LESSON 1

Execution Model

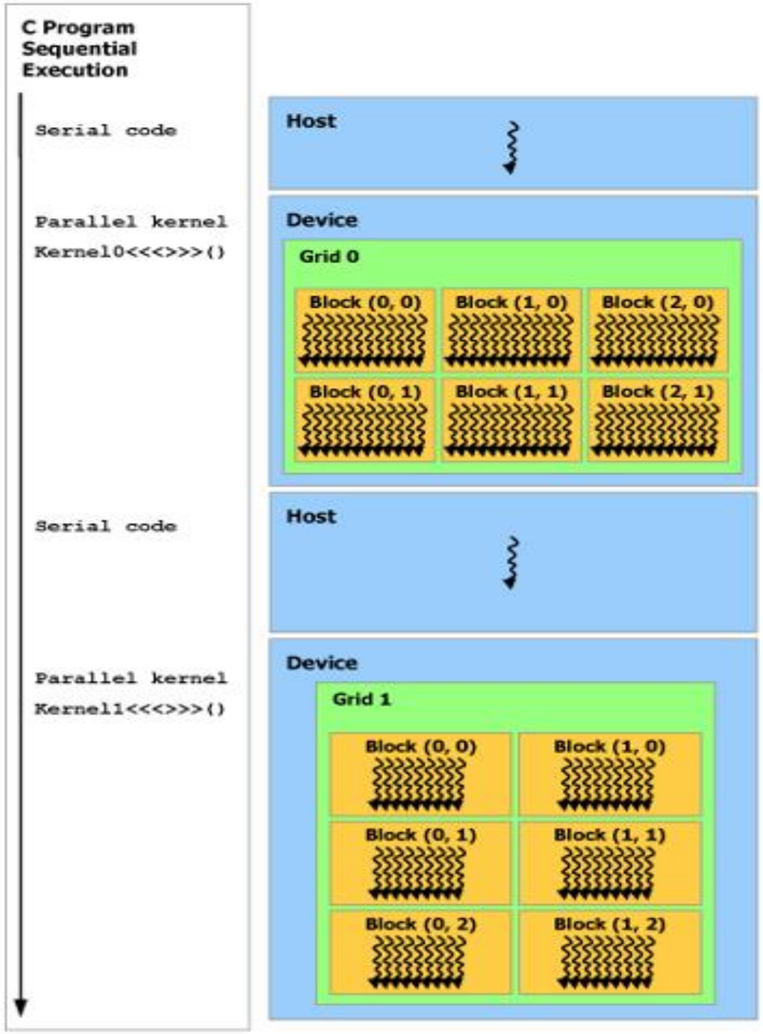
TOPICS

- How the execution configuration works in the device
- How to create different grids in the device to run your code

