Laboratorio del curso GPGPU

2023

Grupo 10

Daniel Padron Simon	5.147.163-4	daniel.padron@fing.edu.uy
Bruno Cabrera Martínez	5.397.585-8	bruno.cabrera@fing.edu.uy

Introducción

Como se explica en la letra del laboratorio, el mismo consiste en ordenar un arreglo de valores numéricos usando la GPU. Para ello habrán varios niveles de ordenamiento que se utilizaran para distintos tamaños.

Inicialmente se ordenará de a 32 valores a nivel de warp utilizando radix sort. Luego se realizará merge sort en sectores del arreglo menores al tamaño del bloque (de CUDA), y cuando los sectores sean de un tamaño no soportado por los bloques se procederá a separar los sectores en segmentos y se realizará el merge entre estos segmentos.

La idea general de dichos algoritmos ya está plasmada en la letra del laboratorio. En el siguiente informe explicaremos brevemente nuestra implementación, cambios que se puedan haber hecho en los algoritmos y al final realizaremos una breve comparación con el Sort de la biblioteca Thrust para CUDA.

Radix Sort

En el Radix Sort se hace un ordenamiento a nivel de warp, al final de su ejecución van a quedar sectores de 32 elementos ordenados internamente. Ya que es a nivel de warp se intenta utilizar lo máximo posible operaciones entre hilos del warp como shuffle y warp sync tanto en la suma prefija como en el algoritmo de ordenamiento.

De todas formas nos fue necesario utilizar memoria compartida para realizar el intercambio entre los valores de cada hilo en el ordenamiento, ya que nos sucedía que con shuffle se puede leer el valor de otro hilo, pero no enviarle un valor.

Merge Sort

El mergesort se implementó al igual que la letra. Para poder distinguir la posición final de un valor en el arreglo final se tuvo que realizar una política de búsqueda de posiciones distintas para A y B. En el caso donde dos elementos iguales a_i y b_i (uno de A y otro de B en la posición i) ocurre que para ambos valores su posición en el arreglo final será la misma. Para poder solucionar esto se debió realizar una política de búsqueda distinta, discriminando según si el valor es A o B.

La idea es que los elementos de A se inserten previo a todos los valores iguales en B. Cuando se realiza la búsqueda por bipartición (para obtener la posición de un valor en el arreglo opuesto) debemos indicar si el hilo que la ejecuta pertenece a A o B.

Merge de sectores mayor al tamaño de bloque

Como menciona la letra, para el caso donde no se puede realizar el merge sort debido a que cada sector a ordenar es mayor al tamaño de un bloque se deben particionar A y B en sectores y realizar el ordenamiento entre ellos.

Para esto se utilizarán dos kernels que se irán ejecutando en bucle hasta que el arreglo quede completamente ordenado. Estos son separators_kernel y merge_segments_kernel.

Kernel de separadores

El primer kernel se encarga de obtener separadores de A y B con una distancia t (tamaño del bloque / 2). El primer elemento de cada A y B y el último también serán separadores. La salida del kernel serán dos arreglos con las posiciones en A y B respectivamente. Estas posiciones vendrán ordenadas según el valor original del separador.

En las primeras interacciones del sort, este kernel procesa muy pocos separadores, por lo tanto, con el fin de aumentar la utilización de la GPU, permitirnos que un mismo kernel procese separadores de diferentes pares de conjunto A-B a ordenar. De forma tal que cuando varios conjuntos de separadores se pueden procesar en un mismo warp, este los ejecute en simultáneo.

Merge de segmentos

En el segundo kernel cada hilo tomará un par de separadores consecutivos y realizará el merge de las secciones entre ambos separadores en A y B.

Utilizamos bloques bidimensionales para esta sección, donde la dimensión Y identifica la pareja A-B a unir, mientras que la dimensión X identifica la pareja de separadores que delimitan el segmento de A y B a unir.

Dado un bloque en particular, tenemos un ini_a y ini_b, los cuales establecen las posiciones de A y B donde comienza el segmento a unir respectivamente, es análogo para las los índices fin_a y fin_b.

Como el índice de inicio de un segmento es el fin del segmento anterior, no podemos incluir el inicio y el final en el merge de cada segmento, por lo que tomamos la decisión de solo incluir al inicio y no al final. Por lo tanto, cada bloque será el encargado de unir los segmento [ini_a, fin_a) con [ini_b, fin_b).

