Computación de propósito general de unidades de procesamiento gráfico

Práctico 1

Impacto del acceso a los datos en CPU

Integrantes

Nombre	CI	Correo
Renzo Gambone	1.234.567-8	renzo.gambone@fing.edu.uy
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy

Índice

1. Introducción

En este informe se detallan los resultados obtenidos durante la evaluación experimental de los ejercicios correspondientes al **Práctico 1**. Dichos resultados fueron analizados, adjuntando las observaciones y conjeturas extraídas de dicho análisis. A continuación, se destacan ciertos aspectos a tener en cuenta del análisis y los entregables:

1. La evaluación experimental se realizó empleando el lenguaje **C** y ejecutando el código compilado en un **CPU AMD Ryzen 5 3600**, cuyas especificaciones pueden ser consultadas en el siguiente enlace. De las mismas se destaca lo siguiente:
   - **Cores:** 6 (12 threads)
   - **CPU max MHz:** 3600 MHz
   - **L1i Cache:** 192 KB (6 x 32 KB)
   - **L1d Cache:** 192 KB (6 x 32 KB)
   - **L2 Cache:** 3 MB (6 x 512 KB)
   - **L3 Cache:** 32 MB (2 x 16 MB)
   - **Cache Latency:** 4 (L1); 12 (L2); 40 (L3)
2. Los datos manipulados fueron vectores y matrices compuestos por números decimales representados en punto flotante de precisión simple (*float* de **C**). Dicho tipo de dato cuenta con un tamaño de almacenamiento de **4 bytes** por unidad. Dado el procesador utilizado, la **caché L1** puede almacenar hasta **49152 celdas** de un vector o una matriz. Es importante tener en cuenta que, para una matriz, el lenguaje **C** almacena dichas celdas por fila.
3. Los tiempos registrados al realizar distintas ejecuciones de un mismo script no fueron siempre constantes, por lo que se decidió realizar **10 ejecuciones** para cada caso de prueba. De esta forma, se trabajó teniendo en cuenta la media de los tiempos, así como los mínimos y máximos obtenidos.
4. Se decidió modificar ligeramente las plantillas otorgadas para los scripts, con el fin de automatizar las múltiples ejecuciones mencionadas en el punto anterior. Es importante destacar que las funcionalidades originales fueron mantenidas, por lo que dichos cambios solo agregan funcionalidad.

