Práctico 5

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

Programación masivamente paralela en procesadores gráficos (GPUs)

Junio 2024

Índice

1. Introducción

En este informe se exploran los beneficios posibles del uso de librerías de CUDA de alto rendimiento, en particular Thrust y Cub. Estas proporcionan funciones de alto nivel eficientes para el uso de GPUs.

En primer lugar se realiza una implementación de la operación de SCAN para arreglos de tamaño $1024 \cdot 2^N$ utilizando CUDA y se compara el rendimiento de la misma con el obtenido al utilizar las librerías mencionadas.

Luego se emplean varias de las funcionalidades provistas por Thrust para paralelizar un algoritmo secuencial destinado a determinar un orden de ejecución de la resolución de ecuaciones correspondientes a filas de una matriz triangular dispersa.

Posteriormente se analizan los resultados obtenidos para varias matrices con distintas características con el fin de determinar bajo que condiciones el programa se ve beneficiado del uso de paralelismo.

2. Ejercicio 1

2.1. Algoritmo Implementado

Se implementa la operación Scan mediante una función que hace uso de dos kernels distintos.

Primero, por limitaciones del tamaño posible para un bloque de hilos, se divide el array en segmentos de 1024 elementos estos bloques son copiados a memoria compartida, de esta forma se logra una mayor eficiencia en el acceso a memoria. Luego, se aplica SCAN sobre cada uno de estos segmentos, siguiendo el algoritmo presentado en el curso y, antes de finalizar la ejecución del kernel, se almacena el último elemento resultante de aplicar la operación sobre cada de cada segmento en un array auxiliar que tendrá, por lo tanto, tamaño igual a la cantidad de bloques utilizados.

A continuación, se aplica recursivamente la misma función sobre este array auxiliar (es decir, se aplica la función scan sobre este), para luego ejecutar el segundo kernel, el cual esta destinado a sumar a cada segmento del array original, su elemento correspondiente del array auxiliar, por ejemplo, a todos los elementos del segundo segmento se le sumarán el elemento 2 del array auxiliar.

La función recursiva tiene como paso base la situación en la que solo es necesario un bloque, es decir, cuando el array de entrada tiene 1024 o menos elementos, en este caso, aplica la operación de scan sobre este único segmento y lo devuelve. Notar que en cada paso, el tamaño del array a computar se divide entre 1024.

