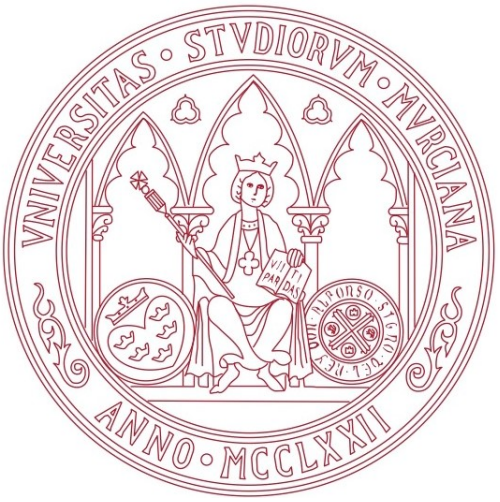

Programación de arquitecturas multinúcleo

Trabajo Práctico 2.3: Proyecto de Programación con OpenMP

Alumno: Alejandro Carmona Martínez

Correo: alejandro.carmonam@um.es

Profesor: Javier Antonio Cuenca Muñoz

Curso Académico: 2023/24

Fecha: 27/02/2024

Índice

1. Diseña el programa paralelo, usando OpenMP, MulMatCua.c para multiplicar matrices cuadradas de números reales en doble precisión, Cnxn=AnxnBnxn. Se debe repartir el trabajo a realizar de manera que el cálculo de cada conjunto de F filas consecutivas de la matriz C sea asignado a un hilo de entre los t hilos generados. 1

1.1. Algoritmo mm 1

1.1.1. Justificación de la planificación 2

1.2. Corrección de los resultados 3

2. Haz un estudio experimental de las prestaciones de MulMatCua.c. Es decir, para cada tamaño de matriz de un conjunto dado: 512,1024,2048, compara razonadamente los tiempos de ejecución que obtienes para las combinaciones de: 5

2.1. Estudio experimental de los resultados 6

2.2. Conclusiones 11

3. Tomando como base el programa MulMatCua.c, escribe un programa secuencial, llamado SecMulMatCuaBlo.c, para calcular el producto de dos matrices cuadradas de números reales de doble precisión, de la forma Cnxn=AnxnBnxn, utilizando un esquema por bloques. En este esquema por bloques, el producto matricial se lleva a cabo gestionando las matrices como conjunto de submatrices (bloques) de tamaño bxb. 11

3.1. Corrección de los resultados 13

4. A partir del programa SecMulMatCuaBlo.c, escribe un programa secuencial, llamado SecMulMatBlo.c, para calcular el producto de dos matrices rectangulares de números reales de doble precisión, de la forma Cmxn=AmxkBkxn, utilizando un esquema por bloques. 14

4.1. Corrección de los resultados 16

5. A partir de los programas anteriores, escribe un programa paralelo usando OpenMP, llamado MulMatCuaBlo.c, para calcular el producto de dos matrices cuadradas de números reales de doble precisión, de la forma Cnxn=AnxnBnxn, utilizando un esquema por bloques. IMPORTANTE: Se debe repartir el trabajo de manera que el cálculo de cada bloque de la matriz C sea asignado a 1 hilo (evidentemente, si el número de bloques de C es mayor que el número de hilos, a cada hilo le puede tocar

calcular más de un bloque). 17

5.1. Corrección de los resultados 19

6. Haz un estudio experimental de las prestaciones de tu implementación paralela MulMatCuaBlo.c. Es decir, para cada tamaño de matriz de un conjunto dado, dim_mat_n=512,1024,2048, compara razonadamente los tiempos de ejecución para todas las combinaciones de: 20

7. Compara razonadamente los tiempos obtenidos con la versión paralela que utiliza bloques, MulMatCuaBlo.c, con los tiempos que obtuviste con la versión paralela sin bloques, MulMatCua.c 25

7.1. Máximo rendimiento 27

7.2. Número de hilos 28

7.3. Tamaño de matriz 30

8. A partir de los programas anteriores, escribe un programa paralelo usando OpenMP, llamado MulMatBlo.c, para calcular el producto de dos matrices rectangulares de números reales de doble precisión, de la forma Cmxn=AmxkBkxn, utilizando un esquema por bloques. 31

8.1. Corrección de los resultados 31

9. A partir de los programas anteriores, escribe un programa paralelo usando tareas de OpenMP, llamado TMulMatCuaBlo.c, para calcular el producto de dos matrices cuadradas de números reales de doble precisión, de la forma Cnxn=AnxnBnxn, utilizando un esquema por bloques. Explica cómo has distribuido el trabajo entre las tareas. 32

9.1. Corrección de los resultados 34

10. Haz un estudio experimental de las prestaciones de tu implementación paralela TMulMatCuaBlo.c. Es decir, para cada tamaño de matriz de un conjunto dado, dim_mat_n=512,1024,2048, compara razonadamente los tiempos de ejecución para todas las combinaciones de: 35

11. Compara razonadamente los tiempos obtenidos con las dos versiones paralelas que utilizan bloques. Es decir, compara la versión que usa tareas, TMulMatCuaBlo.c y la que no usa tareas, MulMatCuaBlo.c. 40

11.1. Máximo rendimiento 42

11.2. Número de hilos 43

11.3. Tamaño de bloque 44

11.4. Tamaño de matriz 46

12. Conclusiones **47**

1. Diseña el programa paralelo, usando OpenMP, MulMatCua.c para multiplicar matrices cuadradas de números reales en doble precisión, Cnxn=AnxnBnxn. Se debe repartir el trabajo a realizar de manera que el cálculo de cada conjunto de F filas consecutivas de la matriz C sea asignado a un hilo de entre los t hilos generados.

SINTAXIS:

`MulMatCua <dimension_matriz_n><numero_filas_consecutivas_F><numero_hilos_t>`

Ubicación: /1_MulMatCua

El programa paralelo `MulMatCua.c` se encuentra comentado en su totalidad en `/1_MulMatCua` donde se puede consultar en detalle. En este apartado analizaremos los aspectos más importantes del código que lo hacen cumplir con las especificaciones del enunciado:

1.1. Algoritmo mm

```
// mm
// A = Matriz A, B = Matriz B, C = Matriz C, matrix_size (ldm) = tamaño de la matriz,
// num_threads_t = número de hilos, num_filas_f = número de filas consecutivas
void mm(double *A, double *B, double *C, int matrix_size, int num_threads_t, int num_filas_f)
{
  int ldm = matrix_size;
  // Ajustar el número de hilos
  omp_set_num_threads(num_threads_t);

  int i, j, k, iam, nprocs;
  double sum;
// Los hiños se crean, todos los hilos tienen las variables privadas iam y nprocs
#pragma omp parallel private(iam, nprocs)
  {
    nprocs = omp_get_num_threads();
    iam = omp_get_thread_num();

//La multiplicación de matrices se realiza en paralelo, cada hilo tiene sus propias variables privadas:
// i: fila de la matriz, sum: resultado de la multiplicación para un elemento de C,
//j: columna de la matriz, k: columna de A y fila de B
#pragma omp for private(i, j, k, sum) schedule(static, num_filas_f)
    for (i = 0; i < matrix_size; i++)
    {
      #ifdef DEBUG
      printf("thread %d fila %d \n", iam, i);
      #endif
```

```
27      for (j = 0; j < matrix_size; j++)
28      {
29        sum = 0.0;
30        for (k = 0; k < matrix_size; k++)
31        {
32          sum += A[i * ldm + k] * B[k * ldm + j]; // A[i][k] * B[k][j]
33        }
34        C[i * ldm + j] = sum; // C[i][j]
35      }
36    }
37  }
38 }
```

Listing 1: Código de mm casi descomentado para mejor legibilidad

Tras crear las matrices y leer los argumentos, llamamos a la función mm para realizar la multiplicación en sí. En primer lugar, establecemos el número de hilos y las variables que vamos a usar, tanto las que son privadas a los hilos como las que no lo son.

A continuación con `pragma omp parallel private(iam, nprocs)` creamos los hilos y se les asignan las variables iam y nprocs como privadas.

Y con el siguiente pragma:

```
1 #pragma omp for private(i, j, k, sum) schedule(static, num_filas_f)
```

conseguimos paralelizar el bucle for y cumplimos con el reparto de filas por hilo indicado en el enunciado, ya que se utiliza una planificación estática, schedule(static), para distribuir las iteraciones del bucle entre los hilos de forma equitativa en bloques de tamaño `num_filas_f`. Esto significa que la carga de trabajo se divide en bloques fijos y se asigna a los hilos antes de la ejecución del bucle. En caso de haber más hilos que bloques de filas, habrá hilos que no realicen trabajo y en caso de que haya menos, habrá hilos que realicen la ejecución de más de un bloque. Si tenemos el mismo número de bloques de filas que hilos, todos realizarán una ejecución. En el siguiente apartado comentaremos los motivos de esta decisión.

También se declaran las variables i, j, k, y sum como privadas para cada hilo, garantizando que cada hilo tenga su propia copia de estas variables, lo que previene interferencias entre hilos.

Finalmente, si la macro DEBUG está definida, imprimimos el hilo y la fila que se le ha asignado para verificar que a cada hilo se le asigna un grupo de filas correctamente, más sobre esto en la siguiente sección.

1.1.1. Justificación de la planificación

Por lo tanto, hemos tomado el camino de la planificación estática para cumplir con que todos los hilos tengan trabajo asignado por defecto y no se pierda rendimiento por el overhead de las asignaciones dinámicas.

Esto tiene ciertas ventajas e inconvenientes, principalmente tenemos un menor overhead a cambio de menor versatilidad. Es decir, debido a que las asignaciones se hacen estáticamente, se hacen más rápido que su contraparte, dynamic. Sin embargo, al hacer las asignaciones estáticamente, nos encontramos que si algunos hilos van más lentos de lo normal, nos quedaremos esperando a que estos hilos terminen sus asignaciones, lo cual se manifiesta especialmente en cargas de trabajo desiguales, pero al menos en teoría, en este caso, tenemos cargas de trabajo lo más simétricas posible para todos los hilos (depende de si el número de hilos es múltiplo del tamaño de matriz, pero pueden ser perfectamente simétricas).

Eso es cierto, en teoría y si tenemos hilos con los mismos recursos, la tarea será repartida equitativamente y nos beneficiará una planificación de este tipo; sin embargo, en nuestra máquina tenemos una situación curiosa.

Y es que tenemos 4 cores principales con hyperthreading y 8 cores más livianos, en otras palabras podemos tener 4 hilos simétricos con los cores principales, 8 hilos simétricos con hyperthreading o los cores más pequeños o, finalmente, 16 hilos no simétricos con diferente capacidad de cómputo y que pueden tardar diferentes tiempos para ejecutar una tarea. **Se ha decidido elegir un enfoque estático, puesto que cubre mejor los casos en los que se ejecuta el programa con 4 o 8 hilos a costa de una pequeña latencia extra cuando usemos 16 hilos** para que en la mayoría de configuraciones alcance el máximo rendimiento. No obstante, es de recibo mencionar que**si queremos maximizar el rendimiento cuando tengamos todos los hilos en funcionamiento y tenemos una máquina con hardware no simétrico**, aunque tengamos una tarea que se pueda repartir equitativamente, dynamic puede ofrecer más rendimiento.

1.2. Corrección de los resultados

Para garantizar la corrección de los resultados tenemos el modo DEBUG que, si definimos la macro DEBUG al principio del código, nos permite realizar un estudio de los resultados y comprobarlos contra una implementación en secuencial tradicional de la multiplicación de matrices cuadrada, además de informarnos del reparto de filas por hilo como comentábamos anteriormente:

```
1  //Multiplicación secuencial de matrices
2  void seq_mult(double *A, double *B, double *C_check, int matrix_size)
3  {
4    int i, j, k;
5    double sum;
6    for (i = 0; i < matrix_size; i++)
7    {
8      for (j = 0; j < matrix_size; j++)
9      {
10       sum = 0.0;
11       for (k = 0; k < matrix_size; k++)
12       {
13         sum += A[i * matrix_size + k] * B[k * matrix_size + j]; // A[i][k] * B[k][j]
14       }
15       C_check[i * matrix_size + j] = sum; // C_check[i][j]
16     }
17   }
```

```
}
```

Una vez realizada la multiplicación, ambos resultados son comparados en la siguiente función:

```
int comprobar(double *C, double *C_check, int matrix_size)
{
  int i, j;
  double tolerancia = 0.01; // Definir la tolerancia hasta el segundo decimal
  for (i = 0; i < matrix_size; i++)
  {
    for (j = 0; j < matrix_size; j++)
    {
      // Comprobar si la diferencia absoluta es mayor a la tolerancia
      if (fabs(C[i * matrix_size + j] - C_check[i * matrix_size + j]) > tolerancia)
      {
        return -1;
      }
    }
  }
  return 0;
}
```

Y esta es la salida resultante en modo DEBUG para una multiplicación de matrices cuadradadas de 4x4, 2 filas consecutivas y 16 hilos:

```
ale1@ale1:~/Escritorio/Universidad/PAM/PAM_2_3_Entrega/1_MulMatCua$ ./MulMatCua 4 2 16
thread 1 fila 2
thread 1 fila 3
thread 0 fila 0
thread 0 fila 1

MULTIPLICACION: OK

Matriz A
0.8402 0.3944 0.7831 0.7984
0.9116 0.1976 0.3352 0.7682
0.2778 0.5540 0.4774 0.6289
0.3648 0.5134 0.9522 0.9162
Matriz B
0.6357 0.7173 0.1416 0.6070
0.0163 0.2429 0.1372 0.8042
0.1567 0.4009 0.1298 0.1088
0.9989 0.2183 0.5129 0.8391
Matriz C Resultado
1.4608 1.1867 0.6843 1.5823
1.4027 1.0040 0.5938 1.3933
0.8886 0.6625 0.4999 1.1937
1.3047 0.9681 0.7156 1.5067
```

Para comprobar que el reparto de filas/hilos es correcto podemos realizar más pruebas con distintos valores para los argumentos y comprobar el correcto reparto de los hilos por

fila y resultado; sin embargo, debido al reparto con static, tras realizar unas cuantas pruebas y conociendo el funcionamiento de este pragma, podemos garantizar el correcto reparto de filas en el programa.

Por otro lado, si queremos realizar múltiples pruebas con diferentes tamaños de matriz y filas consecutivas para garantizar que los resultados son numéricamente correctos, podemos probar todos los conjuntos posibles de filas consecutivas e hilos para distintos tamaños de matriz con el programa `MulMatCuaTest.c` que realizando este proceso iterativamente comprueba que se superen todos estos casos con el objetivo de encontrar algún posible fallo de programación. Se puede encontrar en `/1_MulMatCua`.

```
Uso: ./MulMatCuaTest <n_min> <n_max>
```

2. Haz un estudio experimental de las prestaciones de MulMatCua.c. Es decir, para cada tamaño de matriz de un conjunto dado: 512,1024,2048, compara razonadamente los tiempos de ejecución que obtienes para las combinaciones de:

- Número de hilos generados: 1,4,8,16
- Número de filas consecutivas de la matriz C calculadas por cada hilo: 1,4,16,32

Ubicación: /2_MulMatCua

Antes de comenzar con el estudio de los resultados, vamos a indicar cómo los hemos registrado.

Para ello, creamos un nuevo programa llamado `MulMatCua-bench.c` que usa el mismo algoritmo mm, y es prácticamente el mismo programa de la sección anterior con una salida modificada para poder ejecutarse desde un script de shell (`benchmark-runner.sh`), probar todos los posibles casos y escribirlos en un fichero .csv.

