Proyecto final - 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 proyecto final de la materia Computación de Propósito General en Unidades de Procesamiento Gráfico realizando calculos sobre matrices en distintos formatos.**

I. Introducción

Tomando matrices dispersas (matrices con una cantidad de elementos distintos de cero mínima) se desea realizar el cálculo del producto de estas sobre un vector denso.

Dado que las matrices son dispersas, sería un desperdicio nuestro objetivo guardar todos los elementos de esta. Por esta razón tomamos dos representaciones distintas para implementar.

La primera, una matriz en formato comprimida por filas (CSR). La segunda, una matriz comprimida por bloques.

Para ambas representaciones se crearon kernels de CUDA.

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. Hardware y contexto de pruebas

Todas las pruebas fueron realizadas sobre el clusterUy.

Las pruebas fueron realizadas sobre matrices creadas con el mismo seed, por lo que todas las matrices entre las pruebas de la misma cantidad de bloques son iguales.

Las pruebas se establecieron sobre la cantidad de bloques para la matriz dispera en formato de bloques correspondientes. Por lo que la cantidad de filas/columnas real será el número de bloques x8.

Finalmente, es importante notar que todas las matrices utilizadas fueron matrices cuadradas por simplicidad.

IV. Ejercicio 1

El primer kernel es la multiplicación en formato CSR sobre sobre un vector denso.

Inicialmente se realizó dicha implementación utilizando memoria global. Posteriormente se realizó el cambio de esto para utilizar memoria compartida para la los datos de fila.

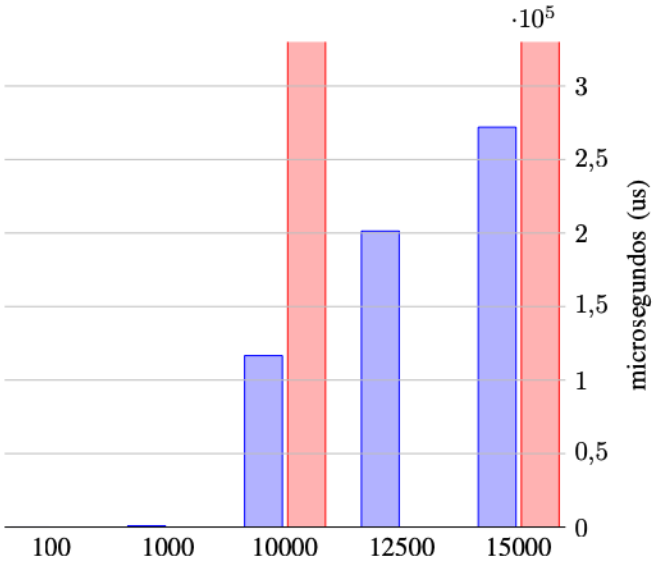
Las implementaciones basan su estrategía en la ejecución de un thread por fila. Se prefirió esta estrategia sobre el uso de un bloque por celda para evitar el uso de atomicAdds o warp reductions.

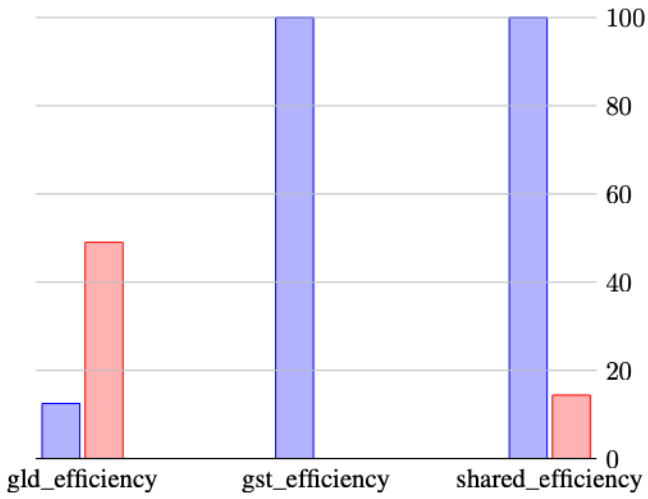
Sin embargo, esta implementación no hace un aprovechamiento del uso de warps para la lecutra de datos dado que cada thread se encarga de una fila entera, no logrando accesos coalesced. Por esto se realizó una implementación donde cada thread realiza operaciones sobre una celda de la matriz. Posteriormente se realiza una reducción del resultado y se escribe de manera atomica el resultado.

Este último sin embargo, sufre de ineficiencia en la escritura de memoria dado que un thread por bloque escribirá en una posición de memoria igual a otros bloques que esten leyendo datos de la misma fila.

En las siguientes figuras se ven los tiempos de ejecución para distintos tamaños de matriz, como a su vez una comparación con la eficiencia de accesos globales y a shared memory.

