Universidad Austral de Chile
Conocimiento y Naturaleza

19/12/23

INFO188

Matriz Dispersa x Vector

Paralelismo en GPU sobre operaciones de matrices

Estudiantes:
Rodrigo Erlandsen,
Patricio Lobos,
Benjamin Parra,
Pascal Salinas.

Profesor:
Cristóbal A. Navarro

Ayudante:
Alejandro Villagrán

Índice

Figuras

1. Introducción:

La optimización de algoritmos y estructuras de datos es esencial para mejorar el rendimiento de las operaciones computacionales, especialmente cuando se trata de matrices. En esta tarea, abordaremos el desafío de implementar eficientemente el producto de una matriz dispersa por un vector, aprovechando la paralelización tanto en la CPU como en la GPU.

Nuestra tarea consiste en proponer una estructura de matriz dispersa que sea eficientemente paralelizable tanto en la CPU como en la GPU. Implementaremos dos versiones de la solución: una utilizando OpenMP para la paralelización en CPU y otra empleando CUDA para la ejecución en GPU. Se explorará el rendimiento en términos de tiempo de ejecución y uso de memoria, comparando nuestro enfoque con la implementación convencional que almacena la matriz completa. Se utilizará el algoritmo Compressed Sparse Row y será comparado con el algoritmo paralelo clásico de multiplicación de matriz por vector.

1.1 Máquina utilizada

Las pruebas serán realizadas sobre una máquina con las siguientes características:

MOTHERBOARD:	Godlike x99a carbon
SO:	Arch Linux
RAM:	130 GiB DDR4
GPU:	NVIDIA GeForce RTX 3090 Ti
CPU:	Intel Core i7-6950X; 10 Nucleos; 2 Threads/Núcleo
PCI-e GEN:	3.0

Características técnicas.

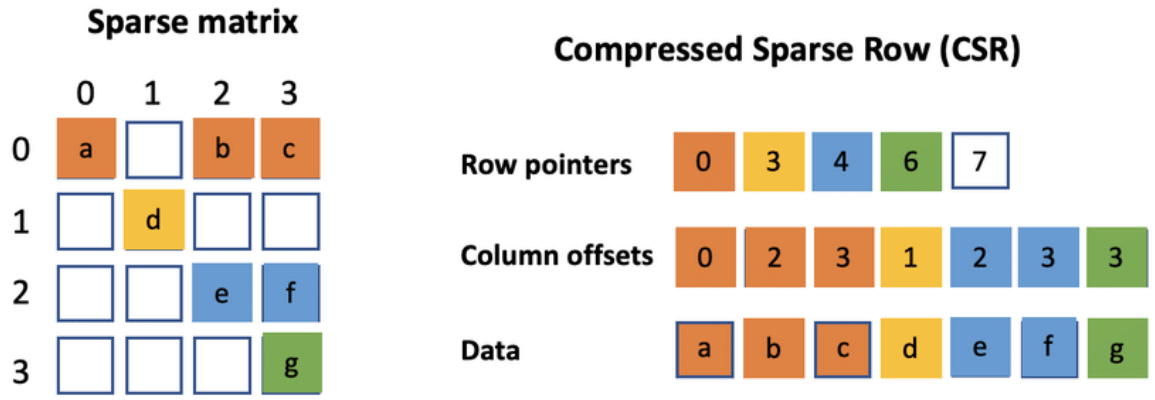
2. Desarrollo:

Para el desarrollo de este ejercicio usamos una estructura para la matriz llamada CSR (Compressed Sparse Row) para poder cargar la matriz sin ningún 0 en la RAM, para su implementación tuvimos que pasar una matriz bidimensional a tres listas: RI (row index), CI (column index), CSR(que es donde guardamos los valores).

RI guarda desde que dato comienza la fila de la matriz. ej: RI = 0,3,4,6,7. CSR = abcdefg significa que de 0 a 3, que en este caso los valores abc son pertenecientes a la primera fila.

CI guarda en qué posición de la columna que está cada valor del CSR. ej: CSR anterior y CI = 0,2,3,1,2,3,3 siendo así el dato a esta en la posición 0 para el cálculo de la matriz con el vector.

CSR es donde se guardan los valores de la matriz distintos de 0 en una lista unidimensional.



