Práctica 4: Programación en CUDA

Luis Jafet Ramos Castillo, Walter Martínez Santana y
Eduardo David Martínez Neri

1. Introducción - suma de vectores

Un ejercicio introductorio simple está disponible en el folder incluido en la práctica. Contiene un archivo con código de CUDA para ser editado y ejecutado. Las secciones del archivo que tienen que ser editadas están marcadas por comentarios, por ejemplo:

```
/* Parte 1A. Reservar la memoria en el GPU */
```

Además del código proporcionado, también se puede consultar la documentación oficial en *http://developer.nvidia.com/nvidia-gpu-computing-documentation*.

1.1. Uso de la memoria del GPU

El ejercicio introductoria es un código simple que suma dos vectores de enteros. Introduce los conceptos del manejo de la memoria del GPU y cómo llamar a los kernels para ejecutarlos.

La versión final debe copiar dos arreglos de enteros de la memoria del CPU a la del GPU, sumarlos en el GPU, y luego copiarlos de vuelta al CPU.

En el archivo intro.cu, se encuentran las siguientes partes relacionadas con el uso de memoria:

- Parte 1A: Reserva de la memoria en el GPU
- Parte 1B: Copiar los arreglos del CPU al GPU
- Parte 1C: Copiar el resultado de la suma al CPU
- Parte 1D: Liberar la memoria del GPU

1.2. Llamado de Kernels

Las siguientes secciones del archivo intro.cu son relevantes al llamado de kernels y su ejecución:

- Parte 2A: Llamar al kernel usando un grid de una dimensión y un sólo bloque de hilos NUM_BLOCKS y THREADS_PER_BLOCK están configurados para esto)
- Parte 2B: Implementación del kernel para realizar la suma de los vectores en el GPU
- Parte 2C: Implementación del kernel pero esta vez permitiendo múltiples bloques de hilos. La implementación es similar, salvo por el índice:

```
int idx = threadIdx.x + (blockIdx.x * blockDim.x);
```

El diagrama 3 es útil para recordar cómo hacer el cálculo de índices.

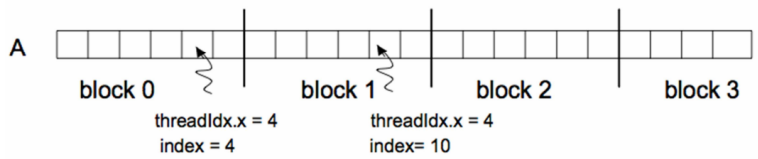

Figura 1: Direccionamiento de hilos. blockDim.x contiene el tamaño del grid en la dimensión x, en este caso 4

Se ejecutaron 3 kernels. El kernel vec_add usa solo 1 bloque y 1024 hilos. Los otros dos usan vect_add_multiblock que usan 4 bloques 256 hilos por bloque y 8 bloques con 128 hilos por bloque respectivamente.

Observamos diferencias en el desempeño de cada una de las configuraciones de las diferentes mallas (Grids).

En la primera configuración, se obtuvo 0.009280 ms, en las otras dos 0.007616 ms. Se debe tomar en cuenta que todo está en función del tamaño del arreglo. Con un arreglo de tamaño mayor se tendrían diferencias más grandes. Sin pérdida de generalidad, una malla trabaja mejor con un balance entre los bloques y el número de hilosx bloque.

Es un desperdicio y un ineficiente desempeño tener un hilo por cada bloque (el caso de vec_add) porque se debe generar una copia del kernel por cada uno de los bloques. No obstante, cuando hay varios hilos en un bloque

se aprovechan mejor los recursos. En el caso de los otros dos tienen un mejor *performance*, ya que en un bloque tiene varios hilos.

2. Configuración de bloques e hilos

En esta sección se analizará el efecto en el desempeño al configurar de explícitamente el número de bloques y de hilos al invocar la ejecución de un kernel. El desempeño se maximiza cuando la especificación de hilos y bloques está en línea con las características de hardware del GPU.

Si todos los hilos ejecutan exactamente el mismo código, el paralelismo es óptimo, mientras que si están ejecutando código distinto, deben dividirse en grupos de hilos que se ejecutan secuencialmente, lo que reduce el rendimiento.