Además, `MulMatCua-bench.c` ejecuta una serie de ejecuciones de warmUp para calentar la caché y simular un comportamiento más realista y luego realiza un número fijo de repeticiones y calcula la media del tiempo de ejecución e imprime los resultados de tiempo y GFLOPS además de la configuración utilizada.

Ambos programas están en `/2_Estudio1`.

2.1. Estudio experimental de los resultados

Debido a las 48 posibles combinaciones y a tener tres variables con 4,4 y 3 posibles valores respectivamente, vamos a atacar cada posible parámetro de ejecución individualmente y a evaluar su relación cuando sea relevante.

Para ello, introducimos una gráfica con la media de GFLOPS de todas las configuraciones con valores fijos para los valores de los parámetros a analizar, así podremos fijarnos más fácilmente en el efecto del número de hilos, filas consecutivas y dimensiones de la matriz sobre el rendimiento.

Utilizamos los GFlops porque los tiempos dependen mucho del tamaño de matriz, aunque también se comentarán en los siguientes sub-apartados. A continuación, introducimos los datos registrados:

Configuración		**Tiempo Medio (segundos) por Tamaño de Matriz**		
Filas	**Hilos**	**512**	**1024**	**2048**
1	1	0.2789	2.1949	58.7595
1	4	0.0679	0.6250	10.1957
1	8	0.0733	0.4836	7.1791
1	16	0.0506	0.3964	5.5858
4	1	0.2729	2.3034	38.3161
4	4	0.0717	0.6208	8.0102
4	8	0.0661	0.4892	5.7740
4	16	0.0510	0.3892	5.6267
16	1	0.2657	2.3519	35.4920
16	4	0.0706	0.6219	6.4076
16	8	0.0719	0.5037	5.8257
16	16	0.0502	0.3942	4.6105
32	1	0.2493	2.3417	43.1176
32	4	0.0690	0.6220	7.3008
32	8	0.0745	0.5103	6.1997
32	16	0.0500	0.3845	4.6074

Tabla 1: Tiempo Medio por Tamaño de Matriz y Configuración

Configuración		GFlops por Tamaño de Matriz		
Filas	**Hilos**	**512**	**1024**	**2048**
1	1	0.9616	0.9779	0.2923
1	4	3.9489	3.4346	1.6846
1	8	3.6579	4.4381	2.3924
1	16	5.3034	5.4142	3.0749
4	1	0.9828	0.9319	0.4483
4	4	3.7407	3.4578	2.1442
4	8	4.0565	4.3877	2.9747
4	16	5.2615	5.5143	3.0525
16	1	1.0092	0.9126	0.4839
16	4	3.7992	3.4512	2.6805
16	8	3.7294	4.2610	2.9483
16	16	5.3390	5.4450	3.7253
32	1	1.0758	0.9166	0.3983
32	4	3.8888	3.4511	2.3526
32	8	3.6004	4.2063	2.7704
32	16	5.3652	5.5824	3.7278

Tabla 2: GFlops por Tamaño de Matriz y Configuración

Número de Hilos (t)

En primer lugar, si en la tabla 2.1 nos fijamos en los tiempos para un solo tamaño, el de 512, por ejemplo, vemos que estos son reducidos drásticamente, pasando de alrededor de 3 décimas de segundo al orden de centésimas de segundo, independientemente del número de filas e hilos, logrando un speedUp de alrededor de 5.5x si comparamos entre el menor y mayor número de hilos.

Este fenómeno se repite en el tamaño de 1024 y aumenta para 2048 hasta alrededor de 10x.

Con esta información podemos concluir que el número de hilos es el mayor causante de aumento de rendimiento a nivel global, como se valida en 2.1 y que el número de filas consecutivas es casi irrisorio en comparación, puesto que se pueden ver resultados con muy poca diferencia temporal si mantenemos el tamaño fijo.

Esto es coherente con las ideas de que para obtener un mayor rendimiento es necesario paralelizar hasta cierto número de hilos dependiendo de nuestro hardware para realizar un mayor trabajo, pero también tiene que ver con ese hardware que usemos.

Como se vio comentó en el ejercicio anterior y más en detalle en la primera práctica, en nuestra máquina tenemos 12 cores, 4 con hyperthreading y caché L1 y L2 independientes y 8 cores más livianos con L1 privada y 2 L2 que comparten entre ellos. El motivo de que reintroduzca esto es que si ejecutamos un solo hilo, tan solo necesitará de un core y aunque nos pongamos en el caso de que sea uno de los 4 cores más potentes con caché L1

y L2 dedicada, no podremos acceder al resto de cachés privadas a los cores y estaremos no solo desperdiciando el cómputo que podrían estar realizando sino el almacenamiento y acceso rápido a los datos necesarios. Se expandirá el rol del sistema de memoria en el apartado de tamaño de matriz, pero ya nos podemos hacer una idea de que maximizar el uso de las cachés y los cores da un mayor rendimiento, como se ve a continuación:

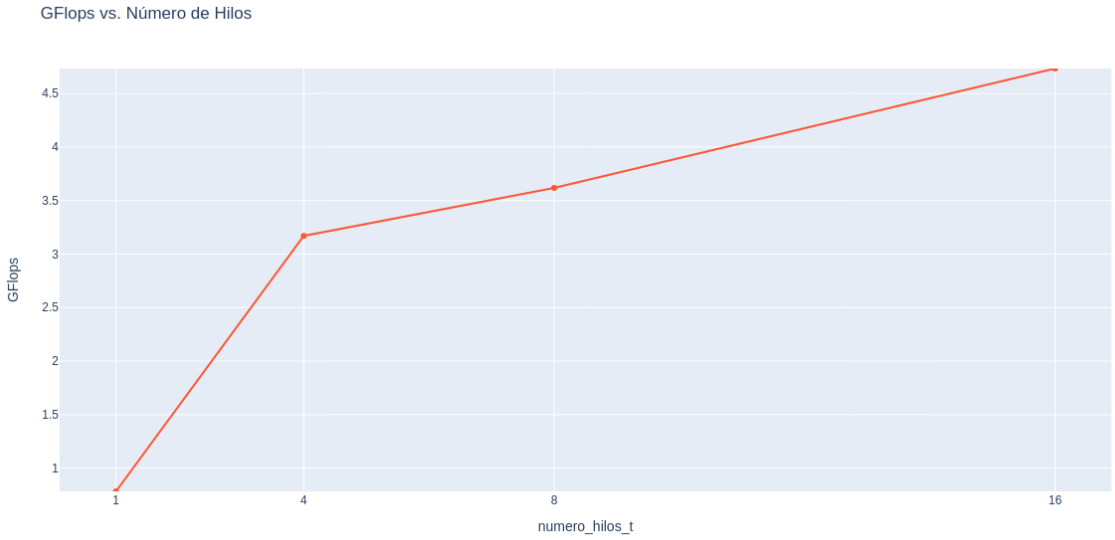

Figura 1: Gráfica FLOPS con número de hilos variable

Como vemos, el speedUp va en aumento constante, aunque crezca con menor rapidez al aumentar el número de hilos de 4 a 8 debido al overhead.

Al tener nuestra máquina soporte para hasta 16 hilos nativos y cachés L1 dedicadas en cada core y L2 muy replicadas, el crecimiento del rendimiento reflota al pasar de 8 a 16 hilos

Número de filas consecutivas (F)

Como ya hemos anticipado antes, este parámetro no tiene tanta importancia como el número de hilos, ya que las magnitudes de las ganancias de rendimiento entre ambos son más bien bajas, sobre todo con los tamaños de matriz más bajos, como se puede apreciar en el gráfico de rendimiento a continuación:

Sin embargo, es cierto que con el tamaño de matriz más grande (2048) y sobre todo cuantos menos hilos hay en ejecución, se obtienen diferencias de tiempo notables debido a los tamaños más grandes de las entradas.

Y es que, al pasar de 1 fila a 4 con un solo hilo se produce un speedUp de alrededor de 1.52x, mientras que al pasar a 16, conseguimos 1.65x 2.1. Sin embargo, baja levemente al subir a 32 filas aunque se consiga una mejoría sobre el caso base (1 fila).

Si aumentamos el número de hilos a su valor máximo, parece que se produce una

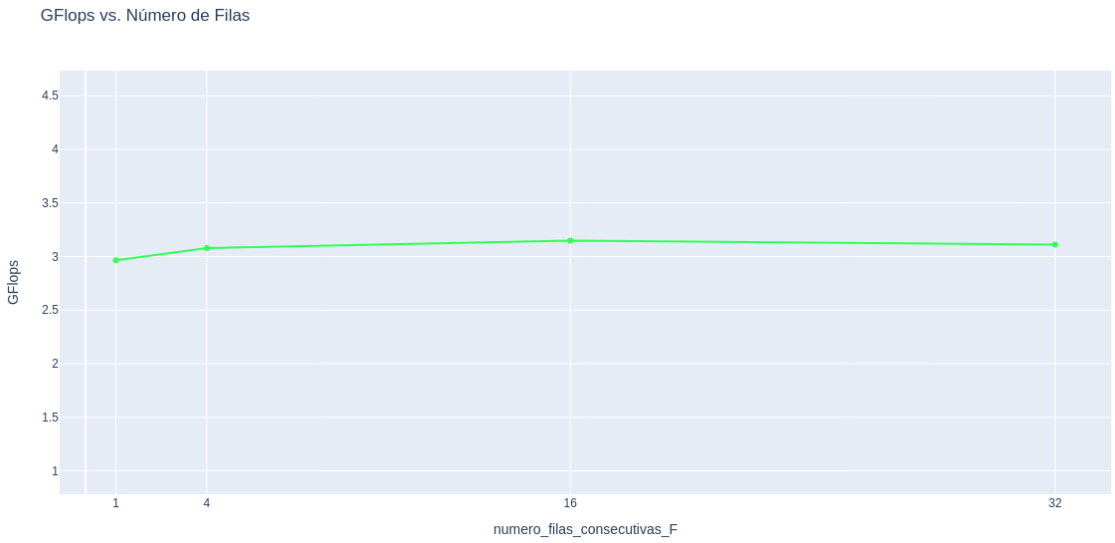

Figura 2: Gráfica FLOPS con número de filas consecutivas variable

ligerísima mejoría constante si subimos el número de filas hasta llegar a 32, aunque la diferencia con 16 es nula.

El motivo de la generalización de que a mayor tamaño de filas consecutivas, ligeramente más rendimiento se debe a que al tener mayor tamaño de filas que asignar de golpe a un hilo, se realizan menos asignaciones. Permitiendo que a mayor tamaño de filas consecutivas, más rendimiento hasta que se alcanza cierto punto, y este baja, sin embargo, con mayores tamaños de hilos y de matriz, este punto parece no alcanzarse.

En conclusión, este punto puede ser importante con menor número de hilos y mayor tamaño de matriz, más sobre esto en los siguientes apartados, pero concluyendo con lo que se ve en la gráfica, este parámetro apenas produce una diferencia considerable por sí solo, puesto que no es comparable a tener múltiples hilos trabajando en paralelo y además depende del número de hilos, puesto que si tenemos 1 solo hilo, realizar asignaciones constantes a él, es completamente innecesario y solo conlleva sobrecarga, como se ve de manera exagerada en los FLOPS obtenidos si usamos el tamaño de matriz 2048 2.1, aunque esa caída de rendimiento también se puede achacar al tamaño de matriz como veremos a continuación.

Dimensión Matriz (n)

Lógicamente y como se ve en la tabla de tiempos 2.1, tenemos que a mayor tamaño de matriz, mayor tiempo, lo cual es coherente con el orden de `N al cubo - N al cuadrado` de la multiplicación de matrices, pero eso, teóricamente no debería afectar al rendimiento en GFLOPS, sin embargo, está claro que a mayor tamaño de matriz, menor rendimiento:

Vemos que el rendimiento es prácticamente el mismo, al pasar de 512 a 1024, pero tiene una caída importante en 2048.

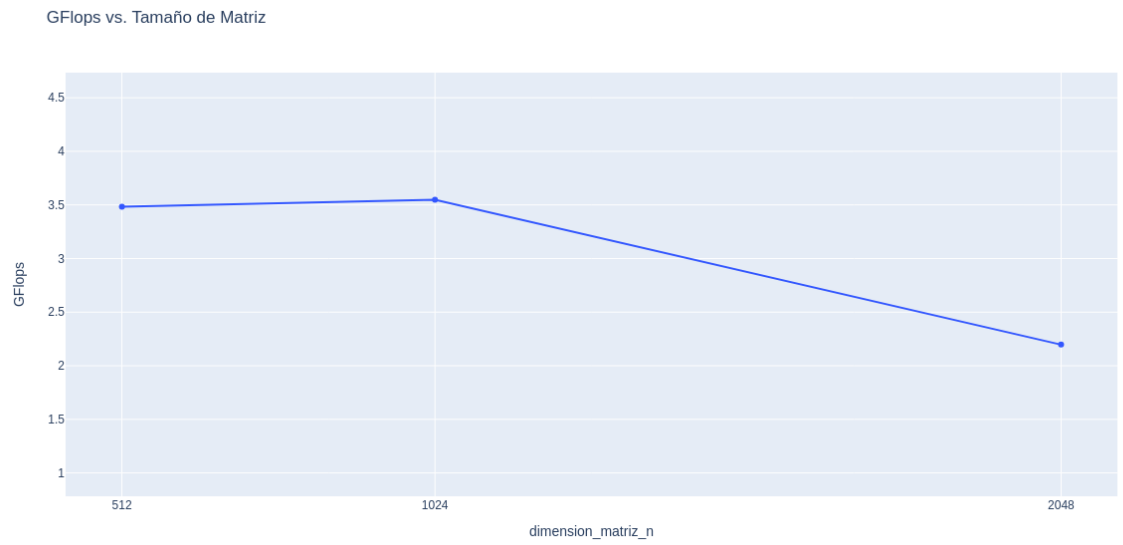

Figura 3: Gráfica FLOPS con tamaño de matriz variable

Como introducíamos antes, el hardware multicore, y este en particular, tiene una jerarquía de memoria con L1 y L2 dedicadas, pero las cachés tienen una capacidad de almacenamiento finita, supongamos que usamos los 16 hilos, estas serían las dimensiones acumuladas de las cachés:

```
Caches (sum of all):
 L1d: 448 KiB (12 instances)
 L1i: 640 KiB (12 instances)
 L2: 9 MiB (6 instances)
 L3: 18 MiB (1 instance)
```

Y estos serían los tamaños en KiB de los datos con los que operamos:

```
TamañoProblema_512=NUM_ELEM_MATRIZ*(SIZE_OF_DOUBLE_bytes)*(NUM_MATRICES)
TamañoProblema_512=(((512)*(512))*(8)*3) / 1024
TamañoProblema_512=6144 Kb
TamañoProblema_1024=((1024)2))*8)*3/1024=24576 Kb
TamanñoProblema_2048=((2048)2))*8)*3/1024=98304 Kb
```

Mientras que para 512 tenemos todos los datos nos caben en L2 y en 1024, alrededor de un cuarto están en L2 y el resto están en L3 y se pueden enmascarar esos accesos a L3, además de que la matriz C tampoco tiene que estar en su totalidad en caché y la A tampoco, si subimos a 2048, es imposible tener la matriz B al completo en las cachés, por lo que tenemos que hacer accesos a memoria que empeoran el rendimiento si o sí.

