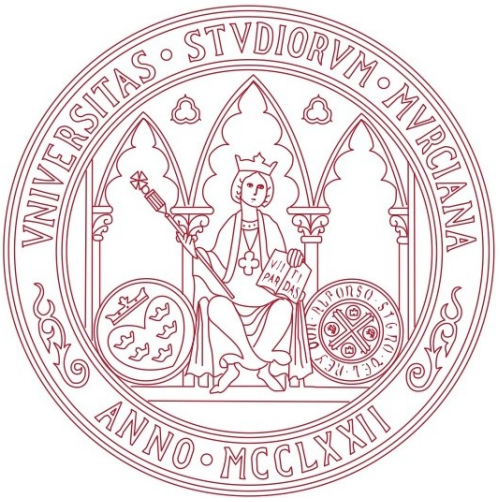

Programación de arquitecturas multinúcleo

T.P.3.3. Proyecto de programación con CUDA

Alumno: Alejandro Carmona Martínez

Correo: alejandro.carmonam@um.es

Profesor: Antonio Javier Cuenca Muñoz

Curso Académico: 2023/24

Fecha: 28/04/2024

Índice

1. Escribe un programa usando CUDA llamado compara_kernels que permita comparar las tres versiones de kernels de multiplicación de matrices descritos en clase: simpleMultiply, coalescedMultiply y sharedABMultiply. El programa debe estar preparado para multiplicar matrices rectangulares de números reales de la forma CNxN =ANxWBWxN, usando el kernel que decida el usuario. Se tienen fijados estos parámetros de configuración:

1. Bloque de hilos: WxW.
2. Bloque (tile) de datos: WxW.
3. Por simplificar, N debe ser múltiplo de W. Es decir, hay un entero T tal que N=WxT.
4. Grid: TxT bloques de hilos.
5. Cada hilo calcula un elemento de C a partir de una fila de A y una columna de B.
6. Cada bloque de hilos calcula los elementos de un tile de C a partir de multiplicar un
7. tile de A y un tile de B.

Compara razonadamente los tiempos de ejecución de la multiplicación completa, para N=512,1024,2048,4096 y W=4,8,16,32i, cuando se usa cada uno de estos kernels: (1) simpleMultiply, (2) coalescedMultiply y (3) sharedMultiply.

SINTAXIS:

```
compara_kernels N=<dim_mat> -W=<dim_bloq> -K=<kernel>
```

Código completo programa en: `1_/compara_kernels.cu`
Comprobación de resultados con: `1_/compara_kernels_test_v1.cu`
Análisis de tiempo con: `1_/compara_kernels_bench_v1.cu`

En primer lugar, dada la naturaleza de este ejercicio en ele que se nos dan los kernels ya programados con el objetivo de compararlos, no vamos a indagar mucho en cómo se logra ejecutar la multiplicación de matrices en sí a diferencia de los próximos ejercicios.

El programa `1_/compara_kernels.cu` actúa como un front para realizar la prueba de los kernels usando los conceptos de las prácticas anteriores como reserva de memoria, uso de kernels, medición de tiempo y memoria compartida. En cuanto a este último concepto, sí que merece la pena comentar que debido al uso de esta por parte de los kernels, el primer kernel (simpleMultiply) no usará memoria compartida, al contrario de los otros dos. coalescedMultiply traerá a memoria compartida la matriz A al completo en tiles por cada bloque, por tanto, asignamos como tamaño de memoria compartida el tamaño de un tile *tile_ dim*tile_ dim*sizeof(float)* y sharedABMultiply hará lo mismo, pero con la matriz A y B, por lo que necesitará *2*tile_ dim*tile_ dim*sizeof(float)* bytes de memoria compartida.

1.1. Validación de los resultados

La validación de los kernels se puede realizar en `1_/compara_kernels.cu` definiendo la macro DEBUG, pero la validación del correcto funcionamiento de los kernels se realiza mediante un programa test, `1_/compara_kernels_test_v1.cu`, encargado de lanzar una serie de casos de prueba para distintas dimensiones seleccionables por el usuario, cumpliendo las condiciones determinadas en el enunciado.

El resultado obtenido por cada kernel es comprobado contra la implementación con blocking+hilos para matrices rectangulares de la práctica anterior. Se ha optado por esta solución sobre la implementación secuencial debido a que para los tamaños a comprobar, esta implementación ya está comprobada y cumple con dichos tamaños, ofreciendo un mejor rendimiento frente a la implementación secuencial.

También se consideró usar la función Sgemm de cublas para realizar una multiplicación de matrices, sin embargo, esto hubiera supuesto la necesidad de la operación de reducción para realizar la comprobación de los resultados habiendo de traer de GPU la matriz con las diferencias entre el resultado de Sgemm y nuestro kernel además de la propia matriz resultado. Aunque puede resultar interesante a partir de ciertos tamaños, esta penalización de tiempo de transferencia causa cierto retardo, además de reducir la simplicidad y facilidad de uso a la hora de procesar y mostrar los errores. No obstante, en otros ejercicios en los que el tamaño es superior, si se ha utilizado con resultados positivos para tamaños grandes.

1.2. Comparación de los tiempos de ejecución para distintas configuraciones y kernel

Al igual que en prácticas previas, tenemos una comparativa multiparámetro, sin embargo, hemos agrupado N y W en un solo parámetro, puesto que sus combinaciones son conocidas y dan lugar siempre al mismo tamaño de matrices. Una vez, atajado eso, podemos centrarnos en cada comparar cada kernel para cada tamaño de N. Para ello, resumiremos la comparativa en una gráfica con los tiempos de las ejecuciones para los distintos valores de N,W y los respectivos kernel y diferentes tablas de tiempos y SpeedUps en los que podemos realizar una comparación más detallada usando los datos obtenidos.

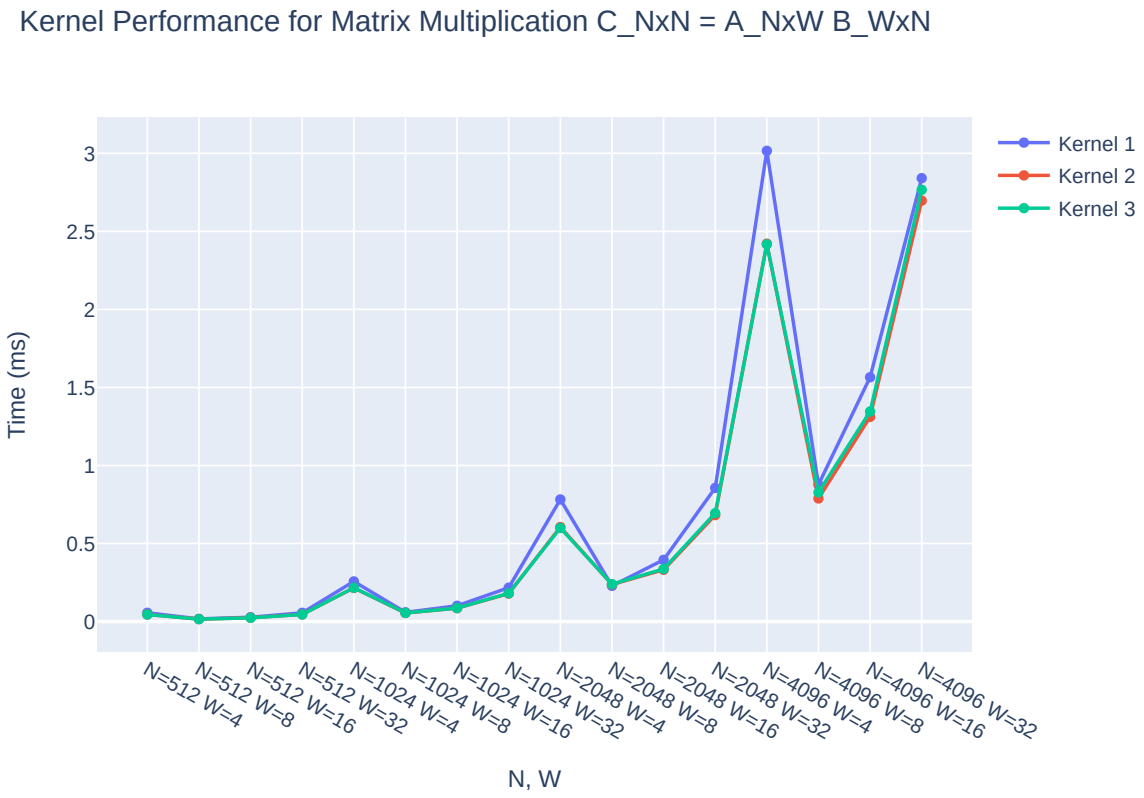

Figura 1: Gráfica tiempos de ejecución de la multiplicación completa, para N=(512,1024,2048,4096) y W=(4,8,16,32)i.