Es importante notar que los último elemento de A y B son separadores, sean s_a y s_b, si s_a < s_b, entonces r(s_b, B) = length(B) - 1 y r(s_b, S) = length(S) - 1. En otras palabras, si s_b es el separador más grande, entonces no habrá ningún s en S tal que r(s, B) sea mayor que r(s_b, B), entonces ninguno de los segmentos a hacer merge incluye a s_b, ya se excluye el borde superior a hacer megre.

La solución al problema es considerar a fin_A y fin_B del último segmento de la pareja A-B igual al length de A y B respectivamente, eso soluciona el problema antes mencionado, ya que sucederá que el último segmento hará el merge de los siguientes segmentos A[length(A), length(A)] (porque como s_b > s_a, entonces r(s_b, A) = length(A)) y B[length(B) - 1, length(B)], lo que genera que s_b ahora si tomado en cuenta en la generación del merge de A y B.

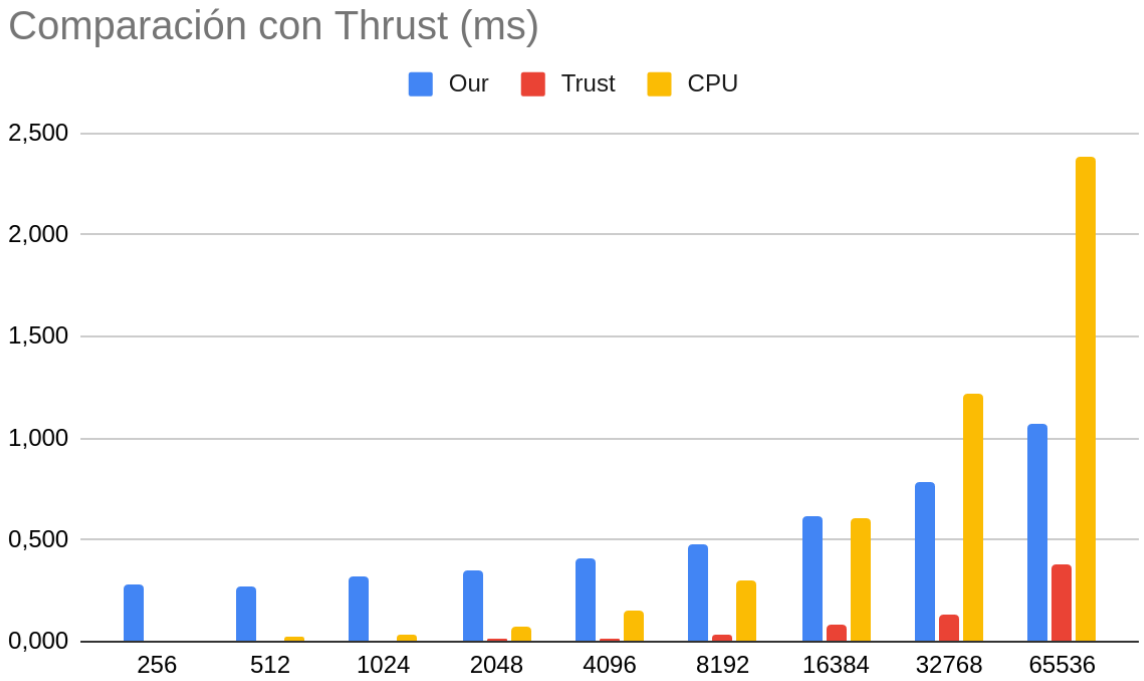
Comparación con Thrust

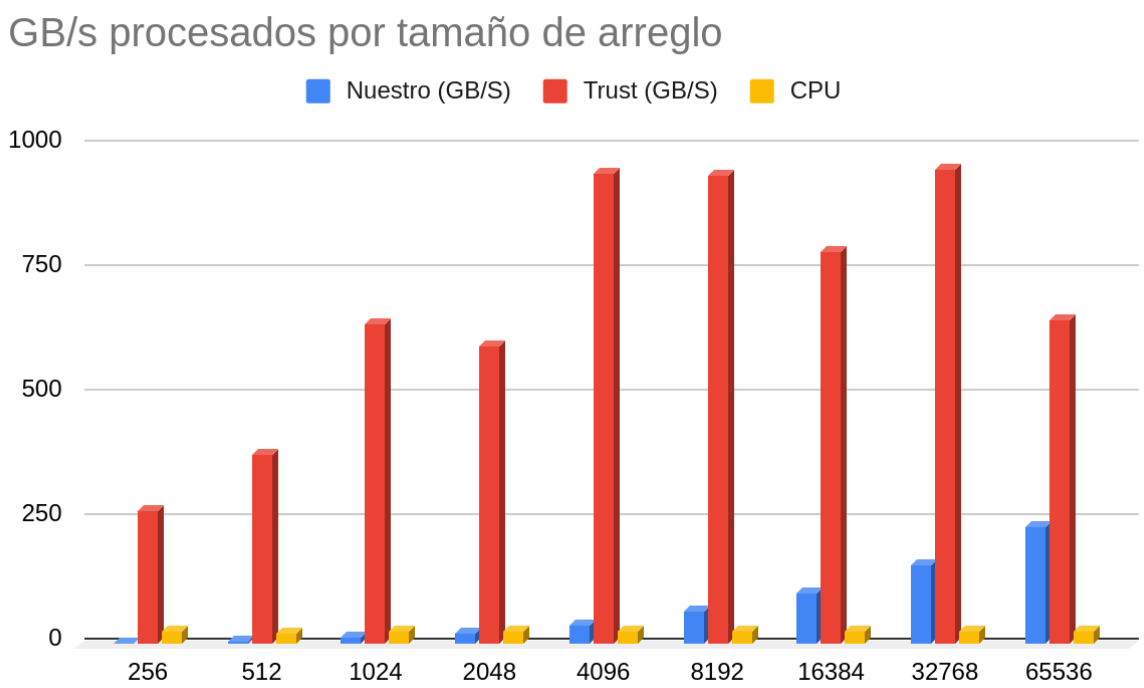
Se realiza una comparación de nuestro algoritmo con la implementación del sort en la biblioteca Thrust de CUDA y con la implementación del Sort en CPU de C++.

Los siguientes tiempos de ejecución en milisegundos fueron obtenidos realizando varias corridas con distintos tamaños de arreglo a ordenar. En el caso de nuestro algoritmo decidimos incluir el tiempo de demora que conlleva enviar los arreglos a la memoria de la GPU (ya que entendemos que el algoritmo de Thrust realiza la misma operación).

Size	Nuestro	Thrust	CPU
256	0,285	0,004	0,009
512	0,272	0,005	0,020
1024	0,322	0,006	0,039
2048	0,353	0,013	0,077
4096	0,410	0,017	0,154
8192	0,476	0,033	0,298
16384	0,617	0,079	0,609
32768	0,781	0,131	1,218
65536	1,066	0,384	2,383

La diferencia de performance es muy alta, Thrust es mucho más eficiente que nuestro algoritmo. Entendemos que el algoritmo de Thrust está mucho más optimizado que el nuestro.





