Programación Paralela: Cuda

Agustin Colazo

September 2018

1 Introducción

CUDA es una plataforma de computación en paralelo y un modelo de programación que hace el usar una GPU (Unidad de procesamiento gráfico) una tarea simple.

2 Descripción

Implementar un algoritmo de ordenamiento de selecćon en forma paralela utilizando reducciones en forma eficiente en CUDA. En cada iteración se encontrará el mayor valor utilizando la reducción y se dejará en la última posición del arreglo. Luego se llamará recursivamente a la función con la longitud del arreglo disminuida en uno, hasta que finalmente quede el arreglo ordenado sobre la placa GPU. Luego se devolverá a la CPU el arreglo ordenado. El trabajo se realizará utilizando el lenguaje de programación C++ con extensiones CUDA.

3 Marco Teórico

3.1 Funciones

En CUDA a las funciones que se ejecutan en la GPU se les llama kernel. Al escribir un programa que utilizará CPU y GPU, para diferenciar los dispositivos en que se ejecutará cada código se utilizan distintas directivas. Cuando se diga host haremos referencia al CPU, y dispositivo será la GPU.
__global__: Este código es llamado desde el host, y ejecutado en el dispositivo.
__device__: Este código es llamado desde el dispositivo, y ejecutado en el dispositivo.
__host__: Este código es llamado en el host, y ejecutado en el host.

3.2 Hilos

3.2.1 Grilla y bloques

En CUDA los hilos son el elemento fundamental para la paralelización. Para organizar los hilos en CUDA estos se dividen en bloques.

Un grilla de tres dimensiones contiene varios bloques. Estos bloques también de tres dimensiones se dividen en hilos.
Cada hilo en la grilla tiene un identificador único el cual se puede armar utilizando variables proporcionadas por CUDA: blockDim, blockIdx y threadIdx.//
Al llamar a una función kernel se especifica las dimensiones de la grilla y de los bloques.
Los hilos se pueden sincronizar en el mismo bloque usando la instrucción _syncthreads(), pero no hay sincronización entre bloques. La única forma de sincronizar entre bloques es terminar un kernel y lanzar otro.

3.2.2 Asignación de bloques

Los bloques se ejecutan en multiprocesadores. Cuantos bloques puede ejecutar cada multiprocesador dependerá del modelo de la placa de video. Por ejemplo, en la placa de video GT200 cada multiprocesador puede ejecutar hasta 8 bloques simultaneamente y hasta 1024 hilos. Y la placa de video GTX680 que es la que se usa en este trabajo, cada multiprocesador puede ejecutar hasta 2048 hilos
A su vez cuando un bloque es asignado a un multiprocesador, este se divide en warps.

3.2.3 Warps

El warp es la unidad de *thread scheduling* en multiprocesadores. Estos consisten en hilos con identificadores de hilo consecutivos.
Cuando se ejecuta un warp, las instrucciones se ejecutan sincronizadamente. Es por esto que puede no usarse _syncthreads() para sincronizar hilos al estar dentro del mismo warp. Tanto la GT200 como la GTX680 tiene un tamaño de warp de 32.

3.3 Memoria

Al programar placas de video es importante saber sobre los distintos tipos de memoria disponible ya que el uso de estas tiene un impacto importante en el rendimiento de los programas.
Hay cuatro tipos de memoria: *global, constant, shared y registers.*
La memoria global y constante son las más lentas. La memoria global tiene largos tiempos de acceso, puede ser escrita y leida tanto por el dispositivo como por el host.
La memoria constante se guarda en memoria global, pero es cacheada para acceso mas eficiente. Esta puede ser escrita y leida por el host, pero solo leida por el dispositivo. Tiene más ancho de banda que la memoria global. También soporta el acceso simultaneo a la misma ubicación.
La memoria compartida puede ser escrita y leida por bloque. Se encuentra en el chip y es de alta velocidad y altamente paralela.
Los registros puede ser escrita y leída por hilo. Es la más rapida de todas y

se encuentra en el chip. También es la de menor tamaño. Normalmente se usa para varibales privadas de cada hilo.
Los registros y la memoria compartida existen dentro de las llamadas a kernel. Mientras que la memoria global y constante existen fuera de estas.
La memoria compartida se puede utilizar para que los hilos dentro de un mismo bloque colaboren. Mientras que la memoria global se puede utilizar para que hilos de distintos bloques colaboren, o para pasar datos entre un kernel y otro.

4 Desarrollo

En el programa a desarrollar se aplico una reducción con las optimizaciones que se encuentran en el documento. [1]
Al código allí expuesto solo funciona para arreglos de tamaño potencia de 2. Se modifico ese código para que funciona con arreglos de cualquier tamaño.
También, en este caso se necesita hacer la reducción para obtener el valor máximo y luego intercambiar este valor con el último de la lista. Para esto se necesita conocer el índice de cada valor. Pero al reducir los valores en la placa de video, se pierde el índice de estos. Es por esto que primero se envuelve a los datos en una estructura que contiene el valor y el índice. Luego se pasa esta estructura al algoritmo de reducción. Finalmente, se intercambia los valores en la placa de video y se repite el proceso recursivamente hasta el primer elemento de la lista. Antes de devolver los datos al host, se los desenvuelve y se devuelve solo los valores.
En el programa se contempla el caso cuando la cantidad de datos excede a la memoria compartida disponible. En este caso se hace varias llamadas a la función de reducción con distintas partes del total de los datos. Pero no se contempla el caso cuando la cantidad de memoria necesitada excede a la memoria global.
Cuando hay muchos elementos en la lista las llamadas recursivas exceden la memoria del stack, entonces el programa lanza un error de *segmentation fault*. Por esto, también se planteo una solución iterativa que soluciona este problema.
Para compilar el programa se uso el compilador nvcc de Nvidia.

5 Resultados

Se hicieron realizaron varias pruebas con distintos tamaños de lista, hasta listas de 150.000 elementos. Los casos de prueba fueron una lista ordenada, y una lista ordenada invertida. Los resultados fueron en todos los casos correctos.
El tiempo de ejecución promedio de la reducción con 150.000 enteros fue de 0,735 milisegundos.
El tiempo de ejecución promedio del ordenamiento de 150.000 enteros fue de 23,861 segundos.

6 Conclusión

El mayor desafío al escribir este programa no fue escribir la función del kernel en forma eficiente, sino en como llamar a esta función desde el host. Al escribir programas usando CUDA hay que tener varias consideraciones en cuenta.
Determinar la cantidad de hilos y bloques necesarios según la cantidad de datos, como también la cantidad de memoria compartida que debe alocarse. También debe llamarse recursivamente a la función de reducción si la cantidad de bloques era mayor a 1, y la cantidad de bloques, hilos y memoria compartida necesaria irá cambiando con cada llamada. Determinar si se debe dividir el total de los datos en varias llamadas a kernel si se excede el espacio de memoria compartida necesario; esto a su vez requiere llamar al kernel devuelta cuando se redujo todas las partes de los datos, con los resultados de cada parte. Alocar el espacio necesario en memoria global según la cantidad de datos tanto del arreglo que contendrá a los datos, como los arreglos auxiliares. También hay que tener en cuenta si se necesita más memoria global de la disponible, en este caso deberia pasarse una parte de los datos a la placa de video, hacer la reducción, devolver los datos al host, y repetir el proceso con otra parte de los datos.

7 Comandos

nvcc name.cu -o name

References

[1] Mark Harris. Optimizing parallel reduction in cuda. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[2] David B. Kirk Wen mei W. Hwu. *Programming Massively Parallel Processors.*