1.2.1. N = 512

Tiempo (ms) Kernel / (N,W)	(512,4)	(512,8)	(512,16)	(512,32)
simpleMultiply	0.056336	0.017107	0.027786	0.055898
coalescedMultiply	0.045606	0.015622	0.024240	0.045901
sharedMultiply	0.044979	0.015382	0.023606	0.045248

Tabla 1: Comparación de tiempos para N=512

1.2.2. N = 1024

Tiempo (ms) Kernel / (N,W)	(1024,4)	(1024,8)	(1024,16)	(1024,32)
simpleMultiply	0.257120	0.059142	0.101008	0.216973
coalescedMultiply	0.215318	0.054611	0.085779	0.178906
sharedMultiply	0.216704	0.055274	0.086189	0.181110

Tabla 2: Comparación de tiempos para N=1024

1.2.3. N = 2048

Tiempo (ms) Kernel / (N,W)	(2048,4)	(2048,8)	(2048,16)	(2048,32)
simpleMultiply	0.782282	0.228454	0.395488	0.856109
coalescedMultiply	0.605558	0.237466	0.331366	0.682138
sharedMultiply	0.599600	0.239635	0.337005	0.694477

Tabla 3: Comparación de tiempos para N=2048

1.2.4. N = 4096

Tiempo (ms) Kernel / (N,W)	(4096,4)	(4096,8)	(4096,16)	(4096,32)
simpleMultiply	3.016128	0.877187	1.565315	2.840810
coalescedMultiply	2.421248	0.788563	1.311360	2.696797
sharedMultiply	2.417040	0.826835	1.345638	2.766285

Tabla 4: Comparación de tiempos para N=4096

1.2.5. SpeedUps

Tiempo (ms) Kernel / (N,W)	(512,4)	(512,8)	(512,16)	(512,32)
simpleMultiply	1x	1x	1x	1x
coalescedMultiply	1.23x	1.09x	1.15x	1.22x
sharedMultiply	1.25x	1.11x	1.18x	1.24x

Tabla 5: Kernel speedups para N = 512

Tiempo (ms) Kernel / (N,W)	(1024,4)	(1024,8)	(1024,16)	(1024,32)
simpleMultiply	1x	1x	1x	1x
coalescedMultiply	1.19x	1.08x	1.18x	1.21x
sharedMultiply	1.19x	1.07x	1.17x	1.20x

Tabla 6: Kernel speedups para N = 1024

Tiempo (ms) Kernel / (N,W)	(2048,4)	(2048,8)	(2048,16)	(2048,32)
simpleMultiply	1x	1x	1x	1x
coalescedMultiply	1.29x	0.96x	1.19x	1.25x
sharedMultiply	1.30x	0.95x	1.17x	1.23x

Tabla 7: Kernel speedups para N = 2048

Tiempo (ms) Kernel / (N,W)	(4096,4)	(4096,8)	(4096,16)	(4096,32)
simpleMultiply	1x	1x	1x	1x
coalescedMultiply	1.25x	1.11x	1.19x	1.05x
sharedMultiply	1.25x	1.06x	1.16x	1.03x

Tabla 8: Kernel speedups para N = 4096

1.3. Análisis de los resultados

1.3.1. Efecto de N,W en el tiempo de ejecución

A primera vista, en la gráfica 1, podemos apreciar, dos ideas claras, en primer lugar, independientemente del kernel, cada vez que el tamaño de N se incrementa, se produce un pico de tiempo provocado por el nuevo valor de N junto a W en su valor más bajo, 4.

Una vez pasado el pico, el tiempo cae a su mínimo local para ese N con W=8 y, aunque se produce una ligera subida para los siguientes W, estas es más progresiva y con valores más bajos que el pico, pero superiores a los valores para N anteriores. Este patrón continúa hasta el siguiente N, donde vuelve a repetirse con un pico más acentuado. Lo vemos en los 3 cambios de N, de 512 a 1024, de 1024 a 2048 y de 2048 a 4096.

Es evidente que podemos achacar esta subida a la naturaleza del problema de la multiplicación de matrices (orden N^3), por lo que es razonable que mayores tamaños de N provoquen un tiempo considerablemente mayor.

Sin embargo, como hemos indicado y se aprecia en la gráfica, el pico viene dado por W=4, lo cual es contraproducente, puesto que W=4 produce unas matrices de A=N*4, B=4*N y C=N*N, es decir, hay menos trabajo que con otras matrices de W mayor, pero W también determina el número de hilos por bloque (WxW).

Si este número de hilos es excesivamente bajo, no se puede obtener un alto rendimiento de la GPU por dos motivos, en primer lugar, al no tener suficientes warps para esconder latencias en memoria, perdemos throughput, pero no necesariamente rendimiento, tardaríamos lo mismo que con un tamaño mayor, pero haríamos menos.

La caída de rendimiento (mayor tiempo de ejecución) viene dada porque estos warps están compuestos de menos de 32 hilos, ya que un bloque solo puede tener 16 hilos y los warps han de tener a todos sus hilos en un mismo bloque, creando un warp de tamaño máximo 16 hilos que hace menos trabajo del que podría.

Con la finalidad de justificar y visualizar esta posibilidad, podemos usar la herramienta Nsight System o equivalente para nuestra arquitectura y observar la ocupación a nivel de warp en los SM para N=1024 y W=4/8. El kernel seleccionado no importa, puesto que todos acusan del mismo fenómeno, pero hemos elegido sharedMultiply, puesto que es el que menos latencias puede presentar:

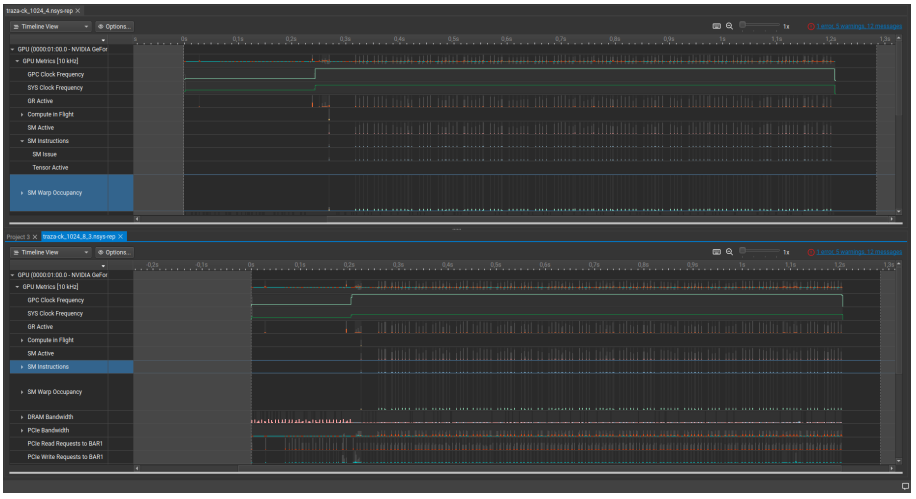

Figura 2: Comparativa de las dos trazas para W=4 (arriba) y W=8 (abajo)

Como vemos, para ambas configuraciones vemos que los SMs se llenan en ráfagas entre las cuales no están en uso. Para ver la ocupación durante esas ráfagas en las que se producen los picos de ocupación hacemoz zoom en ellas:

Como se aprecia, visualmente, la ocupación media es considerablemente superior para W=8, con la ocupación para W=4 bajando por debajo del 40 % en ciertas ocasiones y un porcentaje de warps desocupados que aunque bajo, nos indica también que no tenemos el número máximo de warps soportados por la GPU por un porcentaje mayor que en W=8 (56 % vs 90 %).

Todo esto lleva a un mayor tiempo de ejecución, aunque la cantidad de datos a operar sea menor. Este fenómeno también se aprecia de menor manera para W=16 (256 hilos por bloque), puesto que el tiempo aumenta ligeramente en vez de bajar fortuitamente. En este caso, con 256 hilos por bloque llegamos a una ocupación máxima teórica del SM, ya

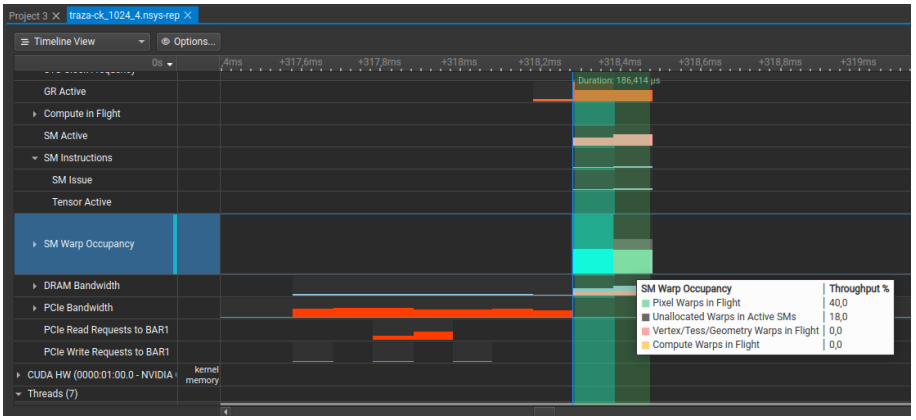

Figura 3: Pico de ocupación de SM para W=4

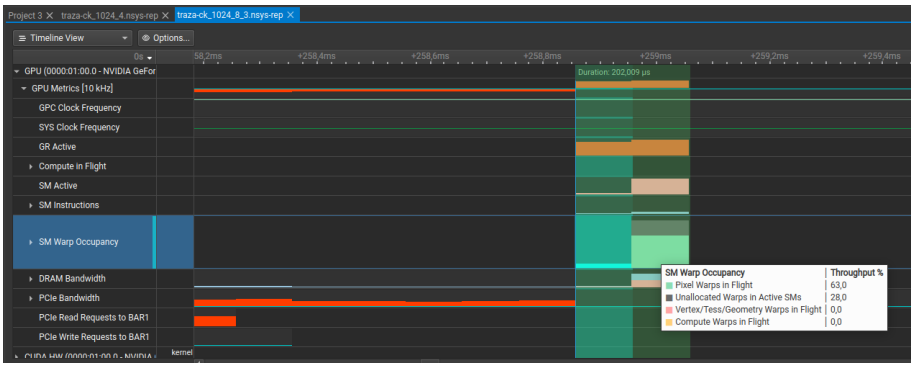

Figura 4: Pico de ocupación de SM para W=8

que lo saturamos de warps que puede lanzar, por lo que suavizamos la aumento de tiempo proveniente de los mayores tamaños de matriz con W mayor, pero en W=32, que ya ha saturado los warps a lanzar en cada SM, vemos el aumento magnificarse, además de que, evidentemente el salto en tamaños de 8 a 16 es menor que de 16 a 32, pero el no poder más leños (hilos) en nuestra hoguera (SM), hace que el fuego ya no caliente más.