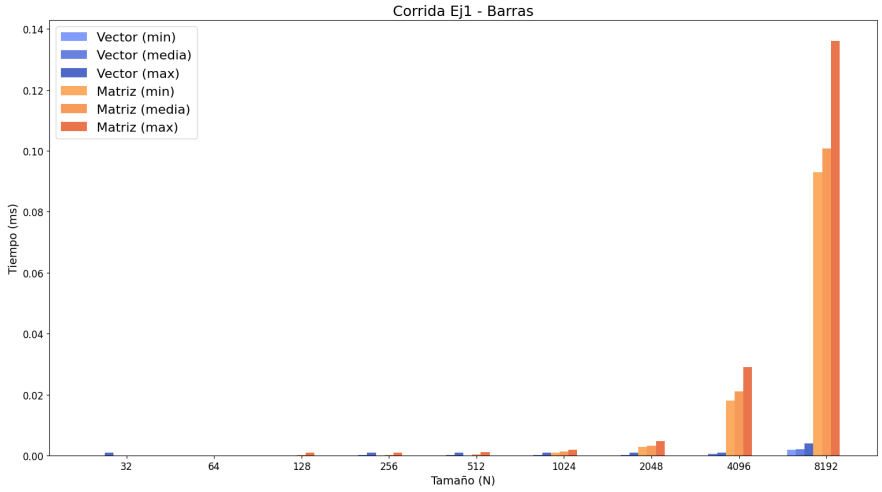
2. Desarrollo

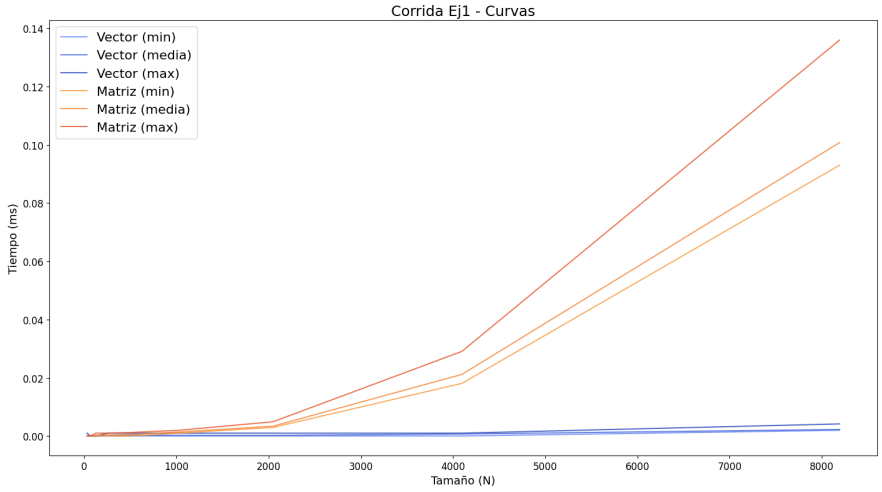
2.1. Ejercicio 1 - Recorridas Lineales

En esta sección se evalúan los resultados obtenidos durante la ejecución de las funciones correspondientes al **ejercicio 1**. Las mismas son:

- **Suma de elementos de vector** *(suma_vector)*: Función que recorre linealmente un vector de tamaño N y suma sus entradas.
- **Suma de elementos de diagonal de matriz** *(suma_matriz)*: Función que recorre linealmente la diagonal de una matriz cuadrada de tamaño N y suma sus entradas.

Con el objetivo de contrastar los tiempos de ejecución entre ambas funciones, se varió el parámetro N, tomando los valores $N = 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192$. A su vez, se realizaron **10 ejecuciones** para cada función y cada valor, midiendo los tiempos y tomando **3 valores** representativos: *mínimo*, *máximo* y *media*. De esta forma, se generaron las siguientes gráficas:





A partir de las gráficas es posible realizar diversas observaciones:

- Las ejecuciones de *suma_vector* fueron más eficientes que las de *suma_matriz*, con una diferencia de más de ***0.08 ms*** entre el mejor tiempo de *suma_matriz* y el peor tiempo de *suma_vector*.
- Las medias de los tiempos de ejecución para ambos casos se encontraron más cerca de los tiempos mínimos que máximos, lo cual podría indicar que los tiempos máximos son valores atípicos en comparación a los mínimos.

Teniendo en cuenta los tiempos, se concluye que acceder y operar con los elementos de un vector unidimensional (recorriéndolo de principio a fin) es más eficiente que operar con la diagonal de una matriz con la misma cantidad de elementos. Si bien virtualmente la diagonal de una matriz cuadrada de N elementos es un vector con dicha longitud, en memoria ambas estructuras son almacenadas de forma distinta.

Particularmente, esto depende de dos factores: los tamaños de la memoria caché, y como **C** carga los datos en la memoria principal. Recorrer un vector en orden garantiza que sólo habrá un *miss* en la caché cuando ya se haya terminado de operar con los datos en la misma y sea necesario cargar nueva memoria (los elementos siguientes, en caso de que falten). En el caso de la matriz, toda una sección de la fila de la matriz es cargada a la memoria para meramente acceder al valor en la diagonal y que luego ocurra un caché *miss*. Para tamaños de vector y matriz suficientemente pequeños, esto no representa un problema ya que pueden alojarse completamente en caché. Sin embargo, a medida que dicho tamaño aumenta, la probabilidad de un *miss* para la solución de la matriz es exponencialmente mayor que la probabilidad de un *miss* para el vector. Esto causa que si bien las iteraciones y operaciones en ambas funciones son del mismo orden computacional (N), se pierda una gran cantidad de tiempo de computo en cargar datos que no son necesarios (los elementos de la matriz que no forman parte de la diagonal).

