Computación de Altas Prestaciones
Seminario sobre GPGPUs
Sesión 2

Contenidos

Sesión 1 Teoría: Introducción a Computación en GPUs con CUDA, Compilación, conceptos básicos

Sesión 2: Programación de algoritmos “trivialmente paralelos”

Sesión 3: Uso de la memoria “Shared”. “Reducciones” en GPUs

Sesión 4: Optimización, temas avanzados, librerías

Programación Paralela en CUDA: threads, bloques, grids

En la sesión anterior, vimos llamadas a kernels desde el código CPU de esta forma:

Kernel <<< x,y>>> (...)

Donde x e y eran números enteros, x es el número de bloques lanzados e y es el número de threads en cada bloque.

En ese caso, dentro del kernel la identidad de un thread se obtenía como

```
int tid=threadId.x+blockIdx.x * blockDim.x;
```

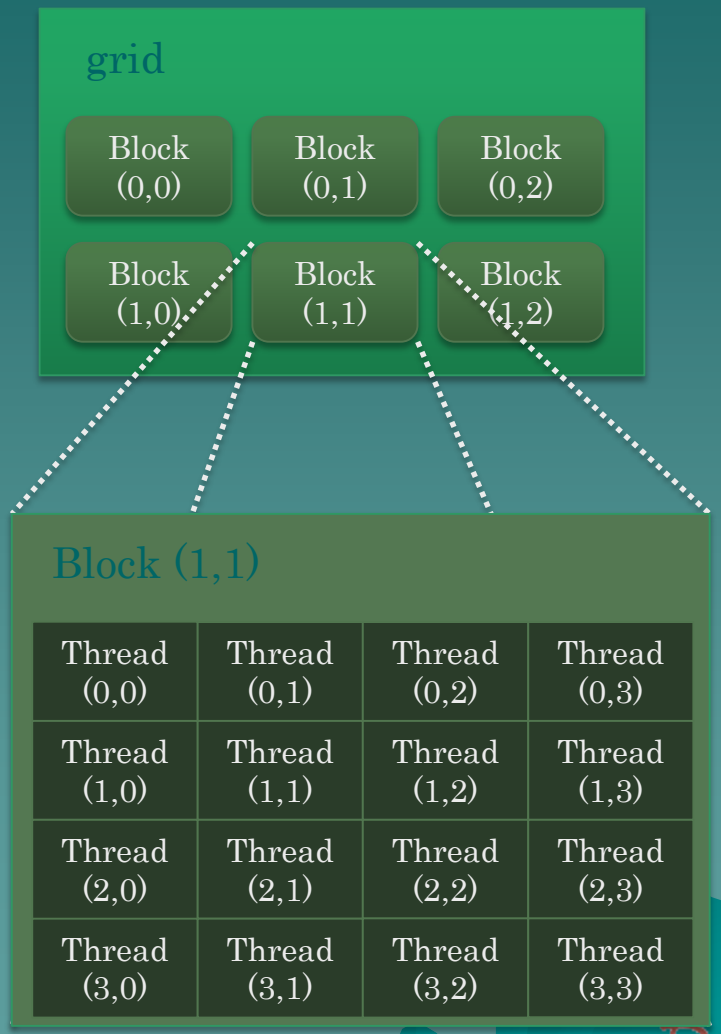
(-threadId.x será un entero entre 0 e y-1;
  -blockIdx.x será un entero entre 0 y x-1;
  -blockDim.x debe ser igual a y.
  -gridDim.x es igual a x.
)

Programación Paralela en CUDA: threads, bloques, grids

También podemos tener bloques o grids bi o tridimensionales

Los bloques pueden ser unidimensionales (como en los ejemplos anteriores), bidimensionales (como una "matriz" de threads) o incluso tridimensionales (como un "cubo de threads").

Los "grids" también pueden ser unidimensionales, bidimensionales o tridimensionales.



Programación Paralela en CUDA: threads, bloques, grids

También podemos tener bloques o grids bi o tridimensionales

Si queremos que un bloque o un grid sea bi o tridimensional, necesitamos un nuevo tipo de variable: **dim3.** Tanto x como y pueden ser de tipo int o de tipo dim3.

Ejemplo 1) Para llamar a un kernel con 20 bloques, cada uno bidimensional de 32 por 32 threads:

```
dim3 thr_p_block(32,32);
...
Kernel <<<20,thr_p_block>>>
```

Ejemplo 2) (transparencia anterior)

```
dim3 block_p_grd(2,3);
dim3 thr_p_block(4,4);
Kernel<<block_p_grd, thr_p_block>>
```