2.2. Conclusiones

1. Usar todos los hilos nativamente permitidos por la máquina es lo más importante.
2. Si trabajamos con tamaños de matrices más grandes que las cachés, podemos perder rendimiento al tener que ir a memoria principal.
3. El número de filas consecutivas, es decir, la planificación estática es muy poco importante a mayor número de hilos, aunque con mayores tamaños de entrada, posiblemente debido al sistema de memoria y cachés ya comentado, mayores tamaños pueden darnos cierto speedUp.

3. Tomando como base el programa MulMatCua.c, escribe un programa secuencial, llamado SecMulMatCuaBlo.c, para calcular el producto de dos matrices cuadradas de números reales de doble precisión, de la forma Cnxn=AnxnBnxn, utilizando un esquema por bloques. En este esquema por bloques, el producto matricial se lleva a cabo gestionando las matrices como conjunto de submatrices (bloques) de tamaño bxb.

NOTA: Suponemos (no hace falta comprobarlo en el código) que la dimensión de la matriz cuadrada, n, siempre va a ser un múltiplo de la dimensión del bloque cuadrado, b, es decir, se cumple que n=eb, siendo e un número entero.

SINTAXIS:

```
SecMulMatCuaBlo <dim_mat_n><tam_blo_b>
```

Ubicación: /3_SecMulMatCuaBlo

Para realizar este programa nos hemos basado directamente en el programa del ejercicio 1 cambiando la entrada para adaptarlo al uso secuencial, pero la gran modificación obviamente es en el algoritmo, `mm`.

Debido a su carácter secuencial, no tenemos que preocuparnos del manejo de los hilos, por lo que no aparecen pragmas a diferencia del programa anterior, sin embargo, el algoritmo gana en complejidad, puesto que ahora recorremos las matrices en forma de bloque, como se ve en el enunciado:

Podemos ver el ejemplo de arriba como un caso en el que las matrices son de 3x3 y el tamaño de bloque es 1x1. Como vemos para un bloque de 1x1 concreto o, para un valor genérico de un bloque, en este caso el valor (2,2) de la matriz recorremos la fila 2 de A y la columna 2 de B.

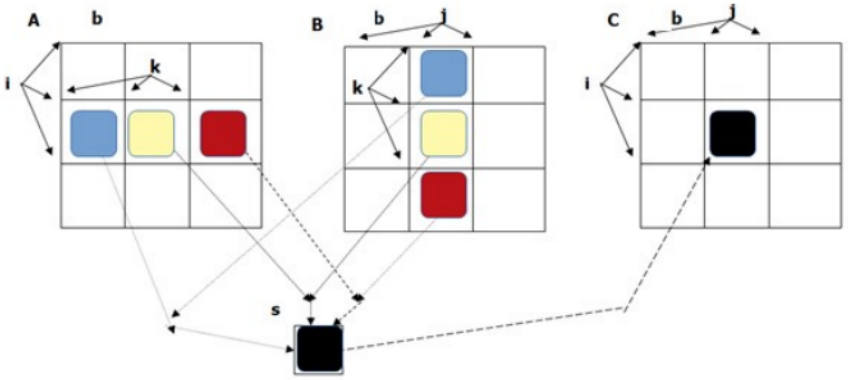

Figura 4: Representación del blocking a alto nivel

Una vez sumadas todas las multiplicaciones, pasaríamos al siguiente bloque, (2,3) y recorreríamos la fila 2 de A de nuevo y la columna 3 de B y así hasta completar todos los bloques.

Si los bloques fueran de NxN, obtendríamos los valores de todos los elementos del bloque y luego pasaríamos al siguiente. Por ejemplo, en una matriz de 4x4 con bloques de 2x2, recorreríamos las dos primeras filas y columnas de A y B para obtener los resultados del primer bloque, el que tiene los valores (1,1);(1,2);(2,1);(2,2).

Luego iríamos al siguiente bloque y así sucesivamente.

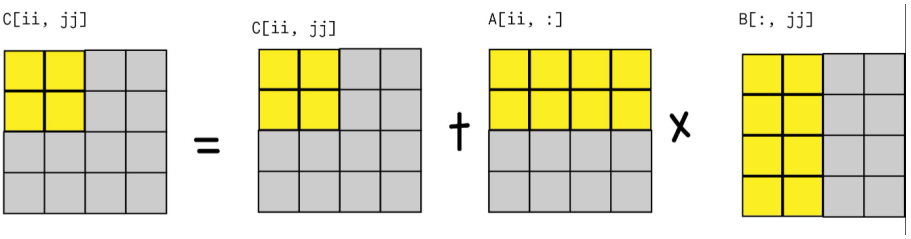

Figura 5: Blocking con bloques de 2x2 para matrices de 4x4. [1]

Esto lo podemos plasmar usando dos bucles for sobre los tres bucles internos de la multiplicación de matrices. Estos iterarán sobre las matrices en incrementos del tamaño del bucle, de forma que indiquen las coordenadas de inicio del bloque sobre la matriz resultado C, con el primer bucle indicando la fila y el segundo bucle indicando la columna, recorriendo así todos los puntos iniciales de cada bloque de la matriz.

Los dos bucles que recorren las filas y columnas de A y B (iteradores i y j) recorrerán desde la fila y columna indicada por los dos fors externos hasta la suma de la fila y columna respectiva más el tamaño de bloque, de forma que obtendremos los resultados de todo un bloque antes de pasar al siguiente.

Finalmente, tenemos el bucle en el que recorremos una fila y columna concreta de A y B (iterador k) para obtener el valor de un elemento de la matriz, solo que ahora obtenemos un elemento del bloque. Este bucle se mantiene igual que en el ejercicio anterior, puesto que itera de la misma forma.

```
1 void mm(double *A, double *B, double *C, int matrix_size, int tam_blo_b)
```

```
{
  double sum; // Variable para almacenar el resultado de la multiplicación
  int ldm = matrix_size; // Tamaño de la matriz = ldm (matriz cuadrada)
  int fila, col, fila_bloque, col_bloque, k; // Variables para los bucles

  //Avanzamos de tamaño de bloque en tamaño de bloque para recorrer la matriz
  //así en el 3er bucle for se opera el bloque concreto Pos Inicio:(fila, fila+tam_blo
      ↪ _b), Pos fin:(col, col+tam_blo_b)
  for (fila = 0; fila < ldm; fila += tam_blo_b) {
    for (col = 0; col < ldm; col += tam_blo_b) {
      #ifdef DEBUG
      printf("BLOQUE (%d, %d)\n", fila, col); // Imprimir el bloque actual con su Pos
          ↪ Inicio
      #endif
      // Hacemos la multiplicación de matrices para bloques
      for (fila_bloque = fila; fila_bloque < fila+tam_blo_b; fila_bloque ++) {
        for(col_bloque = col; col_bloque < col+tam_blo_b; col_bloque++) {
          sum = 0.0;
          //recorremos toda la fila y la columna para hacer la obtener el resultado de
              ↪ 1 elemento del bloque
          for (k = 0; k < ldm; k++) {
            sum += A[fila_bloque * ldm + k] * B[k * ldm + col_bloque];
          }
          C[fila_bloque * ldm + col_bloque] = sum;
          #ifdef DEBUG
          printf("%.4lf ", sum); //Imprimimos el valor de un elemento del bloque
          #endif
        }
        #ifdef DEBUG
        printf("\n"); //Salto de línea para separar las filas del bloque
        #endif
      }
      #ifdef DEBUG
      printf("\n"); //Salto de línea para separar los bloques
      #endif
    }
  }

}
```

NOTA: Podemos avanzar de tamaño de bloque en tamaño de bloque en los bucles, puesto que los tamaños de bloque son siempre múltiplos del tamaño de matriz. En caso contrario, habría que establecer límites u otras técnicas.

3.1. Corrección de los resultados

De forma similar al ejercicio 1, en modo DEBUG imprimimos los resultados, además de imprimir cada bloque para garantizar que la matriz resultado ha sido operada en bloques, y comprobamos contra la implementación secuencial:

```
./SecMulMatCuaBlo 4 2
BLOQUE (0, 0)
1.4608 1.1867
```

```
1.4027 1.0040

BLOQUE (0, 2)
0.6843 1.5823
0.5938 1.3933

BLOQUE (2, 0)
0.8886 0.6625
1.3047 0.9681

BLOQUE (2, 2)
0.4999 1.1937
0.7156 1.5067


MULTIPLICACION: OK

Matriz A
0.8402 0.3944 0.7831 0.7984
0.9116 0.1976 0.3352 0.7682
0.2778 0.5540 0.4774 0.6289
0.3648 0.5134 0.9522 0.9162
Matriz B
0.6357 0.7173 0.1416 0.6070
0.0163 0.2429 0.1372 0.8042
0.1567 0.4009 0.1298 0.1088
0.9989 0.2183 0.5129 0.8391
Matriz C Resultado
1.4608 1.1867 0.6843 1.5823
1.4027 1.0040 0.5938 1.3933
0.8886 0.6625 0.4999 1.1937
1.3047 0.9681 0.7156 1.5067
```

Por supuesto, también tenemos un programa `SecMulMatCuaBloTest.c` en `3_SecMulMatCuaBlo` encargado de ejecutar múltiples escenarios con diferentes tamaños de bloque para matrices cuadradas de tamaños entre dos valores elegidos por el usuario que comprueban que los resultados de las matrices calculadas con blocking encajan con los resultados obtenidos secuencialmente, validando los resultados numéricos.

4. A partir del programa SecMulMatCuaBlo.c, escribe un programa secuencial, llamado SecMulMatBlo.c, para calcular el producto de dos matrices rectangulares de números reales de doble precisión, de la forma Cmxn=AmxkBkxn, utilizando un esquema por bloques.

NOTA: Suponemos (no hace falta comprobarlo en el código) que las dimensiones de las matrices (m, n y k) siempre van a ser múltiplos de la dimensión del bloque cuadrado,

b. Es decir, que se cumple que m=eb, n=gb y k=hb; siendo e, g y h números enteros.

SINTAXIS

`SecMulMatBlo <dim_mat_m><dim_mat_n><dim_mat_k><tam_blo_b>`

Ubicación: /4_SecMulMatBlo

Este programa será muy similar al anterior, con la salvedad de que los argumentos introducidos por el usuario ahora serán para matrices rectangulares y para ello, se han de adaptar las funciones para iterar sobre las distintas dimensiones de las matrices. Estos cambios han de producirse en todas las funciones, pero vamos a utilizar la función mm para verlos a modo de ejemplo para el resto:

```
// mm
// A = Matriz A, B = Matriz B, C = Matriz C, m = filas de A,
// n = columnas de A y filas de B, k = columnas de B y tam_blo_b = tamaño del bloque
void mm(double *A, double *B, double *C, int m, int n, int k, int tam_blo_b)
{
    int ldm = n; // denominamos ldm (leading dimension)
    //al número de columnas de A y filas de B puesto que será el valor que ussemos para
    //recorrer las filas de A y las columnas de B en su totalidad
    int fila, col, fila_bloque, col_bloque, l; // Variables para los bucles
    double sum; // Variable para almacenar el resultado de la multiplicación

    //Avanzamos de tamaño de bloque en tamaño de bloque para recorrer la matriz
    //así en el 3er bucle for se opera el bloque concreto Pos Inicio:(fila, fila+tam_
        ↪ blo_b), Pos fin:(col, col+tam_blo_b)
    for (fila = 0; fila < m; fila += tam_blo_b) //Avanzamos de bloque en bloque hasta m
    {
        for (col = 0; col < k; col += tam_blo_b) //Avanzamos de bloque en bloque hasta
            ↪ k
        {
            #ifdef DEBUG
            printf("BLOQUE (%d, %d)\n", fila, col); // Imprimir el bloque actual con su
                ↪  Pos Inicio
            #endif
            //Estos dos bucles se mantienen igual que en la multiplicación de matrices
                ↪ cuadradas
            // Hacemos la multiplicación de matrices para bloques
            for (fila_bloque = fila; fila_bloque < fila + tam_blo_b; fila_bloque++)
            {
                for (col_bloque = col; col_bloque < col + tam_blo_b; col_bloque++)
                {
                    sum = 0.0;
                    //recorremos toda la fila y la columna para hacer la obtener el
                        ↪ resultado de 1 elemento del bloque
                    for (l = 0; l < ldm; l++)
                    {
                        sum += A[fila_bloque * ldm + l] * B[l * k + col_bloque];
                    }
                    C[fila_bloque * k + col_bloque] = sum;
                    #ifdef DEBUG
                    printf("%.4lf ", sum); //Imprimimos el valor de un elemento del
                        ↪ bloque
                    #endif
                }
                #ifdef DEBUG
                printf("\n"); //Salto de línea para separar las filas del bloque
```

```
            #endif
        }
        #ifdef DEBUG
        printf("\n"); //Salto de línea para separar los bloques
        #endif
    }

}

}
```

Como se ve en la nueva implementación de mm, ahora iteramos sobre las nuevas dimensiones, pero como es una multiplicación de matrices, podemos seguir usando la leading dimension para avanzar en los elementos de la matriz y la implementación de blocking se mantiene igual.

4.1. Corrección de los resultados

Al igual que en el programa anterior, tenemos un modo DEBUG que nos imprime los bloques y realiza la corrección a partir de una implementación secuencial para matrices rectangulares de `comprobar`. Podemos ver que la salida sigue la misma estructura:

```
./SecMulMatBlo 2 4 4 2
BLOQUE (0, 0)
0.8881 1.4236
0.5509 1.0335

BLOQUE (0, 2)
0.9971 2.0071
0.7762 1.5756


MULTIPLICACION: OK

Matriz A
0.8402 0.3944 0.7831 0.7984
0.9116 0.1976 0.3352 0.7682
Matriz B
0.2778 0.5540 0.4774 0.6289
0.3648 0.5134 0.9522 0.9162
0.6357 0.7173 0.1416 0.6070
0.0163 0.2429 0.1372 0.8042
Matriz C Resultado
0.8881 1.4236 0.9971 2.0071
0.5509 1.0335 0.7762 1.5756
```

Como vemos, la única diferencia es que ahora la matriz resultado es de tamaño MxK, pero los bloques siguen siendo cuadrados.

Ubicado en el mismo folder tenemos el fichero `SecMulMatBloTest.c`, que valida la

corrección del algoritmo mediante un set de tamaños de matrices cuadradas predeterminadas con distintos tamaños de bloque. Su funcionamiento es el mismo que el de su homólogo de ejercicios anteriores, solo que no requiere de límite y nos informa sobre si la implementación es correcta o no.

También tenemos `SecMulMatBloTest2.c` que permite ejecutar todas las combinaciones posibles hasta un valor de tamaño de matriz concreto para comprobar el correcto funcionamiento para tamaños mayores. Para ello utiliza una función que itera sobre diferentes dimensiones y tamaños de bloque y comprueba la validez de los argumentos.

5. A partir de los programas anteriores, escribe un programa paralelo usando OpenMP, llamado MulMatCuaBlo.c, para calcular el producto de dos matrices cuadradas de números reales de doble precisión, de la forma Cnxn=AnxnBnxn, utilizando un esquema por bloques. IMPORTANTE: Se debe repartir el trabajo de manera que el cálculo de cada bloque de la matriz C sea asignado a 1 hilo (evidentemente, si el número de bloques de C es mayor que el número de hilos, a cada hilo le puede tocar calcular más de un bloque).

