Práctica 4

Arquitectura de Computadoras
Paralelismo de datos

Fernanda Mora & Luis Román, ITAM 3/05/2016

Objetivos

El objetivo de esta práctica es:

- Implementar los conceptos teóricos de paralelismo de datos.
- Inicializarnos en la programación en CUDA, que permite implementar el paralelismo de datos.

Introducción

CUDA es un lenguaje basado en C/C++ desarrollado por NVIDIA que permite implementar programación en paralelo usando un GPU. En CUDA al CPU le llamaremos *host* y al GPU le llamaremos *device*.

El nivel más bajo de paralelismo en CUDA es el *hilo*, sin embargo para obtener un mejor rendimiento del procesador se deben tener grupos de 32 hilos (denominados *warps*). Los *hilos* tienen *memoria privada* que no es compartida con otros hilos; esta memoria es la más rápida. El CPU no puede tener acceso a la *memoria privada*.

Los hilos a su vez se agrupan en *bloques*. Cada bloque de hilos comparte una *memoria local*; dicha memoria local no es compartida entre bloques y es liberada hasta que todos los hilos de un bloque terminan de ejecutarse. El CPU tampoco puede tener accesor a la *memoria local*.

Finalmente los bloques se agrupan en *grids*. La memoria global es compartida entre grids. El CPU sí puede leer y escribir en la memoria global del GPU. Adicionalmente, es importante mencionar que esta memoria es la más lenta de las tres.

A continuación se muestra el diagrama que ejemplifica los conceptos anteriores.

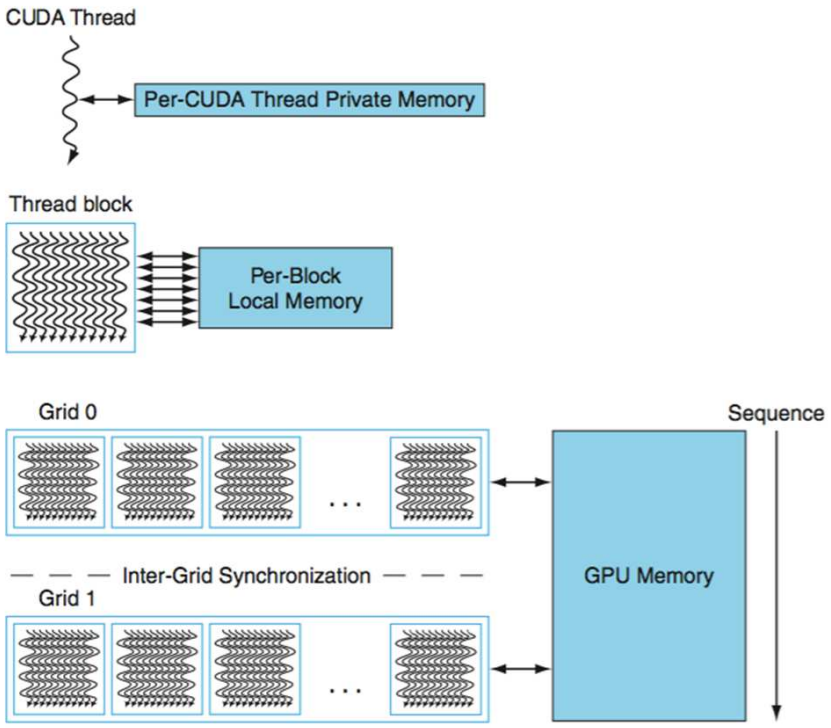

Figura 1: Unidades de trabajo en CUDA

Las funciones que se ejecutan en el *device* se denominan *kernels* y tienen la siguiente sintaxis:

```
1    nombre_kernel<<<dimGrid, dimBlock>>>(param1, param 2, ...)
```

$dimGrid$ y $dimBlock$ especifican las dimensiones del código (en número de bloques) y las dimensiones de un bloque (en número de hilos). A continuación se muestra la relación espacial entre unidades.

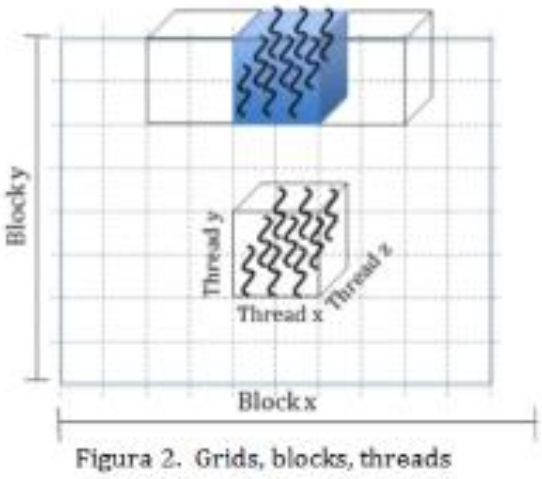

Figura 2: Relación dimensional de unidades

Desarrollo

0.1. Introducción: suma de vectores

Para este primer ejercicio introductorio se usó el código disponible en el folder incluido en la práctica, el cual contiene un archivo con código de CUDA para ser editado y ejecutado.

