Quinto Laboratorio de Algoritmos paralelos y distribuidos
Programacion en CUDA

Prof. Boris Chullo Llave

Dec 2021

1 Organización de hilos en CUDA

Los hilos (*threads*) en CUDA constituyen la base de la ejecución de aplicaciones en paralelo. Como ya hemos comentado anteriormente, cuando se lanza un *kernel* se crea una malla de hilos donde todos ellos ejecutan el mismo programa. Es decir, el *kernel* especifica las instrucciones que van a ser ejecutadas por cada hilo individual. En el ejercicio anterior no hicimos uso de esta propiedad ya que lanzamos lo que podríamos llamar un "*kernel* escalar", mediante la sintaxis $<<< 1,1 >>>$, esto es, un bloque con un solo hilo de ejecución. Esto no es otra cosa que una función estándar de C que se ejecuta de manera secuencial. En esta práctica vamos a hacer uso del paralelismo a nivel de hilo que nos proporciona CUDA lanzando más de un hilo. Las opciones para esta labor son múltiples, ya que podemos lanzar todos los hilos en un único bloque, podemos lanzar varios bloques con un solo hilo cada uno, o la opción más flexible que sería lanzar varios bloques con varios hilos en cada bloque. Por ejemplo, para lanzar un *kernel* con N hilos de ejecución, todos ellos agrupados en un único bloque, la sintaxis sería:

```
myKernel<<<1,N>>>(arg_1,arg_2,...,arg_n);
```

De este modo, la GPU ejecuta simultáneamente N copias de nuestro *kernel*. La forma de aprovechar este paralelismo que nos brinda la GPU es hacer que cada una de esas copias o hilos realice la misma operación pero con datos distintos, es decir, asignar a cada hilo sus propios datos. La pregunta que nos podemos hacer en este punto es ¿y cómo podemos identificar cada uno de los hilos para poder repartir el trabajo? La respuesta está en una variable incorporada (*built-in*) denominada *threadIdx*, que es de tipo `int` y que únicamente puede utilizarse dentro del código del *kernel*. Esta variable adquiere un valor distinto para cada hilo en tiempo de ejecución. Cada hilo puede almacenar su número de identificación en una variable entera:

```
int myID = threadIdx.x;
```

Cuando antes lanzamos el *kernel* mediante la sintaxis anterior, especificamos que queríamos una malla formada por un único bloque y N hilos paralelos. Esto le dice al sistema que queremos una malla unidimensional (los valores escalares se interpretan como unidimensionales) con los hilos repartidos a lo largo del eje x (Figura 4.1), ya que por defecto se considera este eje como la dimensión de trabajo (de ahí que se añada el sufijo ".x" para indicar dicha dirección). De este modo, cada uno de los hilos tendrá un valor distinto de `threadIdx.x` que irá desde 0 hasta $N-1$. Es decir, cada hilo tendrá su propio valor de `myID` que permitirá al programador decidir sobre qué datos debe trabajar cada uno de ellos.

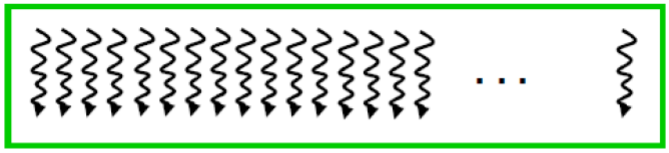

Figura 4.1. Malla (*grid*) formada por un único bloque (*block*) de N hilos paralelos (*threads*) repartidos a lo largo del eje x. Cada hilo se puede identificar mediante la variable `threadIdx.x`.

El hardware de la GPU limita el número de hilos por bloque con que podemos lanzar un *kernel*. Este número puede ser mayor o menor dependiendo de la capacidad de cómputo de nuestra GPU y en particular no puede superar el valor dado por `maxThreadsPerBlock`, que es uno de los campos que forman parte de la estructura de propiedades `cudaDeviceProp`. Para las arquitecturas con capacidad de cómputo 1.0 este límite es de 512 hilos por bloque. Otra alternativa para lanzar un kernel de N hilos consistiría en lanzar N bloques pero de un hilo cada uno. En este caso la sintaxis para el lanzamiento sería:

```
myKernel<<<N,1>>>(arg_1,arg_2,...,arg_n);
```

Mediante esta llamada tendríamos una malla formada por N bloques repartidos a lo largo del eje x y con un hilo cada uno (Figura 4.2).

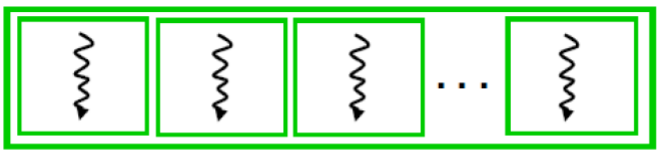

Figura 4.2. Malla formada por N bloques paralelos de 1 hilo repartidos a lo largo del eje x. Cada bloque se puede identificar mediante la variable `blockIdx.x`.

Si ahora queremos identificar los hilos para poder repartir tareas, no podemos utilizar la variable anterior (`threadIdx.x`), ya que esta variable sólo permite identificar a los hilos que pertenecen al mismo bloque. Esto quiere decir que

dentro de cada bloque todos los hilos comienzan a numerarse a partir del número 0, y por lo tanto, en nuestro ejemplo, al haber un único hilo en cada bloque, todos los hilos tendrían el mismo valor de threadIdx.x y por tanto en todos ellos tendríamos myID = 0. Ahora la identificación de los hilos pasa por identificar el bloque en el que se encuentra cada hilo. Esto lo podemos hacer mediante otra variable similar a la anterior denominada `blockIdx`:

```
int myID = blockIdx.x;
```

Al igual que antes, cada uno de los hilos tendrá un valor distinto de `blockIdx.x` que irá desde 0 hasta $N-1$. También existe un límite impuesto por el hardware en el número máximo de bloques que podemos lanzar, que para la mayoría de las arquitecturas es de 65.535 bloques. El valor particular para nuestra GPU lo podemos averiguar a partir del campo `maxGridSize[i]` de nuestra estructura de propiedades, y que nos da el máximo número de bloques permitidos en cada una de las direcciones del espacio ($i = 0, 1, 2 \rightarrow$ ejes x, y, y z respectivamente).

Por último, el caso más general sería el lanzamiento de un kernel con M bloques y N hilos por bloque (Figura 4.3), en cuyo caso tendríamos un total de $M \times N$ hilos . La sintaxis en este caso sería:

```
myKernel<<<M,N>>>(arg_1,arg_2,...,arg_n);
```

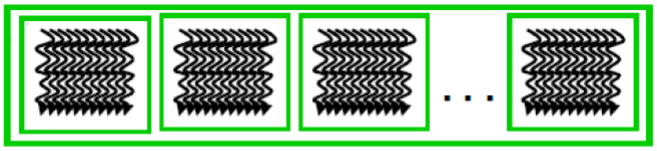

Figura 4.3. Malla formada por M bloques paralelos (`gridDim.x`) de N hilos cada uno (`blockDim.x`) repartidos a lo largo del eje x.

Ahora habrá un total de $M \times N$ hilos ejecutándose en paralelo que para identificarlos será necesario hacer uso conjunto de las dos variables anteriores y de dos nuevas constantes que permitan a la GPU conocer en tiempo de ejecución las dimensiones del kernel que hemos lanzado. Estas dos constantes son `gridDim.x` y `blockDim.x`. La primera nos da el número de bloques (en nuestro caso sería M) y la segunda el número de hilos que tiene cada bloque (N en nuestro ejemplo). De este modo, dentro del kernel cada hilo se puede identificar de forma unívoca mediante la siguiente expresión (figura 4.4):

```
int myID = threadIdx.x + blockDim.x * blockIdx.x;
```

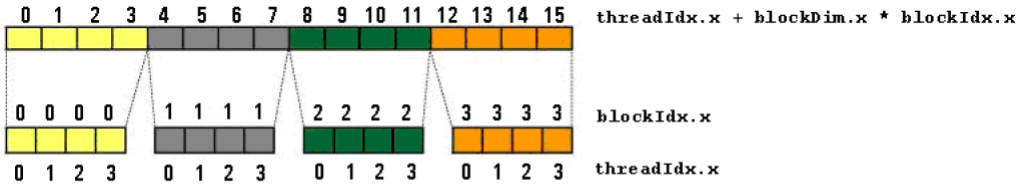

Figura 4.4. Identificación de los hilos dentro de una malla formada por cuatro bloques de cuatro hilos cada uno y repartidos a lo largo del eje x.

2 Ejemplo:

En el siguiente ejemplo se muestra forma de aprovechar el paralelismo de datos programando un *kernel* que realice la suma de dos vectores de longitud N inicializados con valores aleatorios comprendidos entre 0 y 1:

```
%%cu
// includes
#include <stdio.h>
#include <stdlib.h>
// defines
#define N 16 // tamano de los vectores
#define BLOCK 5 // tamano del bloque
// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (
    kernel)
__global__ void suma( float *a, float *b, float *c )
{
    int myID = threadIdx.x + blockDim.x * blockIdx.x;
// Solo trabajan N hilos
    if (myID < N)
    {
        c[myID] = a[myID] + b[myID];
    }
}
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// declaraciones
    float *vector1, *vector2, *resultado;
    float *dev_vector1, *dev_vector2, *dev_resultado;
// reserva en el host
    vector1 = (float *)malloc(N*sizeof(float));
    vector2 = (float *)malloc(N*sizeof(float));
    resultado = (float *)malloc(N*sizeof(float));
// reserva en el device
    cudaMalloc((void**)&dev_vector1, N*sizeof(float));
    cudaMalloc((void**)&dev_vector2, N*sizeof(float));
    cudaMalloc((void**)&dev_resultado, N*sizeof(float));
// inicializacion de vectores
    for (int i = 0; i < N; i++)
    {
        vector1[i] = (float) rand() / RAND_MAX;
        vector2[i] = (float) rand() / RAND_MAX;
```

```
    }

// copia de datos hacia el device
    cudaMemcpy(dev_vector1, vector1, N*sizeof(float),
    cudaMemcpyHostToDevice);
    cudaMemcpy(dev_vector2, vector2, N*sizeof(float),
    cudaMemcpyHostToDevice);
// lanzamiento del kernel
// calculamos el numero de bloques necesario para un tamanho de
    bloque fijo
    int nBloques = N / BLOCK;
    if (N % BLOCK != 0){
        nBloques = nBloques + 1;
    }
    int hilosB = BLOCK;
    printf("Vector de %d elementos\n", N);
    printf("Lanzamiento con %d bloques (%d hilos)\n", nBloques,
    nBloques*hilosB);
    suma<<< nBloques, hilosB >>>( dev_vector1, dev_vector2,
    dev_resultado );
// recogida de datos desde el device
    cudaMemcpy(resultado, dev_resultado, N*sizeof(float),
    cudaMemcpyDeviceToHost);
// impresion de resultados
    printf( "> vector1:\n");
    for (int i = 0; i < N; i++){
        printf("%.2f ", vector1[i]);
    }
    printf("\n");
    printf( "> vector2:\n");
    for (int i = 0; i < N; i++){
        printf("%.2f ", vector2[i]);
    }
    printf("\n");
    printf( "> SUMA:\n");
    for (int i = 0; i < N; i++){
        printf("%.2f ", resultado[i]);
    }
    printf("\n");
// liberamos memoria en el device
    cudaFree( dev_vector1 );
    cudaFree( dev_vector2 );
    cudaFree( dev_resultado );
// salida
    printf("\npulsa INTRO para finalizar...");
    fflush(stdin);
    char tecla = getchar();
    return 0;
}
```

3 Tarea de Laboratorio:

1. Ejecutar un *kernel* compuesto por 24 hilos que se encargue de rellenar tres arrays con los índices de identificación de cada hilo (un array por cada tipo de índice).

2. Cada hilo escribirá en el correspondiente array su índice de hilo (`threadIdx.x`), su índice de bloque (`blockIdx.x`) y su índice global (`threadIdx.x + blockDim.x * blockIdx.x`).

3. El *kernel* se debe ejecutar tres veces, cada una de ellas con diferentes opciones de organización:

   - 1 bloque de 24 hilos.
   - 24 bloques de 1 hilo.
   - 4 bloques de 6 hilos.

4. En cada ejecución se debe mostrar por pantalla el contenido de los tres arrays:

```
>> Opcion 1: 1 bloque 24 hilos
indice de hilo:
0 1 2 3 4 5 6 7 ...
indice de bloque:
0 0 0 0 0 0 0 0 ...
indice global:
0 1 2 3 4 5 6 7 ...
>> Opcion 2: 24 bloques 1 hilo
...
...
...
>> Opcion 3: 4 bloques 6 hilos
...
...
...
```