CUDA Programming

Hello Program

Outline

- ❑CUDA Programming
  - ❑Functions Qualifiers
  - ❑Built-in Device Variables
  - ❑Variable Qualifiers
- ❑Addition on the device
  - ❑Moving to parallel using blocks
  - ❑Moving to parallel using threads
  - ❑Combining blocks and threads

Cuda Programming

- **Kernels are C functions with some restrictions**
  - **Can only access GPU memory**
  - **Must have void return type**
  - **No variable number of arguments ("varargs")**
  - **Not recursive**
  - **No static variables** → local variables, تبقى محفوظة حتى بعد الانتهاء (لا تنحذف)
- **Function arguments automatically copied from CPU to GPU memory**

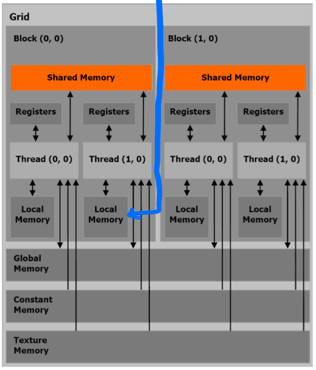
Function Qualifiers

- __ global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

Variable Qualifiers (GPU code)

ex: __shared__ int x;

- __device__
  - Stored in device memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- __shared__
  - Stored in on-chip shared memory (very low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: kernel execution
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - Arrays of more than 4 elements stored in device memory



gridDim.x=3
gridDim.y=2
gridDim.z=1

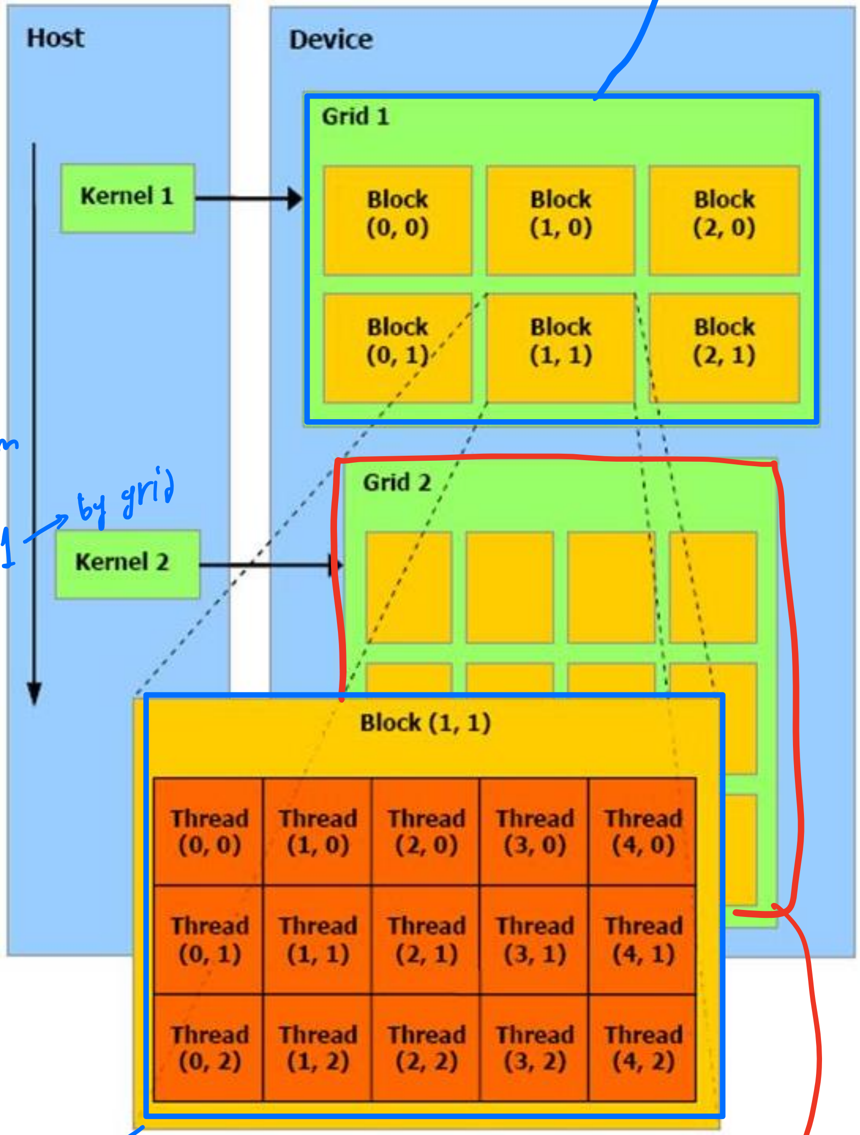
CUDA Built-in Device Variables

All __global__ and __device__ functions have access to these automatically defined variables