NOTA: Suponemos (no hace falta comprobarlo en el código) que la dimensión de la matriz cuadrada, n, es un múltiplo de la dimensión del bloque cuadrado, b, es decir, se cumple que n=eb, siendo e un número entero.

SINTAXIS:
`MulMatCuaBlo <dim_mat_n><tam_blo_b><num_hil_t>`

Ubicación: /5_MulMatCuaBlo_hilos

Para realizar este programa vamos a partir de la implementación rectangular, solo que haremos su uso transparente al usuario, haciéndole introducir un único parámetro para el tamaño de la matriz cuadrada.

En cuanto a la paralelización usando hilos y OpenMP, vamos a explicar la nueva versión del algoritmo existente en el método mm:

```
1 // mm
2 // A = Matriz A, B = Matriz B, C = Matriz C, m = filas de A,
3 // n = columnas de A y filas de B, k = columnas de B y tam_blo_b = tamaño del bloque
4 // num_hilos = número de hilos
5 void mm(double *A, double *B, double *C, int m, int n, int k, int tam_blo_b, int num_
    ↪ hilos)
```

```
{
    int ldm = n;
    int fila, col, fila_bloque, col_bloque, l;
    double sum;

    int iam;
    omp_set_num_threads(num_hilos);
    //private -> cada hilo tiene su propia copia de la variable
    //collapse(2) -> colapsa los 2 primeros bucles anidados en un solo bucle
    //schedule(static, 1) -> divide el trabajo en bloques de tamaño 1
    #pragma omp parallel for private(fila, col, fila_bloque, col_bloque, l, sum)
        ↪ collapse(2) schedule(static, 1)
    for (fila = 0; fila < m; fila += tam_blo_b)
    {
        for (col = 0; col < k; col += tam_blo_b)
        {
            #if defined (_OPENMP)
            iam = omp_get_thread_num();
            #endif
            #ifdef DEBUG
            printf("Soy el Hilo %d haciendo el BLOQUE (%d, %d)\n", iam, fila, col);
            #endif
            for (fila_bloque = fila; fila_bloque < fila + tam_blo_b; fila_bloque++)
            {
                for (col_bloque = col; col_bloque < col + tam_blo_b; col_bloque++)
                {
                    sum = 0.0;
                    for (l = 0; l < ldm; l++)
                    {
                        sum += A[fila_bloque * ldm + l] * B[l * k + col_bloque];
                    }
                    C[fila_bloque * k + col_bloque] = sum;
                }
            }
        }

    }
}
/*Para mas detalle sobre la implementacion de blocking, ir al ejercicio 3 y 4.*/
```

En primer lugar, hemos eliminado algunos comentarios redundantes más relevantes en las otras versiones debido a que el código empezaba a tener problemas de legibilidad debido a lo lleno de comentarios que se encontraba.

Respecto a los cambios introducidos por los hilos, se añade la directiva `omp_set_num_threads(num_hilos);` para establecer el número de hilos introducido por el usuario al lanzar el programa.

También Se establece en un único pragma la creación de hilos para el for que viene a continuación:

```
#pragma omp parallel for private(fila, col, fila_bloque, col_bloque, l, sum) collapse
    ↪ (2) schedule(static, 1)
```

- Con `parallel for` paralelizamos el bucle creando los hilos. Aunque el bucle que realmente se mueve de un bloque a otro por iteración es el bucle siguiente, como se explica en el ejercicio 3, al haber usado `collapse(2)` para combinar ambos bucles en uno solo, podemos declarar los hilos en el primer bucle. Como anotación, `collapse()` solo se puede usar si no hay instrucciones entre los n bucles a colapsar.
- Como variables privadas tenemos los iteradores que cada hilo va a necesitar de forma privada para recorrer los bloques, por ello declaramos todos los iteradores y sum para que el resultado de un valor particular de un hilo no se vea alterado por otro hilo.
- Para conseguir que a cada hilo se le asigne un bloque, usamos, de la misma manera que con las filas, `static, 1` por los mismos motivos que los referenciados en el ejercicio 1, la carga de trabajo es constante, aunque si usamos 16 hilos, el hardware no lo es.

5.1. Corrección de los resultados

Para garantizar que el resultado es correcto y que los hilos son asignados a los bloques, tenemos nuestro modo DEBUG que nos indica qué hilo hace, qué bloque y si el resultado de la multiplicación es correcto:

```
./MulMatCuaBlo 4 2 16
Soy el Hilo 0 haciendo el BLOQUE (0, 0)
Soy el Hilo 3 haciendo el BLOQUE (2, 2)
Soy el Hilo 1 haciendo el BLOQUE (0, 2)
Soy el Hilo 2 haciendo el BLOQUE (2, 0)

MULTIPLICACION: OK

Matriz A
0.8402 0.3944 0.7831 0.7984
0.9116 0.1976 0.3352 0.7682
0.2778 0.5540 0.4774 0.6289
0.3648 0.5134 0.9522 0.9162
Matriz B
0.6357 0.7173 0.1416 0.6070
0.0163 0.2429 0.1372 0.8042
0.1567 0.4009 0.1298 0.1088
0.9989 0.2183 0.5129 0.8391
Matriz C Resultado
1.4608 1.1867 0.6843 1.5823
1.4027 1.0040 0.5938 1.3933
0.8886 0.6625 0.4999 1.1937
1.3047 0.9681 0.7156 1.5067
```

Hemos decidido eliminar los resultados de cada bloque debido a que con los hilos compitiendo por el recurso único que es la pantalla, el resultado era más bien convulso y con su posición inicial podemos identificar qué bloque ha sido asignado a cada hilo de manera satisfactoria.

Para la comprobación numérica tenemos nuestro programa `MulMatCuaTest.c` que realiza la comprobación de los resultados para un rango de tamaños de matrices con bloques de tamaño múltiplo de dicho tamaño de matriz y número de hilos de 1 a 16.

6. Haz un estudio experimental de las prestaciones de tu implementación paralela MulMatCuaBlo.c. Es decir, para cada tamaño de matriz de un conjunto dado, dim_mat_n=512,1024,2048, compara razonadamente los tiempos de ejecución para todas las combinaciones de:

- Tamaño de bloque, tam_blo_b=4,16,32.
- Número de hilos generados, num_hil_t=4,8,16

Ubicación: /6_Estudio5

De forma similar al estudio previo del ejercicio 2, vamos a dividir el impacto de cada parámetro en un apartado en el que comentaremos su impacto individual sobre el rendimiento y su relación con otros parámetros, además de analizar una gráfica con los rendimientos medios para cada parámetro y sus posibles valores. Para ello hemos realizado todas las posibles ejecuciones con las distintas configuraciones y estos son los resultados:

Configuración		**Tiempo Medio (segundos) por Tamaño de Matriz**		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	0.0711	0.6322	10.0397
4	8	0.0684	0.5042	6.6293
4	16	0.0566	0.4123	4.9577
16	4	0.0729	0.6616	7.9344
16	8	0.0648	0.4816	4.9597
16	16	0.0487	0.4043	4.4453
32	4	0.0729	0.6622	5.6743
32	8	0.0647	0.5093	4.1961
32	16	0.0506	0.3969	4.3351

Tabla 3: Tiempo Medio por Tamaño de Matriz y Configuración para MulMatCuaBlo.c

Configuración		GFlops por Tamaño de Matriz		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	3.7711	3.3952	1.7108
4	8	3.9231	4.2570	2.5909
4	16	4.7346	5.2055	3.4645
16	4	3.6808	3.2445	2.1647
16	8	4.1388	4.4565	3.4630
16	16	5.5099	5.3096	3.8638
32	4	3.6794	3.2412	3.0269
32	8	4.1427	4.2142	4.0932
32	16	5.2997	5.4080	3.9620

Tabla 4: GFlops por Tamaño de Matriz y Configuración para MulMatCuaBlo.c

Número de Hilos (t)

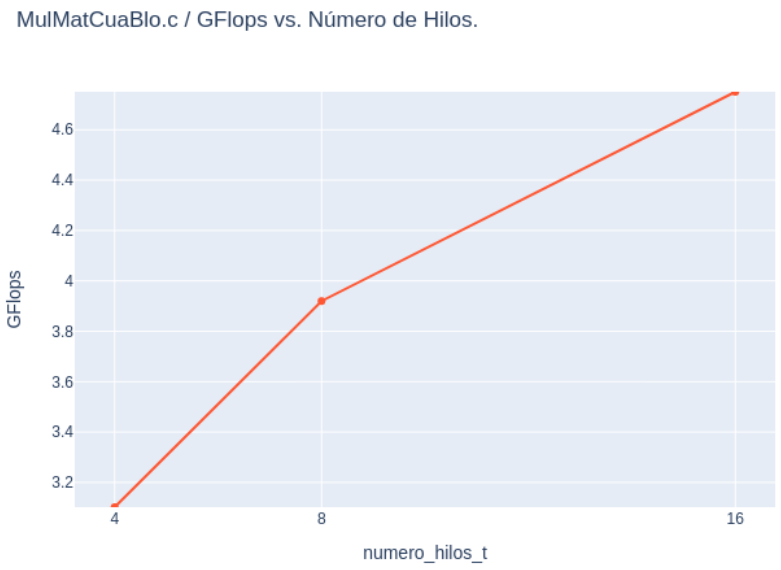

Figura 6: Gráfica FLOPS con número de hilos variable para MulMatCuaBlo.c

De nuevo el número de hilos aumenta el rendimiento como ningún otro parámetro, pasando de alrededor de 3 GFLOPS con 4 hilos hasta los más de 4.6 que alcanzamos con 16 consiguiendo un speedUp de alrededor de 1.5x. Hemos de considerar que la diferencia de speedUp se reduce respecto a la versión de filas consecutivas, ya que en esa, partíamos de 1 hilo, algo que veremos en detalle en la comparativa en el siguiente ejercicio.

Sin embargo, tenemos una diferencia nada despreciable de 1.5 GFLOPS entre usar 4 y 16 hilos.

Respecto a la relación con otros parámetros, vemos que los mejores resultados para

16 hilos, es decir, los mejores resultados por lo general, van acompañados de tamaños de los dos tamaños de matriz más pequeños debido a la relación con el subsistema de caché como era de esperar.

En cuanto al **tamaño de bloque para 16 hilos**, parece que los mejores resultados se logran con tamaño 16 para la matriz de 512 y después se logra una ligera mejoría al usar tamaño de bloque 32.

Tamaño de bloque

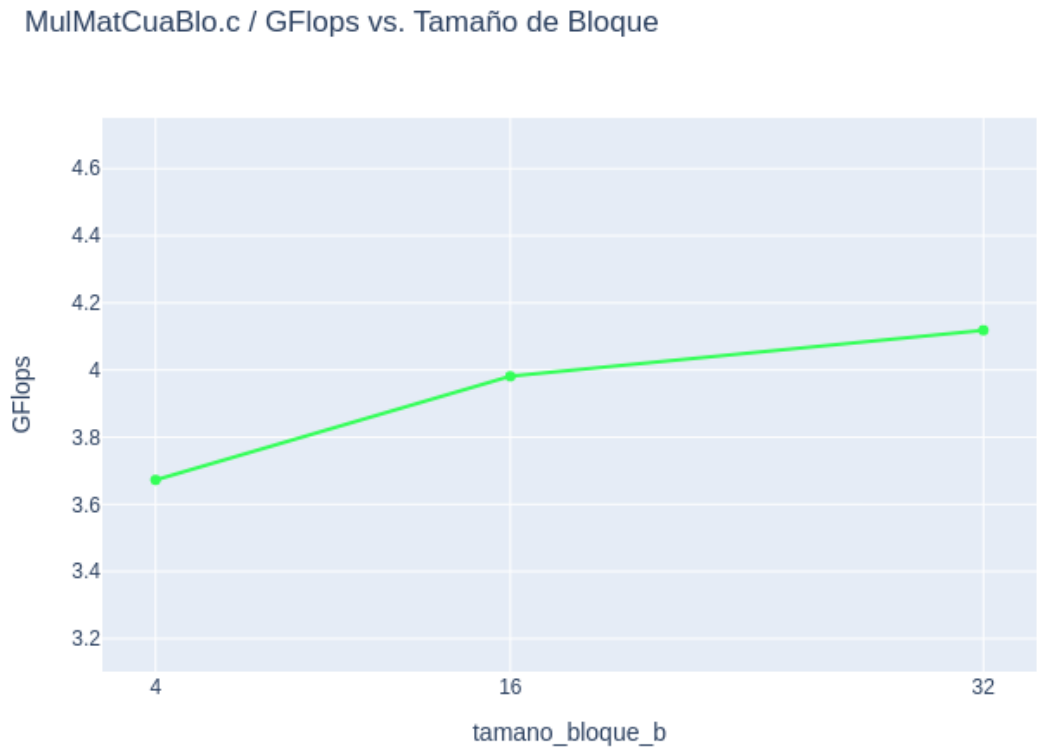

Figura 7: Gráfica FLOPS con tamaño de bloque variable para MulMatCuaBlo.c

Al aumentar el tamaño de bloque en operaciones de multiplicación de matrices, inicialmente se observa una mejora en el rendimiento, medido en GFLOPS (miles de millones de operaciones de punto flotante por segundo). Sin embargo, esta mejora disminuye con bloques más grandes, logrando un incremento promedio de 0.4 GFLOPS del menor a mayor tamaño de bloque.

Un análisis detallado muestra que, para matrices de 512 y de 1024 el tamaño de bloque apenas es de importancia, puesto que, como se ve en el ejercicio 2, **todos los datos para operar las matrices en su totalidad caben en L1+L2+L3 relativamente bien.** Por lo que usar blocking no es un factor muy relevante.

Al aumentar a mayor tamaño (2048), cambiar de un tamaño de bloque pequeño (4)

a uno más grande (16 y 32), si aumenta el rendimiento de forma notable, en particular, al usar 4 hilos se puede incrementar el rendimiento hasta en 1.3 GFLOPS, por lo que a continuación siempre nos referiremos a tamaño de matriz = 2048 a no ser que se diga lo contrario.

El beneficio anterior existe al aumentar el número de hilos, pero se reduce, lo cual se atribuye a la gestión de la memoria caché, ya que, al usar menos hilos, tenemos más tamaño de caché por hilo, independientemente de si usamos los cores más o menos potentes, y esta se reduce al aumentarlo, sin embargo, el trabajo producido por los hilos contrarresta esta pérdida.

En un caso ideal, con 4 hilos se optimiza el uso de la caché L1 y L2, dado que se pueden utilizar los 4 núcleos principales de la máquina, maximizando el espacio de caché disponible por hilo, aunque no puedo garantizar qué hilos se usan durante la ejecución.

Sin embargo, eso no quita que elección óptima del tamaño de bloque dependa de la capacidad de alojar los datos necesarios para la multiplicación de un bloque en la caché L1 o L2 y hacer uso de ellos lo suficiente antes de traer los datos del siguiente bloque. Es por ello que aunque bloques de 4x4 cabrían fácilmente en la caché L1, este enfoque requeriría frecuentes accesos a memoria para traer nuevos bloques que podrían machacar información que se vaya a usar próximamente, lo que reduce la eficiencia debido a esos fallos de caché, especialmente con matrices grandes.

Al incrementar el número de hilos a 16, el máximo soportado por la arquitectura en estudio, el rendimiento no mejora significativamente al aumentar el tamaño de bloque en comparación con el uso de 4 hilos.

