GPGPU – Laboratorio 5

Introducción

El objetivo de este laboratorio es iniciarse en el uso de las bibliotecas de uso general que brinda CUDA como parte de su ecosistema extendido. La principal ventaja de utilizar estas bibliotecas es la relativa sencillez con la que se pueden paralelizar rutinas y obtener buenos speedups, sobre todo en problemas relativamente genéricos. En el ejercicio 1 se busca que el estudiante vea la diferencia de dificultad de programación comprando con la implementación “de 0” y compruebe las ganancias que se pueden obtener sin demasiada inversión del lado del programador. En el ejercicio 2 se intenta mostrar cómo pueden encadenarse varias de estas primitivas utilizándose como “building blocks” para paralelizar una parte de un código que hace de cuello de botella.

Los tiempos de ejecución deben tomarse como el promedio de 10 ejecuciones (presentando únicamente el promedio y desviación estándar en el informe).

Ejercicio 1: Scan

La operación de scan es una de las primitivas más utilizadas. Esta primitiva consiste en aplicar un operador binario (suma, multiplicación, etc.) a los elementos de un vector de forma sucesiva y da como resultado un vector de igual largo con todos los resultados intermedios. Por ejemplo si se tiene el vector [1, 2, 3], se utiliza el operador + y valor inicial 0, el resultado será [0, 0+1, 0+1+2] o [1, 1+2, 1+2+3] dependiendo si la suma es exclusiva o inclusiva respectivamente.

Implemente un kernel que calcule la suma exclusiva de un vector de largo arbitrario. El kernel debe ser relativamente eficiente en su acceso a memoria y se espera que use memoria compartida y/o registros aunque la estrategia de balanceo de carga no sea óptima. En el informe mencione los problemas que detecte de su implementación y, en caso de tenerlo, posibles soluciones para optimizar la implementación (no es necesario hacerlo).

a) Pruebe el kernel con valores de largo del vector del tipo 1024x2^N. Analice los resultados de tiempo de ejecución según la variación de N.

b) Compare los resultados obtenidos en la parte (a) con las implementaciones de las bibliotecas CUB y Thrust.

Ejercicio 2

Para resolver un sistema triangular es necesario resolver las ecuaciones (representadas por una fila en la matriz triangular) en orden ya que cada ecuación depende de las incógnitas correspondientes a las filas anteriores. Sin embargo, al tratar con matrices dispersas no todas las ecuaciones dependen de las anteriores y por tanto las filas pueden ser procesadas a la vez. Por ejemplo en la Figura 1 la ecuación correspondiente a la fila 2 no depende de la de la fila 1.

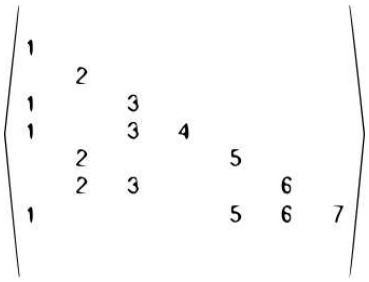

Figura 1: Matriz de ejemplo

Siguiendo esta idea es posible definir una estructura de niveles en función de las dependencias, definiendo como nivel 0 a las que no dependen de ninguna otra y como nivel i+1 a las que dependen de i. La estructura de niveles de la matriz de ejemplo puede verse en la figura 2. Las dos primeras filas son parte del nivel 0 mientras que la fila 3 es de nivel 1 ya que depende de la primera fila.

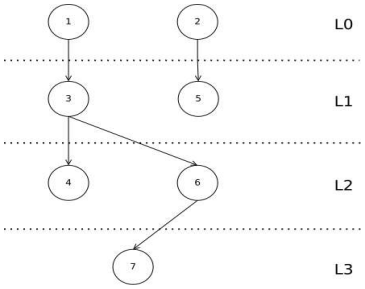

Figura 2: Estructura de niveles de la matriz

Una estrategia para paralelizar las resolución consiste en ejecutar en paralelo cada nivel. Sin embargo, al usar un warp para cada fila, se desaprovecha mucho cómputo en filas con muy pocos elementos distintos de 0. Por esto, una estrategia para optimizar la ejecución consiste en ejecutar filas de un mismo nivel y de similar tamaño en el mismo warp (ej: 16 filas de dos elementos o 8 filas de 4). Esta es la estrategia que se sigue en el código entregado.

El programa brindado lee una matriz de un archivo mtx [1] y la carga en memoria en formato CSR [2], la matriz generada siempre es una matriz triangular. La función *ordenar_filas()* es la encargada de generar un orden de ejecución de las filas en función del nivel y su tamaño que permita agrupar a las del mismo nivel que tienen tamaño similar. Esta función invoca a un kernel que calcula los niveles llamado *kernel_analysis_L*, dicho kernel recibe los vectores de fila (row_ptr), índices de columna (col_idx) y un vector auxiliar (is_solved) inicializado en 0 y retorna un vector que mantiene en cada posición el nivel de la fila. El retorno de la función es la cantidad de warps necesarios para resolver el sistema triangular dado por esa matriz **independientemente** de los valores de b.

Para hacer el ordenamiento, la función ordenar divide los tamaños de fila en 7 clases de equivalencia (menor o igual que 1, 2, 4, 8, 16 y mayor a 16) y redondea el tamaño hacia la cota superior. Luego, ordena las filas por nivel y dentro de cada nivel por tamaño en función de las clases de equivalencia. Finalmente, hace una asignación de filas a warps agrupando las filas de las 6 primeras clases y al terminar obtiene cuántos warps son necesarios,

a) Paralelice la función ordenar_filas() utilizando una de las dos bibliotecas dadas en el teórico. Compruebe los resultados utilizando la matriz *A_matrix.mtx*.
Sugerencia: Recuerde que puede definir sus propios operadores y funciones (a través de structs) que pueden ser aplicados mediante iteradores (o la función transform en el caso de thrust).

b) Pruebe la rutina implementada y compárela con la versión serial. Ejecútela en, al menos, 5 matrices de la colección suitesparse [3] de distintos tamaños. Considere en su análisis tanto el tiempo de la parte paralelizada como total de la función.

Entregar

El código correspondiente a cada parte y un informe en formato pdf con el análisis de los resultados experimentales.

Referencias

[1] https://math.nist.gov/MatrixMarket/formats.html
[2] https://shorturl.at/qaecP
[3] https://sparse.tamu.edu/