Practice 3: Thread Execution in 2D/3D

Objective: To understand organizing thread execution in two or three dimensions.

CUDA Thread Execution:

- Threads in CUDA execute the same kernel (a device function).
- Threads rely on 1D/2D/3D coordinates to distinguish themselves from one another and identify the appropriate portion of data to process.
- Threads are organized into two hierarchical levels:
    - *Grid:* A three-dimensional array of blocks. All threads in a block share the *same* block index (i.e., they possess the same value of the "blockIdx" variable).
    - *Blocks:* A three-dimensional array of threads. Each thread has a unique thread index (the thread index is accessed via the "threadIdx" variable)
- When launching a kernel, we need to specify the size of grid and blocks in each dimension.
- However, we can use fewer than three dimensions (the size of unused dimensions are set to 1 by default).
- The exact organization of grid is determined by the execution configuration parameters (within <<<,>>>) of the kernel launch statement.
- The first parameter specifies the dimensions of the grid in the number of blocks.
- The second parameter specifies the dimensions of each block in the number of threads.
- Each parameter is of the *dim3* type with three unsigned integer fields: x,y,z.

Examples:

- kernel<<<1,64>>>();
  is equivalent to
  kernel<<<dim3(1,1,1), dim3(64,1, 1)>>>();

- The execution configuration parameters below specify 1D grid consisting of 32 1D blocks, each of which consists of 128 threads:

```
dim3 dimGrid(32, 1, 1);
dim3 dimBlock(128, 1, 1);
kernel<<<dimGird, dimBlock>>>();
```

- The execution configuration parameters below specify 1D grid that consists of 1 2D block, each of which consists of 20*20 threads:

```
dim3 dimGrid(1, 1, 1);
dim3 dimBlock(20, 20, 1);
kernel<<<dimGird, dimBlock>>>();
```

Accessing Thread Information via Predefined Variables:

- *threadIdx*: The variable to access information about thread ID.
    - *threadIdx.x* **:** The variable to access thread ID in x-coordinate.
    - *threadIdx.y* **:** The variable to access thread ID in y-coordinate.
    - *threadIdx.z* **:** The variable to access thread ID in z-coordinate.
- *blockIdx:* The variable to access information about block ID.
    - *blockIdx.x* **:** The variable to access block ID in x-coordinate

    - *blockIdx.y* **:** The variable to access block ID in y-coordinate
    - *blockIdx.z* **:** The variable to access block ID in z-coordinate
- *blockDim*: The variable to access information about the number of threads in each block.
    - *blockDim.x* : The variable to access the number of threads in each block in x-coordinate.
    - *blockDim.y* : The variable to access the number of threads in each block in y-coordinate.
    - *blockDim.z* : The variable to access the number of threads in each block in z-coordinate.
- grid*Dim*: The variable to access information about the number of blocks in grid.
    - *gridDim.x*: The variable to access the number of blocks in x-coordinate.
    - *gridDim.y*: The variable to access the number of blocks in y-coordinate.
    - *gridDim.z*: The variable to access the number of blocks in z-coordinate.

Practice 3.1: Write a traditional C program for computing matrix addition. The input to your program should be two 20-*by-20* integer matrices X and Y. Your program should output a 20-*by-20* integer matrix Z such that Z = X + Y.

Practice 3.2: Convert your traditional C program in 3.1 into a CUDA C program. Adjust your code so that it works with the execution configuration parameters below. Note that each thread in your program should be assigned to compute the value of each entry just once.

```
dim3 dimGrid(10, 10, 1);
dim3 dimBlock(2, 2, 1);
kernel<<<dimGird, dimBlock>>>();
```

Practice 3.2: Adjust your codes so your CUDA C program in 3.2 so that it work with n-*by-n* input matrices, for any positive integer *n*. Note that you are allowed to modify the execution configuration parameter of the kernel launch to obtain the best performance.