Práctico 3

Nicolás Alejandro Núñez Hosco
Facundo Diaz
Grupo 17
Profesores: Martín Pedemonte, Pablo Ezzatti y Ernesto Dufrechou

GPU

Mayo 2024

Índice

1. Introducción

Este informe detalla la implementación de diversos kernels en CUDA para realizar operaciones matriciales como la transposición de matrices, sumas de elementos con desplazamiento, y productos matriz-vector. Se analiza el impacto de aplicar diferentes optimizaciones con el objetivo de aprovechar al máximo el paralelismo y minimizar la cantidad de transacciones de memoria requeridas.

A lo largo del informe, se analizan los diferentes factores que impactan en el rendimiento de los kernels, como la alineación de memoria, la cantidad de transacciones requeridas y el grado de paralelismo alcanzado.

Para cada kernel implementado, se presentan los resultados de rendimiento obtenidos al ejecutarlos sobre matrices y vectores de distintas dimensiones. Se estudia la influencia del tamaño de bloque utilizado, así como también de la distribución de los hilos dentro de los bloques.

2. Ejercicio 1

2.1. Ejercicio 1 A

Para computar la transpuesta de la matriz, se la divide en bloques de 32x32 de forma que cada hilo de cada bloque corresponda a una única entrada de la matriz original. El kernel diseñado accede a este elemento para posteriormente copiar su valor en la posición correspondiente de la matriz transpuesta.

Los elementos de la matriz son de tipo int de 4 bytes, a su vez, cada segmento de memoria global es de 32 bytes, por lo que cada segmento almacena 8 elementos de la matriz. Cada bloque es de 32x32 threads y, dado que hilos contiguos de un mismo warp acceden a posiciones contiguas en memoria (haciendo al acceso coalesed), cada transacción es capaz de manipular 8 datos de tipo integer (32bytes), por lo que necesarias 4 transacciones para que un warp acceda a 32 elementos contiguos de la matriz.

Por otro lado, como cada warp lee sobre elementos contiguos de la misma fila, cada hilo del mismo debe escribir sobre una fila distinta de la matriz transpuesta, y, como consecuencia, también un segmento distinto de memoria global, por lo que son necesarias tantas transacciones a memoria como escrituras se realicen.

Cada warp hace entonces 4 transacciones en lecturas, y 32 en escrituras, dando un total de 36 transacciones por warp.

Con este kernel, obtenemos los siguientes resultados.

Filas	**Columnas**	**Tamaño de bloque**	Promedio	Desviación
1024	1024	32x32	29.065,5ns	263,9ns

2.2. Ejercicio 1 B

Para favorecer el acceso coalesed en la escritura, usamos bloques de tamaño 1024x1, bajo estas condiciones cada hilo de un warp lee de filas distintas, pero sobre una misma columna, provocando que, al escribir sobre la matriz transpuesta, se deba escribir sobre una misma fila y, por los mismos motivos que en la parte anterior, sobre espacios de memoria contiguos correspondientes a 4 segmentos, por lo que la escritura realizada por cada warp puede ser completada en 4 transacciones de memoria. Además, cada hilo de un mismo warp lee ahora de una columna diferente, por lo que no habrá acceso coalesed en lectura y cada warp necesitará hacer 32 transacciones para completar las lecturas. Con esto, nuevamente obtenemos un total de 36 transacciones por warp.

Filas	**Columnas**	**Tamaño de bloque**	**Promedio**	**Desviación**
1024	1024	1024x1	58.121,5ns	458,5ns

Se observa como, el ejecutar utilizando un tamaño de bloque que favorece las escrituras en vez de lecturas coalesed se obtiene un tiempo de ejecución mayor. Notar que esta diferencia no se debe a un cambio en la cantidad de transacciones dado que esta es igual en ambos casos.

Adicionalmente, al ejecutar el kernel con un tamaño de bloque de 16x64, obtenemos los siguientes resultados:

Filas	**Columnas**	**Tamaño de bloque**	**Promedio**	**Desviación**
1024	1024	16x64	27.487,6ns	191,1ns

En este caso se realizan 4 transacciones de lectura por warp y 32 de escritura (al igual que en el caso de 32×32). Se observa que aumentar el largo de fila del bloque aumenta el rendimiento del programa. Es posible que la disparidad en el rendimiento del programa que se observa al favorecer las escrituras coalesed pueda deberse a algún factor adicional que influya sobre el mismo, como la utilización de la memoria caché por parte de los hilos o alguna otra mejora de rendimiento a nivel hardware distinto al costo relativo de las transacciones de lectura frente a las de escritura (que se mantiene igual para bloques de 32×32 y 16×64).

3. Ejercicio 2

Para efectuar la operación dada, donde a cada elemento de la matriz se le suma el elemento que se encuentra 4 filas por debajo, se divide la matriz en bloques de 32x32, para que a cada hilo de cada bloque le corresponda una única entrada de la matriz junto con su correspondiente sumando. Cada warp efectúa por lo tanto 32 lecturas de elementos en sus posiciones correspondientes y 32 escrituras en la matriz de destino, ambos accesos requiriendo 4 transacciones cada uno, puesto que el inicio de la matriz se encuentra alineado con una dirección múltiplo de 32B, sin embargo, la lectura de los elementos que se encuentran 4 filas por debajo puede requerir 5 transacciones si la matriz no tiene un tamaño de fila múltiplo de 32B, dando un total de 13 transacciones por warp.