Este fenómeno se va exagerando y pasa de verse muy tímidamente para valores de N más pequeños a mostrarse visible en N=2048 y N=4096, en los que el paso de W=8 a W=16 y a W=32 cuenta deriva en tiempos superiores y provoca un pico en N=2048/4096, W=32 que iguala y supera en tiempo al pico de W=4 debido a que el tamaño que estamos arrojando a la arquitectura es demasiado grande y no obtenemos beneficio por añadir más hilos.

Para afianzar la idea, veámoslo con otro símil. Imaginemos que somos un pastelero que ha de hacer una gran orden de pastelitos, para ello tiene un conjunto de hornos y va a dividir los pastelitos en bandejas que solo puede meter de una en una en cada horno.

Estos pastelitos han de sacarse del horno a media cocción, puesto que hay que echarles chocolate y dejarlos reposar.

En nuestro caso, los pastelitos son los hilos, los warps son bandejas y los hornos los SMs.

Supongamos que tenemos muy pocos pastelitos, en ese caso, tendremos pocas ban-

dejas e incluso puede que no usemos todos los hornos y las bandejas no vayan llenas. Pongamos que hay 4 hornos y 2 bandejas por horno.

Al principio habrá una bandeja por horno y cuando la primera hornada haga su parada, podemos mantener los hornos trabajando con la segunda hornada, pero si al parar esta, no tenemos más bandejas y la primera aún no está lista, tendremos el horno parado. Este desaprovechamiento de recursos nos provoca una pérdida de throughput, pero tardamos lo mismo, a no ser que las bandejas no estén llenas.

En otro caso, si hubiera suficientes bandejas y estas estuvieran llenas, podríamos hacer más pastelitos en menos tiempo, como sucede al pasar de W=4 a W=8, pero si hubieran demasiadas, incluso si están llenas, nos quedaríamos sin hornos en los que hacerlas como se podría intuir que ocurre con W=32. Sin embargo, con estos tamaños, es más probable que en lugar de estar limitados por cómputo, estemos limitados por memoria debido a una intensidad aritmética baja. Esto se puede comprobar lanzando el benchmark para obtener los Flops en lugar del tiempo:

Kernel	N	W	GFlops
sharedMultiply	4096	4	0.053592
sharedMultiply	4096	8	0.316229
sharedMultiply	4096	16	0.376064
sharedMultiply	4096	32	0.371667

Tabla 9: Flops para sharedMultiply con N=4096 y W variable

Como vemos, al llegar a 16 hilos, el rendimiento se estanca, pero este es muy inferior al rendimiento máximo que alcanzaremos en el ejercicio 2 o el que anuncia el fabricante para este chip, por lo que podemos confirmar que con estos, la intensidad aritmética del problema, lo convierte en uno limitado por memoria y ya no importa que usemos mejor o peor el hardware, sino que es el ancho de banda (a pesar de usar Shared Memory) el que nos limita el rendimiento.

1.3.2. Diferencias en tiempo de ejecución entre kernels

La otra tendencia visible en la gráfica se manifiesta en la similitud de rendimiento obtenida por los distintos kernels casi independientemente de los valores de N,W, los kernels presentan funciones muy similares con ciertos patrones como que simpleMultiply siempre es ligeramente más lento que los otros kernels y, notablemente más lento para W=4.

Para explorar con más precisión estos comportamientos, también nos referiremos a las tablas temporales, pero en especial a las tablas de los speedUp5, en las que se manifiesta la escasa diferencia entre kernels.

La otra característica, que ya se veía en la gráfica, pero que se magnifica en las tablas, es que tanto sharedMultiply como coalescedMultiply ofrecen resultados muy cercanos independientemente del caso, con una diferencia de rendimiento prácticamente inexistente, las mayores brechas de rendimiento están entre el 3 y el 5 %, parece que en líneas generales,

la carga a memoria compartida realizada por sharedMultiply es contrarrestada por la falta de sincronización obligatoria en coalescedMultiply.

Aunque hay casos en los que sharedMultiply es ligeramente más rápida, N y W bajos, coalescedMultiply parece sacar algo de ventaja con estos dos valores a un tamaño superior.

Lo que sí parece una generalidad, es que en esta GPU, realizar al menos una transferencia a memoria compartida mejora el rendimiento frente a no hacerlo de ninguna manera como en simpleMultiply. Este kernel solo logra ofrecer un rendimiento mejor en el caso (2048, 8), que parece un outlier, frente a las otras configuraciones con el mismo N o mismo W.

De forma que podemos concluir que para esta GPU es recomendable hacer uso de la memoria compartida siempre al menos de una manera, pero ante el tradeoff de la sincronización y otra carga en memoria compartida parece no haber una gran diferencia ente una y otra, pudiendo optar por cualquiera de forma intercambiable.

2. A partir del programa anterior, escribe un programa usando CUDA llamado mulmatcua_1G que calcule en una GPU el producto de dos matrices cuadradas de números reales de la forma CNxN=ANxNBNxN, usando el kernel sharedMultiply. La resolución se llevará a cabo con una malla cuadrada de TxT bloques de WxW hilos cada uno, siendo T un entero tal que T=N/W. Cada bloque de hilos estará encargado de calcular los elementos del tile(i,j) de C a partir de de multiplicar los T tiles la fila i de tiles de A y los T tiles de la columna j de tiles de B. Por ejemplo, con T=8,i=3,j=4:

SINTAXIS:

```
mulmatcua_1G N=<dim_mat> -W=<dim_bloq>
```

Código completo programa en: `2_/mulmatcua_1G.cu`
Comprobación de resultados con: `2_/mulmatcua_1G_test_v2.cu`
Análisis de tiempo con: `2_/mulmatcua_1G_bench_v2.cu`
Kernel: 2_/sharedABMultiply_kernel_super_tile.cu

2.1. Explicación del algoritmo y el programa

Para poder cumplir con los requisitos del ejercicio, tomaremos el kernel `sharedABMultiply` y realizaremos ciertos cambios que explicaremos a continuación:

```
__global__ void sharedABMultiply(float *a, float* b, float *c, int N, int tile_dim)
{
    /*The shares memory can only be claimed by one 1D array*/
    extern __shared__ float tile[]; // Declare a shared memory
    /*So in order to divide it into two, we assign one array to the start of the
    original array and another one to the middle*/
    float* aTile = tile; // Partition for matrix a
    float* bTile = &tile[tile_dim * tile_dim]; // Partition for matrix

    int row = blockIdx.y * blockDim.y + threadIdx.y; // Calculate the row index
    int col = blockIdx.x * blockDim.x + threadIdx.x; // Calculate the column index

    float sum = 0.0f; // Initialize sum to 0

    int index = threadIdx.y * tile_dim + threadIdx.x; // Calculate the index for the 1D
        ↪  array

    for (int i = 0; i < gridDim.x; i++) // Loop over the tiles of the matrices
    {
        aTile[index] = a[row * N + i * tile_dim + threadIdx.x]; // Copy the elements of
            ↪  matrix a to the shared memory
        bTile[index] = b[(i * tile_dim + threadIdx.y) * N + col]; // Copy the elements
            ↪ of matrix b to the shared memory

        __syncthreads(); // Synchronize the threads: threads need data that they
            ↪ themselves have not written to shared memory

        for (int j = 0; j < tile_dim; j++) // Loop over the dimension of the matrices
        {
            sum += aTile[threadIdx.y * tile_dim + j] * bTile[j * tile_dim + threadIdx.x
                ↪ ]; // Multiply elements of matrices a and b and add to sum
        }

        __syncthreads(); /* Synchronize the threads: If this were not there,
        some threads might start loading data for the next tile before all computations
         for the current tile are complete, leading to incorrect results.*/
    }

    c[row*N+col] = sum; // Store the result in matrix c
}
```

Como podemos contemplar, respecto al kernel *sharedABMultiply* original, hemos mantenido el acceso usando la columna y fila relativa al hilo particular ejecutando el kernel, no así la carga en memoria compartida de los tiles debido a las diferencias fundamentales entre el ejercicio 1 y 2.