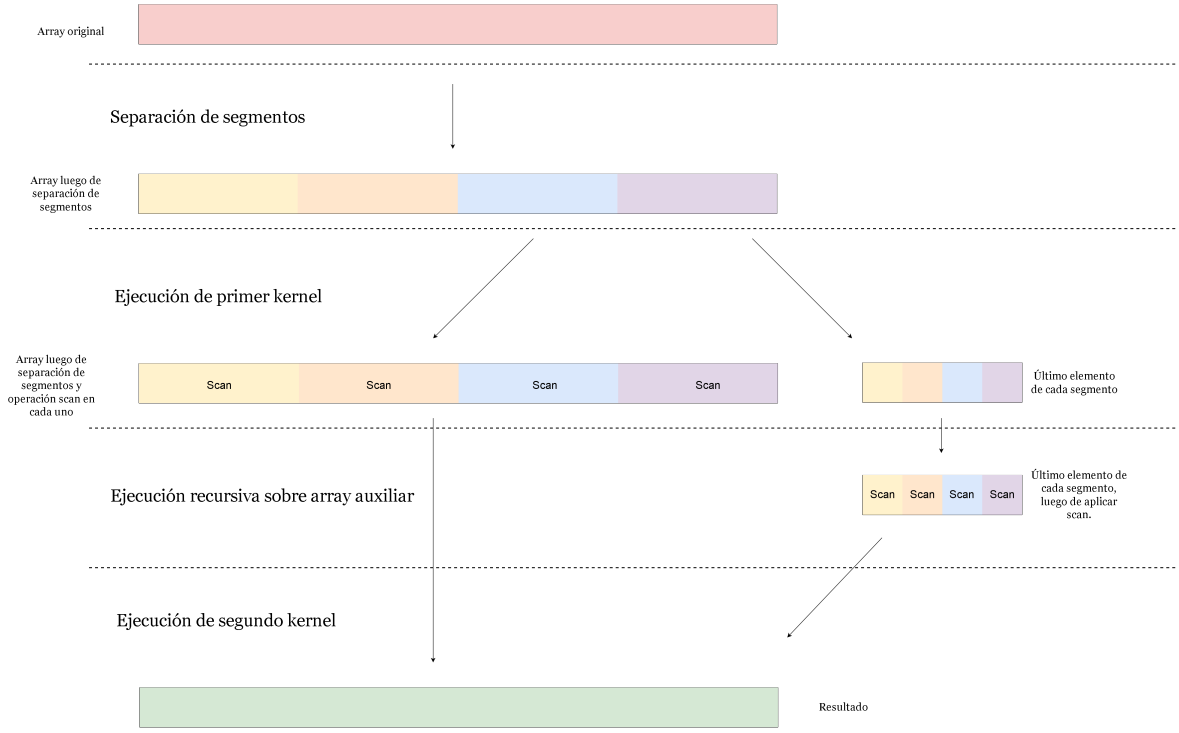

Figura 1: Ejecución de funcion scan

2.2. Resultados experimentales y comparación con librerias

Por último, una vez que la sincronización se ha completado, cada bloque de hilos puede leer el array completo de memoria global y ajustar su segmento en consecuencia.

Con este algoritmo, se consiguen los siguientes resultados para arrays de tamaño 1024×2^N, al ejecutarlo 11 veces, ignorando la primer ejecución:

N	**Tiempo de ejecución promedio (μs)**	**Desviación (μs)**
1	85,8	2,201009869
5	84,3	2,162817093
10	260,9	7,264066813
15	6543,2	715,1367857

Se observa que el tiempo de ejecución aumenta de forma supra lineal, esto es esperado debido a que el arreglo aumenta exponencialmente al aumentar N. Además, la duración de la ejecución presenta diferencias menores para $N = 1$ y $N = 5$, se interpreta que esto ocurre debido a que el aumento en el tamaño del problema no es tan significativo como para impactar notablemente en el rendimiento.

Para asegurarnos de que la implementación de esta función realizada es correcta, se aplica la operación de SCAN utilizando las librerías Thrust y CUB, comparando elemento a elemento los 3 resultados. Además, se comparan los tiempos de ejecución medidos para el programa desarrollado con los resultantes de ejecutar la misma función con las librerías mencionadas.

Se otienen los siguientes resultados:

Thrust

N	**Tiempo de ejecución promedio (μs)**	**Desviación (μs)**
1	43,2	1,475729575
5	44,8	5,996295152
10	59,3	1,946506843
15	1357,4	2,221110833

Thrust muestra un patrón de aumento de tiempo de ejecución similar al observado en la implementación realizada. Si bien en todos los casos esta implementación obtiene tiempos de ejecución comparativamente menores. Esto sugiere que Thrust tiene optimizaciones adicionales.

CUB

N	**Tiempo de ejecución promedio (μs)**	**Desviación (μs)**
1	23,5ms	0,707106781ms
5	22,8ms	0,632455532ms
10	18,1ms	0,316227766ms
15	25,3ms	2,907843798ms

CUB muestra un comportamiento notablemente diferente, con tiempos de ejecución consistentemente bajos.

Según lo descrito en el documento "Single-pass Parallel Prefix Scan with Decoupled Look-back" [1] la implementación de SCAN implementada para CUB utiliza un método llamado "decoupled look-back" que permite realizar la operación en una sola recorrida del conjunto de datos. En cambio otros métodos, como el llamado "scan-then-propagate" implementado por Thrust deben realizar múltiples "pasadas".

En los algoritmos del estado del arte, la mayor limitante en el tiempo de ejecución de algoritmos de SCAN es el costo del movimiento de datos.

La implementación de CUB requiere únicamente de $2n$ accesos a memoria, n para lectura y n para escritura, mientras que otros algoritmos como el utilizado por Thrust $4n$. En particular el algoritmo de CUB, de forma similar al algoritmo implementado, divide al arreglo inicial en particiones que se asignan a distintos bloques. Cada bloque realiza un SCAN local utilizando memoria compartida. Se calculan las sumas locales, denominadas *aggregate* las cuales son consultadas por los bloques correspondientes a segmentos posteriores con el objetivo de calcular la suma total de elementos hasta el mismo. El algoritmo es capaz de, en vez de esperar a que se complete el calculo de la suma total del bloque anterior, utilizar los agregados locales anteriores a medida que se hacen disponibles para comenzar su cálculo lo antes posible, para esto se examinan los estados de los bloques anteriores dinámicamente. Este enfoque es llamado "decoupled look-back". La implementación de CUB incluye también múltiples otras optimizaciones que maximizan el acceso eficiente a memoria y el aprovechamiento del paralelismo.

