ejecutaron los experimentos se compiló utilizando `MINGW64` y `GCC`, y cuenta con las siguientes versiones de las aplicaciones necesarias para compilarlo y ejecutarlo:

- GCC 12.2.0
- GNU Make 4.4

2. Velocidad máxima de acceso a cada nivel de caché

Para poder medir la velocidad máxima de acceso a memoria es necesario medir el tiempo que toma realizar determinadas operaciones sobre cada uno de los niveles de caché.

Inicialmente debemos asegurarnos que se está trabajando sobre el nivel caché deseado. Para trabajar en L1 alcanza con guardar un arreglo de tamaño del nivel de caché, realizando lecturas en cada una de las entradas para almacenar valores. En estos experimentos en los arreglos se almacenan de forma secuencial los números del 0 a n-1, donde n representa la longitud del arreglo.

Para llenar la caché L2, inicialmente se sigue un procedimiento similar al anterior almacenando en caché un arreglo de tamaño L2. A continuación, se introduce un arreglo de tamaño L1 a la caché, lo que provoca que el arreglo L1 desplace al arreglo L2 dentro de la caché L2.

Se sigue un proceso análogo para el llenado de L3. En este caso no es necesario que el arreglo inicial tenga tamaño L3, ya que nos alcanza con tener un arreglo de cualquier tamaño almacenado en ese nivel de caché.

Finalmente, una vez ubicado el arreglo en el nivel de caché deseado, se procede a realizar una operación de suma de sus elementos. Para cada arreglo, se mide el tiempo empleado en terminar las operaciones en nanosegundos. Los resultados se comparan para analizar el desempeño.

Se realizan tres tipos de recorridas sumando los elementos de los arreglos para comparar las velocidades: [1]

- Secuencial: se recorre el arreglo sumando los elementos del primero al último.
- Aleatoria: la recorrida se realiza mediante una búsqueda pseudo-aleatoria a lo largo del arreglo. Este proceso implica almacenar una permutación de los índices del arreglo y luego seguir el orden establecido por estas entradas para la recorrida.
- Saltos: teniendo en cuenta que el tamaño de línea de la caché es de 64 bytes, se realiza una recorrida leyendo un elemento por linea por recorrida, realizando las suficientes recorridas como para leer todos los elementos.

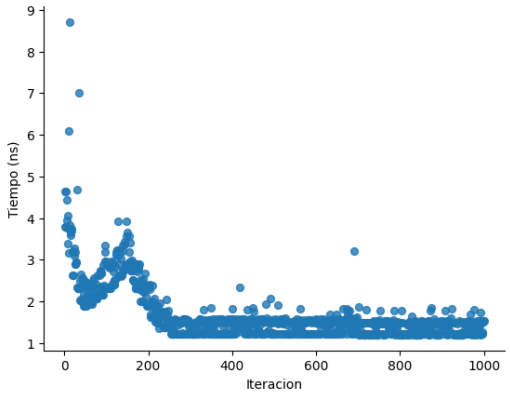
Siguiendo los lineamientos previamente definidos, el experimento consiste en ejecutar los tres distintos tipos de recorrida en cada uno de los tres niveles de caché. Para asegurar la confiabilidad de los resultados y eliminar el ruido estadístico, se repiten las recorridas múltiples veces, limpiando la caché entre recorridas.

[1] Estas implementaciones se encuentran en el archivo `ej1TiempoPromedioAccesoCache.c`