Esto se debe a que la información necesaria para calcular bloques más grandes no cabe completamente en las cachés L1+L2 de los hilos* y la caché L3, compartida por muchos hilos, se satura con más facilidad al tener que estar trayendo datos de muchos hilos que ahora compiten por el espacio en L3, limitando el beneficio del bloqueo (blocking) en el rendimiento en estas matrices y haciendo que no se alcancen las cotas de los tamaños de matriz más pequeños.

En resumen, el rendimiento de las operaciones de multiplicación de matrices utilizando blocking y múltiples hilos depende críticamente del tamaño de bloque elegido y la arquitectura de la memoria caché del sistema. Es esencial seleccionar un tamaño de bloque que permita un uso eficiente de la caché para maximizar el rendimiento.

**NOTA: Si queremos comprobar si los datos para operar un bloque entran en una caché:*

```
T_DATOS=[(T_BLOQUE*T_MATRIZ)*2+(T_BLOQUE*T_BLOQUE)]*BYTES_DATATYPE(double = 8)
T_DATOS_2048x32=((32*2048)*2+(32*32))*8 = 1056768 bytes = 1032 Kb
```

Tamaño de Matriz

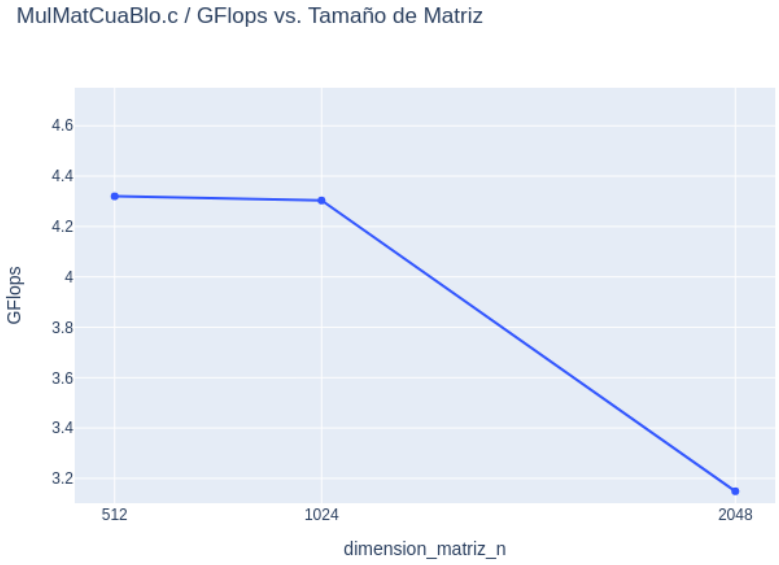

Figura 8: Gráfica FLOPS con tamaño de matriz variable para MulMatCuaBlo.c

El tamaño de matriz ya ha sido comentado bastante en el apartado anterior debido a su relación con el tamaño de bloque y eso mismo que comentábamos se ve muy bien en esta gráfica, puesto que independientemente del número de hilos y tamaño de blocking, seguimos teniendo una caída pronunciada de rendimiento al pasar de 1024 a 2048 debido a que el sistema de caché no es capaz de contener toda la información para bloques lo suficientemente grande como para aprovechar la localidad espacial y temporal con tamaños de matriz grandes al usar múltiples hilos.

Por lo tanto, aunque hagamos blocking, vemos que el tamaño de nuestro subsistema de caché y el tamaño total de las matrices a multiplicar siguen siendo cuellos de botella en esta arquitectura.

7. Compara razonadamente los tiempos obtenidos con la versión paralela que utiliza bloques, MulMatCuaBlo.c, con los tiempos que obtuviste con la versión paralela sin bloques, MulMatCua.c

Configuración		**Tiempo Medio (segundos) por Tamaño de Matriz**		
Filas	**Hilos**	**512**	**1024**	**2048**
1	4	0.0679	0.6250	10.1957
1	8	0.0733	0.4836	7.1791
1	16	0.0506	0.3964	5.5858
4	4	0.0717	0.6208	8.0102
4	8	0.0661	0.4892	5.7740
4	16	0.0510	0.3892	5.6267
16	4	0.0706	0.6219	6.4076
16	8	0.0719	0.5037	5.8257
16	16	0.0502	0.3942	4.6105
32	4	0.0690	0.6220	7.3008
32	8	0.0745	0.5103	6.1997
32	16	0.0500	0.3845	4.6074

Tabla 5: Tiempo Medio por Tamaño de Matriz y Configuración de MulMatCua.c

Configuración		**Tiempo Medio (segundos) por Tamaño de Matriz**		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	0.0711	0.6322	10.0397
4	8	0.0684	0.5042	6.6293
4	16	0.0566	0.4123	4.9577
16	4	0.0729	0.6616	7.9344
16	8	0.0648	0.4816	4.9597
16	16	0.0487	0.4043	4.4453
32	4	0.0729	0.6622	5.6743
32	8	0.0647	0.5093	4.1961
32	16	0.0506	0.3969	4.3351

Tabla 6: Tiempo Medio por Tamaño de Matriz y Configuración de MulMatCuaBlo.c

Configuración		GFlops por Tamaño de Matriz		
Filas	**Hilos**	**512**	**1024**	**2048**
1	4	3.9489	3.4346	1.6846
1	8	3.6579	4.4381	2.3924
1	16	5.3034	5.4142	3.0749
4	4	3.7407	3.4578	2.1442
4	8	4.0565	4.3877	2.9747
4	16	5.2615	5.5143	3.0525
16	4	3.7992	3.4512	2.6805
16	8	3.7294	4.2610	2.9483
16	16	5.3390	5.4450	3.7253
32	4	3.8888	3.4511	2.3526
32	8	3.6004	4.2063	2.7704
32	16	5.3652	5.5824	3.7278

Tabla 7: GFlops por Tamaño de Matriz y Configuración de MulMatCua.c

Configuración		GFlops por Tamaño de Matriz		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	3.7711	3.3952	1.7108
4	8	3.9231	4.2570	2.5909
4	16	4.7346	5.2055	3.4645
16	4	3.6808	3.2445	2.1647
16	8	4.1388	4.4565	3.4630
16	16	5.5099	5.3096	3.8638
32	4	3.6794	3.2412	3.0269
32	8	4.1427	4.2142	4.0932
32	16	5.2997	5.4080	3.9620

Tabla 8: GFlops por Tamaño de Matriz y Configuración de MulMatCuaBlo.c

Aunque ya se han podido observar los resultados de ambos ejercicios, vamos a realizar una comparativa entre ambos, tratando de obtener ciertas conclusiones acerca de la similitud en tiempos de ambas implementaciones. Para ello, vamos a aplicar las claves del rendimiento encontradas en los análisis previos y ponerlas frente a frente para explicar por qué se obtienen ciertos resultados en cada caso.

Para ello, hemos filtrado los resultados para un hilo del primer ejercicio, pero no así los resultados para una fila consecutiva, puesto que a nivel de caché, puede resultar incluso más pesado traer los datos para operar una fila consecutiva que un bloque del mayor tamaño de bloque.

7.1. Máximo rendimiento

A diferencia de los otros análisis, debido a que hemos implementado blocking con el objetivo de maximizar el uso de la caché y aumentar el rendimiento, comenzaremos por comparar los mejores resultados a nivel de GFLOPS de cada implementación para cada tamaño de matriz:

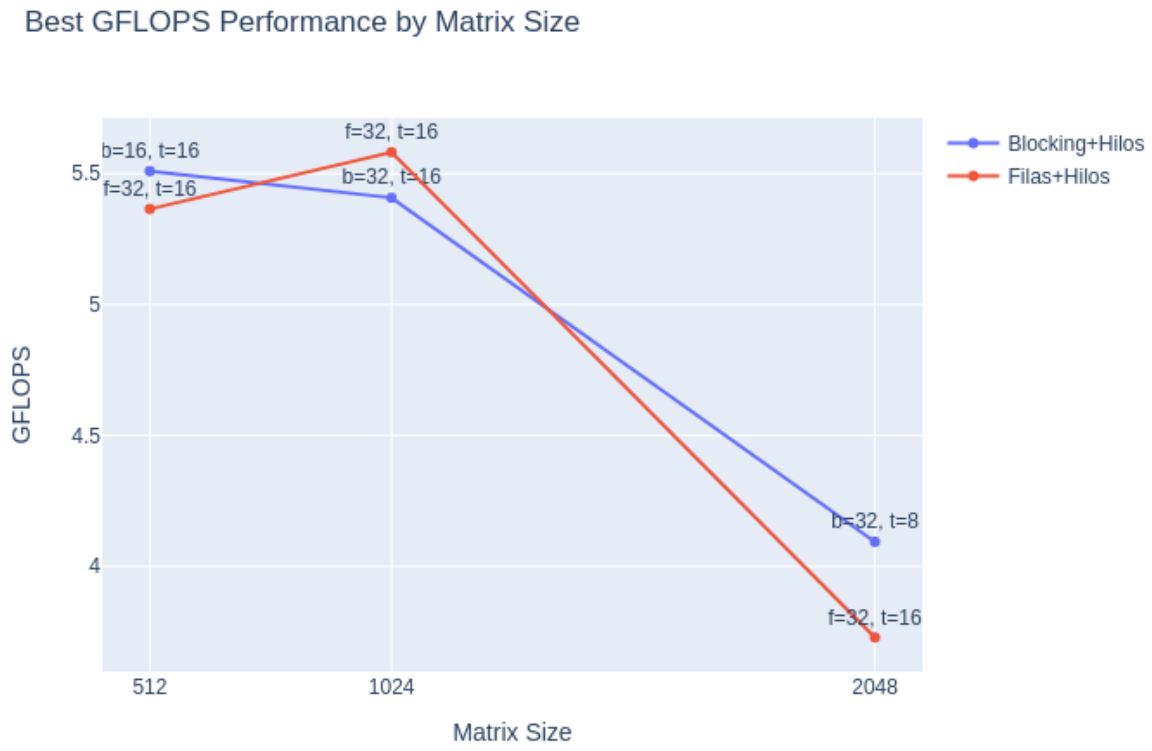

Figura 9: Comparativa de GFLOPS máximos entre MulMatCuaBlo.c (blocking+hilos) y MulMatCua.c (filas+hilos)

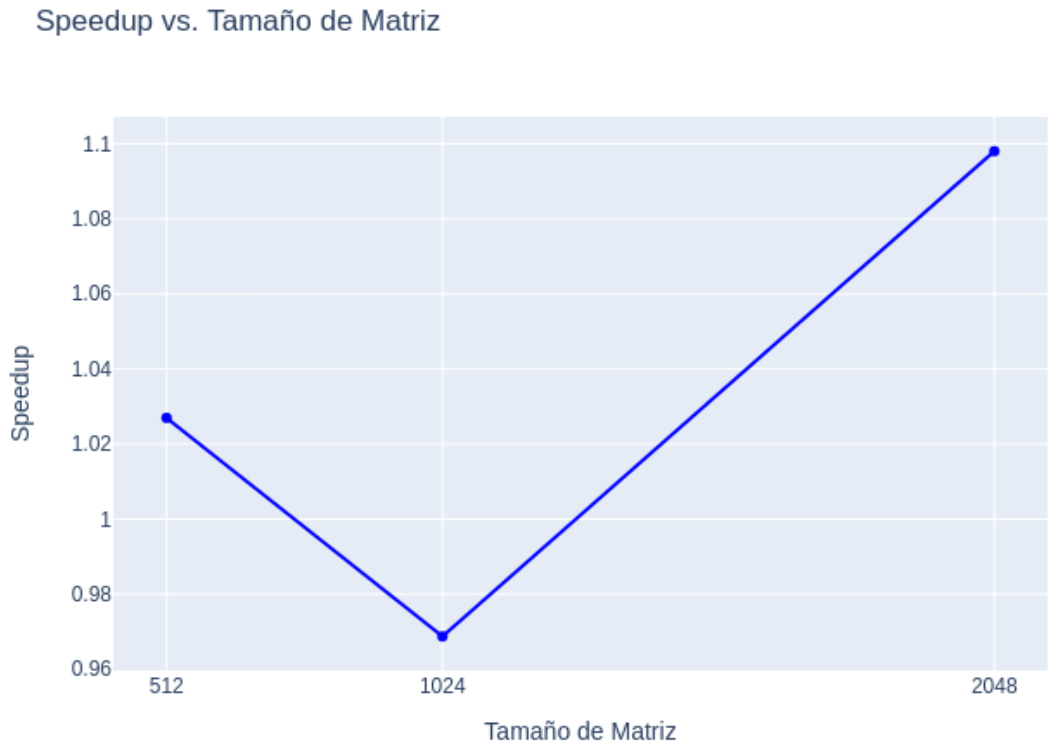

Figura 10: SpeedUp de GFLOPS máximos entre MulMatCuaBlo.c (blocking+hilos) y MulMatCua.c (filas+hilos)

Como se ve en ambas gráficas si tomamos las mejores configuraciones para cada tamaño, hay unas diferencias escasísimas para 512 y 1024, puesto que con esos tamaños de matrices, podemos guardar todos los datos necesarios para operar en caché, sin embargo, con 2048, ya hemos visto en el ejercicio anterior, en la sección del tamaño de bloque que para ciertos tamaños, no toda la información cabe en caché de forma eficiente. Es ahí, con tamaños de matriz más grandes, donde el blocking logra una diferencia que al menos podemos registrar de un speedUp de 1.1x para los mejores casos de ambos programas.

Además, se puede observar que el número de hilos que usa el blocking es de 8 por los 16 de las filas consecutivas. Es posible que con blocking la ganancia de trabajo de usar todos los hilos sea eclipsada por el overhead añadido de los nuevos hilos, lo que unido al hecho de que estos hilos son, posiblemente, los de los cores menos potentes, provoca una pérdida en global.

7.2. Número de hilos

Si comparamos por número de hilos, los resultados también son coherentes con lo que explicamos en ejercicios anteriores:

Lo primero que choca a la vista es que tenemos resultados prácticamente idénticos

Figura 11: Comparativa de GFLOPS entre MulMatCuaBlo.c (blocking+hilos) y MulMatCua.c (filas+hilos) para número de hilos variable

para 4 y 16 hilos, con la mayor diferencia dándose al usar 8. Esto emana del hecho de que con 8 hilos, como ya hemos comentado anteriormente, cada hilo tiene igual o más caché que si usamos 16, si suponemos, en nuestra arquitectura, que usamos los 4 principales con hyperthreading o los 4 principales individualmente y 4 económicos, podemos aprovechar mejor la caché por bloque, maximizando el rendimiento frente a usar 4 o 16.

Sin embargo, recordar que estos datos no tienen en cuenta que con el mayor tamaño de matriz (2048), usar blocking puede darnos una mejoría visible también con 4 hilos si usamos el tamaño de bloque más grande, pasando de 2.7 GFLOPS en la mejor configuración de la implementación por filas a 3 GFLOPS, por los motivos anteriores, lo cual nos hace intuir que el tamaño de matriz puede ser el factor de mayor importancia a la hora de ganar rendimiento usando blocking.