En base a esto se concluye que los tiempos de ejecución medidos en la parte anterior responden adecuadamente a las descripciones teóricas de los algoritmos implementados por CUB y Thrust, donde se observa que el rendimiento de CUB es considerablemente superior a la implementación presentada en este informe y a la utilizada por Thrust.

2.3. Posibles mejoras

Por un lado, debido al patrón de acceso de los hilos dentro de la memoria compartida, en la cual el stride se duplica en cada paso, se identifican conflictos de banco en el momento de crear las sumas parciales al navegar el arbol generado durante el scan local.

Como se describe en [2], esto puede ser solventado agregando un padding igual al valor del indice original dividido entre la cantidad total de bancos.

Otro problema identificado en esta implementación, es el hecho de que el segundo kernel implementado debe esperar a que se complete el paso recursivo antes de continuar, agregando secuencialidad al algoritmo, por lo que una posible mejora es mitigar esto. Se propone por lo tanto, que cada bloque de hilos encargado de un segmento del array, una vez aplicando el scan local, efectúe el ajuste necesario para obtener el valor final sin delegar parte de este trabajo al paso recursivo (haciéndolo innecesario), aunque esto requerriría de sincronizaciones entre bloques.

Una posible estrategia para implementar esta mejora es utilizar la memoria global para almacenar los valores necesarios para el ajuste final de cada segmento. La implementación podría seguir estos pasos:

Primero, cada bloque de hilos aplica el scan a su segmento local del array, almacenando el resultado en memoria compartida.

Luego, el último valor del resultado del scan local de cada bloque se almacena en un array auxiliar en la memoria global, y, en lugar de aplicar scan sobre este array, se permite a los bloques acceder a estos valores para realizar los cálculos necesarios para efectuar el ajuste final, incluso si esto resulta en cálculos repetidos, siguiendo un enfoque similar a la implementación empleada por CUB.

3. Ejercicio 2

Para este ejercicio, se aplicaron las siguientes paralelizaciones a la función "ordenar_filas()"

- En primer lugar, se paraleliza la búsqueda del máximo dentro del array `niveles` utilizando la función de thrust `max_element`, para esto, se utiliza el array en memoria de dispositivo `d_niveles`, y se reutiliza convirtiéndolo a `thrust::device_ptr` para evitar el tiempo debido a la copia de memoria. Durante la experimentación realizada se identifica que sin esta optimización la reducción en el tiempo de ejecución resultante del uso de paralelismo no basta para compensar el tiempo adicional dedicado a la transferencia de datos.
- También se paraleliza el conteo del número de filas de cada nivel, y la clase de equivalencia del tamaño, utilizando la función `for each` de Thrust, junto con un operador propio que contiene el segmento de código original.
- Por último, el exclusive SCAN aplicado para obtener el punto de comienzo de cada par (tamaño, nivel) es paralelizado usando la función `exclusive_scan` de Thrust.

Se planteó paralelizar también la generación de orden (es decir, el segundo for), sin embargo, la dependencias entre iteraciones imposibilitaban esto.

Para comprobar que los resultados generados por la función sigan siendo correctos al aplicar estos cambios, se recorren a la vez los arrays generados por la función secuencial y la paralela, y se comparan uno a uno los elementos.

Con esto, se obtienen los siguientes resultados al aplicarse sobre la matriz A_matriz dada:

Tipo	**Tiempo promedio (μs)**	**Desviación (μs)**
Secuencial	62,6	4,90351348
Paralelo	188,3	2,496664441

Se observa un empeoramiento en los tiempos, posiblemente debido a que el costo correspondiente al "overhead" necesario para el uso de operaciones de CUDA es suficientemente grande como para causar que los beneficios obtenidos por aplicar paralelismo no sean suficientes. Esto se considera razonable debido al reducido tamaño de la matriz 18×18.