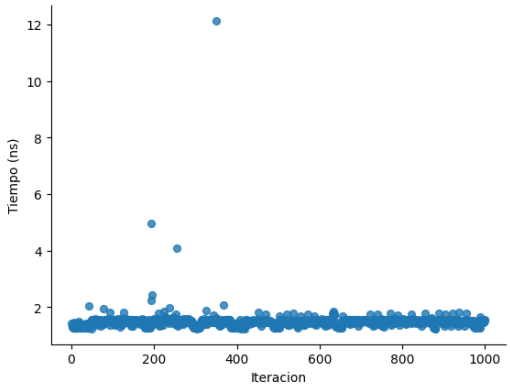
2.1. Análisis de resultados

2.1.1. Overhead al comienzo de la ejecución

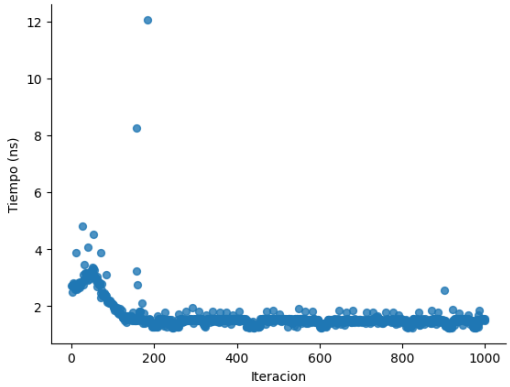
Se detecta que al comienzo De cualquier ejecución del programa la velocidad de lectura es sustancialmente inferior a la del caso general. Se presenta el siguiente testimonio de pruebas de acceso a L1:



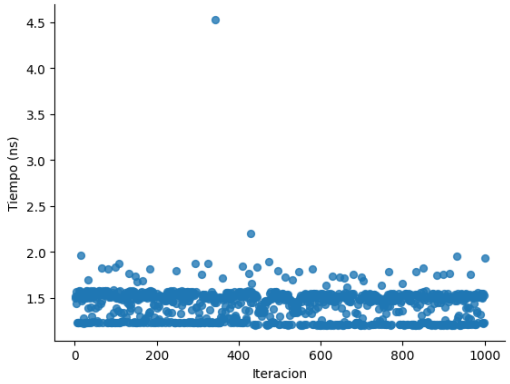
(a) Caso 1: Recorrida secuencial.



(b) Caso 1: Recorrida con saltos



(c) Caso 2: Recorrida con saltos.



(d) Caso 2: Recorrida secuencial.

En el caso 1, primero se ejecuta la recorrida con saltos y luego la recorrida secuencial. Y en el caso 2 se ejecuta primero la secuencial y luego la recorrida con saltos. Se puede observar que las primeras veces que se hace la recorrida (sin importar cual) el tiempo necesario para ejecutarlas es mucho mayor que las recorridas finales.

Si se comienza el programa ejecutando la recorrida de salto fijo y luego se vuelve a ejecutar comenzando con la recorrida secuencial, se ve que en ambos casos para $i \in [1, 100]$ el tiempo promedio de lectura es por lo menos el doble que para iteraciones subsecuentes.

Se podría argumentar que esto es el efecto del prefetch, que una vez detecta los patrones se reduce el "delay". Pero existen dos hechos concretos que contradicen esta afirmación:
1) Estamos en L1.
2) Como se comentó anteriormente, cada iteración es una ejecución de la suma de un arreglo generado para esa iteración, por lo que las ejecuciones son independientes una con otras.

Se asume que este tiempo extra se escapa de lo que podemos controlar en el programa, por lo que no se consideran estas iteraciones para el cálculo del promedio de velocidad de acceso a datos.

2.1.2. caché L1

En este caso como el arreglo a recorrer está en L1, no hay una diferencia notoria entre los distintos tipos de recorrida. Se puede notar que la velocidad máxima de acceso se mantiene constante.

Tipo de Acceso	**Tiempo Promedio de Acceso (ns)**	**Menor tiempo de acceso (ns)**
Secuencial	1.37	1.2
Aleatoria	1.23	1.2
Saltos	1.75	1.25
Saltos (descartando 400 primeras iter.)	1.53	1.23

Cuadro 1: Acceso a memoria caché L1

2.1.3. caché L2

La diferencia entre la velocidad de la caché L1 y L2 es baja. Cabe mencionar que la recorrida secuencial está siendo favorecida por el principio de localidad espacial, ya que al acceder al primer dato del arreglo, la línea que contiene ese dato es movida a L1, por esta razón, por cada miss cometido al buscar el dato en L1 se producen 15 hits.