7.3. Tamaño de matriz

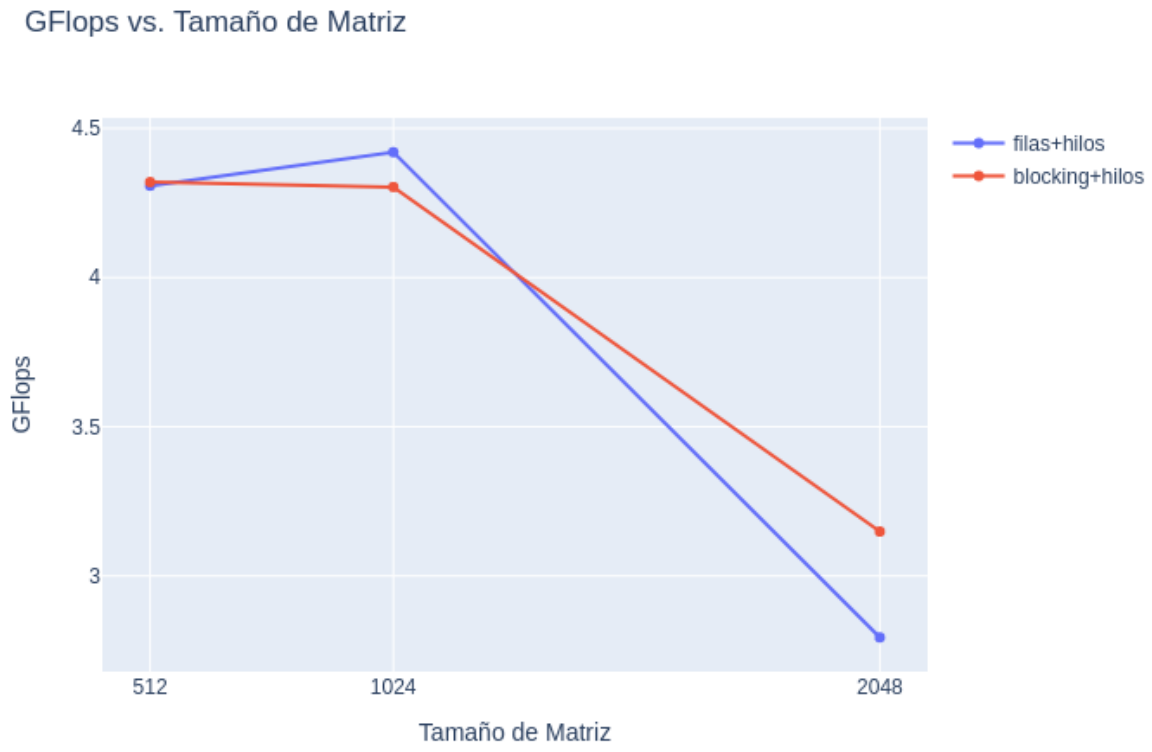

Figura 12: Comparativa de GFLOPS entre MulMatCuaBlo.c (blocking+hilos) y MulMatCua.c (filas+hilos) para tamaño de matriz variable

Como se ve en la gráfica, la mejoría de rendimiento al usar blocking o no, se manifiesta al aumentar el tamaño de matriz a uno significativo para nuestro sistema de caché y es que, por lo general, el blocking funciona mejor cuando el sistema de caché no puede manejar el acceso tradicional a caché para multiplicación de bloques, y aunque hemos visto, que este también está limitado por la caché, podemos ver con esta gráfica y la que medía el rendimiento pico de ambas implementaciones 9 que a partir de cierto tamaño, puede haber una ventaja al usar blocking.

No hemos hablado mucho de las diferencias entre usar un mayor o menor tamaño de filas consecutivas respecto a mayor o menor tamaño de bloque, debido a que no es una comparación directa, sino más bien un parámetro de la implementación. Lo que si se ve, es que en ambos casos, como se ha visto en las gráficas respectivas 7 2 y en las explicaciones previas, este parámetro no es de mucha importancia hasta que se alcanza cierto tamaño de matriz, en los cuales, nos beneficiamos de mayores tamaños para aprovechar la localidad espacial y temporal de mayores cantidades de datos, lo cual se explota más al usar blocking.

En conclusión, no se aprovecha la técnica del blocking sobre la implementación convecional hasta que se alcanza cierto tamaño de matriz, pero usar blocking tampoco produce un gran speedUp al usarse en un sistema de caché con múltiples hilos que no puede alo-

jar los datos para calcular bloques suficientemente grandes que permitan maximizar la localidad espacial y temporal.

Lógicamente con 1 hilo o números de hilos más bajos, si tenemos mayores tamaños de cachés para esos hilos, usar blocking puede darnos un speedUp notable como hemos visto en los ejemplos con 4 hilos con tamaños de bloque altos o como se vería si usasemos blocking secuencial contra filas consecutivas secuencialmente.

8. A partir de los programas anteriores, escribe un programa paralelo usando OpenMP, llamado MulMatBlo.c, para calcular el producto de dos matrices rectangulares de números reales de doble precisión, de la forma Cmxn=AmxkBkxn, utilizando un esquema por bloques.

NOTA: Suponemos (no hace falta comprobarlo en el código) que las dimensiones de las matrices (m, n y k) son todas múltiplos de la dimensión del bloque cuadrado, b. Es decir, se cumple que m=eb, n=gb y k=hb; siendo e, g y h números enteros.

SINTAXIS:

```
MulMatBlo <dim_mat_m><dim_mat_n><dim_mat_k><tam_blo_b><num_hil_t>
```

Ubicación: /8_MulMatBlo

Al haber planteado el ejercicio 5 con un algoritmo que acepta matrices rectangulares, pero usado para matrices cuadradas únicamente, este programa solo sustituye la entrada y su procesamiento para multiplicación de matrices cuadradas por rectangulares y utiliza estas dimensiones en lugar de utilizar el tamaño de matriz como el valor de m, n y k y mm, se mantiene exactamente igual que en el ejercicio 5.

8.1. Corrección de los resultados

Para asegurar la corrección de los resultados de nuevo, comprobamos contra nuestra implementación secuencial usando el programa `MulMatBloTest.c` si queremos unos valores fijos o el programa `MulMatBloTest2.c` si queremos comprobar contra todas las combinaciones posibles de m, n, k, tamaño de bloque y número de hilos hasta cierto límite.

9. A partir de los programas anteriores, escribe un programa paralelo usando tareas de OpenMP, llamado TMulMatCuaBlo.c, para calcular el producto de dos matrices cuadradas de números reales de doble precisión, de la forma Cnxn=AnxnBnxn, utilizando un esquema por bloques. Explica cómo has distribuido el trabajo entre las tareas.

NOTA: Suponemos (no hace falta comprobarlo en el código) que la dimensión de la matriz cuadrada, n, es un múltiplo de la dimensión del bloque cuadrado, b, es decir, se cumple que n=eb, siendo e un número entero.

SINTAXIS:

```
TMulMatCuaBlo <dim_mat_n><tam_blo_b><num_hil_t>
```

Como en los programas anteriores, mantenemos la misma estructura del programa y aunque el enunciado nos pide que el programa use matrices cuadradas, partiremos de la última implementación con matrices rectangulares, blocking y paralelización con hilos del ejercicio anterior, puesto que modificando levemente la entrada de datos y el tratamiento de esta, podemos conseguir que la implementación sea transparente al usuario.

A continuación se muestra la función mm, que realiza la multiplicación de tareas utilizando blocking y tareas, de forma que cada bloque sea asignado a una tarea:

```
1  // A = Matriz A, B = Matriz B, C = Matriz C, m = filas de A,
2  // n = columnas de A y filas de B, k = columnas de B y tam_blo_b = tamaño del bloque
3  void mm(double *A, double *B, double *C, int m, int n, int k, int tam_blo_b, int num_
       ↪ hilos)
4  {
5      int ldm = n;
6      int fila, col, fila_bloque, col_bloque, l;
7      double sum;
8
9      int iam;
10     omp_set_num_threads(num_hilos);
11
12     //TASK PARALLELISM
13
14     #pragma omp parallel
15     {
16         /*single indica que el bloque de código siguiente debe ser ejecutado por un
               ↪ solo hilo.
17         nowait sirve para que los otros hilos no necesiten esperar a que el hilo de
               ↪ single termine,
18         pueden continuar con la ejecución del código que sigue después del bloque
               ↪ single.*/
19         #pragma omp single nowait
20         {
21             for (fila = 0; fila < m; fila += tam_blo_b)
22             {
```

```
23                for (col = 0; col < k; col += tam_blo_b)
24                {
25                    /*task firstprivate define una tarea paralela (task) donde las
                          ↪ variables
26                    fila, col, fila_bloque, col_bloque, l, sum son privadas (
                          ↪ firstprivate) para cada tarea.
27                    firstprivate garantiza que el valor de las variables para cada hilo
                          ↪ será*/
28                    #pragma omp task firstprivate(fila, col, fila_bloque, col_bloque, l,
                          ↪  sum)
29                    {
30                        iam = omp_get_thread_num();
31                        printf("Soy el Hilo %d haciendo el BLOQUE (%d, %d)\n", iam, fila
                              ↪ , col);
32                        for (fila_bloque = fila; fila_bloque < fila + tam_blo_b; fila_
                              ↪ bloque++)
33                        {
34                            for (col_bloque = col; col_bloque < col + tam_blo_b; col_
                                  ↪ bloque++)
35                            {
36                                sum = 0.0;
37                                for (l = 0; l < ldm; l++)
38                                {
39                                    sum += A[fila_bloque * ldm + l] * B[l * k + col_bloque
                                          ↪ ];
40                                }
41                                C[fila_bloque * k + col_bloque] = sum;
42                            }
43                        }
44                    }
45                }
46
47            }
48        }
49    }
50
51}
```

Como vemos, aunque la cabecera de la función se mantenga igual, se producen ciertos cambios a la hora de instaurar el paralelismo de tareas.

En primer lugar, usamos `pragma omp parallel` para declarar el paralelismo de forma que comprenda al `pragma omp single nowait` y todos los bucles, puesto que queremos que el hilo que ejecute el single pueda repartir todos los bloques y haya una barrera implícita al final de todos los bucles donde los hilos esperen cuando terminen su ejecución.

Por ello, ponemos `pragma omp single nowait` alrededor del bucle más externo, para que pueda recorrer todas las instancias de bloques por fila y columna y crear una tarea que será asignada a uno de los hilos restantes que se han quedado ociosos, al final del single, de forma que no hayan de esperar a que el single termine de repartir los bloques para continuar su ejecución.

Para crear dicha tarea enclosamos los tres bucles que realizan la multiplicación de matrices con `pragma omp task firstprivate(fila, col, fila_bloque, col_bloque, l,`

`sum)`, de esta forma conseguimos que se le asigne un trabajo independiente a cada tarea.

Si nos fijamos, esta paralelización en forma de tarea diferente a paralelizar un un bucle for, en el que podemos poner un pragma encima de él y todas las iteraciones se asignaran a un hilo.

Mientras que, usando tareas, tenemos que, dentro del bucle cuyas iteraciones queremos paralelizar, asignar cada iteración independiente, en este caso, cada multiplicación de un bloque* a cada hilo.

Sin embargo, las variables privadas a cada hilo se mantienen igual, puesto que necesitamos que cada tarea mantenga sus propias copias inicializadas a los valores que el single puso de todos los iteradores y la variable suma, que sirve de acumulador para el resultado de un valor de bloque como ya se explicó.

**NOTA: A su vez, la multiplicación de bloque tiene sus tres bucles internos para realizar la multiplicación de matrices en sí, es esta parte la que queremos paralelizar para que cada bloque sea asignado a una tarea.*

9.1. Corrección de los resultados

Para ver si el reparto de tareas y bloques es correcto, además del resultado de una multiplicación particular activamos el modo DEBUG definiendo la macro y podemos verificar el correcto reparto de bloques a cada tarea como se ve en este ejemplo:

```
./TMulMatCuaBlo 4 2 16
Tarea creada para Hilo 15 y BLOQUE (2, 0)
Tarea creada para Hilo 0 y BLOQUE (0, 2)
Tarea creada para Hilo 3 y BLOQUE (0, 0)
Tarea creada para Hilo 11 y BLOQUE (2, 2)

MULTIPLICACION: OK

Matriz A
0.8402 0.3944 0.7831 0.7984
0.9116 0.1976 0.3352 0.7682
0.2778 0.5540 0.4774 0.6289
0.3648 0.5134 0.9522 0.9162
Matriz B
0.6357 0.7173 0.1416 0.6070
0.0163 0.2429 0.1372 0.8042
0.1567 0.4009 0.1298 0.1088
0.9989 0.2183 0.5129 0.8391
Matriz C Resultado
1.4608 1.1867 0.6843 1.5823
1.4027 1.0040 0.5938 1.3933
0.8886 0.6625 0.4999 1.1937
1.3047 0.9681 0.7156 1.5067
```

Se informa de cada hilo para verificar que los hilos asignados a una tarea pueden cambiar de una ejecución a otra, ya que la tarea se asigna al primer hilo preparado en el

momento de la asignación por el hilo que ejecuta la sección single del código.

La corrección numérica de los resultados se hace con, como no, `TMulMatCuaBloTest.c` que, de hecho, mantiene la misma estructura en el main que `TMulMatCuaBloTest.c` y funciona de la misma forma.

10. Haz un estudio experimental de las prestaciones de tu implementación paralela TMulMatCuaBlo.c. Es decir, para cada tamaño de matriz de un conjunto dado, dim_mat_n=512,1024,2048, compara razonadamente los tiempos de ejecución para todas las combinaciones de:

- Tamaño de bloque, tam_blo_b=4,16,32.
- Número de hilos generados, num_hil_t=4,8,16.

Ubicación: /10_Estudio9

De la misma manera que en otros casos, vamos a realizar las ejecuciones para todos los casos, tabularlas y obtener sus tiempos y GFLOPS. Una vez distribuidas las gráficas, analizaremos los distintos parámetros y sus relaciones.