Este ejercicio es un código simple que suma dos vectores de enteros.

El objetivo es introducir 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.

0.1.1. Uso de la memoria del GPU

Las rutinas que tienen que ver con el manejo de memoria en el GPU son las siguientes:

- Parte 1A: Reserva de la memoria en el GPU

```
1    cudaMalloc((void**)&d_a, sz);
2    cudaMalloc((void**)&d_b, sz);
3    cudaMalloc((void**)&d_c, sz);
```

- Parte 1B: Copiar los arreglos del CPU al GPU

```
1    cudaMemcpy(d_a, a, sz, cudaMemcpyHostToDevice);
2    cudaMemcpy(d_b, b, sz, cudaMemcpyHostToDevice);
3    cudaMemcpy(d_b, c, sz, cudaMemcpyHostToDevice);
```

- Parte 1C: Copiar el resultado de la suma al CPU

```
1    cudaMemcpy(c, d_c, sz, cudaMemcpyDeviceToHost);
```

- Part 1D: Liberar la memoria del GPU.

```
1    cudaFree(d_a);
2    cudaFree(d_b);
3    cudaFree(d_c);
```

0.1.2. Llamado de kernels

- Parte 2A: Llamar al kernel usando un grid de una dimensión y un sólo bloque de hilos (NUMBLOCKS y THREADSPERBLOCK están configurados para esto)

```
1     dim3 dimGrid(NUM_BLOCKS, 1, 1);
2     dim3 dimBlock(THREADS_PER_BLOCK, 1, 1);
3     vect_add <<<dimGrid, dimBlock>>> (d_a, d_b, d_c);
```

- Parte 2B: Implementación del kernel para realizar la suma de los vectores en el GPU

```
1  __global__ void vect_add(int* d_a, int *d_b, int *d_c)
2  {
3      if(threadIdx.x <= ARRAY_SIZE){    //chequemos que nuestro indice no se ↩
           salga de nuestro tama o de arreglo
4      int idx = threadIdx.x;  //indice de este hilo
5      int a = d_a[idx];    //accedemos al idx-esimo elemento del arreglo a
6      int b = d_b[idx];     //accedemos al idx-esimo elemento del arreglo b
7      d_c[idx] = a+b;  //guardar la suma en la posicion idx-esima del vector c
8      }
9  }
```

- Parte 2C: Implementación del kernel pero esta vez permitiendo múltiples bloques de hilos. La implementación es similar, salvo por el índice: $intidx = threadIdx.x + (blockIdx.x * blockDim.x)$;

```
1  __global__ void vect_add_multiblock(int *d_a)
2  {
3      int idx = threadIdx.x + (blockIdx.x * blockDim.x);
4      if(idx < ARRAY_SIZE){
5      d_c[idx] = d_a[idx] + d_b[idx];
6    }
7  }
```

0.2. Configuración de bloques e hilos

El objetivo de esta sección es analizar el efecto en el desempeño al configurar de explícitamente el número de bloques y de hilos al invocar la ejecución de un kernel. Como hemos visto, 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.

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.

¿Qué hace este código?

El código inicializa un vector con los elementos del $1 - 2^{10}$.

La dimensión del grid es de $n\_blocks = 256$ y el número de hilos por bloque es de 512 ($n\_threads\_per\_block = 128$).

El procesamiento por bloque se divide a partir de los parámetros STRIDE, OFFSET, y GROUP_SIZE.

Primero se determina el número de elementos por hilo. Posteriormente se determina la dirección de inicio para el hilo, *block_start_idx*, en donde:

```
1 int n_elem_per_thread = N / (gridDim.x * blockDim.x);
2 int block_start_idx = n_elem_per_thread * blockIdx.x * blockDim.x;
3 \\el numero de elementos por hilo, por el tama o de bloque, por el numero de ↩
    hilos en el bloque
```

La dirección de inicio de hilo se obtiene utilizando el parámetro de STRIDE (se toma el módulo del STRIDE más el OFFSET).

```
1 int thread_start_idx = block_start_idx
2     + (threadIdx.x / STRIDE) * n_elem_per_thread * STRIDE
3     + ((threadIdx.x + OFFSET) %STRIDE);
```

La variable GROUP_SIZE afecta la secuencia de instrucciones de los hilos, afectando el grado de paralelismo que se puede obtener con el GPU. Si el último bit del cociente del identificador del hilo entre el parámetro GROUP_SIZE es 0, entonces la instrucción que deberá ejecutar el hilo es una suma. En otro caso, entonces la instrucción es una multiplicación.

```
1     int group = (threadIdx.x / GROUP_SIZE) & 1;
2     for(int idx=thread_start_idx; idx < thread_end_idx; idx+=STRIDE)
3     {
4         if(!group) a[idx] = a[idx] * a[idx];
5         else       a[idx] = a[idx] + a[idx];
6     }
```

Finalmente, los resultados de variar los parámetros son los siguientes:

STRIDE	OFFSET	GROUP_SIZE	T. EJECUCION
32	0	512	0.061
16	0	512	0.076
8	0	512	0. 0.072
32	1	512	0.074
32	0	16	0.070
32	0	8	0.074
8	1	8	0.071

El desempeño óptimo (menor tiempo de ejecución) se obtiene cuando el STRIDE es grande, de 32, el GROUP_SIZE grande, de 512 y el OFFSET nulo. Es decir $STRIDE = 32, OFFSET = 0, GROUP\_SIZE = 512$.