Por otro lado, la recorrida por saltos se ve favorecida por el prefetch, el acceso a lineas consecutivas de la caché es probable que esté prediciendo las lecturas provocando que las lecturas sean en L1.

Finalmente, el método de recorrida aleatoria es el que más se acerca a hacer la mayor parte de sus lecturas en la caché L2. El primer dato se lee desde L2 y luego la linea correspondiente se traslada a L1. Si alguna de las siguientes lecturas es sobre un dato alojado en una línea previamente trasladada a L1, esta lectura se llevará a cabo más rápidamente. Aunque esto puede parecer un problema, la gran diferencia entre los tamaños de la caché L1 y L2 hace que la probabilidad de acceder a un dato ya presente en L1 no sea alta.

Tipo de Acceso	**Tiempo Promedio de Acceso (ns)**	**Menor tiempo de acceso (ns)**
Secuencial	1.56	1.50
Aleatoria	1.64	1.26
Saltos	1.74	1.35

Cuadro 2: Acceso a memoria caché L2

2.1.4. caché L3

En el caso de la recorrida secuencial, al igual que en L2 la recorrida se ve beneficiada por el principio de localidad espacial, teniendo resultados muy similares. Lo mismo sucede con la recorrida en saltos, esta vez beneficiada por el prefetch. Por último, la recorrida aleatoria, que conserva la mayoría de sus accesos en L3, es la que sufre un cambio evidente en el rendimiento casi duplicando el tiempo de acceso.

Tipo de Acceso	**Tiempo Promedio de Acceso (ns)**	**Menor tiempo de acceso (ns)**
Secuencial	1.55	1.41
Aleatoria	2.5	2.06
Saltos	1.74	1.35

Cuadro 3: Acceso a memoria caché L3

3. Acceso alineado y desalineado a memoria

Dado un objeto de n bytes de datos ubicando en una dirección D, está alineado si `D mod n = 0`. [6]

Para comparar el rendimiento de acceso a memoria entre datos alineados y desalineados, se crean dos arreglos que contienen datos de 4 bytes. Uno de estos arreglos se accede de forma alineada, y el otro se accede de manera desalineada para simular un escenario de almacenamiento desalineado. Esto se logra comenzando las lecturas un byte luego del inicio del arreglo desalineado, por lo que el mismo será un poco más grande para compensar el desplazamiento inicial en la lectura y que ambas recorridas lean la misma cantidad de datos.

En ambos casos se realiza una suma de los valores leídos para asegurar que ninguna recorrida tenga ventaja sobre la otra en cuanto a la operación.[2] Los tiempos de acceso a ambos arreglos se miden y comparan para entender el impacto de la alineación en el rendimiento.