En este ejemplo en particular, se puede observar en las gráficas que los tiempos de ejecución entre ambas funciones empiezan a diferir a partir de $N = 1024$, y el crecimiento en tiempo para *suma_vector* es sustancialmente más lento que para *suma_matriz*. En el CPU empleado, el tamaño de la **caché L1d** es de **192 KiB**, la cual puede alojar hasta $192KiB/4B = 49152$ elementos (un número en punto flotante de precisión simple ocupa **4B**). Esto supera el espacio ocupado por los vectores, sea cual sea el N elegido. En cambio para las matrices, ya con $N = 256$ no es posible alojar la totalidad de la matriz en la misma. A medida que N crece, cada vez es más costoso almacenar la matriz, lo cual es coherente con los tiempos observados.

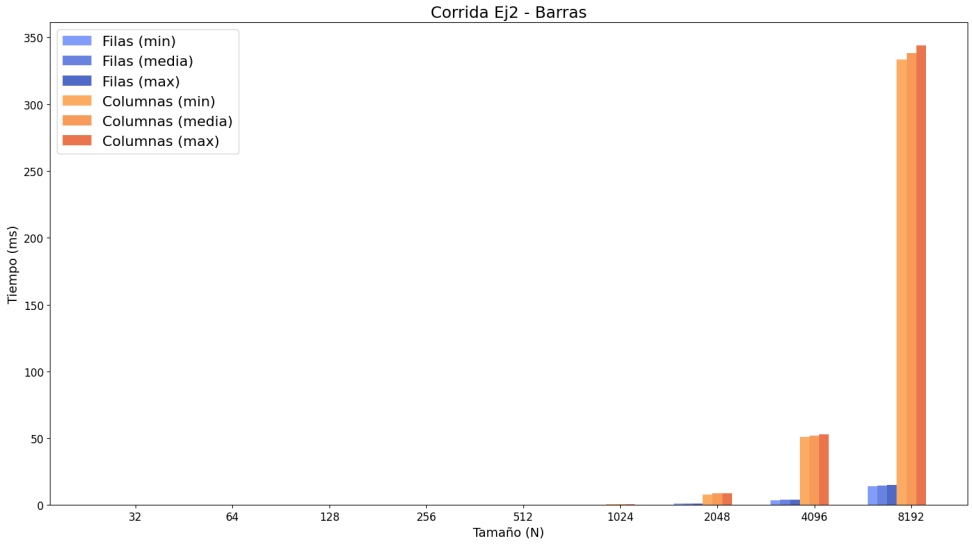
Tomando en cuenta lo anterior, este fenómeno es un buen candidato a explicar las diferencias en cuanto a la performance de ambas recorridas lineales.