Esto se debe a que como dijimos en la introducción, lo ideal es tener grupos de GRIDS (32 hilos). El parámetro GROUP_SIZE de 512 es el más grande e introduce menos heterogeneidad en el código entre hilos, lo cual también mejora el desempeño. Finalmente como puede verse, el OFFSET también tiene resultados negativos en el desempeño.

0.3. Producto matricial

En esta sección se calculará el producto de dos matrices cuadradas A, B de dimensiones compatibles.

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

```
1 for (unsigned int i = 0; i<N; i++){
2           for (unsigned int j = 0; j<N; j++) {
3           float sum = 0;
4           for (unsigned int k = 0; k<n; k++)
5               sum += A[i * n + k] * B[k * n + j];
6           C[i * n + j] = (float) sum;
7       }
8 }
```

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.

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

0.4. Área del conjunto de Mandelbrot

El conjunto de Mandelbrot es el conjunto de números complejos $c \in \mathbb{C}$ tales que $|z_n|$ no diverge con z_n tal que $z_{n+1} = z_n^2 + c$ para $n > 0$ y $z_1 = c$.

Para determinar aproximadamente si un punto c pertenece al conjunto se realiza un número finito de iteraciones con ese número c. Para hacer un cálculo práctico, si la norma al cuadrado

del complejo excede por 4 entonces se considera que dicho c no es un número de Mandelbrot, i.e. $|z| > 2$. Y el *área de Mandelbrot* se aproxima como la proporción de números de Mandelbrot, es decir, los puntos dentro y fuera del conjunto son 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?**

Sí, el código en CUDA es mucho más rápido. Sin embargo, la implementación matricial del ejercicio pasado es más rápida comparativamente, i.e. utilizando arreglos del mismo tamaño el desempeño es menor. La diferencia la podríamos atribuir a la heterogeneidad de las instrucciones que se ejecutan en los hilos, la cual afecta el paralelismo y por lo tanto afecta el desempeño.

Implementación	tiempo ejecución
Secuencial	4,332 seg
CUDA	18 seg

0.5. Uso de memoria compartida

En el producto matricial que se calculó en la secci ón 3 de esta práctica, las matrices A, 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 mem oria 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:

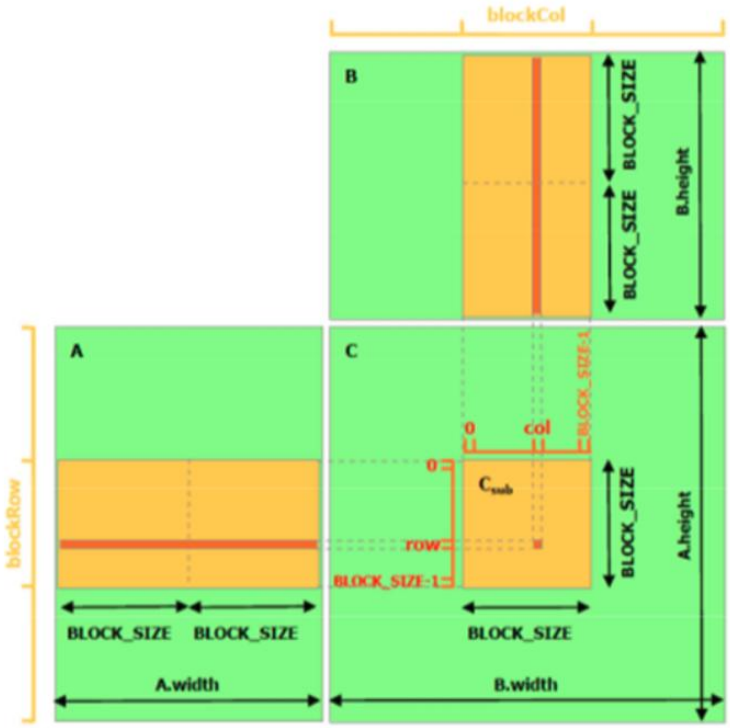

Figura 3: Memoria

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**.

El objetivo de este ejercicio, es, pues, hacer uso de la memoria local para mejorar el tiempo de ejecución al disminuir los accesos a memoria global del GPU.

Conclusiones

Referencias

[1] Stallings, William. Computer organization and architecture: designing for performance. Pearson Education India, 2000.

[2] Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2011.

[3] Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional, 2010.