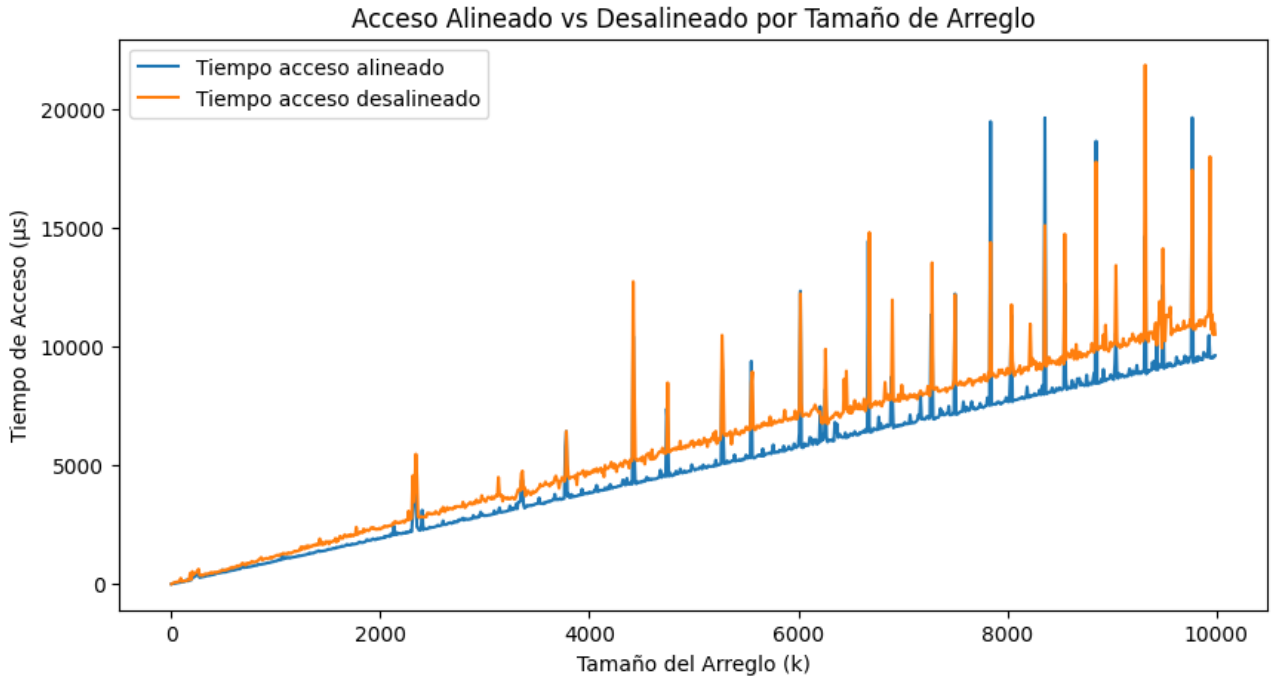


Figura 2: Acceso alineado vs desalineado a memoria.

Se puede notar una diferencia entre los tiempos correspondientes al acceso de memoria alineado y desalineado, donde en la mayoría de los casos el acceso alineado es más rápido que el acceso desalineado esto sucede porque cuando los datos están alineados el sistema puede cargarlos en los registros en una única operación. Acceder a datos desalineados podría requerir múltiples accesos a memoria para ensamblar el dato completo. Incluso el uso de los buses puede ser menos eficiente con accesos desalineados, se podría requerir que el bus realice dos transferencias separadas para mover los datos.

[2] Esta implementación se puede encontrar en el archivo `ej1AccesoDesalineado.c`

4. Prefetch

Los procesadores modernos son capaces de detectar patrones regulares en el acceso a los datos [2]. Esto les permite cargar una línea de caché antes de que sea requerida por una instrucción. También pueden hacerlo los compiladores y los programadores.

En esta sección se realizan experimentos con el prefetch tanto de procesador/compilador (al que llamaremos *implícito*) como el de a nivel de programador (*explícito*) y se analizan sus resultados.

4.1. Prefetch Implícito

Se plantea realizar dos tipos de recorrida de un arreglo hospedado en el nivel 3 de caché:

- **Recorrida con saltos de tamaño 64 bytes:** de esta manera si se accede a un elemento, el siguiente elemento a acceder siempre estará en otra línea de caché. De no ser por el prefetch, cada acceso se daría en L3.
- **Recorrida Aleatoria:** Se realiza una permutación de un arreglo secuencial, se traslada a la caché L3 y se recorre a través de la permutación dada accediendo al siguiente elemento a través de `a[a[i]]`. Dado que el patrón de recorrida es aleatorio, el compilador y el procesador no es capaz de traer a caché la línea correspondiente a ese elemento. Al ser la recorrida aleatoria, no se puede asegurar que todos los accesos se den en L3, por ejemplo, si `a[i]` y `a[a[i]]` se encuentran en la misma línea, entonces se garantiza un hit al acceder al último. Al igual que se mencionó en secciones anteriores, la prueba es prudente ya que dado un tamaño sustantivo de arreglo, muy probablemente la mayoría de los accesos se den en L3, y el prefetch no sería capaz de realizarse.

Las recorridas y sus resultados pueden encontrarse en la sección 2.1.4. Se observa un aumento sustantivo del tiempo promedio de acceso de la recorrida aleatoria en relación a la otra, esto indica que se pudo realizar el prefetch de datos exitosamente en la recorrida por saltos. Consecuentemente, el primer acceso se realiza en L3, y gracias al prefetch, el resto de ellos muy probablemente se estén realizando en L1,

Sin embargo, se esperaba que el tiempo de acceso promedio sea aún mayor dado que, según [2] la tendencia es que cada nivel de caché es 3-10X más lento que el nivel superior.

