Práctica 4. Programación en CUDA

Arquitectura de Computadoras, ITAM[1]
Abril 2016

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
- Part 1B: Copiar los arreglos del CPU al GPU
- Part 1C: Copiar el resultado de la suma al CPU
- Part 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

[1] Práctica realizada con apoyo de Dante Gama

bloques de hilos. La implementación es similar, salvo por el índice:

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

El siguiente diagrama es útil para recordar cómo hacer el cálculo de índices:

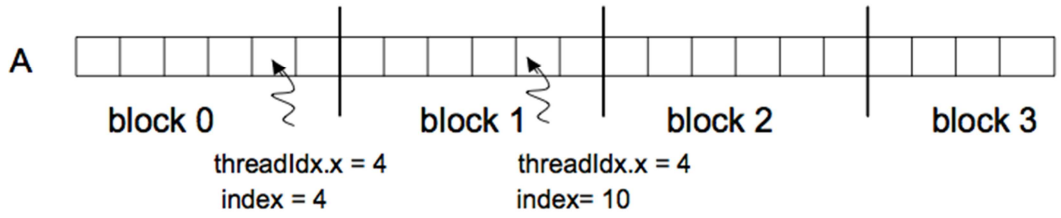


Figura 1. Direccionamiento de hilos. blockDim.x contiene el tamaño del grid en la dimensión x, en este caso 4.

Ejecute el código y comente sobre los resultados

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.

Explique brevemente qué hace este código

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.

Llene la siguiente tabla modificando los valores de STRIDE, OFFSET y GROUP_SIZE y comente sobre el resultado.

Prueba	STRIDE	OFFSET	GROUP_SIZE	T. Ejecución
1	32	0	512	
2	16	0	512	
3	8	0	512	
4	32	1	512	
5	32	0	16	
6	32	0	8	
7	8	1	8	

3. Producto matricial

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

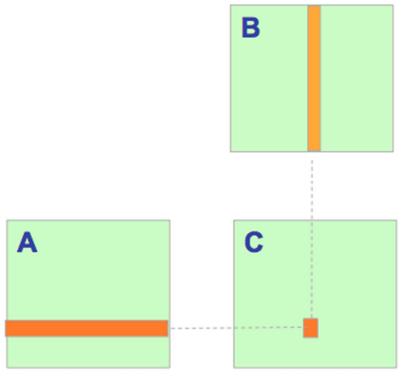
> *Dadas dos matrices cuadradas A y B, su producto es otra matriz C cuyos elementos se obtienen multiplicando las filas de A por las columnas de B (utilizando producto punto).*

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



Implemente el código correspondiente en CUDA y ejecútelo en su GPU ¿Cuántas veces se lee cada una de las entradas de A y B?

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.

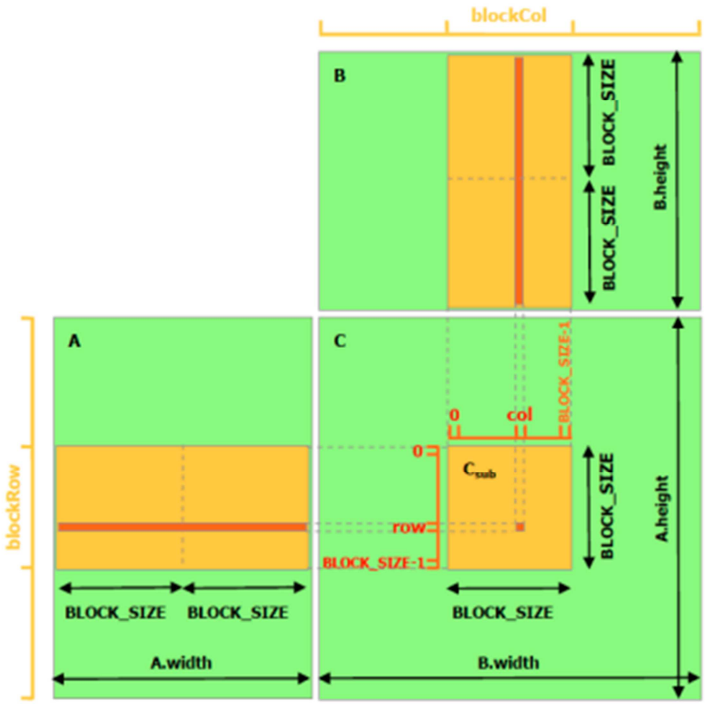
El archivo mandel.c contiene una versión secuencial del código para estimar el área de Mandelbrot.

Implemente el código correspondiente en CUDA y ejecútelo en su GPU ¿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?

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

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.

Modifique el código CUDA de la sección 3 para aprovechar la memoria compartida de los SM. Considere utilizar tamaños de bloque que dividan apropiadamente las dimensiones de las matrices en función de los resultados obtenidos en la sección 2.

Reporte las diferencias de desempeño obtenidas y reflexione sobre sus resultados