En el ejercicio 1, el ancho (número de filas) de A y el alto (número de columnas) de B, lo que sería la dimensión K tradicionalmente, se correspondía con el ancho de bloque (W) según lo indicado, de esta forma, cada bloque se encargaría únicamente de un tile cargando solamente dos tiles, uno de A y otro de B, y el resultado era correcto porque nuestro número de bloques se correspondía con el número de tiles de la matriz resultado y W se correspondía con K. Por lo tanto, para obtener un tile, bastaba con que cada hilo de un bloque operase los W (tile_dim) elementos de una fila de A y columna de B para

obtener el su resultado asignado dentro del tile.

Sin embargo, con las nuevas condiciones, W ya no tiene por qué ser K, de hecho K será N, por lo que ahora cada bloque de hilos habrá de computar T tiles de A y T tiles de B para obtener su tile correspondiente de la matriz resultado.

Por lo tanto, para obtener dicho tile, cada hilo debe hacer lo que hacía en *sharedAB-Multiply* T veces para operar los T tiles actuales y obtener su resultado concreto dentro del tile de su bloque.

Veámoslo en un ejemplo:

- M=4
- W=2
- ThreadBlock.size=2x2
- T=N/W=2
- Grid.size=2x2

Para sharedABMultiply del ejercicio 1, tendríamos esta situación para el tile a calcular por el bloque (0,0):

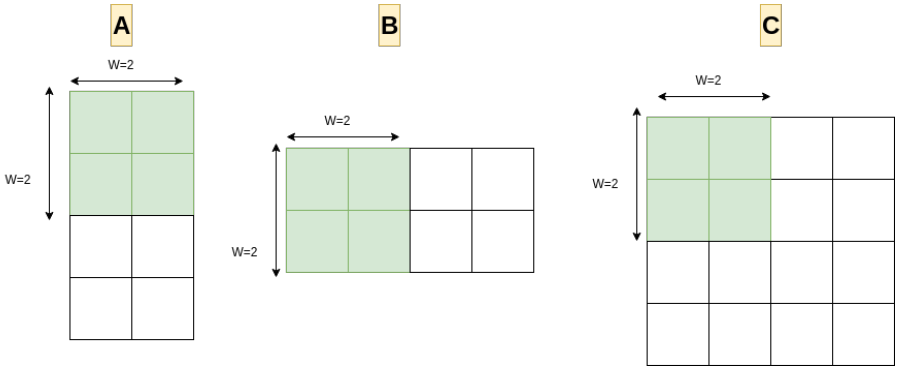

Figura 5: Ejercicio 1: N=4, W=2

Como vemos, el tile resaltado de C, es calculado únicamente con un tile de A y un tile de B en el que, según el algoritmo del ejercicio 1, cada hilo de los 4 que compondrán el bloque traerá uno de los 4 elementos del tile.

Al tener 4 bloques en el grid, cada bloque trae un tile (aunque se repitan) y cada hilo obtiene uno de los elementos finales del tile.

Si usamos los mismos parámetros en el ejercicio 2, estaríamos más bien ante esta situación:

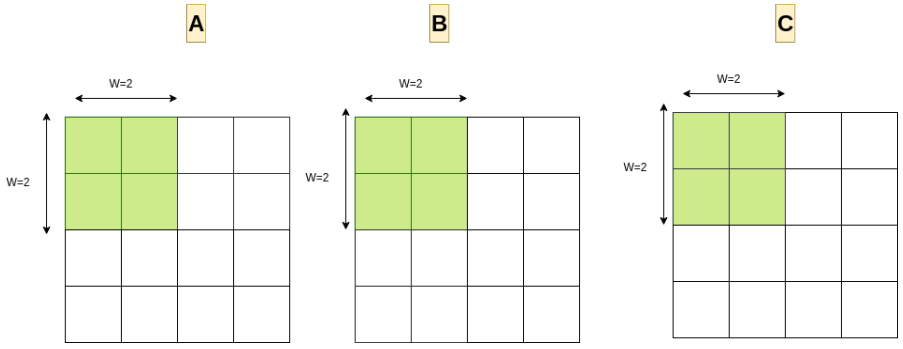

Figura 6: Ejercicio 2: N=4, W=2

Al no ser W=K, tenemos que recorrer las matrices A y B cargando T tiles en cada iteración, en este caso T=2 como se puede ver, traeríamos los tiles amarillos en la primera iteración, acumularíamos sus resultados paraciales en C y los azules en la segunda, con lo que al acumular sus parciales con los actuales de C, obtendríamos el primer tile de C:

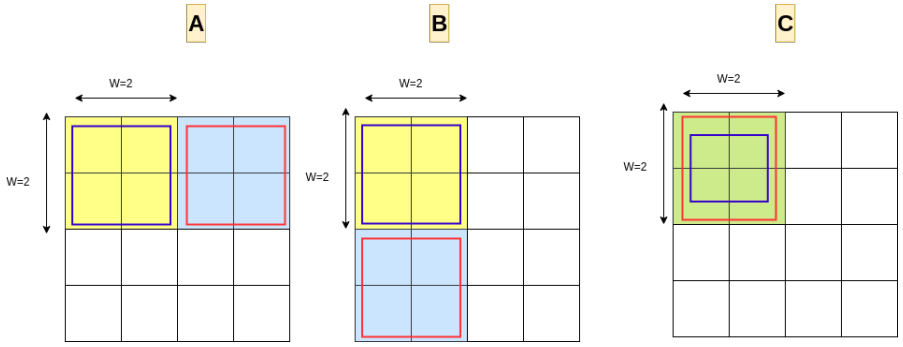

Figura 7: Ejercicio 2: N=4, W=2 Tile Looping

En conclusión, se ha realizar el mismo proceso que en el ejercicio 1 un número T de veces, por lo que podemos utilizar un bucle para llevar la cuenta del tile que estamos obteniendo y mediante accesos a memoria usando esa cuenta y la metainformación del hilo podemos cargar los tiles como hacíamos en el ejercicio anterior y operarlos.

De esta manera terminamos cada iteración del bucle con la acumulación de 1 o más resultados parciales de cada elemento del tile hasta que completamos todas las iteraciones. Es importante destacar que en cada iteración, se ha de sincronizar de nuevo, puesto que un hilo podría terminar antes y empezar a cargar los elementos del nuevo tile, haciendo que otro hilo más lento cargue dicho elemento que no pertenece al tile operado por dicho hilo.

2.2. Validación de los resultados

La validación de los resultados se ha realizado de la misma manera que en el ejercicio 1 ajustado a los parámetros y restricciones de este ejercicio.

Se ha añadido un programa test con cublas para evaluar tamaños grandes, aunque en el programa *mulmatcua_1G* se sigue usando la implementación por CPU e hilos para evitar problemas de compatibilidad.

2.3. Comparación de los tiempos de ejecución para distintas configuraciones y kernel

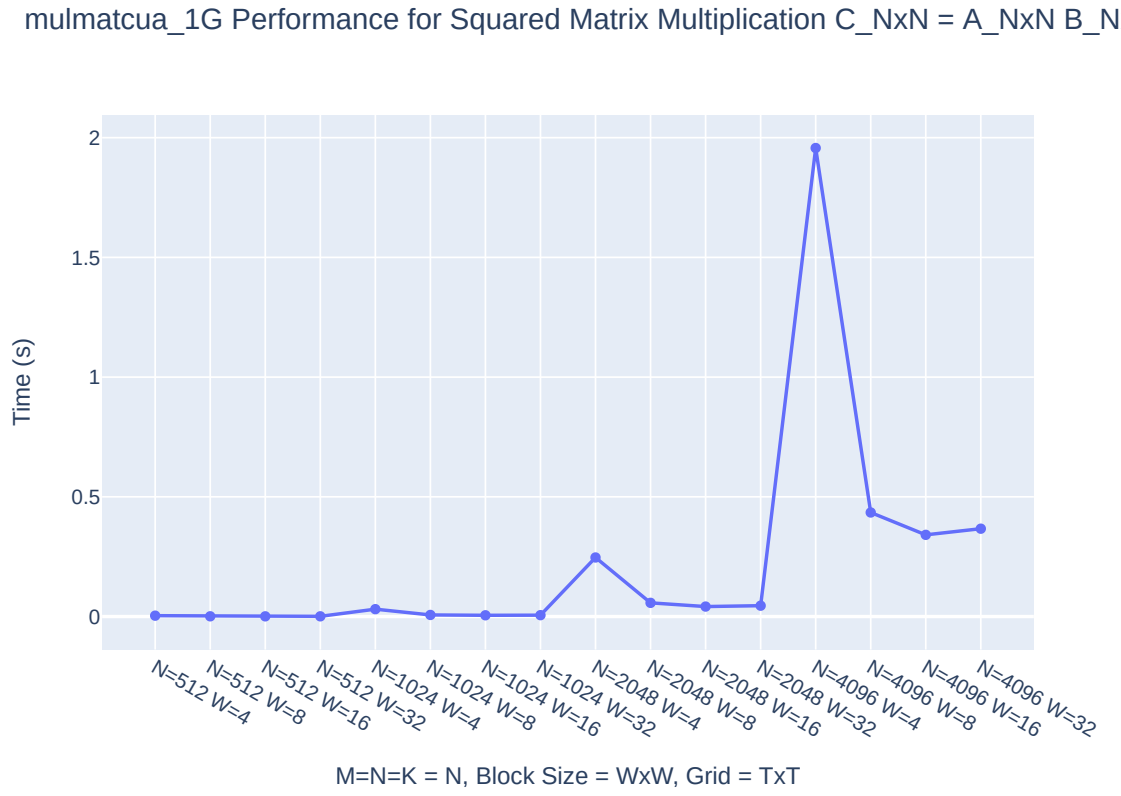

Figura 8: Gráfica tiempos de ejecución de la multiplicación completa, para N=(512,1024,2048,4096) y W=(4,8,16,32)

