CUDA Programming

Recap

1. Write a CUDA program to initialize an array of size 32 to all zeros in parallel.
2. Change the array size to 1024.
3. Create another kernel that adds *i* to *array[i]*.
4. Change the array size to 8000.
5. Check if answer to problem 3 still works.

Thread Organization

- A kernel is launched as a grid of threads.
- A grid is a 3D array of thread-blocks (gridDim.x, gridDim.y and gridDim.z).
  - Thus, each block has blockIdx.x, .y, .z.
- A thread-block is a 3D array of threads (blockDim.x, .y, .z).
  - Thus, each thread has threadIdx.x, .y, .z.

Grids, Blocks, Threads

Each thread uses IDs to decide what data to work on.

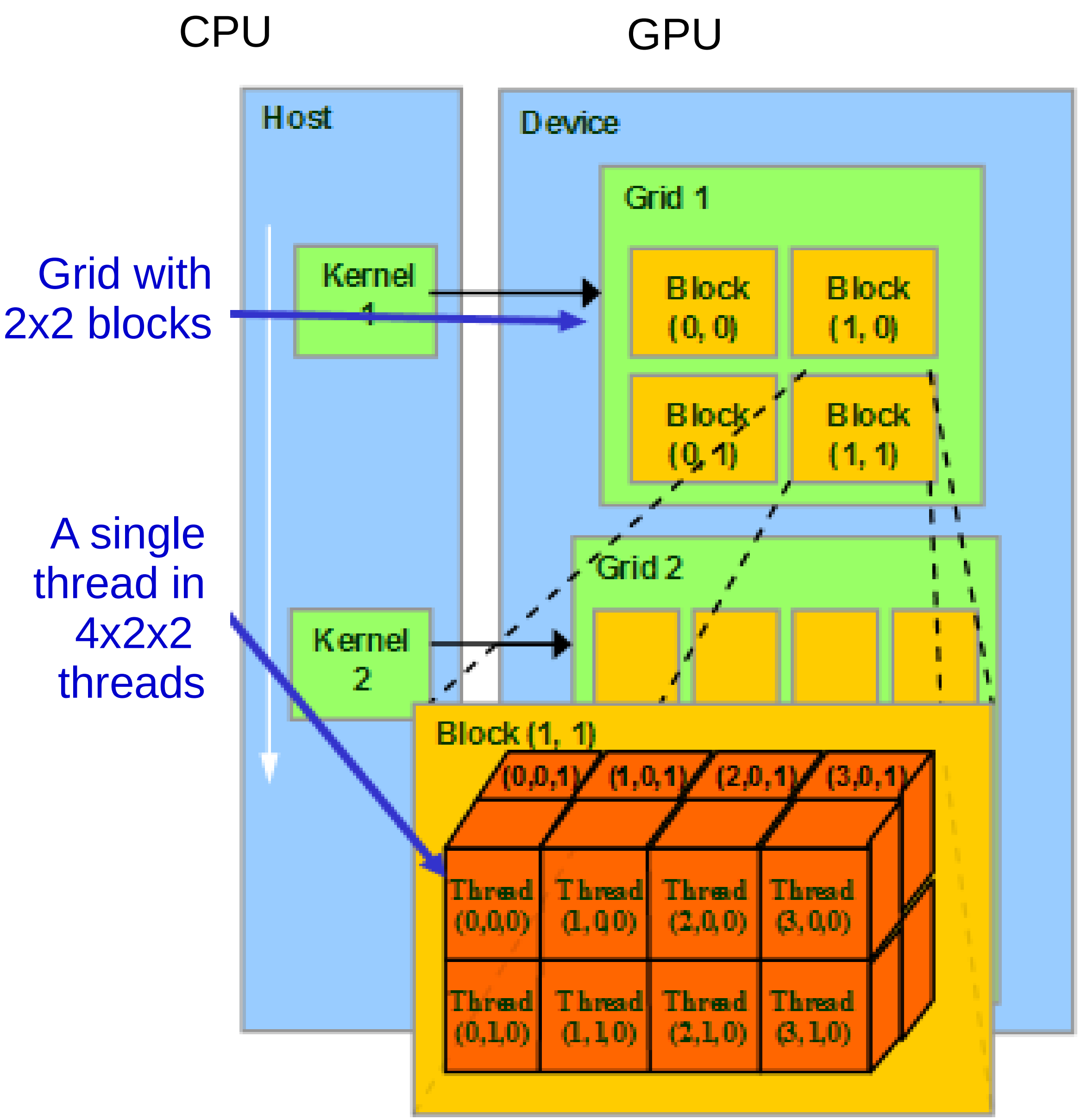
- Block ID: 1D, 2D, or 3D
- Thread ID: 1D, 2D, or 3D

Simplifies memory addressing when processing multi-dimensional data

- Image processing
- Solving PDEs on volumes
- …

Typical configuration:

- 1-5 blocks per SM
- 128-1024 threads per block.
- Total 2K-100K threads.
- You can launch a kernel with millions of threads.