Configuración		**Tiempo Medio (segundos) por Tamaño de Matriz**		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	0.0687	0.6027	18.4816
4	8	0.0466	0.4002	7.7385
4	16	0.0378	0.3137	5.0388
16	4	0.0682	0.6156	6.4051
16	8	0.0429	0.3929	3.9116
16	16	0.0304	0.3111	4.2693
32	4	0.0673	0.6372	5.7887
32	8	0.0513	0.4063	4.5558
32	16	0.0295	0.3496	4.1834

Tabla 9: Tiempo Medio por Tamaño de Matriz y Configuración de TMulMatCuaBlo.c

Configuración		GFlops por Tamaño de Matriz		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	3.9054	3.5613	0.9293
4	8	5.7578	5.3629	2.2195
4	16	7.1014	6.8431	3.4087
16	4	3.9302	3.4869	2.6816
16	8	6.2490	5.4630	4.3909
16	16	8.8128	6.8992	4.0230
32	4	3.9852	3.3684	2.9671
32	8	5.2249	5.2835	3.7701
32	16	9.0788	6.1402	4.1056

Tabla 10: GFlops por Tamaño de Matriz y Configuración de TMulMatCuaBlo.c

Número de hilos

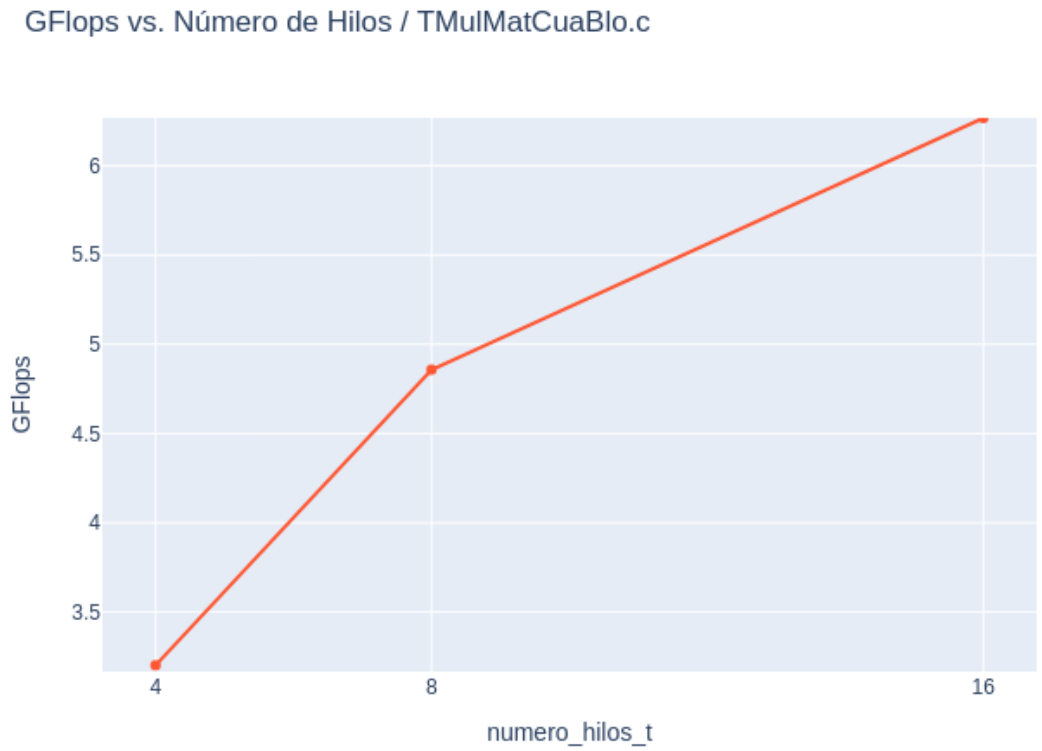

Figura 13: GFlops de TMulMatCuaBlo.c para número de hilos variable

Al igual que en sus homólogos anteriores, tanto usando bloques como sin ellos, a mayor número de hilos, mayor rendimiento. Sin embargo, y como también hablaremos más adelante, el speedUp de pasar de 4 hilos a 8 y luego a 16 es bastante alto y es que, aunque es normal que haya un aumento, se suele reducir más rápido debido al overhead,

cachés más pequeñas, etcétera.

Parece que al usar tareas, hemos desbloqueado cierto rendimiento extra. Este está relacionado con el número de hilos, evidentemente, más hilos ofrecen más rendimiento en esta arquitectura, pero parece que debido al reparto de tareas, que se hace de forma dinámica, podemos conseguir un mejor rendimiento al no asignar estáticamente, como en los ejercicios anteriores los bloques, sino dinámicamente al primer ejercicio listo usando uno de los hilos (el que ejecuta el single) como creador de las tareas.

Y al aumentar el número de hilos, pero no asignar la misma carga a todos los hilos, que, como hemos visto en la arquitectura, no son iguales, podemos conseguir que los hilos más rápidos (los de los 4 cores más potentes) puedan hacer trabajo en lugar de estar esperando a que el resto de hilos (los de los cores menos potentes) terminen, lo cual podía producir cierto cuello de botella.

Una anotación importante es que aunque schedule(dynamic, 1) tambiá asignaría cada iteración (bloque) a un hilo, los resultados pueden no ser iguales debido a que el reparto de tareas e iteraciones respectivamente no es igual.

Usando tareas, un hilo se encarga de crear todas las tareas y el sistema las asigna al hilo que considera adecuado y se encuentre listo sin tener que esperar a que todas las tareas estén creadas. Mientras, en schedule(dynamic, 1), tenemos que la iteración se asigna al primer hilo disponible y puede haber cierto overhead al tener que realizar la asignación de la iteración, mientras que las tareas tienen más overhead al crear las tareas, pero una vez creadas, su asignación puede ser más eficiente.

Tamaño de bloque

El tamaño de bloque para TMulMatCuaBlo.c es muy interesante, puesto que según nos muestra la gráfica, el tamaño que mejor rendimiento medio ofrece para todas las configuraciones es 16. Como hemos comentado antes, un tamaño de bloque muy pequeño puede causar fallos de caché frecuentes y un tamaño de bloque alto puede implicar que la caché no pueda guardar toda la información necesaria por bloque.

Al parecer, usando tareas y aprovechando mejor los hilos, el tamaño de 16 puede ser un mejor compromiso para todas las configuraciones, puesto que consigue un equilibrio entre aprovechar la caché y aprovechar la información por bloque.

Si nos fijamos en la tabla de GFlops, tenemos que usando tamaño de bloque = 16 se obtienen resultados que son casi idénticos a los mejores de cada configuración o los mejores en cuanto a tamaño de matriz e hilos, por lo que aunque en tamaños de matriz más grandes con más hilos, tamaño de bloque=32 funcione mejor, con tamaños de matriz más pequeños y menores hilos, se consigue más rendimiento aprovechando este equilibrio. Más sobre esto en la comparativa del siguiente ejercicio.

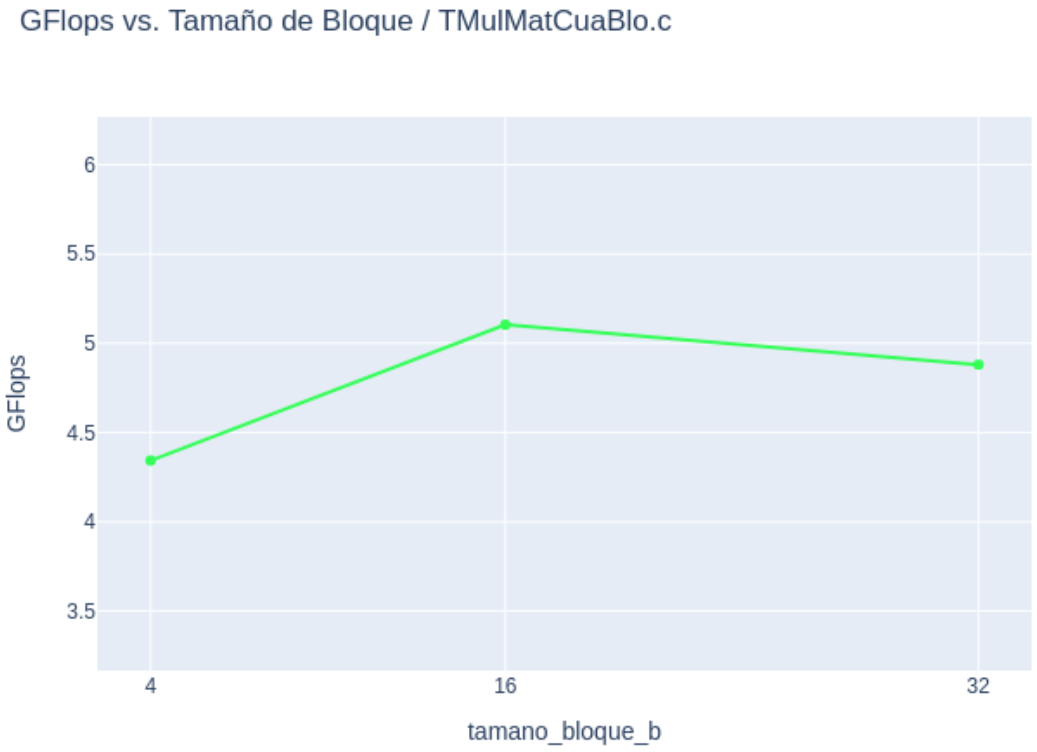

Figura 14: GFlops de TMulMatCuaBlo.c para tamaño de bloque variable

Tamaño de matriz

Como viene siendo habitual, los mejores resultados se obtienen con los tamaños de matriz más pequeños, pero se pueden apreciar algo muy interesante en esta gráfica.

En primer lugar, el rendimiento medio obtenido al utilizar tamaño de matriz = 512 es muy alto. Esto puede estar relacionado con el mejor uso de los hilos que comentábamos antes y al utilizar los hilos a su máximo rendimiento con el tamaño de matriz más pequeño, que, por lo tanto, permite almacenar la información de los bloques en caché fácilmente y del problema en general, podemos reducir los fallos de caché y obtener un alto rendimiento. Sin embargo, como decíamos antes, esta mejora viene del uso más eficiente de los hilos mediante tareas, no del blocking y su relación con el tamaño de matriz, al menos no en su mayoría.

Este efecto va disminuyendo según avanzamos en el tamaño de matriz, puesto que para 1024 obtenemos buenos resultados que, como veremos en la comparación, son superiores a sus homólogos sin tareas y para 2048, parece que las ganancias se han vuelto muy pequeñas.

Esto podría venir de que al tener tamaños de matriz más grandes, el número de fallos de caché aumenta y se vuelve el factor limitante de nuestra mejora, ya que ahora el tiempo de ejecución parece más marcado por dichos fallos que por la alta latencia de algunos hilos

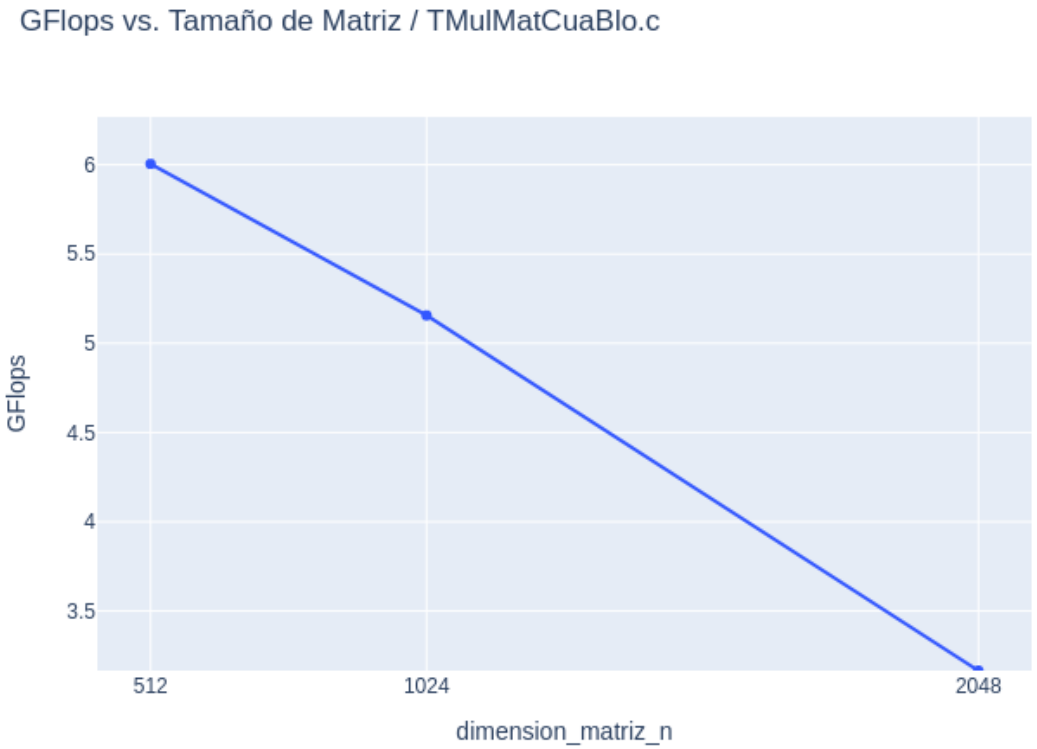

Figura 15: GFlops de TMulMatCuaBlo.c para tamaño de matriz variable

cuyo peor rendimiento puede ocultarse tras dichos fallos.

En conclusión, parece que en nuestra arquitectura se puede obtener un notable mayor rendimiento usando tareas y blocking siempre y cuando no nos encontremos con tamaños de matriz más grandes de lo que nuestra caché pueda manejar. Esto puede estar relacionado no con que las tareas repartidas sean desiguales, sino con que en este equipo en particular algunos hilos rinden peor que otros, creando diferentes tiempos de ejecución para las "mismas"tareas y favoreciendo este esquema sobre la asignación de hilos.

11. Compara razonadamente los tiempos obtenidos con las dos versiones paralelas que utilizan bloques. Es decir, compara la versión que usa tareas, TMulMatCuaBlo.c y la que no usa tareas, MulMatCuaBlo.c.

Configuración		**Tiempo Medio (segundos) por Tamaño de Matriz**		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	0.0711	0.6322	10.0397
4	8	0.0684	0.5042	6.6293
4	16	0.0566	0.4123	4.9577
16	4	0.0729	0.6616	7.9344
16	8	0.0648	0.4816	4.9597
16	16	0.0487	0.4043	4.4453
32	4	0.0729	0.6622	5.6743
32	8	0.0647	0.5093	4.1961
32	16	0.0506	0.3969	4.3351

Tabla 11: Tiempo Medio por Tamaño de Matriz y Configuración para MulMatCuaBlo.c

Configuración		**Tiempo Medio (segundos) por Tamaño de Matriz**		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	0.0687	0.6027	18.4816
4	8	0.0466	0.4002	7.7385
4	16	0.0378	0.3137	5.0388
16	4	0.0682	0.6156	6.4051
16	8	0.0429	0.3929	3.9116
16	16	0.0304	0.3111	4.2693
32	4	0.0673	0.6372	5.7887
32	8	0.0513	0.4063	4.5558
32	16	0.0295	0.3496	4.1834

Tabla 12: Tiempo Medio por Tamaño de Matriz y Configuración de TMulMatCuaBlo.c

Configuración		GFlops por Tamaño de Matriz		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	3.9054	3.5613	0.9293
4	8	5.7578	5.3629	2.2195
4	16	7.1014	6.8431	3.4087
16	4	3.9302	3.4869	2.6816
16	8	6.2490	5.4630	4.3909
16	16	8.8128	6.8992	4.0230
32	4	3.9852	3.3684	2.9671
32	8	5.2249	5.2835	3.7701
32	16	9.0788	6.1402	4.1056

Tabla 13: GFlops por Tamaño de Matriz y Configuración de TMulMatCuaBlo.c

Configuración		GFlops por Tamaño de Matriz		
Tamaño de Bloque	**Hilos**	**512**	**1024**	**2048**
4	4	3.7711	3.3952	1.7108
4	8	3.9231	4.2570	2.5909
4	16	4.7346	5.2055	3.4645
16	4	3.6808	3.2445	2.1647
16	8	4.1388	4.4565	3.4630
16	16	5.5099	5.3096	3.8638
32	4	3.6794	3.2412	3.0269
32	8	4.1427	4.2142	4.0932
32	16	5.2997	5.4080	3.9620

Tabla 14: GFlops por Tamaño de Matriz y Configuración para MulMatCuaBlo.c

11.1. Máximo rendimiento

Como en la última comparativa, comenzaremos por comparar los mejores resultados a nivel de GFLOPS de cada implementación para diferente tamaño de matriz:

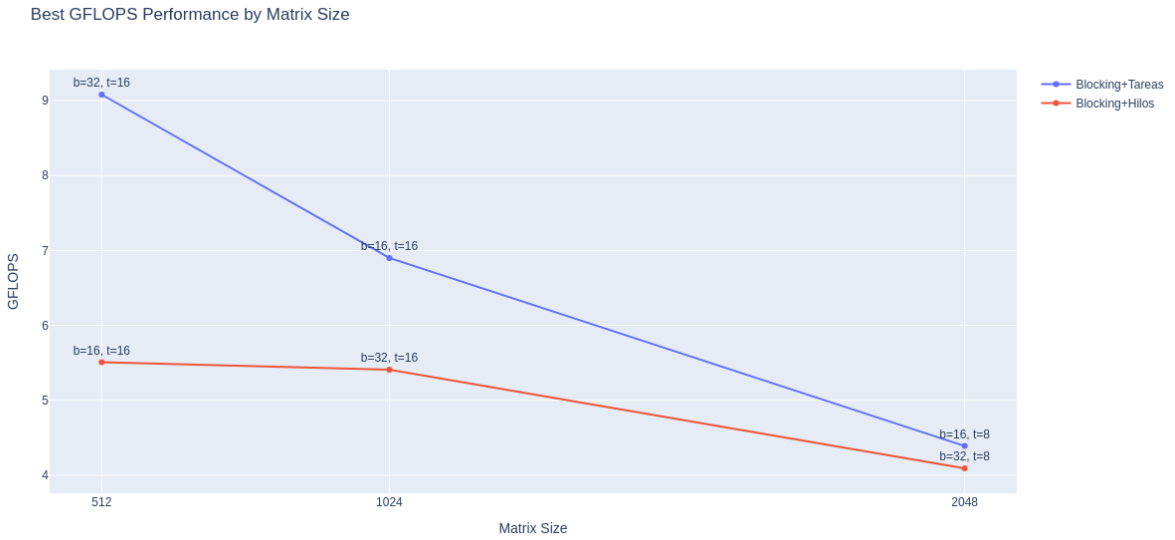

Figura 16: Comparativa de GFLOPS máximos entre TMulMatCuaBlo.c (blocking+hilos) y TMulMatCuaBlo.c (blocking+tareas)

Como intuíamos del ejercicio anterior, los resultados para tamaños de matrices más pequeños, en particular para 512, se disparan al usar tareas por los motivos ya discutidos en dicho ejercicio.