Programación Paralela en CUDA: threads, bloques, grids

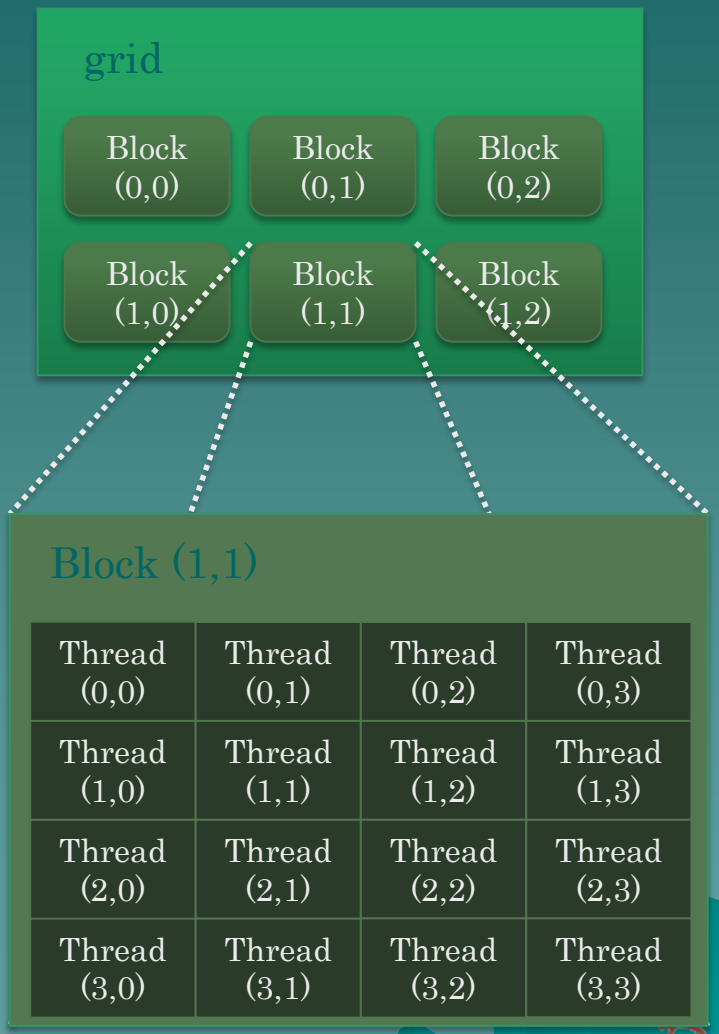
En casos bi o tridimensionales, la identidad de cada bloque/thread se determina con dos o tres variables

Grid DIM: 1D, 2D, or 3D
gridDim.x gridDim.y gridDim.z

- Block DIM: 1D, 2D, or 3D
blockDim.x blockDim.y blockDim.z

- Block ID: 1D, 2D, or 3D
blockIdx.x blockIdx.y blockIdx.z

- Thread ID: 1D, 2D, or 3D
threadIdx.x threadIdx.y threadIdx.z



Programación Paralela en CUDA: threads, bloques, grids

Supongamos que queremos mapear una matriz A con 8 filas y 12 columnas en el esquema de bloques/threads de la transparencia anterior (8 por 12 threads en total) , de forma que cada thread se haga cargo de un solo elemento de la matriz.

¿Como obtendríamos dentro del kernel la identidad de cada thread?

```
tidx= threadIdx.x+blockIdx.x*BlockDim.x
tidy= threadIdx.y + blockIdx.y*BlockDim.y
```

El thread tix,tidy se haría cargo del elemento de la matriz

➔ A[tidx+tidy*8]

Es bastante “estándar” coger bloques de threads de dimensión 16 por 16= 256.

Programación Paralela en CUDA: threads, bloques, grids

¿Como se ejecutan los kernels?

-En principio, en la GPU los kernels se ejecutan uno a uno (aunque ya es posible la ejecución concurrente de varios kernels diferentes).

-cada kernel es ejecutado por muchos threads en paralelo, operando con datos diferentes (usando la id del thread).

-Los bloques se pueden ejecutar **en cualquier orden**

Toma de tiempos en GPU: eventos en CUDA

```
cudaEvent_t start, stop;
cudaEventCreate(&start) ;
cudaEventCreate(&stop) ;

...
 cudaEventRecord(start, NULL) ;

... hacer algo en la GPU

 cudaEventRecord(stop, NULL) ;
 cudaEventSynchronize(stop) ;
cudaEventElapsedTime(&msecCPU, start, stop);
printf("GPU time = %.2f msec.\n",msecGPU);
```

Programación “Defensiva” en CUDA

Los errores en programas CUDA son difíciles de encontrar;

Es conveniente “envolver” todas las llamadas a CudaMalloc, CUDAmemCPY, etc. con una macro como esta:

```
#define CUDA_SAFE_CALL( call ) {                                    \
cudaError_t err = call;                                              \ if(
cudaSuccess != err ) {                                               \
fprintf(stderr,"CUDA: error occurred in cuda routine. Exiting...\n"); \
exit(err);                                                           \ } }

...
CUDA_SAFE_CALL( cudaMalloc((void **) &d_C, mem_size ) );
...
```

Programación Paralela en CUDA: Suma de matrices

Ejercicio 1) A partir del programa de la sesión anterior para sumar dos vectores, amplíalo a una versión bidimensional que sume dos matrices de la misma dimensión. Cada thread debe sumar un solo elemento. Por ejemplo, tomar un grid de bloques como el del ejemplo (grid 2 por 3, bloques 4 por 4), con una matriz 8 por 12.

Ejercicio 2) Queremos implementar en la gpu un filtro para suavizar una imagen (matriz). Dada una matriz Inp de dimensiones M filas por N columnas, queremos calcular una matriz Out de dimensiones M-2 filas por N-2 columnas , de forma que el elemento [i,j] de Out se calcule así:

Out(i,j)=(Inp(i-1,j)+Inp(i+1,j)+Inp(i,j-1)+Inp(i,j+1)+Inp(i,j))/5.0

De forma que cada thread haga el cálculo de un solo elemento de la matriz Out. Hazlo con un bloque bidimensional.