Cuando lo comparamos con el algoritmo sort en CPU podemos notar que en arreglos pequeños el overhead de tener que enviar datos al GPU (en el caso de nuestro algoritmo) genera que sea más eficiente en CPU. Pero a medida que el tamaño aumenta la situación cambia. Es importante notar que las pruebas se realizaron en arreglos relativamente de tamaño pequeño (máximo 64 Kb). A medida que aumente más el tamaño esta diferencia aumentará de forma considerable.

Como conclusión se puede decir que en el caso de necesitar un algoritmo de ordenamiento en una implementación en GPU generalmente es preferible utilizar el provisto por Thrust ya que está muy bien optimizado y probado.

Código

Se deja adjunto el código de esta implementación. Para probar el algoritmo se debe ejecutar el bash run.sh que se encargará de compilar y ejecutar.

El programa generará 9 arreglos de 256 a 65536 valores y los ordenará usando Thrust, nuestro algoritmo y el sort de C++. Los tiempos de ejecución se almacenarán en un archivo de texto (generado por código) llamado output.txt que además puede ser leído como csv.

Posible mejoras

Una posible mejora que identificamos es utilizar los hilos inactivos de un warp con el fin de realizar lectura coaleced de memoria.

A modo de ejemplo, en el kernel 2 de la parte d, el número de thread por bloque son 256, pero hay veces que de elementos del segmento de A y B no superan los 256, en tal caso los thread podrían ser utilizados para poder “alinear” la lectura a memoria global cuando estamos poblando la memoria compartida.

Notar que en la parte C, si se cumple que la lectura es coaleced, ya que estamos ordenando arreglos que son potencia de 2.

Pero cuando ejecutamos el kennel 2 de la parte d, notamos que es muy posible que uno de los dos segmentos a ordenar (en particular del arreglo al que no pertenece el separador que delimita al inferiormente al segmento) no este alineado.