4.2. Prefetch Explícito

Varios lenguajes de programación brindan la posibilidad de realizar prefetch de manera explícita en el código, esto tiene sentido para aquellos algoritmos con patrones lo suficientemente complejos como para que el prefetch automático no los detecte, la búsqueda binaria [3] es un ejemplo de esto.

Se realizan dos implementaciones del algoritmo de búsqueda binaria idénticas línea a línea, con la única diferencia de que en una de ellas se realiza el prefetch tanto del elemento a la mitad derecha (`a[(medio + 1 + alto)/2`) y la mitad izquierda (`a[(bajo + medio - 1)/2]`).[3]

La directiva utilizada es `__builtin_prefetch (dato, 0, 1)` indicando que se debe realizar la instrucción de máquina `FETCH` de la línea de caché donde se encuentra `dato`. En el caso de la búsqueda binaria, se trae a caché las dos posibles opciones de salto, es claro que una de ellas se va a descartar sin ser utilizada.

Los resultados muestran que el tiempo de ejecución con prefetch explícito es sustantivamente menor que su contraparte en todas las ejecuciones realizadas.

Es necesario agregar que, si bien el aumento del rendimiento es sustantivo, hay que tener en cuenta que al realizar el prefetch de ambos posibles elementos (4 bytes) se está cargando en caché dos líneas de 64 bytes (en

[3] Estas implementaciones se encuentran en el archivo `ej1PrefetchExplicito.c`

lugar de una sin prefetch explícito), y consecuentemente, se está desaprovechando la totalidad de una de ellas. El ejemplo presentado es de laboratorio, su ejecución aislada muestra una aumento sustantivo del rendimiento, pero probablemente en un entrono real, presionar a la caché de esta forma sea contraproducente.

5. Distintas técnicas para la multiplicación matricial

5.1. Reordenamiento de bucles

Esta sección se enfoca en optimizar el rendimiento del programa reordenando los bucles con el objetivo de maximizar los hits en caché, utilizandola de forma más eficiente. Los arreglos A y B almacenan matrices por filas, el código se encarga de multiplicar las matrices y almacenar los resultados en una nueva matriz C. Al observar los arreglos como si fueran matrices, el recorrido por las entradas de A se realiza por filas, mientras que para B el recorrido es por columnas. Esto implica, que al acceder a una fila completa en B para realizar las operaciones, sólo se utiliza un dato de la fila en cada paso, resultando en un uso ineficiente de la caché.

El siguiente código muestra como se realiza la multiplicación de matrices actualmente:

```
VALT sum;
for (int row = 0; row < m; row++)
    for (int col = 0; col < n; col++) {
        sum = 0;
        for (int it = 0; it < p; it++)
            sum += A[row * p + it] * B[it * n + col];
        C[row * n + col] = sum;
    }
```

