Práctico 4 - GPGPU

Santiago Calvo
Estudiante de Ingeniería en Computación
5.055.578-2
Montevideo, Uruguay
santiago.calvo.vello@fing.edu.uy

***Resumen*—Informe de los resultados de la realización del práctico 4 de la materia Computación de Propósito General en Unidades de Procesamiento Gráfico.**

I. Introducción

Con el objetivo de realizar un estudio del impacto de la memoria shared sobre distintos algoritmos sobre la plataforma CUDA, se realizan diversos estudios ante 4 funciones con accesos distintos a memoria. De esta manera se obtienen métricas necesaria para el estudio de dicho impacto.

II. Instrucciones de ejecución y compilación

La ejecución de los algoritmos se realizó sobre cluster.uy utilizando los archivos bash para cada ejercicio contenido en el entregable.

Dichos archivos tienen las pruebas realizadas con métricas obtenidas.

Cada launchX.sh escribe sobre su salidaX.out

III. Ejercicio 1

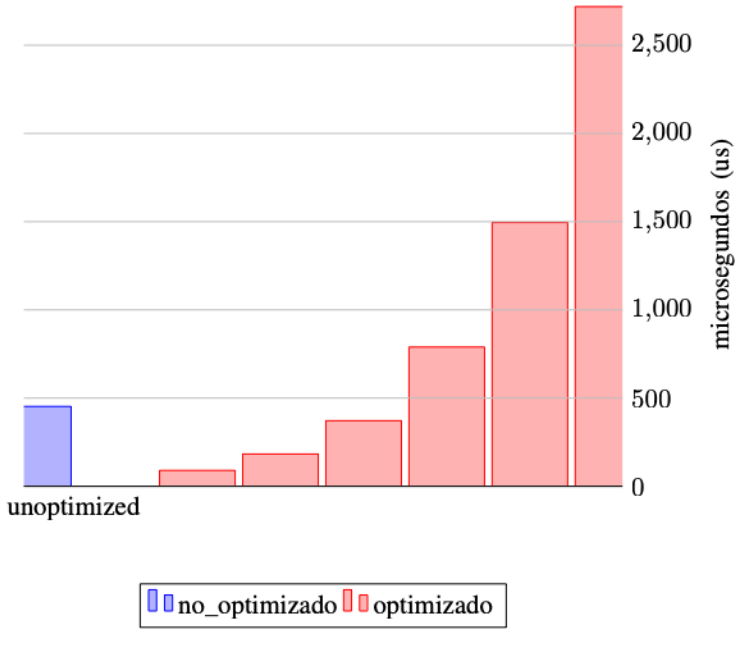
Para contar la ocurrencia de caractéres sobre un texto realizó previamente una implementación naive que creaba un thread por cada caracter en el texto. De esta manera se obtiene su posición en un array de 256 bytes (uno para cada caracter) donde se incrementará un contador.

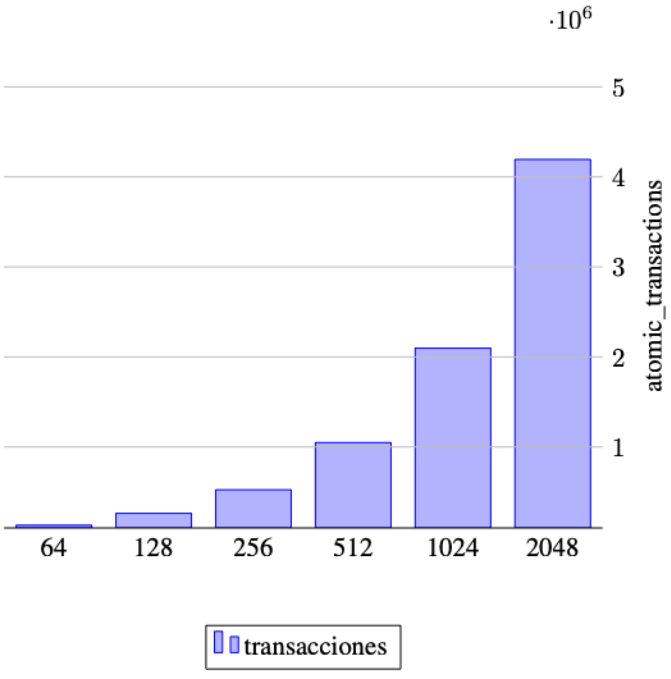
Este contador debe incrementarse de manera atómica dado que todos los threads tendrán acceso a estas direcciones.

Es posible mejorar la velocidad de ejecución sin embargo, si se utiliza memoria compartida con el objetivo de minimizar la cantidad de accesos a memoria global.

Es importante notar que el aumento de la cantidad de bloques implica un aumento importante en la demora del la optimización de este algoritmo. En la gráfica III se muestra como para la cantidad de bloques 64,128,256,512,1024,2048 hay un aumento en el tiempo de ejecución llevando este a tener una demora mayor al algoritmo sin memoria compartida.