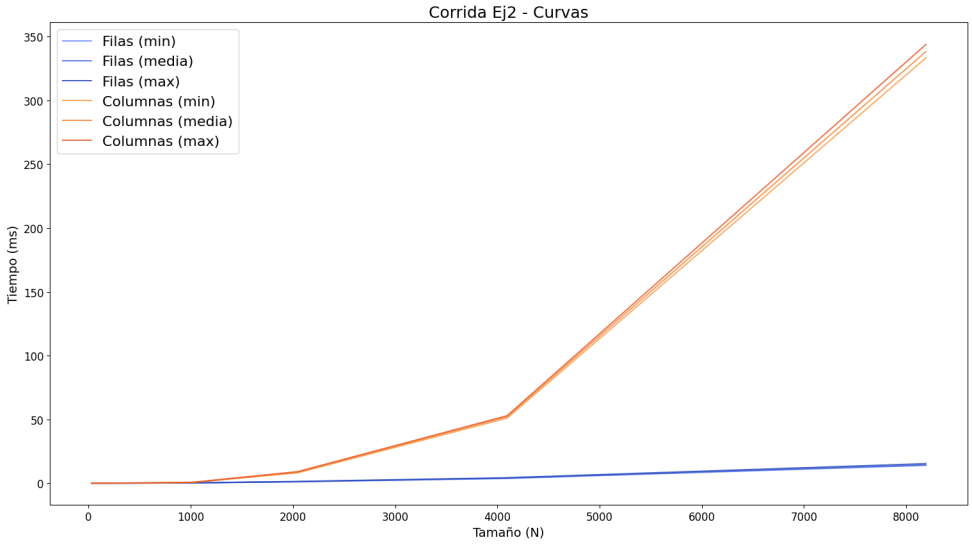
2.2. Ejercicio 2 - Recorridas en Matrices

En esta sección se evalúan los resultados obtenidos durante la ejecución de las funciones correspondientes al **ejercicio 2**. Las mismas son:

- **Suma de elementos de matriz por filas** *(suma_porfilas)*: Función que recorre por filas una matriz cuadrada de tamaño N y suma sus entradas.
- **Suma de elementos de matriz por columnas** *(suma_porcolumnas)*: Función que recorre por columnas una matriz cuadrada de tamaño N y suma sus entradas.

Con el objetivo de contrastar los tiempos de ejecución entre ambas funciones, se varió el parámetro N, tomando los valores $N = 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192$. A su vez, se realizaron **10 ejecuciones** para cada función y cada valor, midiendo los tiempos y tomando **3 valores** representativos: *mínimo*, *máximo* y *media*. De esta forma, se generaron las siguientes gráficas:





A partir de las gráficas es posible realizar diversas observaciones:

- Las ejecuciones de *suma_porfilas* fueron mucho más eficientes que las de *suma_porcolumnas*, con una diferencia de más de ***320 ms*** entre el mejor tiempo de *suma_porfilas* y el peor tiempo de *suma_porcolumnas*.
- Las diferencias entre mínimos y máximos, si bien fueron de varios milisegundos en el peor de los casos, fueron, en comparación con las diferencias del **ejercicio 1**, mucho más acotadas teniendo en cuenta los tiempos manejados. Esto implicaría que no hay valores atípicos ni grandes oscilaciones entre distintas ejecuciones.
- En relación al **ejercicio 1**, los tiempos de ejecución en general fueron mucho mayores, pero también lo fue la diferencia entre el algoritmo más eficiente y el menos eficiente. Esto condice a las características de cada ejercicio: en este caso se opera con la totalidad de elementos de la matriz (N^2 elementos) mientras que en el ejercicio anterior sólo se opera con N elementos.

Se observa en los resultados un patrón similar a los del ejercicio anterior. Ambos algoritmos son de orden computacional N^2, mientras que por cada elemento de la matriz se realiza la misma operación de suma, sin embargo se observan grandes diferencias en los tiempos registrados. Tomando en cuenta los tiempos, se concluye que recorrer una matriz por filas es mucho más eficiente que hacerlo por columnas. Esto es coherente con cómo **C** almacena las matrices en memoria (por filas). De hecho, en este ejercicio recorrer la matriz por filas presenta la misma característica que recorrer el vector en el **ejercicio 1**, los datos cargados a memoria y copiados a la caché son todos utilizados antes de volver a cargar datos. De forma opuesta, recorrer la matriz por columna no sólo carga datos que no serán prontamente utilizados, sino que esos datos no utilizados tendrán que volver a ser cargados en el resto de la ejecución.