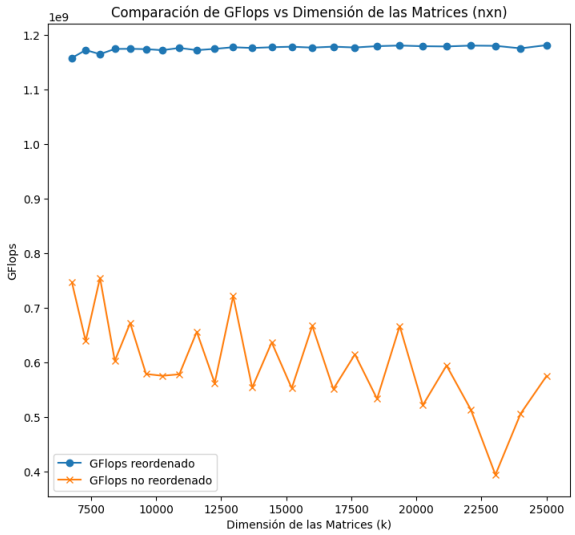
Figura 3: Multiplicación sin reordenar

Para optimizar el rendimiento del programa y hacer un uso más eficiente de la caché durante la multiplicación de matrices, se propone un reordenamiento de los bucles. La propuesta busca mantener el patrón de recorrida por filas en la matriz A, mientras que la matriz B se modifica para aprovechar los datos cargados en la caché, realizando todas las operaciones necesarias en la fila actual antes de avanzar a la siguiente. Además, se introduce una inicialización de la matriz C, consecuencia del reordenamiento, se evita el uso de condicionales para asignar los primeros valores de cada entrada de C, lo cual podría ser menos eficiente debido al uso de predicción de saltos. Esta estrategia busca una comparación más justa entre los distintos reordenamientos.

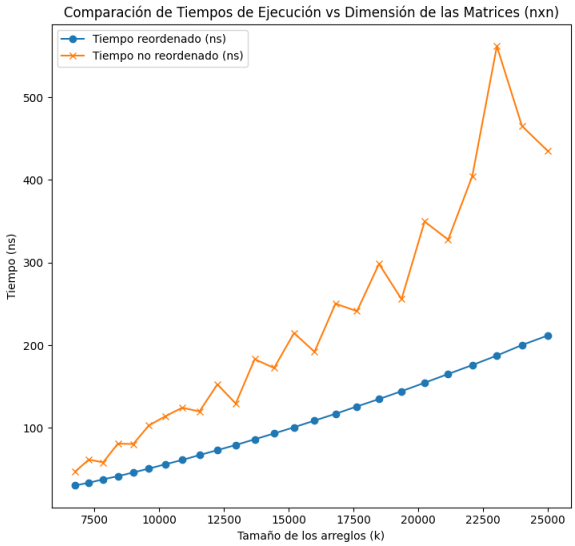
```
    for (int i = 0; i < m * n; i++) {
        C[i] = 0;
    }

    for (int row = 0; row < m; row++) {
        for (int it = 0; it < p; it++) {
            for (int col = 0; col < n; col++) {
                C[row * n + col] += A[row * p + it] * B[it * n + col];
            }
        }
    }
```

Figura 4: Multiplicación reordenada

En la solución propuesta se aprovecha el principio de localidad espacial, primero con la inicialización de la matriz C almacenando sus valores iniciales de manera secuencial, y luego con la modificación sobre la recorrida sobre B para aprovechar las líneas de caché en los niveles más altos. Este enfoque aumenta la cantidad de hits a los niveles más altos de caché.



(a) GFlops reordenado vs GFlops no reordenado



(b) Tiempo de ejecución reordenado vs. Tiempo de ejecución no reordenado

Las pruebas realizadas, buscan ocupar por completo la caché, aumentando gradualmente el tamaño de las matrices hasta llegar a un máximo de 5000×5000 elementos. Los resultados muestran una diferencia marcada entre la versión reordenada y no reordenada. La versión reordenada muestra un tiempo de ejecución menor y un crecimiento casi lineal a medida que el tamaño de las matrices aumenta. Por otro lado, el tiempo requerido por el código no reordenado crece con más rapidez con respecto al tamaño de las matrices, además su crecimiento es oscilante, lo cual sugiere una menor eficiencia en la utilización de la caché.

Por otro lado, el número de operaciones por segundo (medido en GFlops) que logra el código reordenado es sustancialmente más alto y se mantiene estable a medida que aumentan las dimensiones de las matrices utilizadas. Esto podría indicar que el código reordenado podría estar operando cerca de su capacidad máxima, optimizando el uso de caché. En cambio, la versión no reordenada muestra un rendimiento mucho menor e irregular, lo cual implica una utilización ineficiente de los recursos del procesador teniendo un menor rendimiento global.

5.2. Aplicando la técnica de Blocking

En esta parte se utiliza la técnica de Blocking para el producto de matrices, el blocking se basa en acceder a una porción de los datos que entre en caché repetidamente, antes de pasar a la siguiente porción [2]. Esta técnica explota de mejor manera el principio de localidad espacial.