Al hacer un estudio de accesos a memoria global por los atomic add, vemos como a medida que aumenta la cantidad de bloques, existe un aumento en la fragmentación del algoritmo produciendo un aumento signficativo en la cantidad de atomic adds necesarios. Esto aumenta particularmente el acceso a memoria pues cada atomicAdd necesario realiza una lectura a memoria previa para luego realizar la escritura, que a su vez, causa demoras entre threads dada le atomicidad necesaria de la operación. A su vez existe la falta de primitivas que permite la cooperación entre threads, como lo es en caso de memoria compartida.





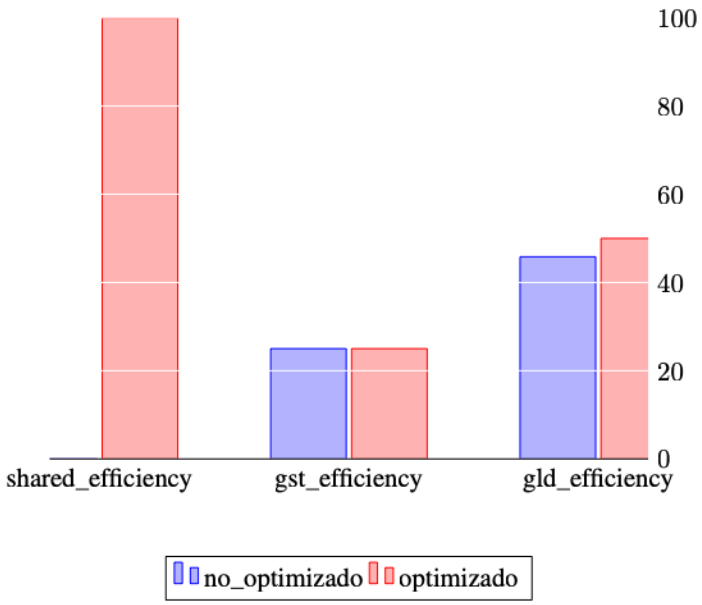
IV. Ejercicio 2

Dado el stencil dado, se requiere realizar optimizaciones en los accesos a memoria para evitar el uso de la memoria global. De vuelta, mediante el uso de shared memory esto se puede lograr.

Dado que la memoria shared es considerablemente más rápida, su uso permite reducir el tiempo de ejecución de nuestro función con stencil a la mitad como se puede ver en IV.

Se puede apreciar su uso de memoria en la gráfica V donde se ve el la eficiencia de 100 % de shared memory por la función optimizada.

Un agregado es ver como el aumento del tamaño de la matriz llevó a un aumento de tiempo de ejecución el cual fue aún mayor para la versión no optimizada debido a la utilización de memoria global. Este creció en mayor medida puesto que los warps en la versión optimizada obtendrían su fragmento de shared memory disminuyendo la cantidad de lecturas a memoria global e incrementando entonces la eficientes de lectura como eficiencia en el uso de memoria compartida.