Otro factor que afecta el rendimiento es el acceso a memoria global. Las transferencias con memoria global son mucho más eficientes si se fusionan en bloques de palabras consecutivas. Por ello, las direcciones generadas por los hilos en un warp, son consecutivas con respecto a sus índices. Es decir, el hilo N debe acceder la dirección Base+N, donde Base es un apuntador alineado a una frontera de 16 bytes.

En la página del sitio encontrará el código Ej2Pr3.cu, el cual se utilizará para evaluar el desempeño de un GPU variando el número de bloques e hilos.

Este programa está hecho para llevar a cabo cambios en el manejo de los accesos a memoria, de tal manera que las tareas sean repartidas de manera simétrica, es decir, distribuir la tarea de manera equitativa en cada hilo.

Además de evitar conflictos de datos al acceder a localidades contiguas de memoria, lo que hace es evitar estos conflictos usando las variables de *offset* y *stride* para asignar, idealmente, el mismo número de tareas por hilo.

El código del kernel está diseñado para que cada hilo ejecute un ciclo sobre un conjunto de elementos del arreglo usando un patrón de memoria que puede ser modificado junto con la homogeneidad de los hilos. Cada hilo en un bloque recibe una dirección de inicio de los elementos que procesará, determinado por el índice del hilo y el valor del *stride* y del *offset*.

El parámetro GROUP_SIZE afecta la secuencia de instrucciones de los hilos y rompe la capacidad de paralelización que se puede obtener con el GPU.

Prueba	STRIDE	OFFSET	GROUP_SIZE	T. Ejecución
1	32	0	512	0.197248
2	16	0	512	0.139968
3	8	0	512	0.205216
4	32	1	512	0.172320
5	32	0	16	0.226464
6	32	0	8	0.152768
7	8	1	8	0.227264

Tabla 1: Tabla de pruebas

3. Producto matricial

En esta sección se calculará el producto de dos matrices, el cual se define de la siguiente manera:

Dadas dos matrices cuadradas A y B, su producto es otra matriz C cuyos elementos se obtienen multiplicando las filas de A por las columnas de B (utilizando producto punto).

Un código secuencial simple para ser ejecutado por el CPU es:

```
for (unsigned int i = 0; i<N; i++){
    for (unsigned int j = 0; j<N; j++) {
        float sum = 0;
        for (unsigned int k = 0; k<n; k++)
        sum += A[i * n + k] * B[k * n + j];
        C[i * n + j] = (float) sum;
    }
}
```

Una manera adecuada de llevar este código a CUDA, consiste en que cada hilo calcule un elemento de C a partir del renglón y columna que le corresponde:

- Lee un renglón de A
- Lee una columna de B
- Realiza el producto punto

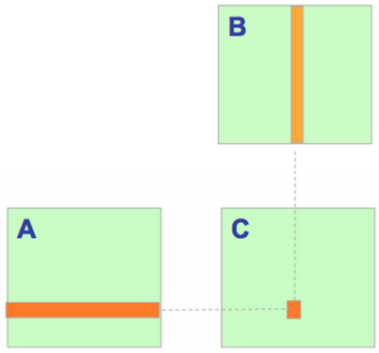

Figura 2

- ¿Cuántas veces se lee cada una de las entradas de A y B?

  Las entradas de A se leen n veces las columnas de B. Las entradas de B se leen m veces las filas de A

4. Área del conjunto de Mandelbrot

El conjunto de Mandelbrot es el conjunto de números complejos c para los cuales la iteración: $z = z_2 + c$ no diverge para la condición inicial $z = c$. Para determinar aproximadamente si un punto c pertenece al conjunto se realiza un número finito de iteraciones con ese número c. Si se cumple la condición: $z > 2$.

Entonces se considera que el punto está fuera del conjunto de Mandelbrot. El radio de puntos dentro y fuera del conjunto es un estimador del área del conjunto.

El archivo mandel.c contiene una versión secuencial del código para estimar el área de Mandelbrot.