3.1. Evaluación de paralelizaciones realizadas

En la siguiente sección se evalúa la mejora de rendimiento obtenida como resultado de la paralelización de cada una de las secciones de código descritas anteriormente en matrices de distintos tamaños y cantidad de entradas no nulas. Todas las matrices utilizadas son cuadradas, por lo que la columna "Tamaño" hace referencia a la cantidad de filas (y por lo tanto, también de columnas).

Se obtienen los siguientes resultados:

3.1.1. Paralelización de máximo

Máximo

Tipo	Tamaño	No-ceros	Tiempo promedio (μs)	Desviación (μs)
Secuencial	100.000	3.200.000	49	1,054092553
Paralelo	100.000	3.200.000	96,1	17,61596498
Secuencial	100.000	1.000.000	51,1	1,791957341
Paralelo	100.000	1.000.000	105,2	9,919677414
Secuencial	1.000.000	64.000.000	712,8	14,1719598
Paralelo	1.000.000	64.000.000	397,6	16,5677062
Secuencial	5.000.000	79.000.000	2.240,7	130,2826415
Paralelo	5.000.000	79.000.000	396,7	111,2555117
Secuencial	16.000.000	47.000.000	6.501,4	105,8491589
Paralelo	16.000.000	47.000.000	1.142,4	1.309,111675
Secuencial	36.000.000	74.000.000	14.153,4	271,5319502
Paralelo	36.000.000	74.000.000	1.335,3	1.609,458228
Secuencial	55.000.000	113.000.000	21.976	814,8020891
Paralelo	55.000.000	113.000.000	1.825,3	1.925,517019

Nota: Los valores de tamaño y cantidad de No-ceros están redondeados.

Se observa que la implementación realizada se ve beneficiada por el uso de paralelismo para el cálculo del máximo para matrices de tamaños superior a cien mil. En particular se encuentran mejoras de entre 44 % y 91,6 %.

Si bien la mejora es consistente y considerable, se observa que la desviación estándar del algoritmo que utiliza Thrust es sustancial. Por ejemplo, para la matriz de mayor tamaño, si bien la mayoría de los resultados obtenidos son de alrededor de $900\mu s$, se detectan "outliers" que alcanzan hasta $5948\mu s$ (pese a esto, el algoritmo paralelo sigue resultando beneficioso incluso en sus peores casos, como es de esperar).

3.1.2. Paralelización de contar filas

Contar filas

Tipo	Tamaño	No-ceros	Tiempo promedio (μs)	Desviación (μs)
Secuencial	100.000	3.200.000	513,1	31,77158059
Paralelo	100.000	3.200.000	207,2	17,38326143
Secuencial	100.000	1.000.000	1.571,3	20,38545233
Paralelo	100.000	1.000.000	1.348,6	25,14491334
Secuencial	1.000.000	64.000.000	7.987,9	83,07754075
Paralelo	1.000.000	64.000.000	3.813,8	37,83825225
Secuencial	5.000.000	79.000.000	53.780,4	3189,221751
Paralelo	5.000.000	79.000.000	12.973,7	1755,18521
Secuencial	16.000.000	47.000.000	97.163,2	595,2549034
Paralelo	16.000.000	47.000.000	39.198,7	100,5032338
Secuencial	36.000.000	74.000.000	116.095,7	1.125,685179
Paralelo	36.000.000	74.000.000	85.384,5	1.539,899943
Secuencial	55.000.000	113.000.000	303.534,2	3.386,625748
Paralelo	55.000.000	113.000.000	131.728	1.267,540488

Nota: Los valores de tamaño y cantidad de No-ceros están redondeados.

La paralelización en la operación de contar filas muestra una mejora significativa en el tiempo de ejecución para matrices de gran tamaño. Para matrices pequeñas (100.000 con 1.000.000 no-ceros), el rendimiento paralelo no es superior al secuencial. Sin embargo, a medida que aumenta el tamaño de la matriz, el rendimiento de la versión paralela mejora significativamente. En algunos casos, se observa una reducción de más del 50 % en el tiempo de ejecución.