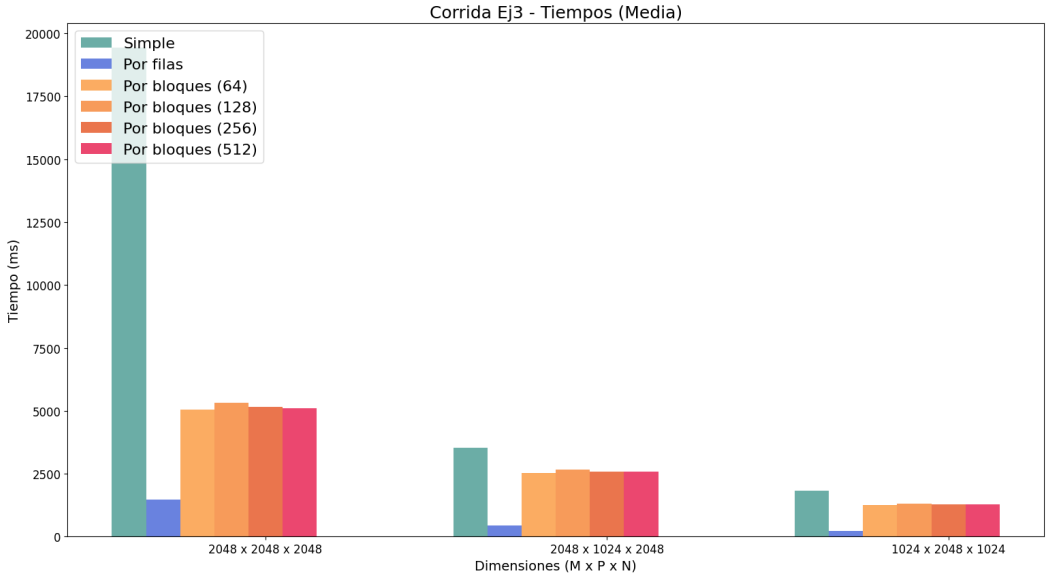
Un dato interesante a acotar es que mientras que *C* almacena los datos de una matriz según sus filas, *Fortran* toma un enfoque opuesto, almacenando los datos de una matriz a partir de sus columnas. Es por ende que se puede conjeturar que estos dos algoritmos mostrarían performance opuesta si se ejecutaran en *Fortran*.

2.3. Ejercicio 3 - Multiplicación en Matrices

En esta sección se evalúan los resultados obtenidos al ejecutar distintos algoritmos de multiplicación de matrices, multiplicando $A_{m\times p} \times B_{p\times n}$. Los algoritmos implementados y empleados en la experimentación son:

- **Multiplicación simple**
  Algoritmo tradicional de multiplicación de matrices, en el cual se recorre A por sus filas y B por sus columnas, computando: $c_{ij} = \sum_{k=1}^{p} a_{ik} \times b_{kj}$
- **Multiplicación por filas**
  Alteración del algoritmo convencional de multiplicación de matrices, en el cual tanto la matriz A como la matriz B se recorren por filas.
- **Multiplicación por bloques**
  En este enfoque *divide and conquer*, la matriz es subdividida en *tiles* o bloques de tamaño $nb \times nb$, en donde se toma cada par posible de bloques entre A y B y se multiplican, acumulando el resultado parcial en el lugar correspondiente de la matriz C. Se destaca que cada multiplicación dentro del bloque sigue el algoritmo de *multiplicación por filas* descrito anteriormente.

Con el objetivo de contrastar los tiempos de ejecución entre las tres funciones, se realizaron **10 ejecuciones** para cada función y cada configuración, midiendo los tiempos y tomando la *media* de los mismos. Las configuraciones implementadas fueron las siguientes: $m = p = n = 2048$; $m = n = 1024$, $p = 2048$; $p = 1024$, $m = n = 2048$. Para el algoritmo de multiplicación con bloques, se experimentó variando el tamaño del bloque con $nb = 64, 128, 256, 512$. De esta forma, se generó la siguiente gráfica:



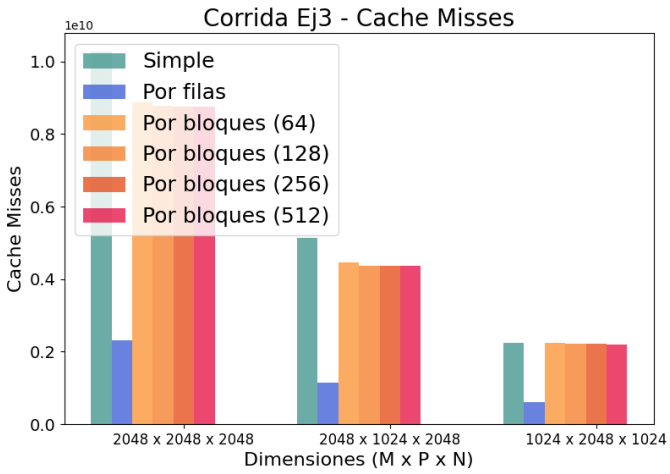
A partir de estos datos se realizan las siguientes observaciones:

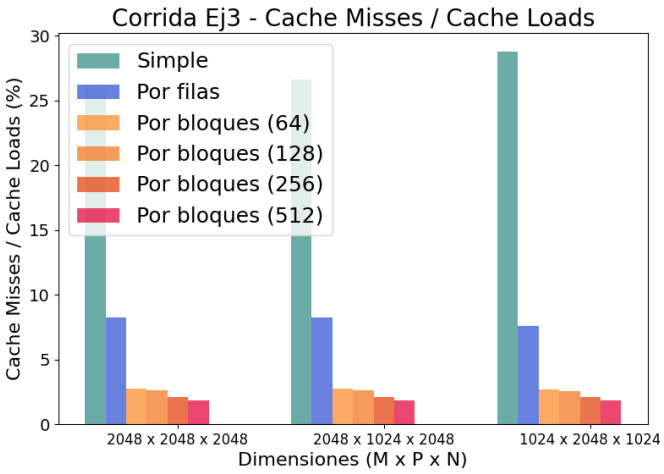
- El algoritmo de multiplicación por filas obtuvo drásticamente mejores tiempos que el resto.
- El algoritmo de multiplicación por bloques tuvo resultados similares independientemente del tamaño de bloque elegido.
- El algoritmo de multiplicación simple tuvo una performance similar (aunque peor) al de multiplicación por bloques cuando $p = 1024$; $m = n = 2048$ y cuando $p = 2048$; $m = n = 1024$. No obstante, esto cambió drásticamente cuando $m = n = p = 2048$.

En base a estas observaciones, parece pertinente establecer que el algoritmo más eficiente es el de multiplicación por filas, lo cual es coherente con como **C** almacena las matrices en memoria. No obstante, este conclusión fue considerada prematura, por lo cual se realizó un análisis en mayor profundidad, capturando distintas métricas para cada algoritmo y configuración. Para lograr esto se empleó la herramienta perf. Entre las métricas capturadas, se destacan las siguientes:

- **L1d cache misses:** Cantidad de *misses* que ocurrieron al leer la caché (y en las que se tuvo que invalidar la misma para traer nuevos datos).
- **L1d cache misses/L1d cache loads (%):** Porcentaje de *misses* que hubo de la cantidad de cache *loads*.
- **Ciclos:** Ciclos de reloj consumidos durante la ejecución en el procesador.
- **Instrucciones:** Cantidad de instrucciones ejecutadas. Las mismas están fuertemente vinculadas a los ciclos de reloj, sin embargo, según que tan bien se logre aprovechar el pipeline y el branch predictor la cantidad de instrucciones ejecutadas por ciclo puede variar entre 1 y 5 para el procesador con el que se experimentó.