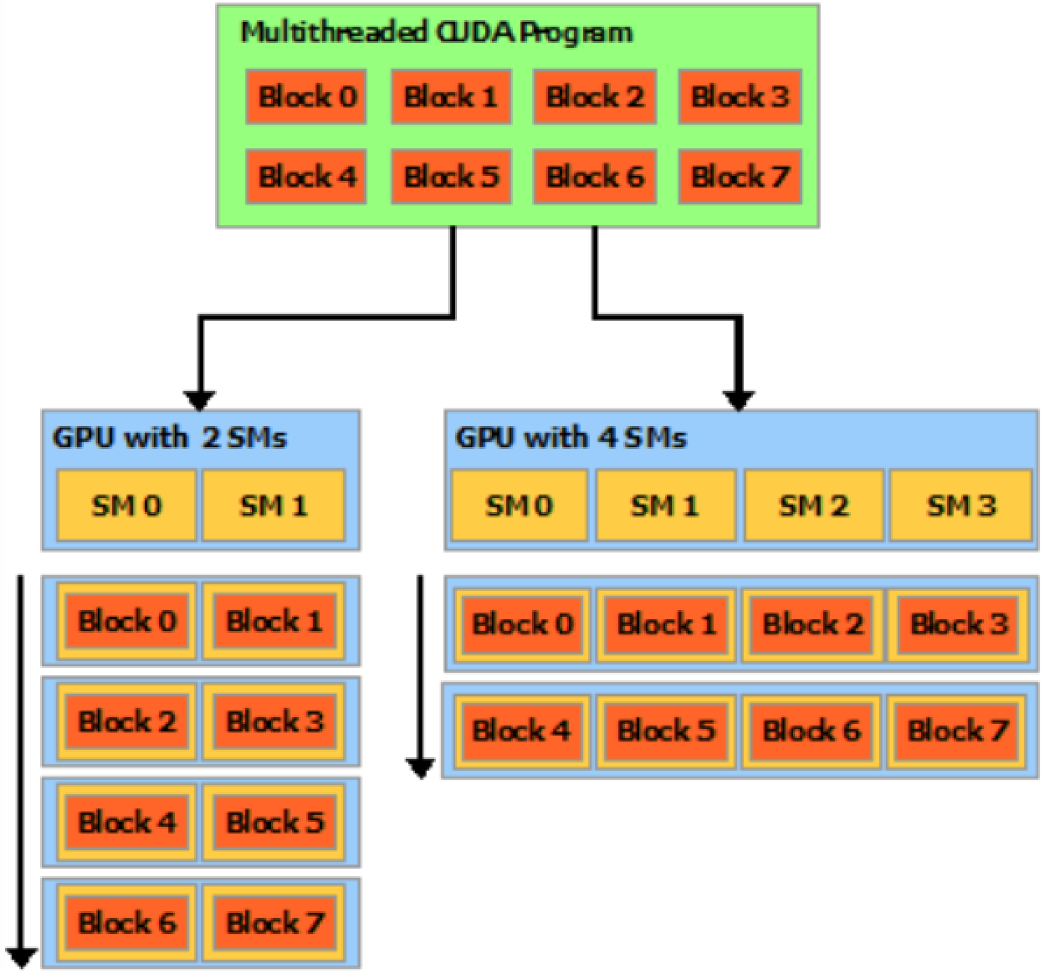
Key words: *execution, configuration, threads, blocks, grid, threadIdx, blockIdx, blockDim, gridDim.*

Heterogeneous host + device application C programming model

- The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program.

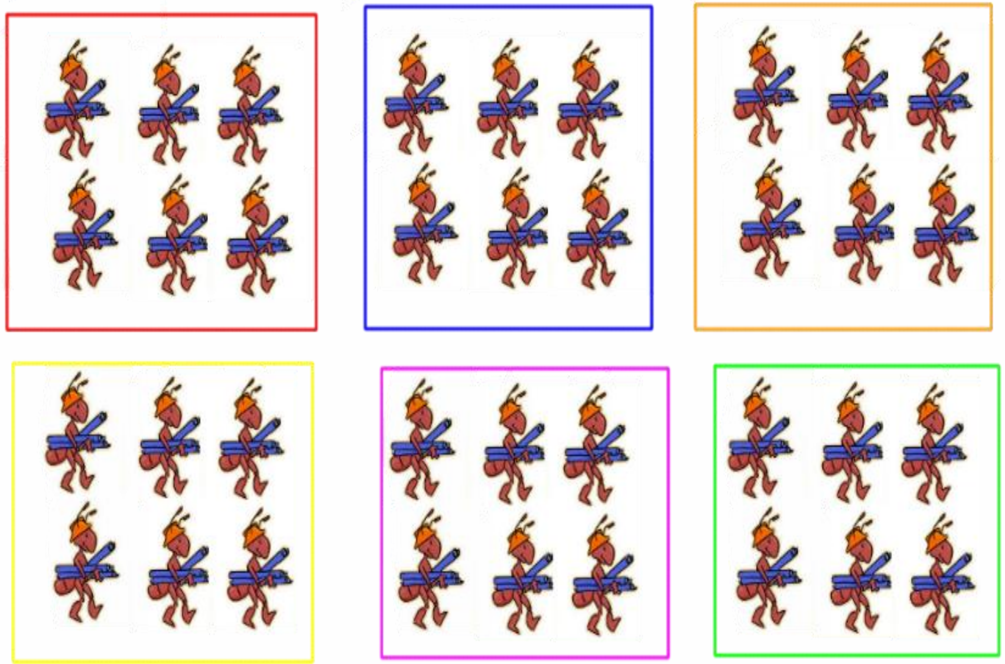


- This scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions.