Implementacion	1	2
100	47.80 us	21.76 us
1000	920.70 us	Error
10000	116620.00 us	435060.00 us
12500	201330.00 us	Error
15000	272000.00 us	1162890.00 us
gld_efficiency	12.50 %	49.03 %
gst_efficiency	100.0 %	0.00 %
shared_efficiency	100.0 %	14.40 %





Como se puede ver en los resultados, en el caso de matrices de 100 bloques x 100 bloques, el resultado fue benefifcioso para la implementación que utiliza un kernel por celda. Quizá debido al uso de atomicAdds y la ineficiencia en la escritura, este no fue el caso para el resto de las pruebas (dos de ellas fallaron por problemas de acceso).

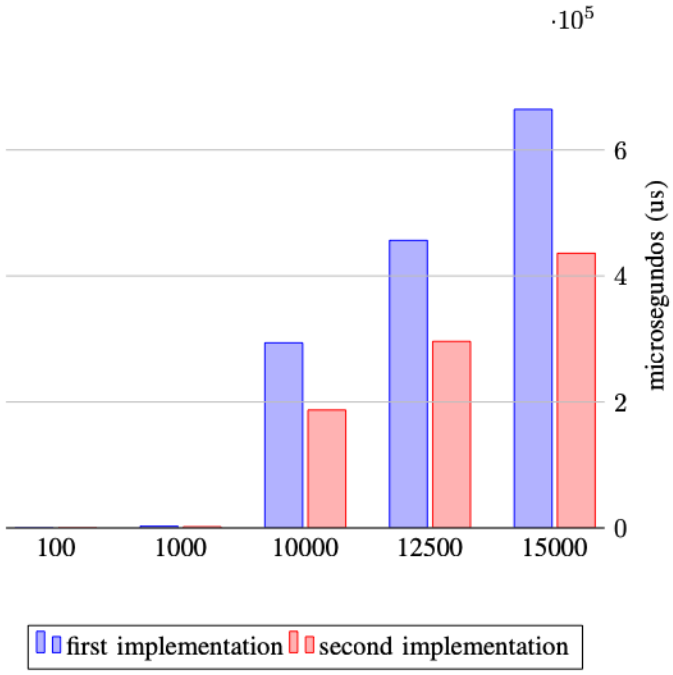
V. Ejercicio 2

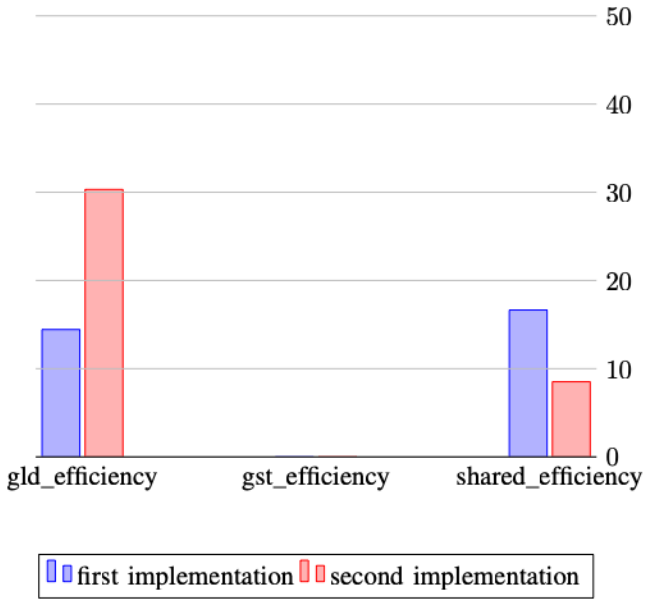
Para el cálculo de la multiplicación entre una matriz diseprsa en formato de bloques, se realizó un kernel que para cada celda de la matríz se evaluaría su valor y se realizaría la multiplicación con su posición correspondiente en el vector denso.

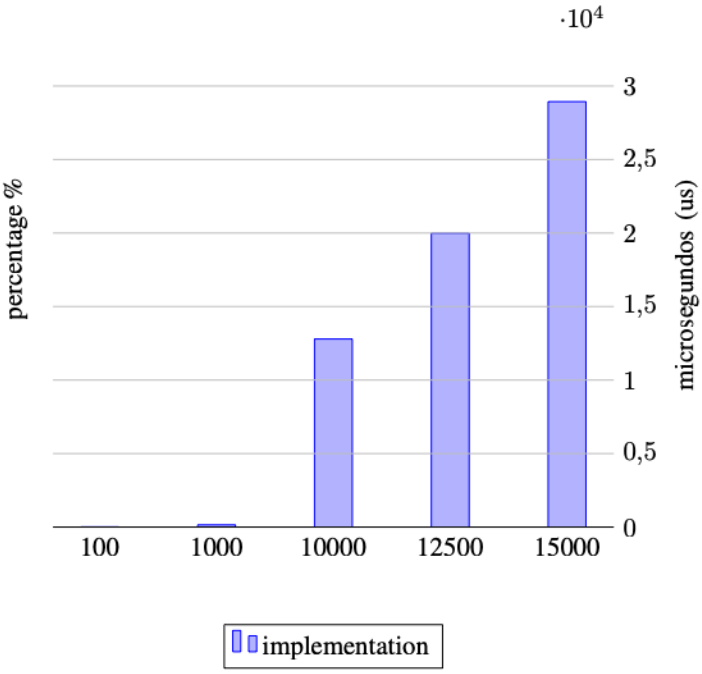
Se realizaron dos implementaciones para esto. Una de ellas crea un bloque denso sobre cada thread, y finalmente el thread inicial realiza la suma de las filas y realiza una suma atomica al espacio de memoria correspondiente del vector resultado.