Teniendo en cuenta dichas métricas, se realizaron las siguientes gráficas para comparar el acceso a caché de cada algoritmo:



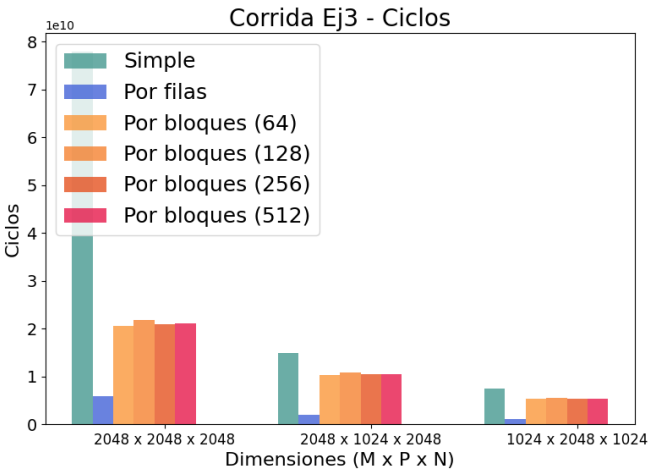


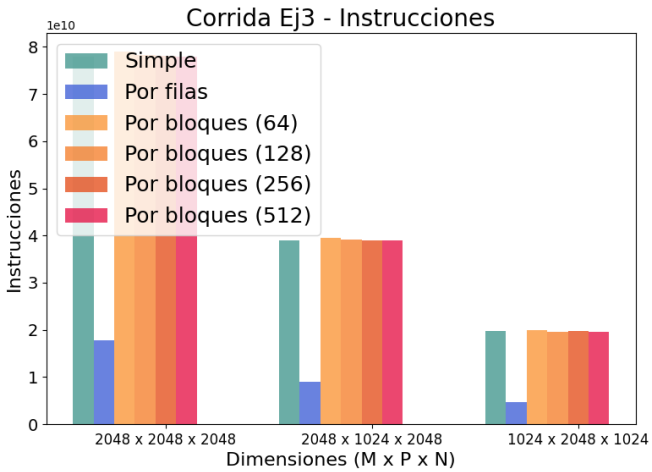
Se puede observar que:

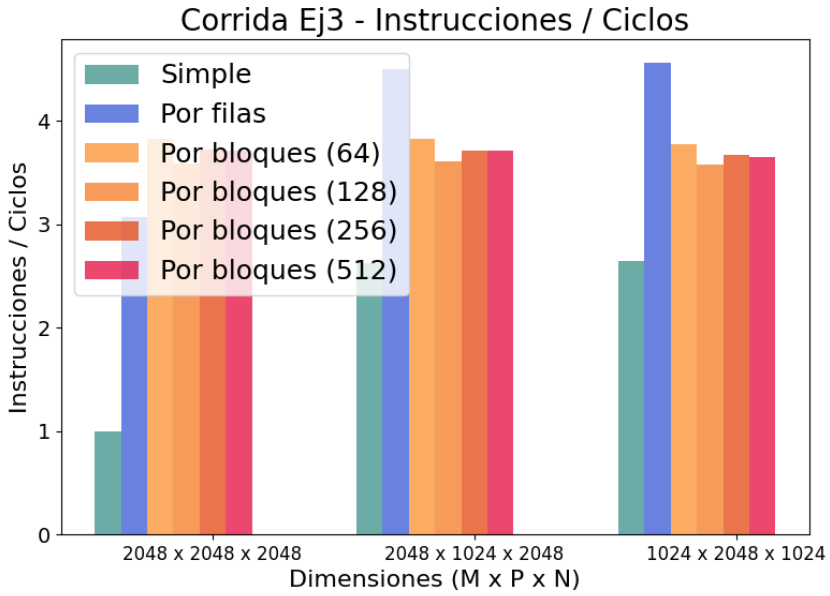
- La multiplicación por filas tuvo la menor cantidad total de *cache misses* (como era de esperarse, recordando que **C** almacena matrices por filas).
- La multiplicación por bloques tuvo un % drásticamente menor de *cache misses* respecto a *cache loads* (y el mismo disminuye al aumentar el largo del bloque). Sin embargo, el algoritmo realizó una cantidad extremadamente alta de accesos *cache loads* respecto al de multiplicación por filas; requiriendo de más accesos a memoria caché para terminar de ejecutar.
- La multiplicación simple tuvo la mayor cantidad de *cache misses* y el % más alto de los mismos frente a la cantidad total de *cache loads*.