Tiempo (s) N/W	4	8	16	32
512	0.003609	0.000985	0.000821	0.000904
1024	0.030763	0.006628	0.005191	0.005736
2048	0.246631	0.057171	0.041658	0.045332
4096	1.956845	0.434562	0.341130	0.366901

Tabla 10: Comparación de tiempos para N y W

2.3.1. SpeedUps y Gráficas para N constante

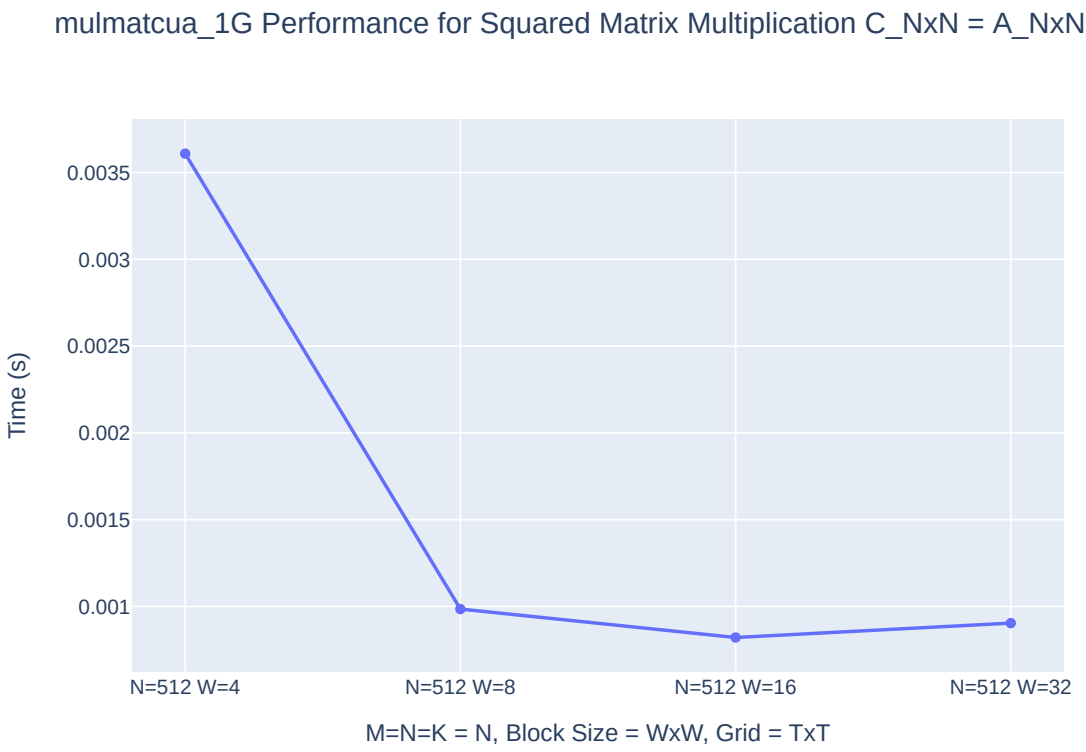

Figura 9: Gráfica tiempos de ejecución de la multiplicación completa, para N=512 y W=(4,8,16,32)

W	Speedup
4	1x
8	3.66x
16	4.39x
32	3.99x

Tabla 11: Speedup para N = 512

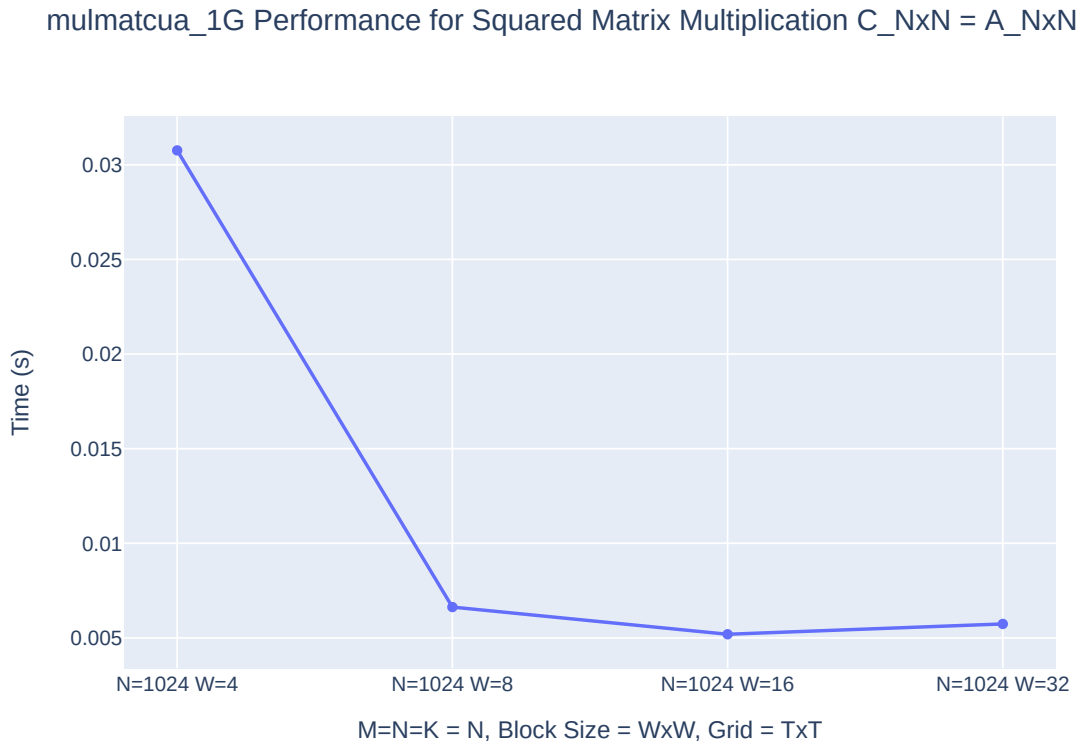

Figura 10: Gráfica tiempos de ejecución de la multiplicación completa, para N=1024 y W=(4,8,16,32)

W	Speedup
4	1x
8	4.64x
16	5.92x
32	5.36x

Tabla 12: Speedup para N = 1024

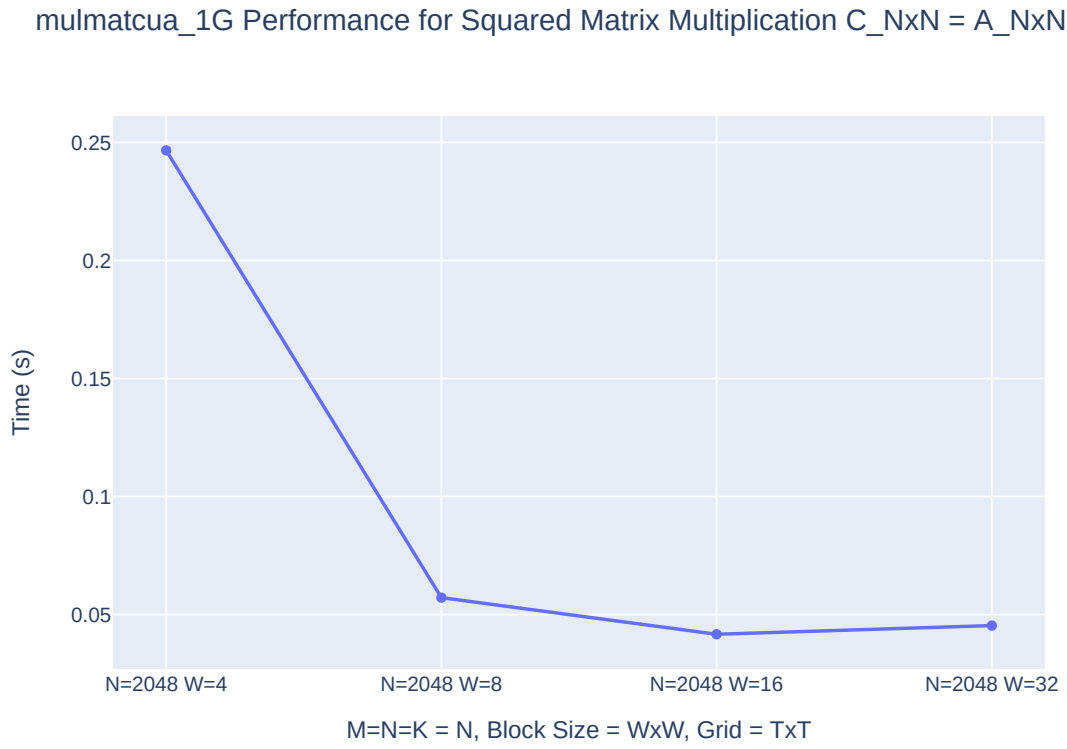

Figura 11: Gráfica tiempos de ejecución de la multiplicación completa, para N=2048 y W=(4,8,16,32)