CPU

GPU

Grid with 2x2 blocks

A single thread in 4x2x2 threads

Accessing Dimensions

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
        if (threadIdx.x == 0 && blockIdx.x == 0 &&
            threadIdx.y == 0 && blockIdx.y == 0 &&
            threadIdx.z == 0 && blockIdx.z == 0) {
                printf("%d %d %d %d %d %d.\n", gridDim.x, gridDim.y, gridDim.z,
                                                        blockDim.x, blockDim.y, blockDim.z);
        }
}
int main() {
        dim3 grid(2, 3, 4);
        dim3 block(5, 6, 7);
        dkernel<<<grid, block>>>();
        cudaDeviceSynchronize();
        return 0;
}
```

How many times the kernel printf gets executed when the *if* condition is changed to *if (threadIdx.x == 0)* ?

Number of threads launched = 2 * 3 * 4 * 5 * 6 * 7.
Number of threads in a thread-block = 5 * 6 * 7.
Number of thread-blocks in the grid = 2 * 3 * 4.

ThreadId in x dimension is in [0..5).
BlockId in y dimension is in [0..3).

2D

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *matrix) {
        unsigned id = threadIdx.x * blockDim.y + threadIdx.y;
        matrix[id] = id;
}
#define N       5
#define M       6

int main() {
        dim3 block(N, M, 1);
        unsigned *matrix, *hmatrix;

        cudaMalloc(&matrix, N * M * sizeof(unsigned));
        hmatrix = (unsigned *)malloc(N * M * sizeof(unsigned));

        dkernel<<<1, block>>>(matrix);
        cudaMemcpy(hmatrix, matrix, N * M * sizeof(unsigned), cudaMemcpyDeviceToHost);

        for (unsigned ii = 0; ii < N; ++ii) {
                for (unsigned jj = 0; jj < M; ++jj) {
                        printf("%2d ", hmatrix[ii * M + jj]);
                }
                printf("\n");
        }
        return 0;
}
```

What is the output of this program?

```
$ a.out
 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
```

2D

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *matrix) {
        unsigned id = threadIdx.y * blockDim.x + threadIdx.x;
        matrix[id] = id;
}
#define N       5
#define M       6

int main() {
        dim3 block(N, M, 1);
        unsigned *matrix, *hmatrix;

        cudaMalloc(&matrix, N * M * sizeof(unsigned));
        hmatrix = (unsigned *)malloc(N * M * sizeof(unsigned));

        dkernel<<<1, block>>>(matrix);
        cudaMemcpy(hmatrix, matrix, N * M * sizeof(unsigned), cudaMemcpyDeviceToHost);

        for (unsigned ii = 0; ii < N; ++ii) {
                for (unsigned jj = 0; jj < M; ++jj) {
                        printf("%2d ", hmatrix[ii * M + jj]);
                }
                printf("\n");
        }
        return 0;
}
```

What is the output of this program?

```
$ a.out
 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
```

Write the kernel to initialize the matrix to unique ids.

1D

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *matrix) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        matrix[id] = id;
}
#define N       5
#define M       6
int main() {
        unsigned *matrix, *hmatrix;

        cudaMalloc(&matrix, N * M * sizeof(unsigned));
        hmatrix = (unsigned *)malloc(N * M * sizeof(unsigned));

        dkernel<<<N, M>>>(matrix);
        cudaMemcpy(hmatrix, matrix, N * M * sizeof(unsigned), cudaMemcpyDeviceToHost);

        for (unsigned ii = 0; ii < N; ++ii) {
                for (unsigned jj = 0; jj < M; ++jj) {
                        printf("%2d ", hmatrix[ii * M + jj]);
                }
                printf("\n");
        }
    return 0;
}
```

Takeaway

One can perform computation on multi-dimensional data using a one-dimensional block.

If I want the launch configuration to be <<<2, X>>>, what is X?
The rest of the code should be intact.