Se utiliza la implementación del blocking utilizada en Üsing Blocking to Increase Temporal Locality"[4].

Para este análisis se trabaja con datos (int) de tamaño 4 bytes y matrices cuadradas. La caché L1 dispone de un espacio total de 49152 bytes. Por lo que la cantidad de enteros que entran en la caché L1 en el caso de un P-Core es $\frac{49152}{sizeof(int)} = \frac{49152}{4} = 12288$

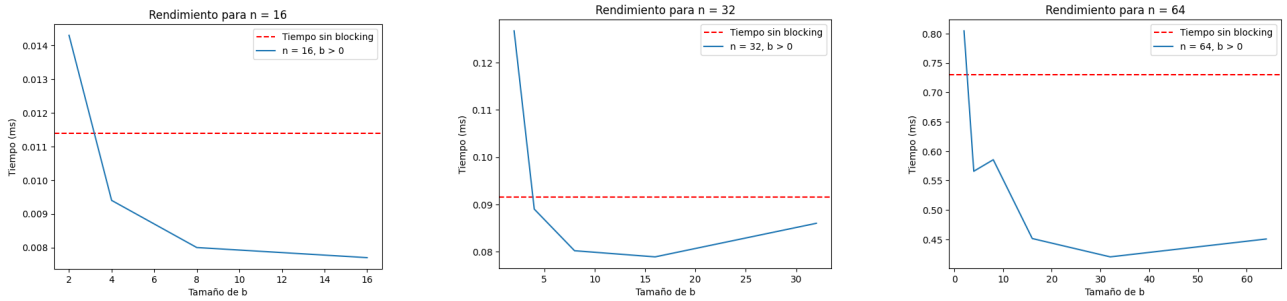
Esto significa que entran 12288 enteros de 4b en la Caché L1, si queremos agruparlos en una matriz A, esta deberá ser de $dim(A) = \lfloor\sqrt{12288}\rfloor = 110$. Esta es la matriz más grande posible que entra en el primer nivel de caché.

Aplicando este razonamiento para una ejecución en E-Core tenemos que Tamaño de $L_1 = 32{,}768 \rightarrow dim(A) = 90$.

Los datos experimentales muestran que en un entorno de las dimensiones calculadas anteriormente (90) se encuentra el tamaño de bloque que minimiza el tiempo de ejecución. (ver Figuras 6, 7, 8).

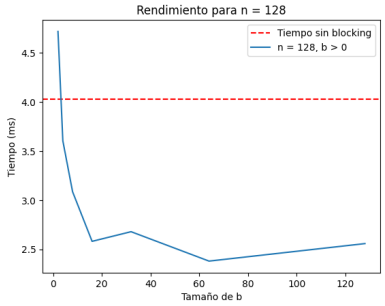
En dichas figuras se muestra una línea punteada roja que indica la velocidad de ejecución para el código presentado en la Figura 4. Se observa que en todos los casos existe un tamaño de bloque que mejora la velocidad frente al no uso de blocking.

Esto coincide con la noción que presentan los autores de la implementación, dado que el programa está estructurado de manera que carga un fragmento en el caché L1, realiza todas las lecturas y escrituras que necesita en ese fragmento, luego lo descarta, carga el siguiente fragmento, y así sucesivamente.

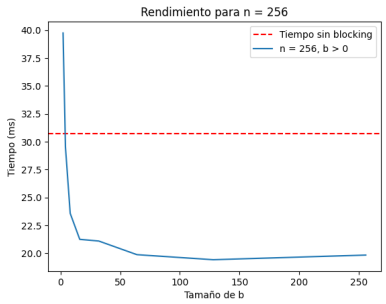


(a) blocking_n_16 (b) blocking_n_32 (c) blocking_n_64

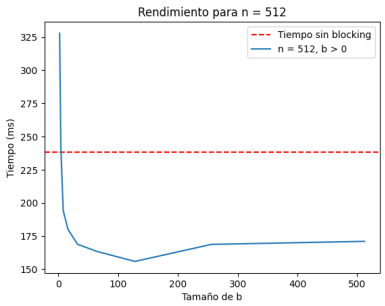
Figura 6: Imágenes con blocking 16, 32 y 64