W	Speedup
4	1x
8	4.31x
16	5.92x
32	5.44x

Tabla 13: Speedup para N = 2048

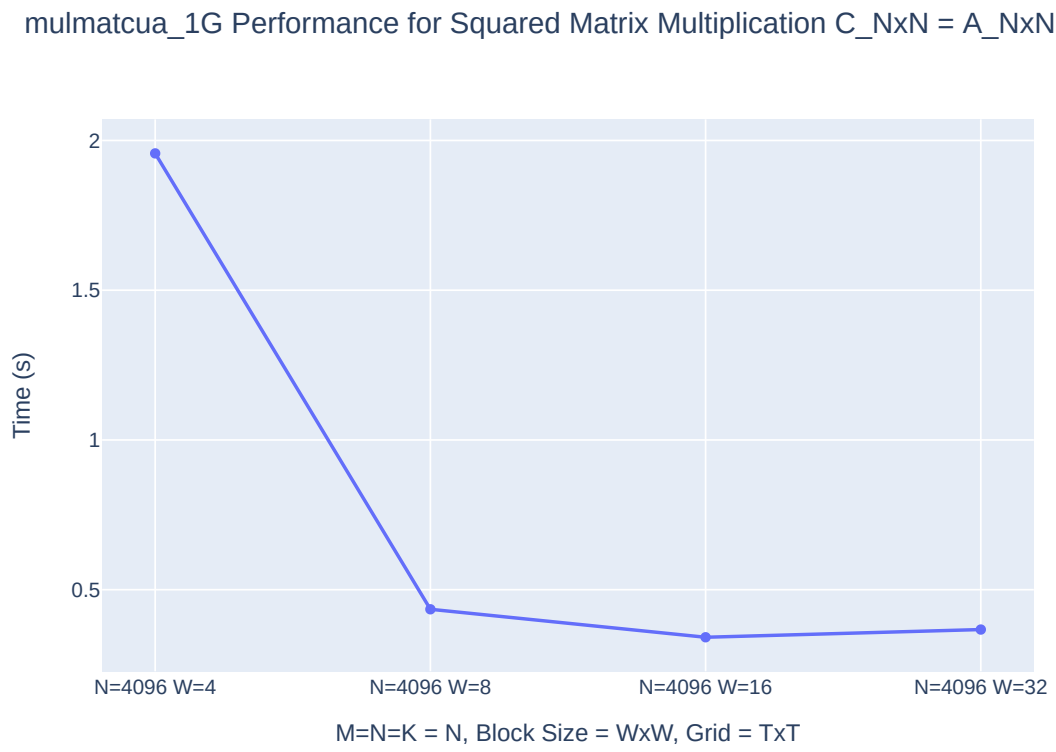

Figura 12: Gráfica tiempos de ejecución de la multiplicación completa, para N=4096 y W=(4,8,16,32)

W	Speedup
4	1x
8	4.50x
16	5.73x
32	5.33x

Tabla 14: Speedup para N = 4096

2.4. Análisis de los resultados

Como ya hemos explicado en la sección anterior, este ejercicio cuenta con un approach fundamentalmente diferente al ejercicio anterior, pero para nuestro análisis de rendimiento, las diferencias se reducen a que W ahora no tiene efecto en el tamaño del problema (M=N=K = N) y, por tanto, este es constante para todas los valores de W, por lo que podemos ver de forma más justa como el tamaño de bloque (y de tile) afecta al rendimiento.

En 8 podemos ver la disparidad de rendimiento entre N=512 y N=4096, de hecho, dicha diferencia es tal, que para poder analizar correctamente el impacto de W gráficamente se han dispuesto sendas gráficas con N fijo.

2.4.1. Efecto de N y W en el tiempo de ejecución

Como vemos, la diferencia entre N=512 y N=4096, es abrumadora, pasamos de diezmilésimas a segundos, pero esta diferencia no es tal entre N=512 y N=1024 o N=2048, quedándose estos valores en las milésimas salvo por N=2048/W=4 que sube a las décimas por su bajo valor de W como explicamos en el ejercicio anterior.

Este comportamiento es muy similar en patrón a lo que sucede en el ejercicio anterior, con tiempos, que aún mayores debido al salto de tamaño de la mulltiplicación (pasamos de NxW a NxN), encajan con lo descrito en el ejercicio 1. No obstante, como ya hemos comentado antes, al tener tamaño de matriz constante, vemos que una vez pasado el pico de tiempo por un W bajo, este solo baja en vez de subir ligeramente hasta alcanzar W=16 como se ve en cualquiera de las gráficas con N constante. Dichas gráficas siguen el mismo patrón con la única diferencia siendo el orden de magnitud de los tiempos, lo cual se manifiesta en speedUps muy similares independientemente de N, (orden de 3.5 - 6x) al aumentar el tamaño de bloque hasta 16 con una ligera caída en W=32, todo por los motivos ya comentados anteriormente.

Este análisis confirma que el número de hilos de un threadblock siguen un comportamiento similar al número de hilos en CPU, con el rendimiento aumentando hasta que se explota todo el hardware de la máquina, llegando a un punto dulce antes de caer debido a la sobrecarga de hilos por encima de la capacidad del hardware.

Sin embargo, eso no es lo más interesante. Aunque las gráficas del ejercicio 1 1 y este son muy similares en forma, las diferencias de tiempo son mucho mayores que cuando antes aumentábamos el tamaño de N como comentábamos antes, si cogemos los tiempos más rápidos (W=16) para N=(512, 1024, 2048, 4096) cogiendo como 1x N=4096 tenemos lo siguiente:

N	Speedup
512	426,25x
1024	68.2x
2048	8.31x
4096	1x

Tabla 15: Speedup W=16, N variable

Mientras que en el ejercicio anterior para W=16/K=3 teníamos:

N	Speedup
512	58,47x
1024	15.64x
2048	4x
4096	1x

Tabla 16: Speedup W=16/K=3, N variable

Lo cual nos indica que, volviendo al símil del ejercicio 1, ahora si nos estamos quedando sin hornos, es decir, estamos limitados por cómputo, no por memoria y, por tanto, el tiempo de ejecución no escala respecto al tiempo de ejecución ideal si tuviéramos hardware ilimitado, alcanzando tiempos realmente elevados para N más grandes.

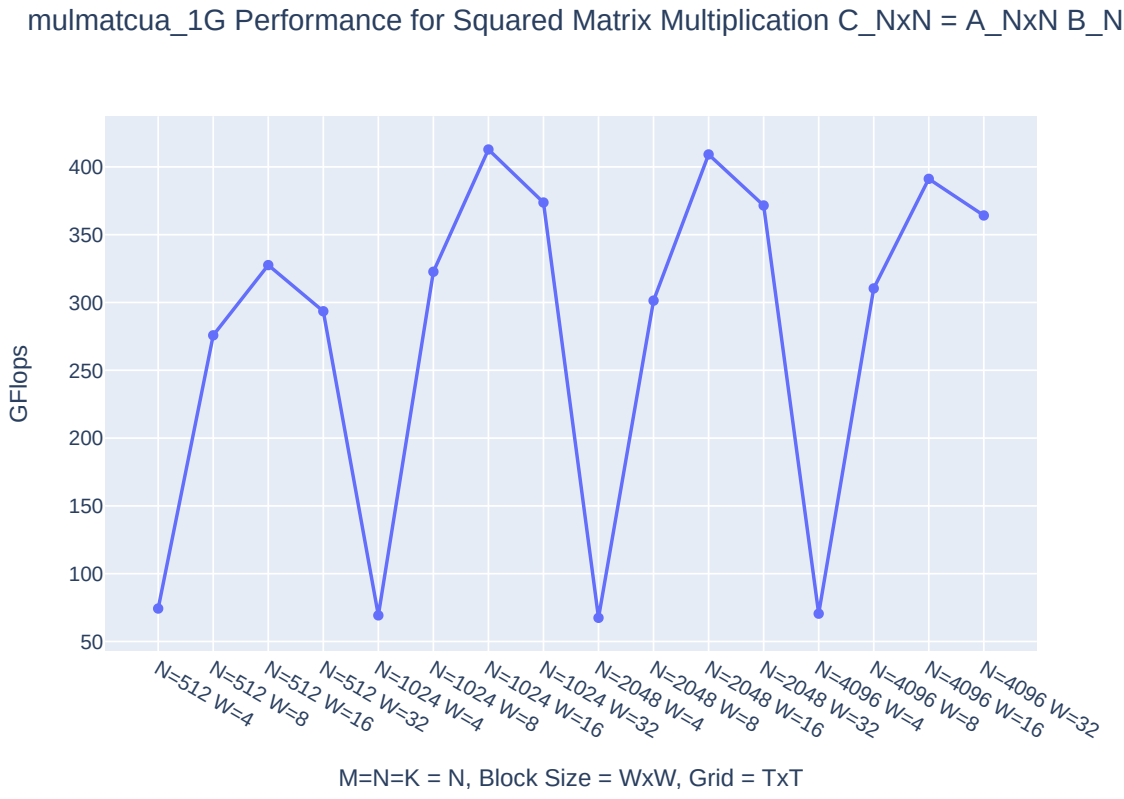

Figura 13: GFLOPS para N=(512,1024,2048,4096); W=(4,8,16,32)

Como vemos, al llegar a N=1024, W=16, hemos alcanzado la mejor configuración

para este programa y no importa que aumentemos el tamaño de N, ya estamos limitados por la potencia de cómputo.