Estructura de datos a utilizar.

fuente:
https://www.researchgate.net/figure/The-Compressed-Sparse-Row-CSR-format-for-representing-sparse-matrices-provides-a_fig1_357418189

Con esta estructura pudimos utilizar la matriz ahorrando cargar los 0 en la RAM de manera eficiente.

3. Metodologías:

Se realizaron pruebas variando el número de threads para una matriz cuadrada de lado 2^{15} y un vector también de tamaño 2^{15}. Estas pruebas fueron realizadas sobre la máquina especificada en el punto **1.1** de este documento. Los tiempos se midieron desde un programa hecho para pruebas que consiste en ejecutar sucesivamente los programas con las configuraciones establecidas. Luego de medir los tiempos, se tabularon con excel y se realizó un gráfico. Para cada dato se sacaron las medias de 3 pruebas consecutivas. El blocksize utilizado fue de 1024, la densidad fue de 0.3. Las pruebas se realizaron de esta manera:

- Variando la cantidad de threads con una matriz de lado 2^{15}.
- Variando el tamaño de la matriz con un thread.

Los programas fueron escritos en CUDA y C++, estos se pueden encontrar en el siguiente repositorio: **https://github.com/Paskiben/Matriz-Dispersa-x-Vector/tree/main**

4. Resultados:

4.1 Pruebas variando la cantidad de threads

n° threads	tiempo	speedup
1	0,404456	1
2	0,209014667	1,93506038
3	0,1453215	2,78318074
4	0,112347333	3,60004985
5	0,090265	4,4807622
6	0,077263667	5,23475027
7	0,068550667	5,90010309
8	0,061024333	6,62778236
9	0,056609333	7,14468757
10	0,053177333	7,60579696
11	0,065226667	6,20077678
12	0,061302	6,5977619
13	0,059460333	6,80211458
14	0,055055333	7,34635458
15	0,056275	7,18713461
16	0,055152333	7,33343406
17	0,055079	7,34319795
18	0,053705	7,53106787
19	0,054971333	7,35758032
20	0,055438333	7,29560172

Tabla Speedup según threads CPU.

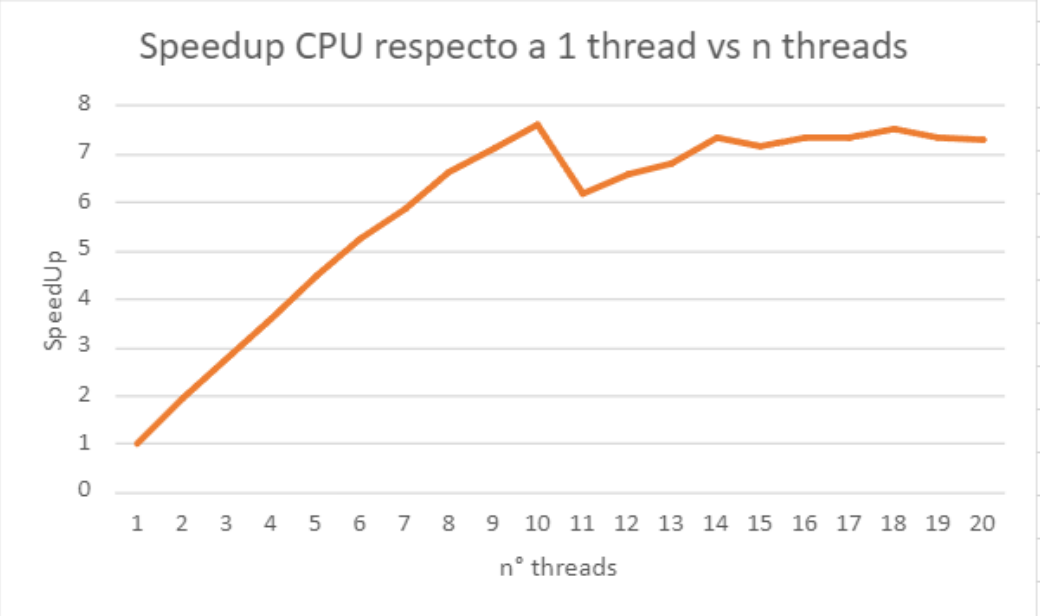


Gráfico Speedup según threads CPU.