(a) blocking_n_128

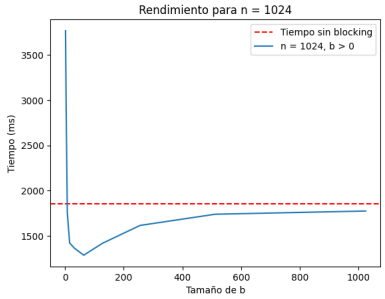


(b) blocking_n_256

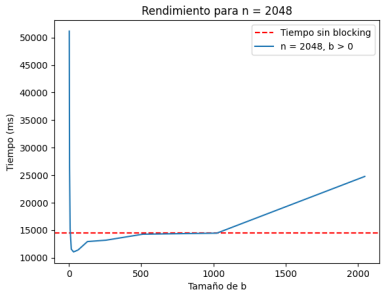


(c) blocking_n_512

Figura 7: Imágenes con blocking 128, 256 y 512



(a) blocking_n_1024



(b) blocking_n_2048

Figura 8: Imágenes con blocking 1024 y 2048

5.3. Produciendo Caché Miss debido a la competencia de los accesos por un set específico de la caché

En esta parte se busca encontrar un tamaño de matriz y bloque tal que que se minimicen los caché hit, con la condición que estos miss se generen a causa de competitivdad por el set, para esto recordamos los siguientes datos de la caché:

- L1 Size: 32768 bytes.
- L1 Asociatividad: 8 lineas por set.
- Tamaño de Línea: 64 bytes por línea.

Considerando las unidades anteriores, contamos con $\frac{32768}{64} = 512$ lineas y $\frac{512}{8} = 64$ sets.

De este cálculo se desprende una propuesta para elegir bloques, utilizar bloques rectangulares sobre una matriz de tamaño $n \times n$ con $n \equiv 0 \pmod{1024}$ (porque cada 1024 enteros se repite el set, esto se justifica por las cuentas realizadas anteriormente y que se utilizan integers de 4 bytes). Con esta configuración, tenemos que fijando una columna y, `a[i][y]` es mapeado al mismo set para todo i. Esto nos permite elegir bloques de múltiplos de 1024 filas y una cantidad arbitraria de columnas (siempre y cuando sea múltiplo del tamaño de la matriz).

Para lograr esto fue necesario adaptar el código de la multiplicación de matrices utilizando blocking, esta implementación puede encontrarse en el archivo (`ej2CompetenciaSetCache.c`).

Para matrices de tamaño mediano (1024×1024 y 2048×2048) se observa consistentemente una bajada de rendimiento de aproximadamente 50 % frente a la recorrida con bloque de tamaño 128.

Se presenta como ejemplo la siguiente ejecución:

N	Bloque	Tiempo (s)	Gflops
2048	1024×512	22.288923	0.770781
2048	128×128	14.278936	1.203162

Cuadro 4: Resultados de rendimiento

Referencias

[1] Cómo Funcionan los Procesadores Intel® Core™, <intel.la/content/www/xl/es/gaming/resources/how-hybrid-design-works.html>.

[2] Letra del Práctico 1 | GPGPU 2024 | FING | Udelar <eva.fing.edu.uy/pluginfile.php/285821/mod_resource/content/7/Presenta_pr_1.pdf>. eva.fing.edu.uy/pluginfile.php/285821/mod_resource/content/7/Presenta_pr_1.pdf

[3] Búsqueda Binaria <es.wikipedia.org/wiki/BÃžsqueda_binaria>.

[4] Blocking <csapp.cs.cmu.edu/public/waside/waside-blocking.pdf>.

[5] Intel® 64 and IA-32 Architectures Software Developers Manual <intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>. Página 3009

[6] Arquitectura del procesador, página 19. <fdi.ucm.es/profesor/jjruz/ec-is/temas/Tema%202-Arquitectura%20del%20procesador.pdf>.