CPU

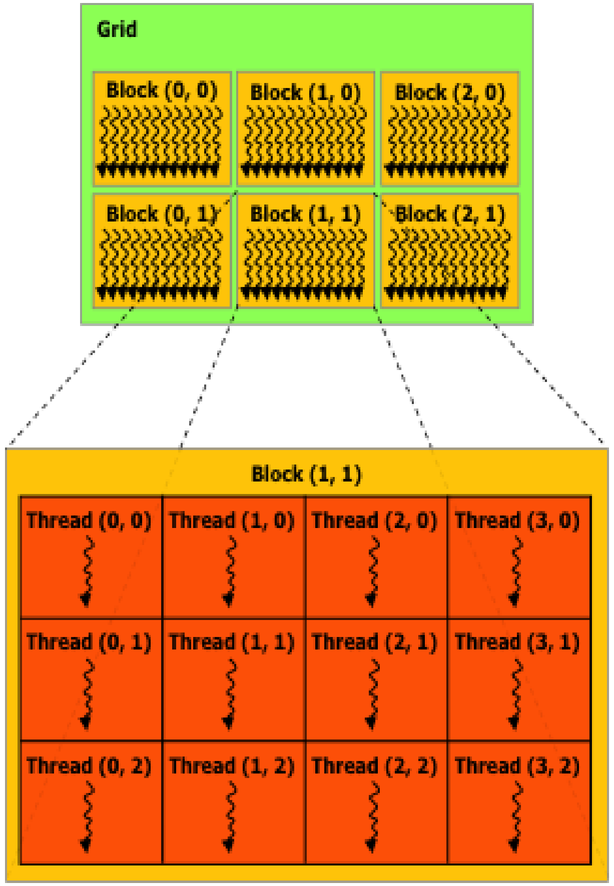
GPU



Execution configuration

- CUDA parallel execution model operates on parallel threads.
- The basic unit of work in the device is the thread, and every thread has its identity and registers.
- The number of threads that are possible to create depends on the hardware.
- Every thread must be part of a block, and a group of blocks belongs to a grid
- *The main objective of your execution configuration is the use of many threads working in parallel to execute the work with the best performance.*

- Therefore, the execution configuration defines the number of threads that will run the kernel, and the block configuration in one, two or three dimensions to generate the grid.



- All the threads have indices in order to compute memory addresses and make control decisions.
- The way to identify every thread in the configuration is through the **threadIdx** variable. This variable is a 3-component vector and it helps to identify the thread in one, two or three dimensions.
- In one dimension (*x*), the variable is **threadIdx.x**, in two dimensions (*x,y*) the variables are **threadIdx.x** and **threadIdx.y;** and in three dimensions the variables are **threadIdx.x, threadIdx.y** and **threadIdx.z.**
- This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

- Just like the threads, the blocks are organized into one, two or three dimensions to generate the grid.

- The block index is accessible through the blockIdx variable. You have to notice that the kernel will be copied as many times as the generated blocks.

- A kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

- For example, the kernel *add <<<1,10>>>* creates one block with 10 threads. Other configuration of the kernel is *add <<<10,10>>>*, where 10 blocks with 10 threads each one are created, and the kernel is copied to each block; finally *add <<<10,1>>>* with 10 blocks and just 1 thread.

- To know the dimension of the threads in a block, the blockDim variable is used. All these variables are the bult-in variables that CUDA runtime defines automatically.

- The variable gridDim indicates the number of blocks in the grid.

- For example:
  - index of the threads in x -> threadIdx.x
  - index of the blocks in x -> blockIdx.x
  - number of threads per block in x -> blockDim.x
  - number of blocks per grid in x -> gridDim.x
- The index of a thread and its thread ID relate to each other in a straightforward way:
  - For a one-dimensional block, they are the same
  - For a two-dimensional block of size (D_x, D_y),the thread ID of a thread of index (x, y) is $(x + y D_x)$
  - For a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + y D_x + z D_x D_y)$

Example

- 01add_kernel_bad: This program should add two vectors in the GPU (wrong way).

- 02add_kernel_good: This program should add two vectors in the GPU (right way).