- ¿Detecta alguna diferencia en el desempeño obtenido en este ejercicio y el de producto matricial? De ser positiva su respuesta, ¿A qué factores atribuye esta diferencia?

  Si hay diferencias. Si se considera que en los dos problemas estamos trabajando con matrices (en el producto matricial esto es evidente, en el mandelbrot se genera una malla que equivale a una matriz), aunque el tamaño se puede igualar, el desempeño es totalmente diferente.

Esto se debe a que en el producto matricial el acceso a cada uno de los datos lo hacemos n veces la columna de la otra matriz y m veces los renglones de la primera, además de hacer las sumas de cada producto renglonXcolumna. En cambio, en el de mandelbrot el acceso solo es una vez por cada dato, aunque la carga está en el loop por cada dato.

Entonces las diferencias están en el número de veces en que se accede cada dato (el acceso a memoria global del GPU es muy lenta) y en la carga de los loop de cada una que va a estar en función del tamaño de la matriz y el tamaño de la malla.

5. Uso de memoria compartida

En el producto matricial que se calculó en la sección 3 de esta práctica, las matrices A y B se encuentran en la memoria global del GPU; acceder las matrices desde ahí no permite aprovechar la memoria local compartida, que es mucho más rápida, aunque solo es accesible para los hilos en un bloque.

Idealmente las matrices podrían colocarse en la memoria local compartida, pero ésta es de tamaño limitado (típicamente de 48 KB). Por ello, la forma de utilizar esta memoria es cargando secciones de las matrices originales, como se ejemplifica en la figura siguiente:

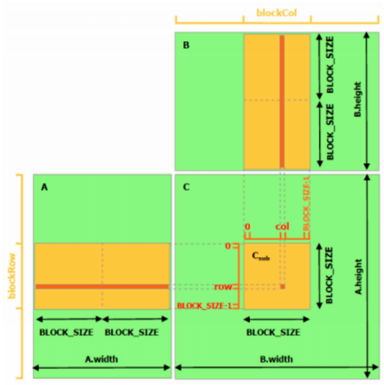

Figura 3

En cada bloque se cargan las secciones en naranja de las matrices A y B, y cada hilo sigue calculando entradas individuales del producto de esas secciones.

En la tabla de ejecución con diferentes tamaños de matrices y variando el número de bloque podemos observar resultados muy interesantes en cuanto al tiempo de ejecución, podemos observar en la columna TSecuential el tiempo que toma a un CPU ejecutar la multiplicación sin paralelismo, en TShared el tiempo que toma usando memoria compartida en el GPU y en TGlobal el tiempo que toma la ejecución utilizando memoria global del GPU.

Al variar el tamaño de bloque en el ejemplo de 512 x 512 con 16, 32 y 64 bloques de iguales dimensiones de threads en el GPU observamos como al aumentar el tamaño de threads en el bloque el tiempo de ejecución disminuye notablemente, esto es porque nos acercamos a una memoria global y la matriz no es de grandes dimensiones, para este entonces nosotros deberíamos suponer que la memoria global beneficia más que la memoria compartida.

En ningún momento podemos deducir otra cosa si solo observamos la tabla, lamentablemente no pudimos realizar multiplicaciones de mayores dimensiones con el hardware utilizado, para poder suponer lo contrario.

Matriz	Bloque	TSecuential	TShared	TGlobal
32 x 32	16	0.000000	0.860000	0.000000
64 x 64	16	0.030000	1.230000	0.000000
128 x 128	16	0.180000	1.090000	0.030000
256 x 256	16	0.870000	1.160000	0.180000
384 x 384	16	1.730000	1.910000	0.620000
512 x 512	16	5.050000	3.800000	1.380000
512 x 512	32	5.330000	3.190000	1.440000
512 x 512	64	5.040000	0.610000	0.020000
640 x 640	16	9.900000	6.610000	2.740000
768 x 768	16	17.020000	11.020000	4.780000
896 x 896	16	27.160000	16.960000	7.450000
1024 x 1024	16	43.180000	25.120000	11.170000

Tabla 2: Ejecución con diferentes configuraciones (Milisegundos).