4.2 Pruebas variando el tamaño de la matriz

lado matriz	tiempo CPU	Tiempo GPU	SpeedUp
2^6	0,000018	0,000581	0,030981
2^7	0,000025	0,000192	0,130208
2^8	0,000043	0,00020033	0,214642
2^9	0,000121667	0,00024967	0,487316
2^10	0,000402	0,00053433	0,752339
2^11	0,001414667	0,00088367	1,600905
2^12	0,005633333	0,001641	3,432866
2^13	0,022848333	0,003065	7,454595
2^14	0,090578667	0,006451	14,04103
2^15	0,377326	0,06585833	5,729358
2^16	1,7359987	0,33826133	5,132123

Tabla Speedup GPU a CPU según tamaño.

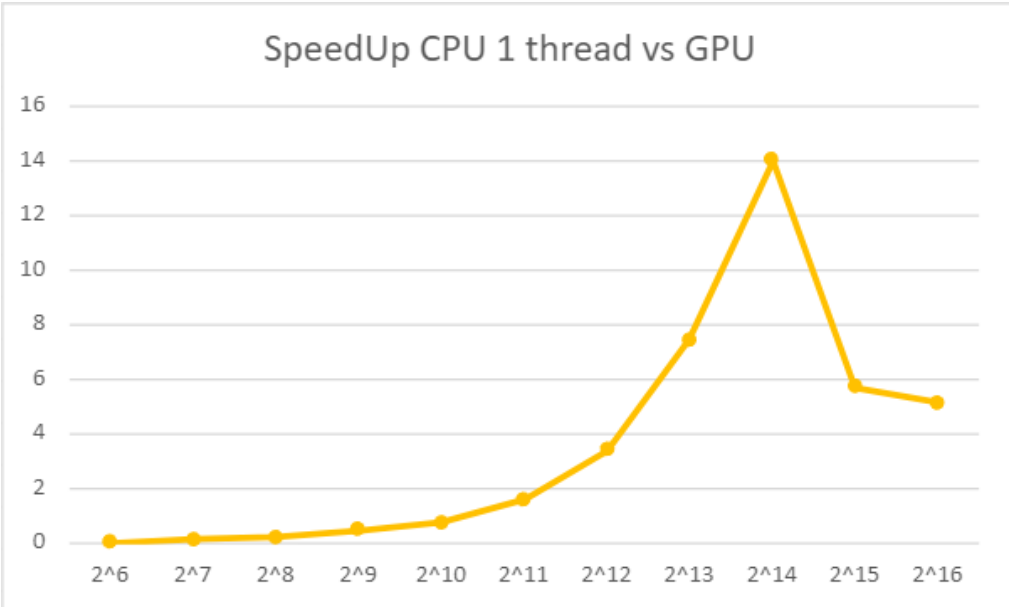
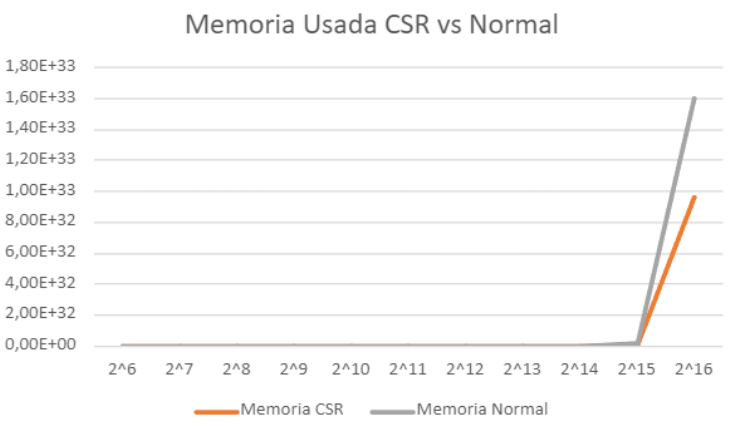


Gráfico Speedup GPU a CPU según tamaño de entrada.

4.3 Memoria utilizada con nuestro método vs Memoria utilizada por el método tradicional .

lado matriz	Memoria CSR	Memoria Normal
2^6	9,60E+12	1,60E+13
2^7	9,60E+14	1,60E+15
2^8	9,60E+16	1,60E+17
2^9	9,60E+18	1,60E+19
2^10	9,60E+20	1,60E+21
2^11	9,60E+22	1,60E+23
2^12	9,60E+24	1,60E+25
2^13	9,60E+26	1,60E+27
2^14	9,60E+28	1,60E+29
2^15	9,60E+30	1,60E+31
2^16	9,60E+32	1,60E+33



Comparación de memoria utilizada.