3. A partir del programa anterior, escribe un programa usando CUDA llamado mulmat_1G que calcule en una GPU el producto de dos matrices de números reales de la forma CMxN=AMxKBKxN, usando el kernel sharedMultiply. La resolución se llevará a cabo con una malla de SxT bloques (S filas, T columnas) de WxW hilos cada uno, siendo S y T dos enteros tal que S=M/W y T=N/W. Cada bloque de hilos estará encargado de calcular los elementos del tile (i,j) de C a partir de los R tiles de la fila i de tiles de A y los R tiles de la columna j de tiles de B, siendo R un entero tal que R=K/W. Por simplificar, M, N y K deben ser múltiplo de W. Así, por ejemplo, si W=32, M=224, N=256, K=192; entonces tendríamos que S=7, T=8, R=6:

SINTAXIS:

```
1 mulmat_1G M=<dim_mat> N=<dim_mat> K=<dim_mat> -W=<dim_bloq>
```

Código completo programa en: 3_/mulmat_1G.cu
Comprobación de resultados con: 3_/mulmat_1G_test_v3.cu
Análisis de tiempo con: 3_/mulmat_1G_bench_v3.cu
Kernel: 3_/sharedABMultiply_kernel_hyper_tile.cu

3.1. Explicación del algoritmo y el programa

Al tener que lidiar con matrices rectangulares, deberemos tener en cuenta las nuevas dimensiones de las matrices, por lo que deberemos modificar los índices de manera acorde para poder acceder a ellas, no obstante, la restricción que impone que M,N y K (leading dimension) sean múltiplos de W permite mantener el algoritmo prácticamente inalterado más allá de dichos cambios:

En esta imagen planteamos una multiplicación rectangular que cumple con los requisitos del ejercicio. Es posible ver que a diferencia de operar con matrices rectangulares, A y B tienen diferente número de columnas, aunque mismo número de filas. Como consecuencia, para realizar los accesos con los índices habremos de multiplicar por K en el caso de A para acceder a la fila correspondiente y por N en el caso de B y C.

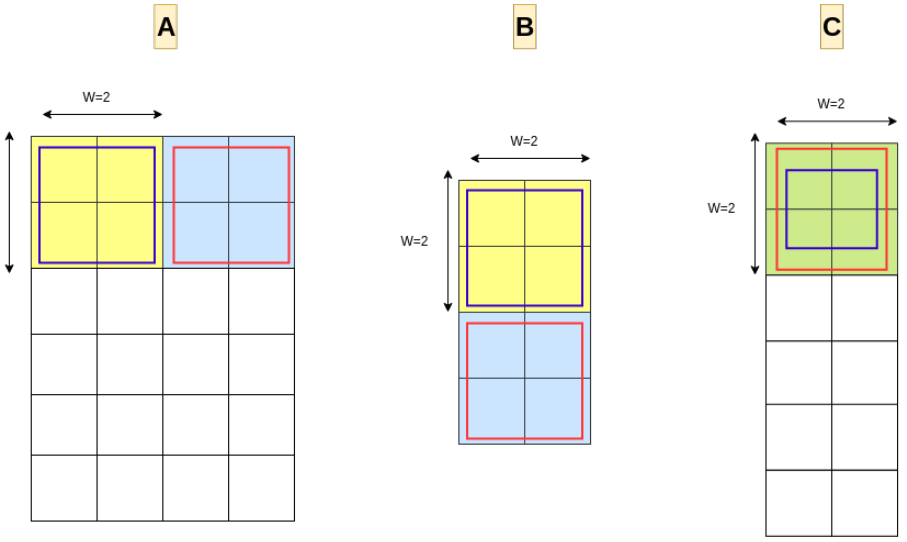

Figura 14: Ejercicio 3: M=6, N=2, K=4, W=2

Dicho esquema también se ajusta a la malla de SxT bloques (S filas, T columnas) siendo S y T dos enteros tal que S=M/W y T=N/W, lo cual se ajusta a las nuevas posibles dimensiones como en el ejemplo anterior en el que tenemos 3x1 bloques que coinciden con los bloques necesarios para obtener la matriz C y sus dimensiones en bloques, ya que podríamos ver la matriz como una matriz de 3 filas y 1 columna si en lugar de elementos, consideramos bloques.

```
__global__ void sharedABMultiply(float *a, float* b, float *c, int N, int K)
{
    /*The shares memory can only be claimed by one 1D array*/
    extern __shared__ float tile[]; // Declare a shared memory
    /*So in order to divide it into two, we assign one array to the start of the
    original array and another one to the middle*/
    float* aTile = tile; // Partition for matrix a
    int tile_dim = blockDim.x; // Tile dimension
    float* bTile = &tile[tile_dim * tile_dim]; // Partition for matrix

    int row = blockIdx.y * blockDim.y + threadIdx.y; // Calculate the row index
    int col = blockIdx.x * blockDim.x + threadIdx.x; // Calculate the column index

    float sum = 0.0f; // Initialize sum to 0

    int index = threadIdx.y * tile_dim + threadIdx.x; // Calculate the index for the 1D
        ↪  array

    // Calculate the number of tiles (R = K/W)
    int R = K / tile_dim;

    for (int i = 0; i < R; i++) // Loop over the tiles of the matrices
    {
        aTile[index] = a[row * K + i * tile_dim + threadIdx.x]; // Copy the elements of
            ↪  matrix a to the shared memory
        bTile[index] = b[(i * tile_dim + threadIdx.y) * N + col]; // Copy the elements
            ↪ of matrix b to the shared memory

        __syncthreads(); // Synchronize the threads: threads need data that they
            ↪ themselves have not written to shared memory

```

```
28          for (int j = 0; j < tile_dim; j++) // Loop over the dimension of the matrices
29          {
30              sum += aTile[threadIdx.y * tile_dim + j] * bTile[j * tile_dim + threadIdx.x
                    ↪ ]; // Multiply elements of matrices a and b and add to sum
31              printf("aTile[%d] = %f, bTile[%d] = %f\n", threadIdx.y * tile_dim + j,
                    ↪ aTile[threadIdx.y * tile_dim + j], j * tile_dim + threadIdx.x, bTile[
                    ↪ j * tile_dim + threadIdx.x]);
32          }
33
34          __syncthreads(); /* Synchronize the threads: If this were not there,
35          some threads might start loading data for the next tile before all computations
36           for the current tile are complete, leading to incorrect results.*/
37      }
38
39      c[row*N+col] = sum; // Store the result in matrix c
40  }
```

Por lo tanto, en el código del kernel, basta con realizar dichos cambios para realizar los accesos a A y B con base en sus respectivas dimensiones y calcular la variable R con el número de tiles a obtener.

3.2. Validación de los resultados

La validación de los resultados se ha realizado de la misma manera que en el ejercicio 1 ajustado a los parámetros y restricciones de este ejercicio.

4. A partir del programa anterior, escribe un programa usando CUDA llamado mulmat_1G que calcule en una GPU el producto de dos matrices de números reales de la forma CMxN=AMxKBKxN, usando el kernel sharedMultiply. La resolución se llevará a cabo con una malla de SxT bloques (S filas, T columnas) de WxW hilos cada uno, siendo S y T dos enteros tal que S=M/W y T=N/W. Cada bloque de hilos estará encargado de calcular los elementos del tile (i,j) de C a partir de los R tiles de la fila i de tiles de A y los R tiles de la columna j de tiles de B, siendo R un entero tal que R=K/W. Por simplificar, M, N y K deben ser múltiplo de W. Así, por ejemplo, si W=32, M=224, N=256, K=192; entonces tendríamos que S=7, T=8, R=6:

SINTAXIS:

```
mulmat_1G M=<dim_mat> N=<dim_mat> K=<dim_mat> -W=<dim_bloq>
```

Código completo programa en: 4_/mulmat_1G1C.cu
Comprobación de resultados con: 4_/mulmat_1G1C_test_v4.cu
Análisis de tiempo con: 4_/mulmat_1G1C_test_v4.cu
Kernel: 3_/sharedABMultiply_kernel_hyper_tile.cu

4.1. Explicación del algoritmo y el programa

En este ejercicio no hemos introducido ningún nuevo algoritmo, sino que hemos reutilizado el kernel del ejercicio anterior, además de utilizar la función del ejercicio 8 de la práctica 2.3.

