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.
En el desarrollo del software hay que tener varias consideraciones. Al reducir elementos sobre un arreglo, uno pierde los valores originales, pero al necesitar devolver la lista entera ordenada esto no es aceptable.
Entonces, se debera copiar los datos a un buffer, ejecutar la reduccion en este buffer, y luego los resultados escribirlos en un segundo buffer. Este segundo buffer servira como entrada para el proximo kernel (pero en este segundo caso ya no será necesario mantener los datos.
La segunda consideración es que no solo hay que comparar los valores, sino que lo que se necesita pasar es el índice del valor máximo en el arreglo original. Ese es el valor que queremos pasar. Luego con ese índice podremos intercambiar ese valor con el último de la lista.
Se hicieron varias programas que resuelven esto de distintas maneras.

4.1 Envolviendo datos, sin rellenar, usando memoria compartida

En esta versión 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.
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. En este caso el kernel usa condiciones "if" para que los hilos no tomen datos que estan fuera de la memoria alocada. La memoria alocada coincide con el tamaño de los datos.
Al usar condiciones if hay menos lecturas a memoria ya que cuando un dato esta fuera del rango no se lee. Tambien hay menos escrituras ya que no se rellena

los datos.
Un ejemplo de esto:

```
if (tid < 512) {
            if ((i + 512) < size) sdata[tid] = sdata[tid].value > sdata[tid + 51
        }
        __syncthreads();
```

4.2 Envolviendo los datos, rellenando, usando memoria compartida

Similar al caso anterior pero esta vez la memoria que se reserva es la potencia mayor más cercana al tamaño de los datos. La memoria que sobra es rellenada con estructuras que contienen el valor mínimo posible para el tipo de dato correspondiente.
Una vez encontrado un máximo este valor se mueve al final de la lista y se cambia su valor por el mínimo. El valor es guardado en un arreglo de valores ordenado.
En este caso no son necesarias las condiciones "if", por lo tanto los kernel tienen menos instrucciones.
Se rellena los datos en cada invocación al kernel cuando se carga los datos de memoria global.

```
if((i + blockDim.x )< size){
        sdata[tid] = g_input[i].value > g_input[i + blockDim.x].value ? g_input[
        }
    else if (i<size){
        sdata[tid] = g_input[i];
    }

    else{
        DATATYPE min_value;
        min_value.value = MINVALUE;
        sdata[tid] = min_value;
    }
.
.
.
if (tid < 512) {
            sdata[tid] = sdata[tid].value > sdata[tid + 512].value ? sdata[tid]
        }
        __syncthreads();
```

4.3 Envolviendo los datos, sin rellenar, usando memoria global

En este caso se uso memoria global con condiciones if. Asi se puede comparar el uso de memoria global con el uso de memoria compartida.

4.4 Envolviendo los datos, rellenando, usando memoria global

Esta versión se hizo para ver el impacto en el rendimiento que tiene usar las condiciones "if" para manejar tamaños de datos distintos a potencias de dos.

4.5 Sin envolver los datos, rellenando, usando memoria global

La última versión es sin envolver a los datos. Aquí se usan dos kernel para la reducción.
El primer kernel acepta como entrada la lista y como salida guarda un índice en un arreglo de índices.
El segundo kernel acepta como entrada un arreglo de índices y como salida índices.

4.6 Kernel vacío

También se hizo un programa con kernel vacío para medir la sobrecarga de la invocación de kernels.

5 Resultados

Tiempo de invocación de kernel: 0.00717 ms.

Se hicieron realizaron varias pruebas con distintos tamaños de lista, hasta listas de 200.000 elementos. Los casos de prueba fueron una lista ordenada, y una lista ordenada invertida. Los resultados fueron en todos los casos correctos. Se muestra el tiempo de ejecución para cada programa en el mismo orden que se presentaron en la sección anterior.

5.1 Ordenamiento de 200.000 elementos[ms]

1. 4944
2. 5251
3. 13246
4. 16462

5. 9948

También se probo una versión del segundo programa rellenando los datos en memoria global, en vez de rellenandolos en memoria compartida dentro del kernel. Esto significa que el kernel de reducción tenia menos uso de CPU, pero más uso de la memoria global. El tiempo de ejecución de ese kernel fue de 5831 ms (tardo más que la versión con que se compara de 5251 ms).

5.2 Reducción de 500.000 elementos[ms]

1. 0,121024
2. 0,121536
3. 0,253376
4. 0,267232
5. 0,159776

6 Conclusión

El resultado interesante es que en memoria global pasar índices y datos en arreglos distintos tiene mejor rendimiento que envolver a los datos en estructuras. Si bien la implementación es más complicada (ya que dependiendo de la cantidad de datos deberan ejecutarse dos kernels distintos) el tiempo de ejecución es mejor.
Como vemos en los resultados, el programa con mejor tiempo de ejecución es el que hace un uso más eficiente de la memoria, y no del CPU. El programa con más condiciones if tiene peor uso del CPU, pero hace menos escrituras y lecturas en memoria. Se concluye que esta reducción es más dependiente del uso de la memoria que del uso del CPU.

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.*