Nuestro método utiliza { $2n^2$ • densidad • 4_{bytes}+ (n+1)•4_{bytes} + n•4_{bytes} } de memoria, mientras que el método tradicional utiliza { n^2 • 4_{bytes} + n • 4_{bytes}, } cuya, por lo que, con densidad 0,3 la intersección y momento donde nuestro método utiliza menos memoria es en n=4 por lo que no es significativo. El cambio proporcional asintótico sería el siguiente:

$L = \lim_{n \to \infty} \frac{2n^2*d*4 + (n+1)*4 + n*4}{n^2*4 + n*4}$, luego como los coeficientes de los polinomios son iguales:

$L = \frac{2*d*4}{4} = 2d.$ Finalmente como $0 < d < 1$, queda que $0 < L < 2$ = Constante.

De esto queda probado que la memoria proporcional extra utilizada es menor con densidades entre]0, 0.5] la memoria utilizada es <= a la utilizada por el algoritmo clásico. Con densidades de]0.5, 1] la memoria extra es a lo más el doble

5. Reflexiones y conclusión del experimento:

A partir de los resultados, se puede notar un buen uso del paralelismo, aumentando la velocidad considerablemente al utilizar tamaños de entrada grandes.

La paralelización fue efectiva en este experimento, mostrando beneficios significativos al performance de la operación según la cantidad de distribución del trabajo asignada. En el *Gráfico Speedup según threads* se observa como las diferencias de rendimiento se van reduciendo.

La ganancia de rendimiento en el algoritmo clásico utilizando paralelismo en CPU muestra un estancamiento luego de los 10 threads. Esto ocurre porque el CPU de la máquina solo cuenta con 10 núcleos, por lo que el resto de los threads no mejoran el rendimiento.

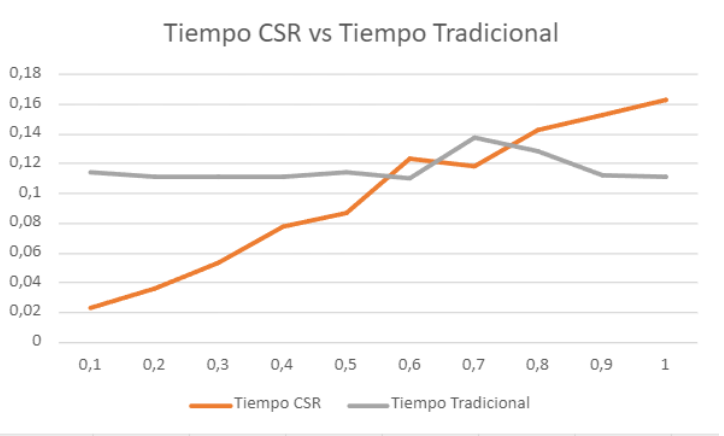
Algo parecido ocurre cuando se tratan con tamaños de matriz de lado 2^{15}, el Speedup disminuye considerablemente, esto puede suceder porque la GPU se satura desde ese punto, pues el programa podría estar invirtiendo más tiempo en coordinar y gestionar los hilos o los bloques en comparación con el beneficio real obtenido por la ejecución paralela. Otra opción puede ser que se esté sobrepasando la capacidad del caché de la GPU. Es requerido un análisis futuro para resolver este punto y aclarar el culpable del decremento de rendimiento.

En resumen, la paralelización mostró mejoras de rendimiento hasta cierto punto, con beneficios considerables tanto a la memoria usada como al tiempo utilizado.

6. Acerca de la densidad óptima:

Otro resultado que obtuvimos fue el punto de densidad en el cual la matriz dispersa comienza a ocupar más memoria, que es en 0.5 como se demostró anteriormente en el punto 4.3. El punto de densidad en el cual el cálculo de la matriz dispersa CSR es notablemente menos eficiente que el clásico en d≈0.8 lo cual se puede ver en el siguiente gráfico:

densidad	Tiempo CSR	Tiempo Tradicional
0,1	0,022611	0,11397433
0,2	0,036592667	0,111723
0,3	0,053920333	0,11093367
0,4	0,077589	0,11185733
0,5	0,087242	0,11412533
0,6	0,123963667	0,11053967
0,7	0,118259667	0,13791667
0,8	0,142424333	0,128158
0,9	0,152897333	0,11205567
1	0,163122	0,11099367



Comparación de tiempo según densidad

Alrededor de d=0.5 también, el tiempo y la memoria utilizada son similares.