- dim3 **gridDim**;
  - Dimensions of the grid in blocks (at most 2D)
- dim3 **blockDim**;
  - Dimensions of the block in threads
- dim3 **blockIdx**;
  - Block index within the grid
- dim3 **threadIdx**;
  - Thread index within the block

data type

gridDim.x => colum
gridDim.y => row
gridDim.x => layer=1 → by grid



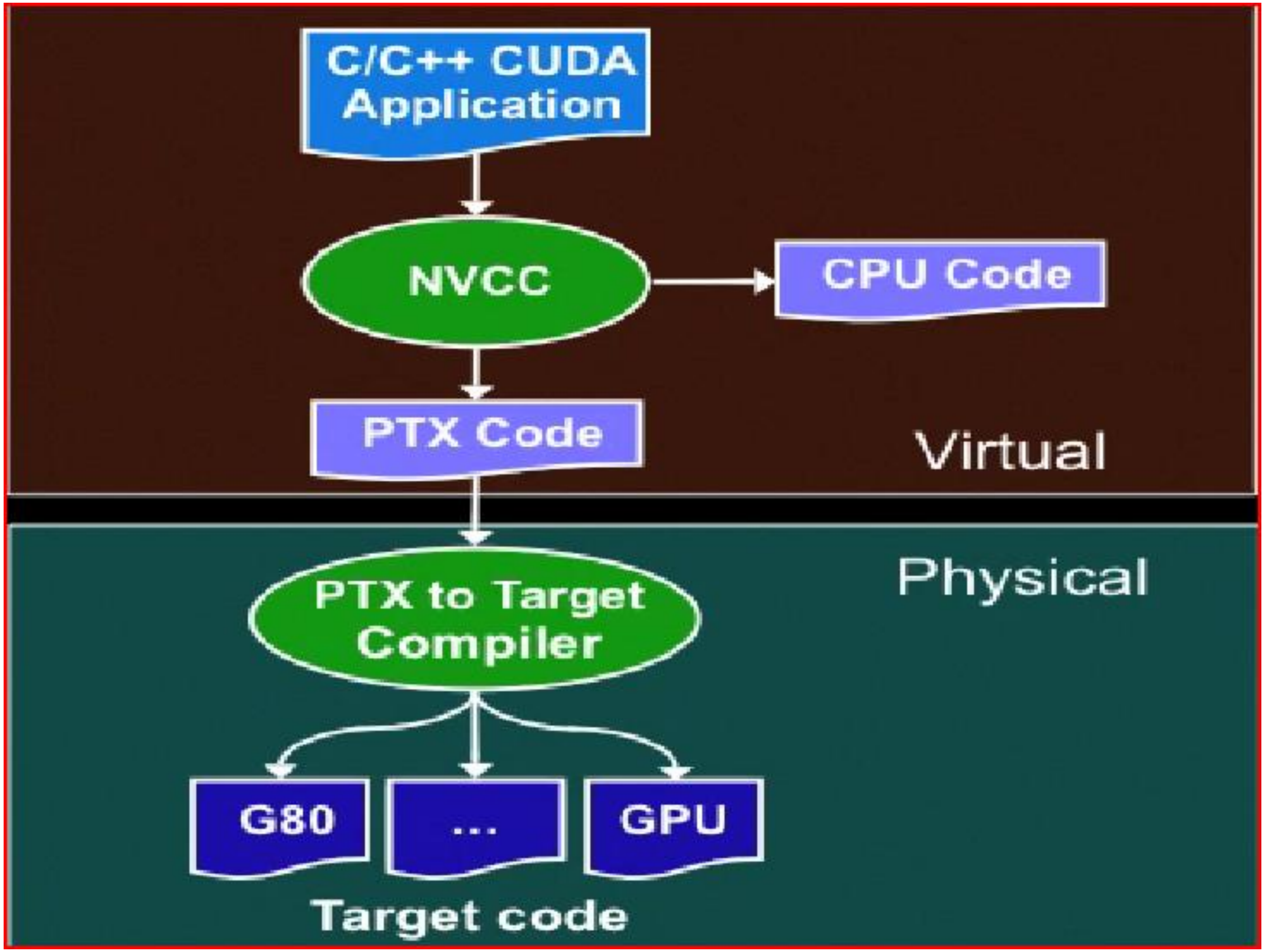
© NVIDIA Corporation

blockDim.x=5

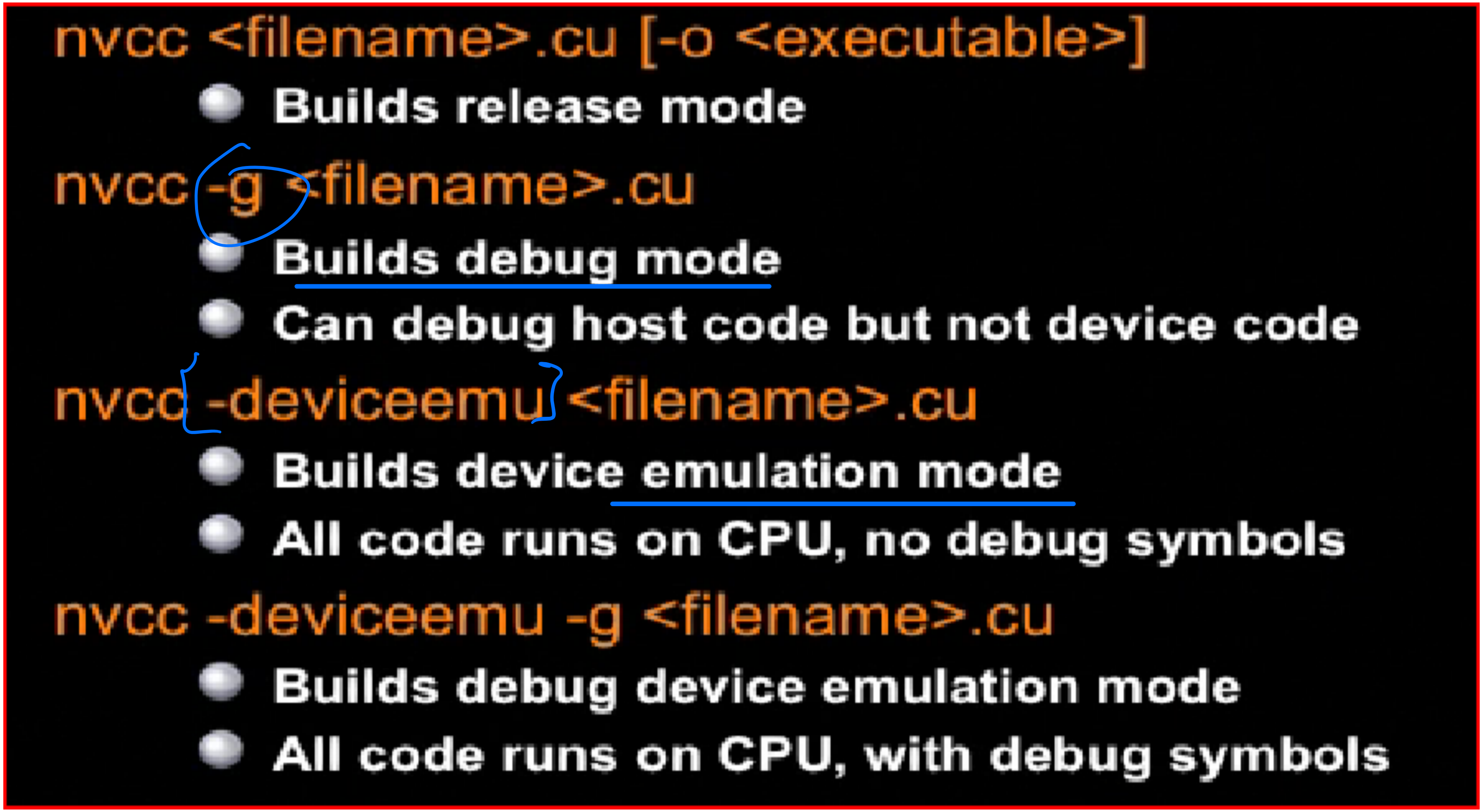
blackDim.y=3
blackDim.z=1

gridDim.x=4
gridDim.y=3
gridDim.x=1

CUDA Compile



CUDA Compile



```
nvcc <filename>.cu [-o <executable>]
```

- Builds release mode

```
nvcc -g <filename>.cu
```

- Builds debug mode
- Can debug host code but not device code

```
nvcc -deviceemu <filename>.cu
```

- Builds device emulation mode
- All code runs on CPU, no debug symbols

```
nvcc -deviceemu -g <filename>.cu
```

- Builds debug device emulation mode
- All code runs on CPU, with debug symbols

Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C that runs on the host

- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc
hello_world.cu
$ a.out
Hello World!
$
```

out put

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
      mykernel<<<1,1>>>();
      printf("Hello World!\n");
      return 0;
}
```

- Two new syntactic elements…

Hello World! with Device COde

```
    mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!

Hello World! with Device Code

```
__global__ void mykernel(void){
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

cpu
print it

- `mykernel()` does nothing

Hello World! with Device Code

```
__global__ void mykernel(void){
        printf("Hello World!\n");
}

int main(void) {
        mykernel<<<1,1>>>();
        return 0;
}
```

Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

GPU
print it

Hello World! with Device Code

```
__global__ void mykernel(void){
        printf("Hello World!\n");
}

int main(void) {
        mykernel<<<2,2>>>();
        return 0;
}
```

2 block

2 thread

GPU

Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
Hello World!
Hello World!
Hello World!
$
```