De esta manera, el programa `4_/mulmat_1G1C.cu`, combina ambos métodos, lanzándolos simultáneamente usando, en un primer momento, dos hilos. Uno que se encargará del trabajo en GPU como copiar y recuperar los datos, además de la ejecución del kernel y otro que lanzará la función `mm_blo` con tamaño de bloque W que, a su vez, lanzará 8 hilos para computar las F filas restantes de la matriz resultado. Hemos elegido 9 hilos, 1 en GPU y 8 en CPU debido a que 8 hilos ofreció el máximo rendimiento en la práctica 2.3 y, aunque no podemos elegir qué hilos se lanzan en qué cores, si fuera posible, la ejecución

del hilo de GPU en los cores menos potentes dejaría a los 4 cores con hyperthreading ejecutar la parte más pesada.

```
void hybrid_mm(float *A, float *B, float *C, int M, int N, int K, int W, int F, double
    ↪  *times)
{
    // GPU Parameter setup
    int T = N/W;
    int S = M/W;
    dim3 grid(T, S); //Grid size: SxT
    dim3 block(W, W); //Threads per Block: WxW
    int tile_dim = W; //Tile size: WxW
    // M_gpu = M-F; M_cpu = F
    int M_gpu = M - F;


    //CPU Parameter setup
    //We must advance the pointer of A and C to the F-th row
    float *A_cpu_ptr = A + (M_gpu) * K;
    float *C_cpu_ptr = C + (M_gpu) * N;

    double cpu_time = 0.0;
    double start, stop;

    //Allow nested parallelism so that the main thread can run the GPU kernel
    //and one thread can then create more threads to run the CPU kernel
    omp_set_nested(1);
    omp_set_num_threads(2);

    // Copy the matrices A (partially) and B to the device
    #pragma omp parallel
    {
        int iam = omp_get_thread_num();
        if (iam == 0) {
            hybrid_kernel_run_chrono(M_gpu, K, N, A, B, C, grid, block, tile_dim);

        } else {
            // The rest of the threads execute the rest of the matrix
            start = omp_get_wtime();
            mm_blo(A_cpu_ptr, B, C_cpu_ptr, F, K, N, W, NUM_THREADS-1);
            stop = omp_get_wtime();
            cpu_time = stop - start;
        }
    }

    times[GPU_KERNEL_TIME] = gpu_kernel_time;
    times[GPU_TRANSFER_TIME] = gpu_transfer_time;
    times[CPU_TIME] = cpu_time;
}
```

A diferencia del ejercicio 1 y 2, para este caso, si era de interés medir el tiempo de transferencia entre el device y la CPU, por lo tanto, en la función `hybrid_kernel_run_chrono` hemos ejecutado tanto dichas transferencias de datos como el propio kernel, obteniendo así dichos tiempos:

```
void hybrid_kernel_run_chrono(int M_gpu, int K, int N, float *A, float *B, float *C,
    ↪ dim3 grid, dim3 block, int tile_dim) {
```

```
    // Allocate memory for matrices A_d, B_d and C_d in the device
    float *A_d, *B_d, *C_d;
    allocate_device_matrices(M_gpu, K, N, &A_d, &B_d, &C_d);

    // Recording events GPU
    cudaEvent_t start, stop;
    float milliseconds = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Record event: matrix A (partially) and B copy to the device
    cudaEventRecord(start, 0);
    checkCudaErrors(cudaMemcpy(A_d, A, M_gpu * K * sizeof(float),
        ↪ cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(B_d, B, K * N * sizeof(float), cudaMemcpyHostToDevice));
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&milliseconds, start, stop);
    gpu_transfer_time = milliseconds / 1000.0;

    cudaEventRecord(start, 0);
    sharedABMultiply<<<grid, block, 2 * tile_dim * tile_dim * sizeof(float)>>>(A_d, B_d
        ↪ , C_d, N, K);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&milliseconds, start, stop);
    gpu_kernel_time = milliseconds / 1000.0;

    // Copy the partial result from the device to the host
    cudaEventRecord(start, 0);
    checkCudaErrors(cudaMemcpy(C, C_d, M_gpu * N * sizeof(float),
        ↪ cudaMemcpyDeviceToHost));
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&milliseconds, start, stop);
    gpu_transfer_time += milliseconds / 1000.0;

    //GPU Cleanup
    checkCudaErrors(cudaFree(A_d));
    checkCudaErrors(cudaFree(B_d));
    checkCudaErrors(cudaFree(C_d));
}
```

4.2. Validación de los resultados

La validación de los resultados se ha realizado de la misma manera que en el ejercicio 1 ajustado a los parámetros y restricciones de este ejercicio.

Además, se incluye test con cublas para tamaños grandes.

W	F	KERNEL (s)	TRANSFER (s)	GPU (s)	CPU (s)	TOTAL
4	0	0.245448	0.004317	0.249766	0.000000	0.249766
4	4	0.245609	0.004708	0.250317	0.021924	0.250317
4	8	0.245791	0.004444	0.250235	0.043961	0.250235
4	12	0.246662	0.004345	0.251007	0.070308	0.251007
4	16	0.246757	0.004755	0.251512	0.090823	0.251512
4	20	0.246987	0.004343	0.251331	0.107912	0.251331
4	24	0.247073	0.004326	0.251399	0.135224	0.251399
4	28	0.247923	0.004295	0.252218	0.150617	0.252218
4	32	0.248081	0.004277	0.252357	0.161984	0.252357

Tabla 17: Tabla de tiempos para mulmat_1G1C con F=0,4,8,12,16,20,24,28,32, N=M=K=2048 y W=4.

4.3. Comparación de tiempos y análisis de los resultados

En esta tabla, hemos desglosado los tiempos obtenidos por el programa para diferentes casos en los que la CPU va tomando mayor importancia según el valor de F aumenta. El tiempo de la columna GPU resulta de la suma entre KERNEL y TRANSFER, lo que da el tiempo total necesario para computar datos en GPU desde el momento que los tenemos en CPU. El tiempo de la función `mm_blo` se mide como en la práctica 2.3 y el total resulta del menor tiempo entre GPU y CPU debido a que, como el ejercicio indica, ambas computaciones se ejecutan en paralelo.

A primera vista, choca el hecho de que los resultados son prácticamente idénticos, con tan solo unas 2 milésimas de diferencia entre el mejor y el peor y, si nos vamos a las columnas, GPU y CPU, vemos que el tiempo de GPU siempre es mayor que el de CPU, lo cual significa que estamos usando la CPU eficientemente, puesto que no alarga la duración del programa. Esta sería la situación hasta que el tiempo de CPU superará al de GPU, en ese momento, las dimensiones de las matrices a computar por la CPU son demasiado elevadas y su uso solo nos provocará esperas hasta que esta termine.

Sin embargo, otro patrón emerge y es que, aunque la CPU no empeora el tiempo de GPU y KERNEL, estos sí parecen alargarse al tiempo al que se aumenta F. Como ya hemos dicho antes, estas diferencias son superfluas e incluso podrían achacarse a no haber hecho suficientes ejecuciones para regular los tiempos, sin embargo, las nulas diferencias entre kernels manifiestan un hecho: a la GPU no le importa hacer 32 filas más.

Eso es lo que se ve principalmente en la tabla, el uso de la CPU es tan bajo que hace que la GPU no pueda disminuir su carga de trabajo lo suficiente como para que se note dicho uso. Esto se debe a la naturaleza de la GPU y es que por cada ciclo, se tarda lo mismo, usando 20 SMs o 1, por lo tanto, imaginemos el caso de que llega el último ciclo de ejecuciones y con F=0, se necesitan todos los SMs (suposición). Cuando F=4,16,... seguiremos teniendo que realizar ese ciclo para computar a pesar de que la CPU esté interviniendo.

Esto es lo que se denomina **Wave Quantization** en términos de Nvidia:

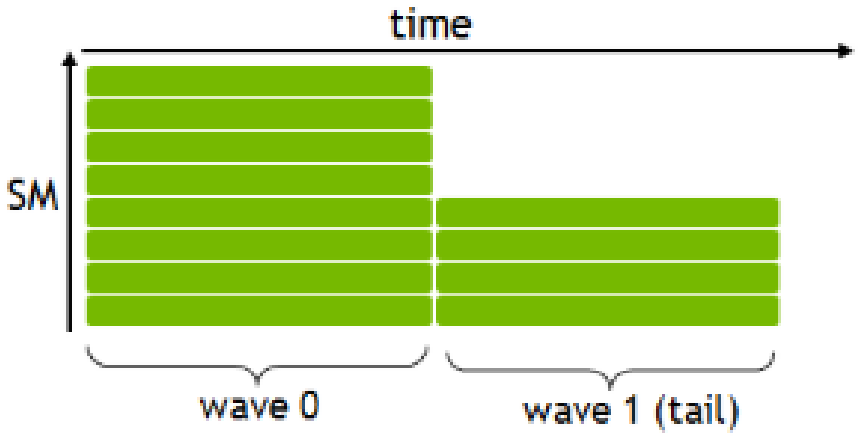

Figura 15: Representación Wave Quantization. Link: https://developer.nvidia.com/blog/optimizing-gpu-performance-tensor-cores/

A pesar de que el segundo wave tenga menos elementos (como nuestro último ciclo en el que F filas se han delegado a CPU), la GPU tarda lo mismo y, por tanto, podemos concluir que usar la CPU muy poco puede que no nos devuelva ganancia frente a no usarla y usarla demasiado podría provocar esperas indeseadas a la propia CPU. El punto que deseamos encontrar sería aquel en el que el uso de la CPU reduzca el número de waves, disminuyendo el tiempo de GPU sin superar dicho tiempo de GPU al realizar el cómputo de su parte de la matriz resultado.