Y como también se introdujo y se ve especialmente bien en estas gráficas, a mayor tamaño, menor diferencia se produce al usar las tareas, puesto que la ganancia del reparto de trabajo se va apagando según el tamaño de matriz aumenta y con él los fallos de caché.

También se ve que para los diferentes tamaños aparecen diferentes combinaciones para obtener el máximo rendimiento. Mientras que el blocking con hilos hace uso de los bloques de 16 porque quizás le permita tener un tiempo de trabajo más similar a todos los hilos al ser más pequeño, blocking+tareas usa tamaño de bloque = 32. Para 1024 se intercambian los valores, puesto que 16 ofrece un mayor equilibrio entre aprovechar la caché y hacer mejor uso del subsistema y finalmente al llegar a 2048 ambos obtienen sus mejores resultados con 8 hilos. Este suceso puede darse debido a que, al igual que en el ejercicio 7, el overhead de tener 8 hilos más añadido, al hecho de que estos hilos puede que sean los basados en los cores menos potentes, puede que añada una latencia que sobrepase la ganancia de tener más hilos tanto para tareas como para hilos como pasaba en el ejercicio 7.

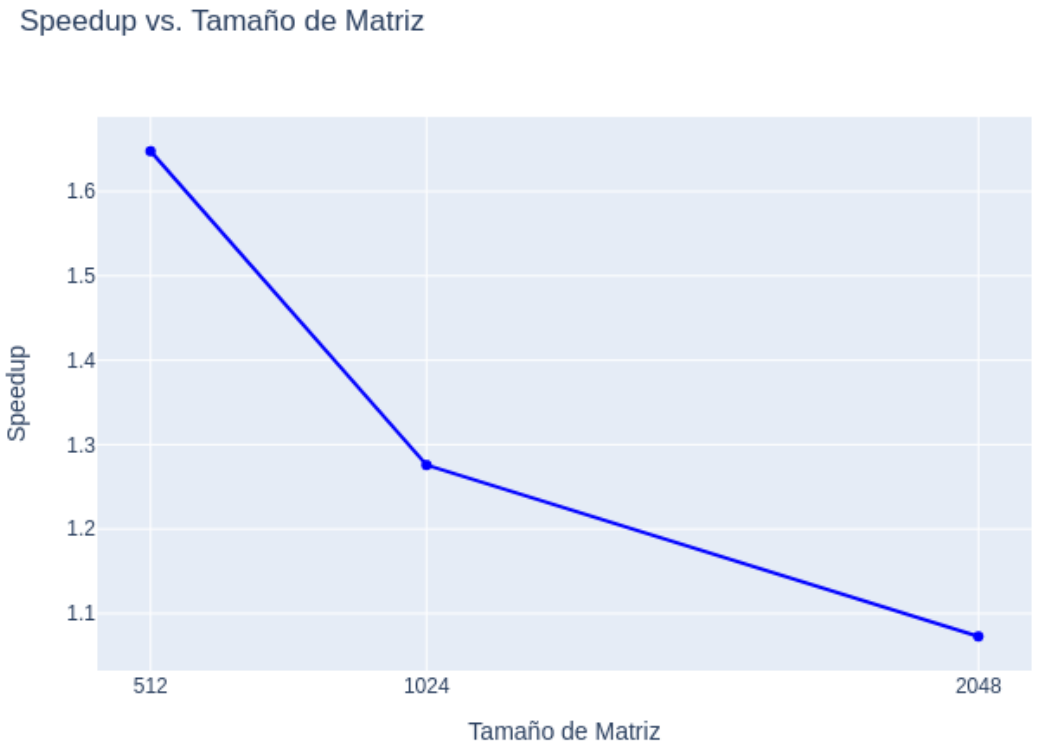

Figura 17: speedUp entre máximo rendimiento de TMulMatCuaBlo.c (blocking+hilos) y TMulMatCuaBlo.c (blocking+tareas)

11.2. Número de hilos

Figura 18: Comparativa de GFLOPS entre TMulMatCuaBlo.c (blocking+tareas) y MulMatCuaBlo.c (blocking+hilos) para número de hilos variable

Si nos centramos en el número de hilos, únicamente vemos un crecimiento razonable en ambas implementaciones, pero claramente más pronunciado para TMulMatCuaBlo.c. Como hemos visto, al usar tareas, se pueden administrar mejor que usando únicamente hilos sobre todo en este hardware con hilos de capacidades desiguales, lo cual explica que el crecimiento se reduzca tanto en MulMatCuaBlo.c.

11.3. Tamaño de bloque

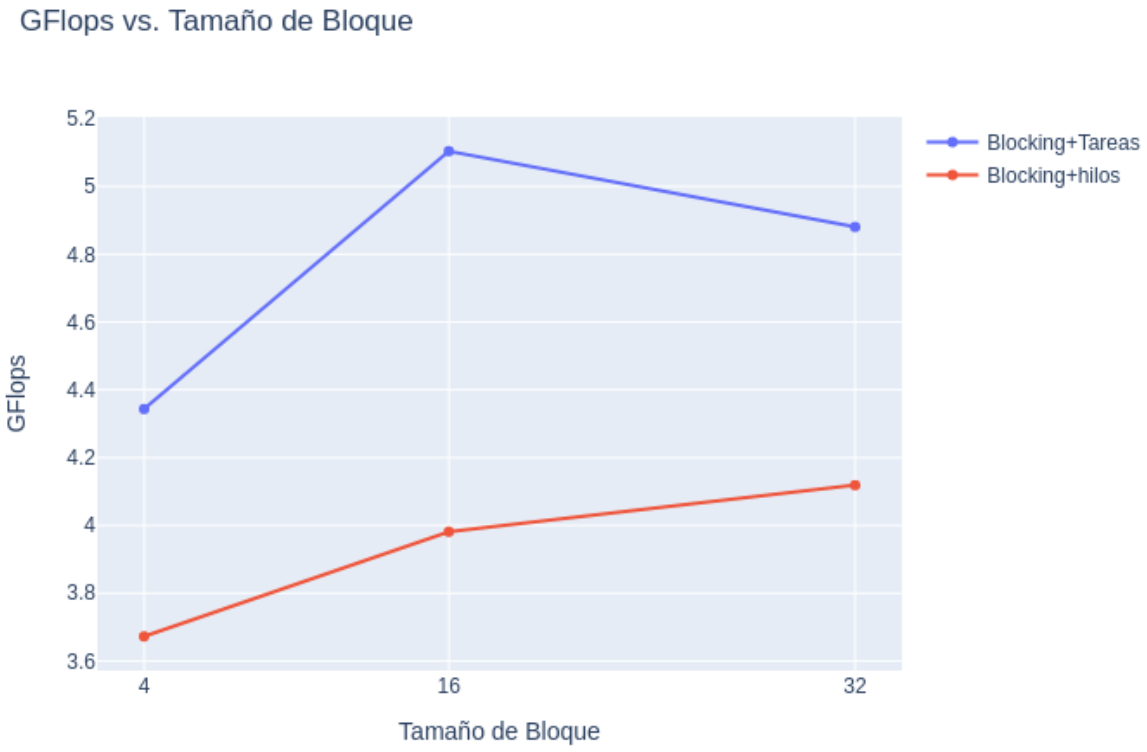

Figura 19: Comparativa de GFLOPS entre TMulMatCuaBlo.c (blocking+tareas) y MulMatCuaBlo.c (blocking+hilos) para tamaño de bloque variable

Como ya comentábamos en sus respectivos apartados, el tamaño de bloque se ve afectado en gran medida por el tamaño de los datos caché para operar un bloque y, por tanto, por el número de hilos y, principalmente, el tamaño de matriz.

Es por ello que estas dos implementaciones tienen un tamaño de bloque mejor en la mayoría de casos diferente debido a su diferente gestión de los hilos, puesto que ambos tamaños de bloque conllevan la misma cantidad de datos a operar en ambas implementaciones y, por tanto, ocupan lo mismo en caché.

11.4. Tamaño de matriz

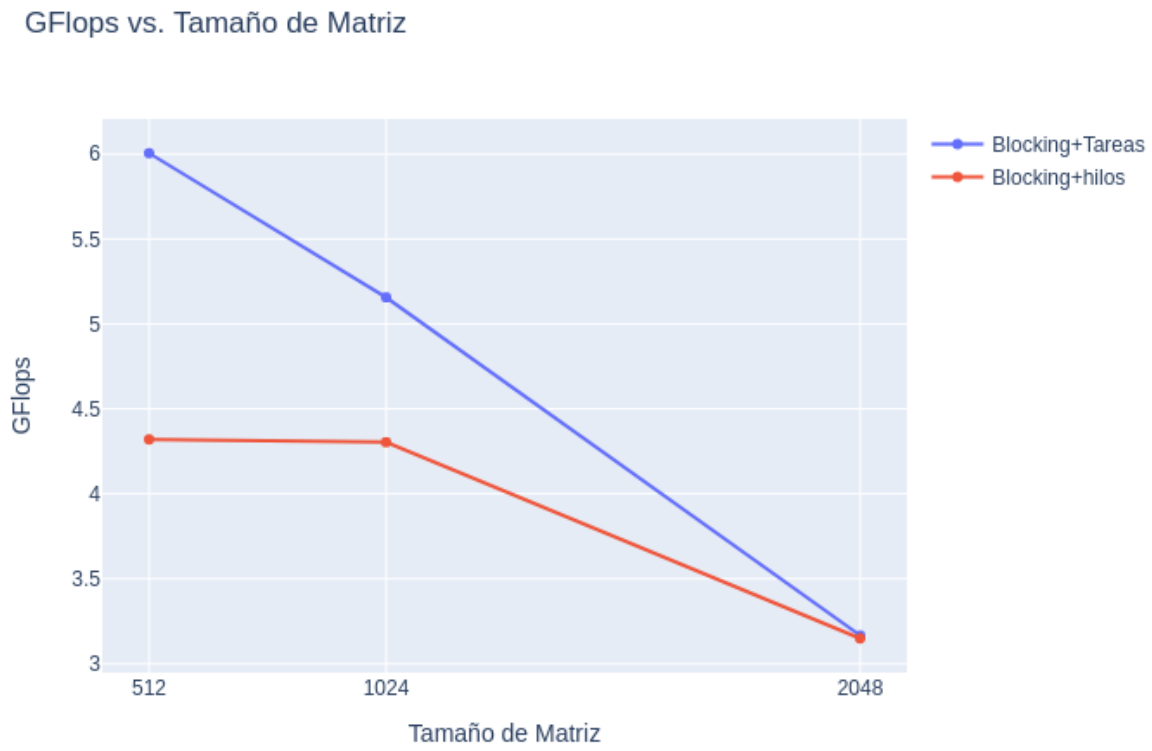

Figura 20: Comparativa de GFLOPS entre TMulMatCuaBlo.c (blocking+tareas) y MulMatCuaBlo.c (blocking+hilos) para tamaño de matriz variable

Como se ha dicho anteriormente, utilizando tareas, los tamaños de matriz más pequeños, 512 y 1024, ofrecen un gran rendimiento, con speedUps considerables y diferencias de alrededor de 1.5 GFLOPS y 1 GFLOP al pasar de hilos a tareas.

Sin embargo, al alcanzar cierto tamaño, podemos ver que la diferencia se minimiza debido a los fallos de caché que van de la mano de las matrices más grandes, 2048 en este caso. En ambos casos, estamos limitados por ellos, aunque como se ve, la elección de asignación dinámica de tareas o estática/dinámica de hilo puede conllevar una mejoría considerable en tamaños que no producen tantos fallos de caché en esta arquitectura para esta carga de trabajo. No hay más que comparar los resultados de tamaño de bloque 16/32 para 16 hilos en tamaño de matriz 512/1024 en las respectivas tablas.

Otro apunte parece ser que con menores números de hilos, en especial 4, los resultados entre ambas implementaciones se acercan e incluso se decaen de MulMatCuaBlo.c. Esto se exagera si usamos tamaño de bloque 4, ya que al tener un tamaño de bloque menor, la carga de cómputo es menor y la latencia de los hilos menos potentes se puede esconder mejor ayudados de los frecuentes fallos de caché.

12. Conclusiones

En este apartado espero resaltar las conclusiones que de manera más o menos explícita he resaltado a lo largo de este documento como estudio de la programación multihilo a través de la multiplicación de matrices.

En primer lugar, que no hay una mejor única solución que sirva para todos los casos y diversas configuraciones. Sí, podemos quedarnos con algunas generalidades como que aumentar el número de hilos lleva a un mejor rendimiento y que a mayor tamaño de matriz, peor rendimiento, sobre todo cuando nos salimos de la caché y aunque esta última es una restricción que se cumple a raja tabla, el número de hilos, nuestra mejor herramienta para ganar rendimiento no siempre ofrece el mejor rendimiento con mayor número de hilos ya que todos los parámetros, hilos, bloque/filas, tamaño de matriz están interrelacionados y, por lo tanto, cada configuración puede dar resultados inesperados a primera vista en los que hay que indagar.

Y normalmente esta investigación nos lleva al hardware. La realidad es que mis resultados y explicaciones no tiene por qué trasladarse a otras máquinas que quizás no consigan aumentos de rendimiento con los mismos parámetros.

Así que si queremos obtener el máximo rendimiento de una multiplicación de matrices para un tamaño concreto, recomiendo conocer bien el hardware y memoria caché de la máquina, intentar razonar qué configuración obtiene los mejores resultados y comprobarlo vía ejecuciones de benchmarks.

Un saludo.

Alejandro Carmona.

Referencias

[1] Malith Jayaweera. *Blocked Matrix Multiplication*. https://malithjayaweera.com/2020/07/blocked-matrix-multiplication/. Accedido el: 27/01/2024. Jul. de 2020.