- CPU allocates in the GPU

```
const int ARRAY_SIZE = 10;
const int ARRAY_BYTES = ARRAY_SIZE * sizeof(int);


int *h_a, *h_b, *h_result;
int *d_a, *d_b, *d_result;

//allocate memory on the host
h_a = (int*)malloc(ARRAY_BYTES);
h_b = (int*)malloc(ARRAY_BYTES);
h_result = (int*)malloc(ARRAY_BYTES);

//allocate memory on the device
cudaMalloc( (void**)&d_a, ARRAY_BYTES );
cudaMalloc( (void**)&d_b, ARRAY_BYTES );
cudaMalloc( (void**)&d_result, ARRAY_BYTES );
```

- CPU copies data from CPU to GPU

```
//copythe arrays 'a' and 'b' to the device
cudaMemcpy( d_a, h_a, ARRAY_BYTES, cudaMemcpyHostToDevice );
cudaMemcpy( d_b, h_b, ARRAY_BYTES, cudaMemcpyHostToDevice );
```

- CPU launches kernel(s) on the GPU to process the data

```
//run the kernel
addKernel<<<1,ARRAY_SIZE>>>( d_a, d_b, d_result);
```

- CPU copies back data from GPU to CPU

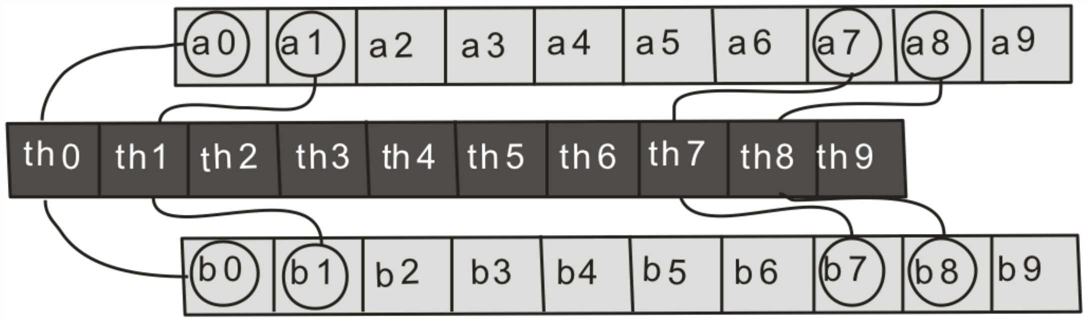
```
// copy the array 'result' back from the device to the CPU
cudaMemcpy( h_result, d_result, ARRAY_BYTES, cudaMemcpyDeviceToHost );
```

- CPU frees memory

```
// free device memory
cudaFree( d_a );
cudaFree( d_b );
cudaFree( d_result );
```

- Each thread takes one element of the array *a* and *b* to do the addition. The element that the thread takes from the arrays corresponds to the same index of the thread, so 10 threads are executing the same operation in parallel.

- As you can see in the next figure, the thread (*th*) takes one element with the same index of the thread of each array to execute the operation.



- Kernel

```
//kernel
__global__ void addKernel(  int  *d_a, int *d_b, int *d_result){
    int idx = threadIdx.x;
    d_result[idx] = d_a[idx] + d_b[idx];
}
```

Example

- 03add_kernelv2: This program must add two vectors of size 100 in the GPU using 10 blocks and 10 threads per block.