Habiendo comparado estos datos, es posible extender la conclusión obtenida en base a la primer gráfica: el algoritmo por filas fue más eficiente en esta situación en concreto pero cuenta con una cantidad superior de *caché misses* en comparación al algoritmo por bloques. Pareciera que la demora asociada al algoritmo por bloques tiene que ver con que la cantidad de accesos a caché fue ampliamente superior, pero es importante destacar que dichos accesos tuvieron un excelente ratio de *caché hit*. Este dato podría implicar que bajo ciertas condiciones, este algoritmo podría ser más eficiente que el algoritmo por filas. Respecto al algoritmo simple, no se obtuvieron datos que contradigan lo anteriormente observado.

Finalmente, se realizaron las siguientes gráficas para comparar el uso óptimo del *pipeline* y comprobar si esto tuvo influencia en los tiempos de ejecución:







De estas gráficas se observa que:

- La relación entre instrucciones y ciclos para el algoritmo simple es de 1 para el caso $m = n = p = 2048$, lo cual implicaría que el pipeline no está siendo aprovechado.
- La relación entre instrucciones y ciclos para los algoritmos de filas y bloques es mucho mejor, aprovechando más el pipeline.
- El algoritmo por bloques ejecuta tantas instrucciones como el simple, pero tiene menor cantidad de ciclos por lo que el pipeline es más aprovechado.

Juntando la información extraída de todas estas métricas, se concluye que el algoritmo por filas es el más eficiente ya que ejecuta menos instrucciones, aprovecha mejor el pipeline, realiza menos accesos a caché y aprovecha mejor la disposición de la caché (por ser por filas). Es importante observar que el algoritmo por bloques aprovecha muy bien el pipeline pero cuenta con más instrucciones y más accesos a caché, lo cual lo enlentece. En un entorno de programación paralela, estas contras podrían mitigarse, llegando a un resultado incluso mejor que en filas.

3. Conclusiones

Como conclusión principal, se pudo observar el impacto en la performance de algoritmos del mismo orden, según como se aprovecha la memoria que se carga a la caché.

En los **ejercicios 1 y 2** se observó y especuló sobre la influencia de como **C** almacena estructuras en memoria, y el impacto directo que esto tiene en la performance de un algoritmo simple.

En el **ejercicio 3** se realizó una experimentación de mayor profundidad, aprovechando que los algoritmos a implementar eran de mayor complejidad a los de ejercicios anteriores. También se confirmaron las especulaciones conjeturadas, a partir de las métricas obtenidas mediante la herramienta perf.

Si bien se podría afirmar que al trabajar con el lenguaje **C** el algoritmo de multiplicación recorriendo por filas es superior, esta sería una afirmación incompleta, puesto a que la multiplicación por bloques tiene una gran ventaja que no fue aprovechada durante esta experimentación. Dicha ventaja radica en su enfoque *divide and conquer*, el cual ocasiona que el algoritmo se preste mucho más para la programación paralela, dividiendo la carga en múltiples procesadores y sus respectivos hilos.

Teniendo esto último en cuenta, se puede afirmar que para un enfoque de computación simple que disponga de un sólo procesador, el algoritmo de multiplicación de matrices por filas da mejores resultados en C que el de multiplicación simple y multiplicación por bloques. En el caso de disponer con herramientas de paralelización en varios núcleos o inclusive procesadores, el algoritmo de multiplicación por bloques con una buena implementación en su paralelización probablemente de aún mejores resultados que el algoritmo de multiplicación por filas.