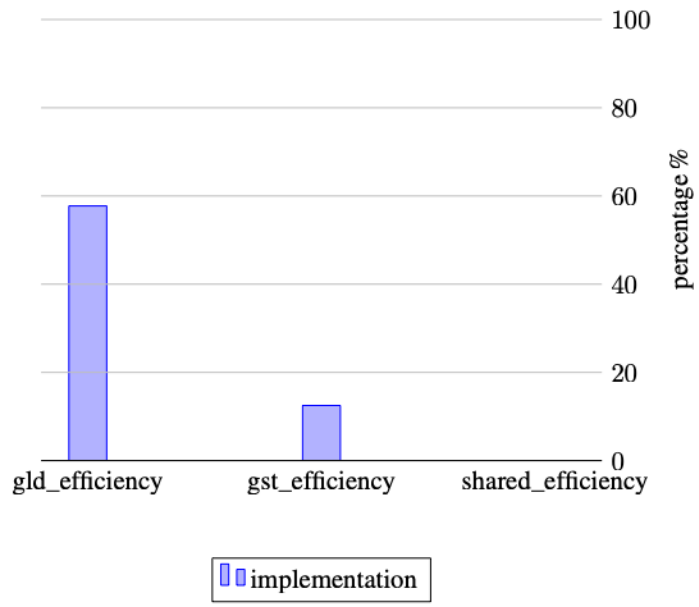
Como optimización se realizó un kernel similar pero que la suma se realizaría en vez utilizando las primitivas de warps para $\_\_shfl\_xor$ permitiendo evitar el uso de la memoria compartida y mantener la reducción de atomic adds.

Implementacion	1	2
100	28.99 us	Error
1000	2687.60 us	1874.10 us
10000	293830.00 us	187200.00 us
12500	456370.00 us	296090.00 us
15000	664350.00 us	435950.00 us
gld_efficiency	14.44 %	30.32 %
gst_efficiency	0.00 %	0.00 %
shared_efficiency	16.65 %	8.52 %









VI. Ejercicio 3

Para realizar una comparación de la velocidad de estos algoritmos se realiza una implementación de CSR utilizando la librerías cuSparse de CUDA. Esta librerías realiza un análsisis de la matríz logrando resultados muy rápidos.

Implementacion	1
100	17.055 us
1000	151.52 us
10000	12793.00 us
12500	19989.00 us
15000	28938.00 us
gld_efficiency	57.74 %
gst_efficiency	12.50 %
shared_efficiency	0.00 %

VII. Comparación de las implementaciones

Aún hay espacios para mejoras. Particularmente en los accesos a memoria. Un mejor uso de los accesos shared podrían ser beneficiosos. Dada la baja eficiencia de carga global en la mayoría de los kernels implementados.

Como se puede ver en los resultados, la implementación del producto de CSR cuSparse toma un tiempo mucho menor, por lo que sería la opción ideal si se desea realizar este tipo de operaciones. A su vez esta podría obtener mejores optimizaciones en el caso de utilizar CUDA Graph Capture (permitiendo definir tareas como pasos en un grafo) y Hardware Memory Compression.

Una posible mejora de estas implementaciones es utilizar el primer algoritmo de CSR pero separando cada fila en varios

threads. De esta manera se podría hacer una reducción de toda la fila obteniendo los beneficios de la segunda implementación sin utilizar atomicAdds.

Algo similar se podría realizar sobre la implementación por bloques si se implementara el kernel de tal manera que vaya por fila en vez de por columna, reduciendo la necesidad de atomicAdds.

Una extensión interesante a probar es la comparación de todos estos algoritmos con cuBLAS (una implementación de rutinas básicas de algebra lineal sobre CUDA). Se ha reportado que las optimizaciones de estas librerías logran velocidades mayor que cuSparse dependiendo de la densidad de la matriz, por lo que un estudio interesante sería la comparación entre estos algoritmos sobre distintas densidades.