Al ejecutar el programa con matrices de igual tamaño pero distinta cantidad de valores no nulos, se observa como el tiempo de ejecución es altamente sensible a esta característica. Se identifica que cuando la proporción de valores no nulos presentes en la matriz es más alta, el beneficio relativo obtenido como consecuencia del uso de paralelismo disminuye.

3.1.3. Paralelización de SCAN

SCAN

Tipo	Tamaño	No-ceros	Tiempo promedio (μs)	Desviación (μs)
Secuencial	100.000	3.200.000	8,1	0,316227766
Paralelo	100.000	3.200.000	57	9,603240194
Secuencial	100.000	1.000.000	351,7	8,420213774
Paralelo	100.000	1.000.000	503,1	12,83614688
Secuencial	1.000.000	64.000.000	23,2	5,223876806
Paralelo	1.000.000	64.000.000	84,5	3,40750805
Secuencial	5.000.000	79.000.000	170	17,72004515
Paralelo	5.000.000	79.000.000	1.250,5	82,7314934
Secuencial	16.000.000	47.000.000	0,1	0,316227766
Paralelo	16.000.000	47.000.000	38,1	0,875595036
Secuencial	36.000.000	74.000.000	0	0
Paralelo	36.000.000	74.000.000	40,6	2,366431913
Secuencial	55.000.000	113.000.000	0,2	0,421637021
Paralelo	55.000.000	113.000.000	39	2,748737084

Nota: Los valores de tamaño y cantidad de No-ceros están redondeados.

La operación de SCAN presenta un comportamiento distinto.

Se observa que en todos los casos el beneficio obtenido por el uso de paralelismo no es suficiente para justificar su overhead correspondiente.

Se identifica que el tiempo de ejecución de SCAN para particularmente este programa es altamente dependiente del valor de `n_levels` que a su vez no toma valores de magnitud alta y, por lo tanto, la ejecución de SCAN presente no es tan costosa como podría serlo en problemas de otra naturaleza. A su vez la dependencia en este valor justifica la observación de resultados inesperados como tiempos reducidos en la ejecución del algoritmo secuencial para matrices grandes.

3.2. Tiempo de ejecución total

Aplicando ambos algoritmos para diferentes tamaños de matrices de *suitesparce*, obtenemos los siguientes resultados para el tiempo total de ejecución:

Total

Tipo	Tamaño	No-ceros	Tiempo promedio (μs)	Desviación (μs)
Secuencial	100.000	3.200.000	31.348,9	23.599,46512
Paralelo	100.000	3.200.000	27.443,3	19.013,1105
Secuencial	100.000	1.000.000	140.602.7	6.896,497115
Paralelo	100.000	1.000.000	139.835	7.242,106722
Secuencial	1.000.000	64.000.000	177.692.6	1.363,002503
Paralelo	1.000.000	64.000.000	171.821.1	612,8825245
Secuencial	5.000.000	79.000.000	352.379.6	4.611,826204
Paralelo	5.000.000	79.000.000	310.464.5	4.052,64559
Secuencial	16.000.000	47.000.000	503.941,8	4.346,641815
Paralelo	16.000.000	47.000.000	442.071,7	3.423,079285
Secuencial	36.000.000	74.000.000	5.416.886,4	722.103,3212
Paralelo	36.000.000	74.000.000	5.272.461,5	618.262,4948
Secuencial	55.000.000	113.000.000	1.252.962,2	5.469,656313
Paralelo	55.000.000	113.000.000	1.075.418,4	4.961,974611

Nota: Los valores de tamaño y cantidad de No-ceros están redondeados.

Al aplicar todas las operaciones paralelas a las matrices de diferentes tamaños, los resultados muestran mejoras en el tiempo total de ejecución. Para matrices pequeñas y medianas, la paralelización ofrece una mejora moderada en el rendimiento.

Para matrices grandes, la reducción en el tiempo de ejecución es más significativa, con mejoras que van desde un 20 % hasta un 35 % (aproximadamente). En general, la implementación paralela del código completo muestra una mejora clara en el rendimiento para matrices grandes.

4. Bibliografía

(1) Merrill, D., & Garland, M. (2016). Single-pass Parallel Prefix Scan with Decoupled Look-back. NVIDIA Technical Report NVR-2016-002, March 2016.

(2) Chapter 39. Parallel Prefix Sum (Scan) with CUDA
https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda