UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ingeniería

Introducción a CUDA C

Laboratorio de Intel para la Academia y Cómputo de alto desempeño

Elaboran: Ariel Ulloa Trejo
Fernando Pérez Franco
Revisión: Ing. Laura Sandoval Montaño

Temario

1. Antecedentes
2. El GPU
3. Funciones y vectores
4. Manejo de matrices
5. Memoria compartida

4 Matrices

Errores

Vimos en las funciones pasadas que regresan un valor *cudaError_t.*
Éste es usado para detectar errores; si la ejecución fue exitosa, regresa un *cudaSucces*. En otro caso, el código del error será regresado.

Para poder leer los códigos de los errores, usamos *cudaGetErrorString().*
cudaGetLastError() envía el último error. Si hubo uno antes de éste, no es reportado.

```
cudaError_t cudaGetLastError ( void )
```

```
const char* cudaGetErrorString (cudaError_t error)
error – code to convert to string
```

Errores comunes:

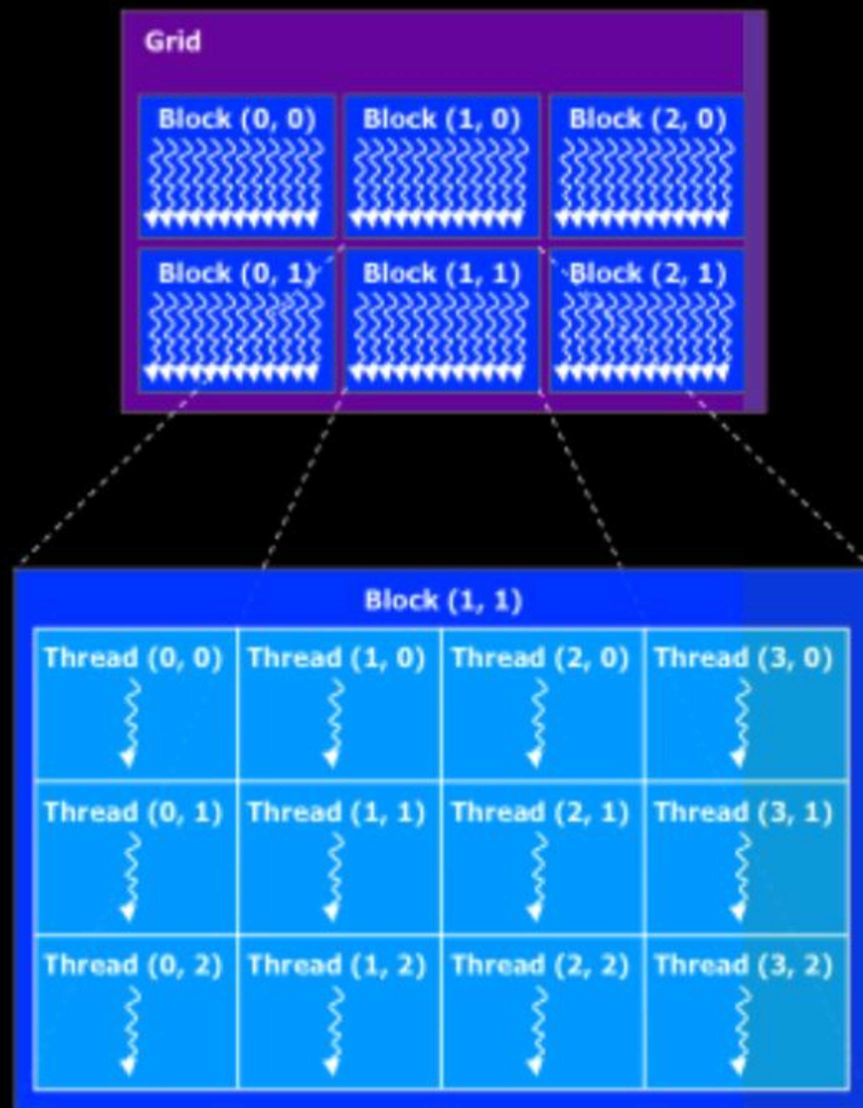
Como los kernels son asíncronos, es necesario bloquear la ejecución hasta que el dispositivo haya terminado su trabajo. Para esto utilizamos la función cudaDeviceSynchronice().

Utilizar variables en un segmento de código donde no existen.
La configuración de ejecución es inválida.

05_errores_01.cu

Más de grids, bloques e hilos:

CUDA permite la ejecución de grids compuestos por bloques hasta de 3 dimensiones, y a su vez, cada bloque compuesto por hilos en 3d.
Para hacerlo, utilizamos dim3 (para encapsular datos multidimensionales).



```
Kernel<<<dim3(Ax, Ay, Az), dim3(Bx,By,Bz)>>>()

Kernel<<<dim3(10, 32), dim3(10, 10)>>>()
```

En caso de que una dimensión no sea especificada, es reemplazada automáticamente por 1.

Ejemplo:

Una matriz de 6x6 es mapeada en un arreglo de 36 elementos. El objetivo es dividir la matriz en 4 bloques de 3x3 y el kernel sumará a cada elemento el primero de cada bloque. El resultado se guarda en otra matriz del mismo tamaño.

0	1	2	0	1	2
3	4	5	3	4	5
6	7	8	6	7	8
0	1	2	0	1	2
3	4	5	3	4	5
6	7	8	6	7	8

Consideremos dos datos:

- BLOCK_SIZE = 3 (longitud de cada bloque)
- STRIDE = 6 (longitud de la matriz completa)

En el dispositivo, cada pequeño bloque será mapeado en un bloque dentro del grid y para cada operación, existirá un hilo. Por lo tanto existen tantos hilos como elementos de la matriz.

0	1	2	0	1	2
3	4	5	3	4	5
6	7	8	6	7	8
0	1	2	0	1	2
3	4	5	3	4	5
6	7	8	6	7	8

La configuración de la ejecución será, por lo tanto:

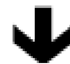
dim3 ThreadsBlocks(3, 3);
dim3 GridBlocks(2 , 2);

Block 0,0	Block 1,0
Block 0,1	Block 1,1

El truco está en generar los índices viendo a la memoria como un arreglo de 36 en lugar de una matriz de 6x6.

Los índices de los valores a sumar son entonces:
{0, 3, 18, 21}

0	1	2	0	1	2
3	4	5	3	4	5
6	7	8	6	7	8
0	1	2	0	1	2
3	4	5	3	4	5
6	7	8	6	7	8



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
0	1	2	0	1	2	3	4	5	3	4	5	6	7	8	6	7	8	0	1	2	0	1	2	3	4	5	3	4	5	6	7	8	6	7	8

Para asignar los índices de cada elemento de la matriz a los hilos:

```
int bx = blockIdx.x;
int by = blockidx.y;
Int tx = threadIdx.x;
Int ty = threadIdx.y;
d_a[ (by * BLOCK_SIZE + ty) * STRIDE + (bx * BLOCKSIZE + tx)
];
```

Para encontrar el elemento que se sumará a cada uno, simplemente cambiamos el índice del hilo a 0:

```
d_a[ (0*3+0)*6+(0*3+0)] = a[0]
d_a[ (0*3+0)*6+(1*3+0)] = a[3]
d_a[ (1*3+0)*6+(0*3+0)] = a[18]
d_a[ (1*3+0)*6+(1*3+0)] = a[21]
```

06_matrices_01.cu

Ejemplo: Programa que suma dos matrices cuadradas.

Ejercicio: Hacer un programa que sume dos matrices cuadradas de 250x250.

07_matrices_02.cu

08_matrices_03.cu

Ejemplo: Programa que hace la transpuesta de una matriz pequeña.

09_matrices_04.cu

Ejercicio: Programa que obtiene la transpuesta de una matriz grande.

10_matrices_05.cu