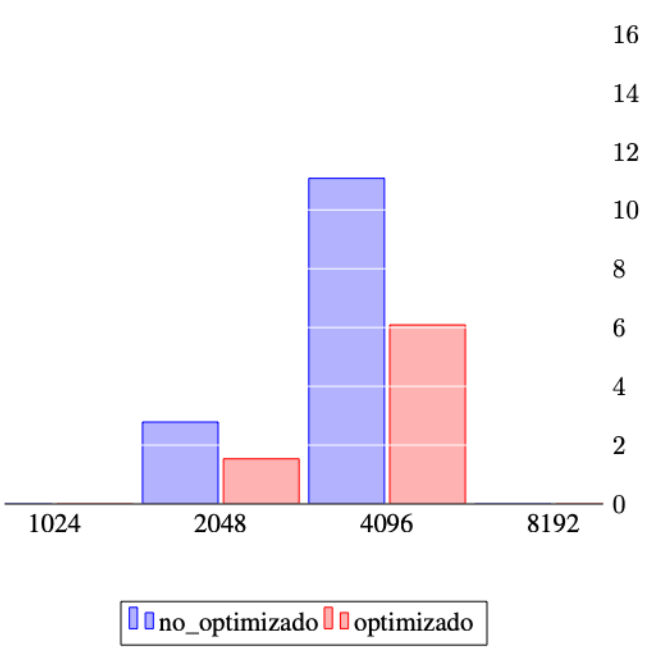


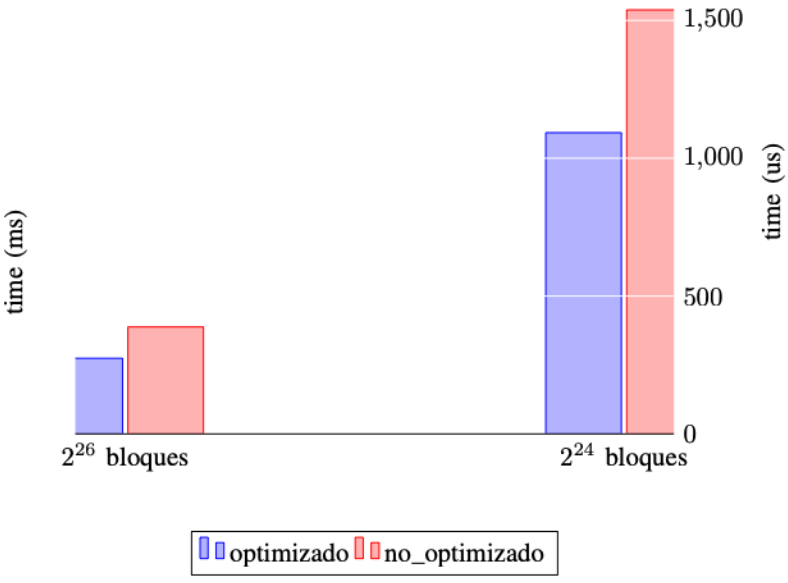
V. Ejercicio 3

Para la optimización de esta función utilizaremos memoria compartida para mejorar la eficiencia de a memoria global.

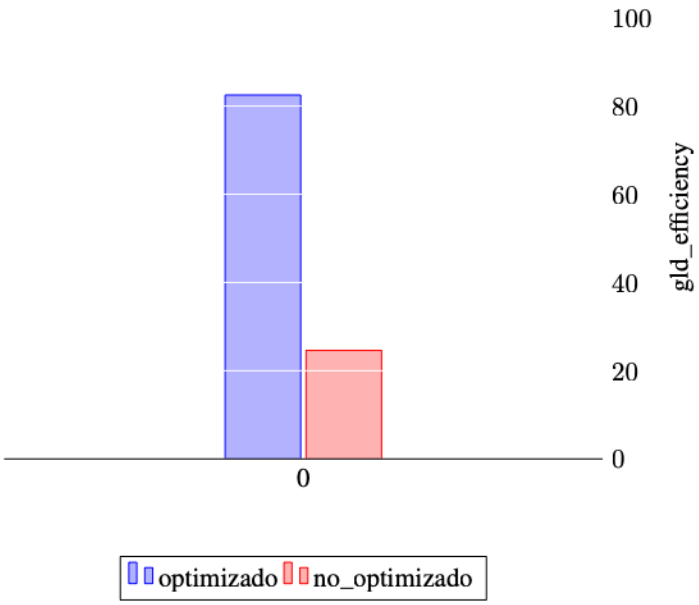
El algorítmo realiza permutaciones dentro de bloques. Lo que realiza significativos accesos a memoria global. Estos pueden evitarse llevando el espacio de datos leídos (restringido al tamaño de bloque) a memoría local, permitiendo entonces aprovechar mejor las lecturas a memoria global.

Como se demuestra en el siguiente gráfico, esto resulta en un incremento en la velocidad de ejecución de la función.





Para ambas cantidad de bloques, la métrica gld_efficiency es:



VI. Ejercicio 4

Dada una función con accesos por fila a direcciones de memoria, se busca realizar optimizaciones para lograr disminuir su tiempo de ejecución.

Esta función realiza la reducción por columna de una matríz, por bloque, teniendo que realizar acceso a cada fila dentro de un warp de threads.

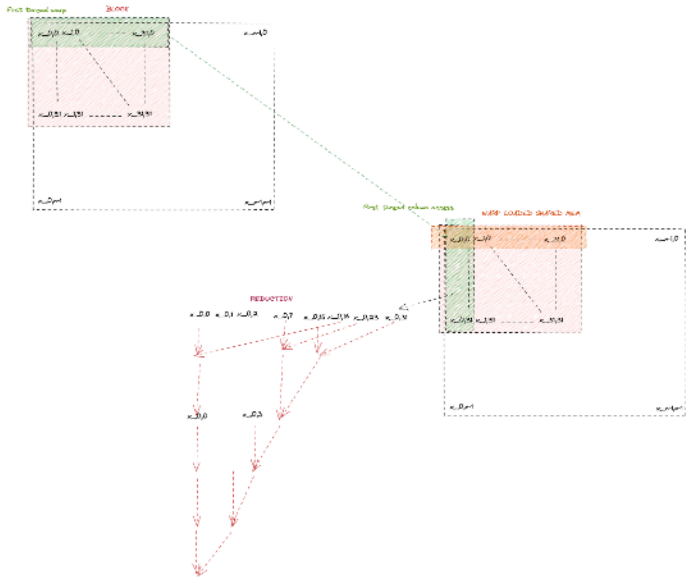


Figura 1. Image showing works of function

Al ser las lecturas realizadas por los threads por columna, fila existe un desaprovechamiento de los warps provocado por conflictos en la lectura de los bancos de memoria.

La solución a esto es realizar la lectura sobre la matriz de manera traspuesta. De esta forma evitamos el conflicto entre bancos de memoria permitiendo a los warps tener acceso a memoria rápido.

Así pasamos de tener un shared efficiency de 0 % a 100 %. Esto sin embargo no implica una velocidad en la ejecución de la función dado que este cambió produce un aumento en el número de accesos necesarios a la matriz debido al mal acceso desaprovechando la cache.