• Changing execution configuration

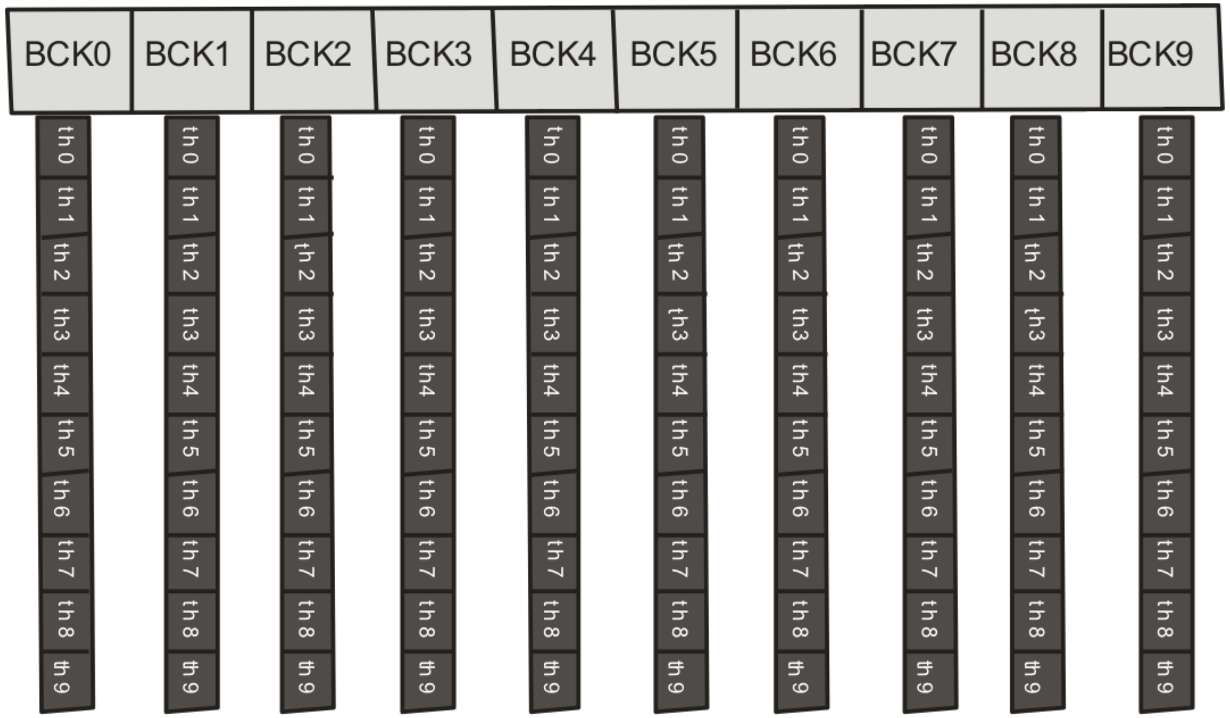
- We are going to increase 10 times the data to create 10 blocks with 10 threads each one (<<<*10,10*>>>), but it is necessary to change the kernel a little bit.

- The thread's index is calculated using the block number, and the total number of blocks in the grid (*int tid = threadIdx.x + blockIdx.x * blockDim.x;*).

- Each block executes the same kernel, but with different data, they take just 10 elements of the arrays depending of the *tid* calculated.

- Grid organization

  - Like in the last example, each thread takes one element of the different arrays to execute the operation.



- Calculating the indexes

  - For example, the 10 indexes in block 5 (*blockIdx.x=5*) and block 9 (*blockIdx.x=9*) are:

  - int idx= threadIdx.x + blockIdx.x * blockDim.x;

  - Every block has the same code of the kernel, but they are going to compute different data.

BLOCK 5	BLOCK 9
tid= 0 + 5* 10 = 50	tid= 0 + 9* 10 = 90
tid= 1 + 5* 10 = 51	tid= 1 + 9* 10 = 91
tid= 2 + 5* 10 = 52	tid= 2 + 9* 10 = 92
tid= 3 + 5* 10 = 53	tid= 3 + 9* 10 = 93
tid= 4 + 5* 10 = 54	tid= 4 + 9* 10 = 94
tid= 5 + 5* 10 = 55	tid= 5 + 9* 10 = 95
tid= 6 + 5* 10 = 56	tid= 6 + 9* 10 = 96
tid= 7 + 5* 10 = 57	tid= 7 + 9* 10 = 97
tid= 8 + 5* 10 = 58	tid= 8 + 9* 10 = 98
tid= 9 + 5* 10 = 59	tid= 9 + 9* 10 = 99

- New kernel

```
//kernel
__global__ void addKernel(  int  *d_a, int *d_b, int *d_result){
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d_result[idx] = d_a[idx] + d_b[idx];
}
```

```
const int ARRAY_SIZE = 100;
```

- Kernel

```
//run the kernel
addKernel<<<10,ARRAY_SIZE>>>( d_a, d_b, d_result);
```

Practice

- 04add_kernelv3: This program must add two vectors in the GPU using 10 blocks and 5 threads per block, with the same 100 elements per array.