Con este algoritmo, obtenemos los siguientes resultados

Filas	**Columnas**	**Tamaño de bloque**	**Promedio**	**Desviación**
4801	4801	32x32	2.037.300,7ns	1.875,0ns

Para mitigar las consecuencias negativas del acceso desalineado, se opta por definir un nuevo kernel de forma que cada hilo que ejecuta al mismo resuelve el valor objetivo para cuatro entradas de la matriz final, por lo que cada warp efectuará la operación sobre 128 elementos. Si bien esto disminuye el grado de paralelismo permite también reducir la cantidad de transacciones de memoria necesarias. Como cada warp debe leer 128 elementos que se encuentran en su fila correspondiente, necesitará 16 transacciones, además de otras 16 transacciones para la escritura del resultado, pero para leer los elementos que se encuentran 4 filas por debajo, como el tamaño de fila no necesariamente es múltiplo del tamaño de bloque, estos elementos podrían estar desalineados respecto al tamaño de los segmentos, haciendo necesaria una transacción adicional, para un total de 17 transacciones. En conclusión esta forma de acceso hace necesarias $\frac{16+16+17}{128} = \frac{49}{128}$ transacciones por elemento, en cambio el primer caso presentado requiere $\frac{13}{32}$ transacciones por elemento, disminuyendo las transacciones totales en un 5.76 %

Con estos cambios en el algoritmo, obtenemos los siguientes resultados:

Filas	**Columnas**	**Tamaño de bloque**	**Promedio**	**Desviación**
4801	4801	32x32	1.607.678,8ns	7.959,4ns

Se observa una mejora en el tiempo al compararlo con el algoritmo anterior.

4. Ejercicio 3

4.1. Ejercicio 3 A

Para implementar la multiplicación planteada, se define una grid 10 con bloques unidimensionales de 1024 hilos cada uno. Cada uno de estos hilos multiplicará una fila de la matriz con el vector $\vec{v}$, y escribirá el resultado en la posición de su id dentro del vector resultado $\vec{r}$.

Para este algoritmo, obtenemos los siguientes resultados:

Filas	**Columnas**	**Tamaño de bloque**	**Promedio**	**Desviación**
10240	256	1024	406.462,44ns	3.907,1ns

4.2. Ejercicio 3 B

Este kernel presenta ineficiencias relacionadas con el acceso a memoria, puesto que cada hilo debe acceder a 1024 bytes contiguos en memoria, el acceso no es coalesed, por lo que cada hilo realiza 256 transacciones para la lectura de la fila, 256 transacciones para la lectura de $\vec{v}$, y una transacción para la escritura del resultado del producto fila-vector en $\vec{r}$, dando un total de $(256 + 256 + 1) * 10240 = 5{,}253{,}120$ transacciones en total.

Para reducir el número de transacciones, modificamos el kernel, asignando un bloque por cada fila, y dividiendo cada bloque en 256 hilos, por lo que se asignará a un hilo por cada elemento de la matriz, de forma que hilos contiguos utilicen información de direcciones contiguas de memoria. Luego, cada hilo efectúa la multiplicación de su elemento en la fila, por su correspondiente elemento en $\vec{v}$, y guarda este resultado en un arreglo almacenado en memoria compartida, por lo que esta disponible para todo el bloque. Notar que si bien todos los hilos del bloque escriben sobre este arreglo, no lo hacen sobre la misma dirección de memoria por lo que no es necesario utilizar operaciones atómicas.

Por último, se calcula la sumatoria de los elementos de este arreglo, y se almacena el resultado en la posición correspondiente al bloque en $\vec{v}$. Esta suma se implementa de dos formas diferentes, por un lado, antes de finalizar el kernel el thread con id = 0 la efectúa, y por otro lado, con el objetivo de optimizar la operación paralelizando esta sumatoria, se utiliza un algoritmo de reducción, en el que cada hilo del bloque suma elementos de a pares, reduciendo en cada paso a la mitad el arreglo a sumar. Notar que esta implementación paraleliza el calculo de los productos escalares requeridos para obtener el resultado final. En esta implementación cada warp necesitará hacer 1 transacción para obtener 8 elementos contiguos de la matriz, además de 1 transacción para obtener 8 entradas del vector, y se necesitará una transacción de escritura por cada bloque. Dando un total de $\frac{256}{8} \cdot 10240 \cdot 2 + 10240 = 665{,}600$ transacciones, esto es casi 8 veces menos transacciones que el caso anterior.

Se obtienen los siguientes resultados:

Sumatoria	**Filas**	**Columnas**	**Tamaño de bloque**	**Promedio**	**Desviación**
Secuencial	10240	256	1x256	60.786,7ns	1.020,0ns
Reduce	10240	256	1x256	133.035,5ns	281,5ns
Secuencial	10240	1024	1x1024	457.596,5ns	3.247,5ns
Reduce	10240	1024	1x1024	667.118,0ns	279,3ns

En primer lugar se observa que las optimizaciones realizadas reducen efectivamente el tiempo de ejecución a casi un séptimo del original en promedio. En segundo lugar, el algoritmo que utiliza reduce presenta un tiempo de ejecución mayor obtenido al utilizar suma secuencial. Esto puede deberse a que el overhead añadido por el uso de esta técnica es mayor a la ganancia de tiempo que la misma provee. Al aumentar el número de columnas de la matriz multiplicadora se puede observar como esta diferencia de tiempo de ejecución es reducida, como consecuencia de que el tiempo de overhead representa un porcentaje menor del tiempo total de ejecución.