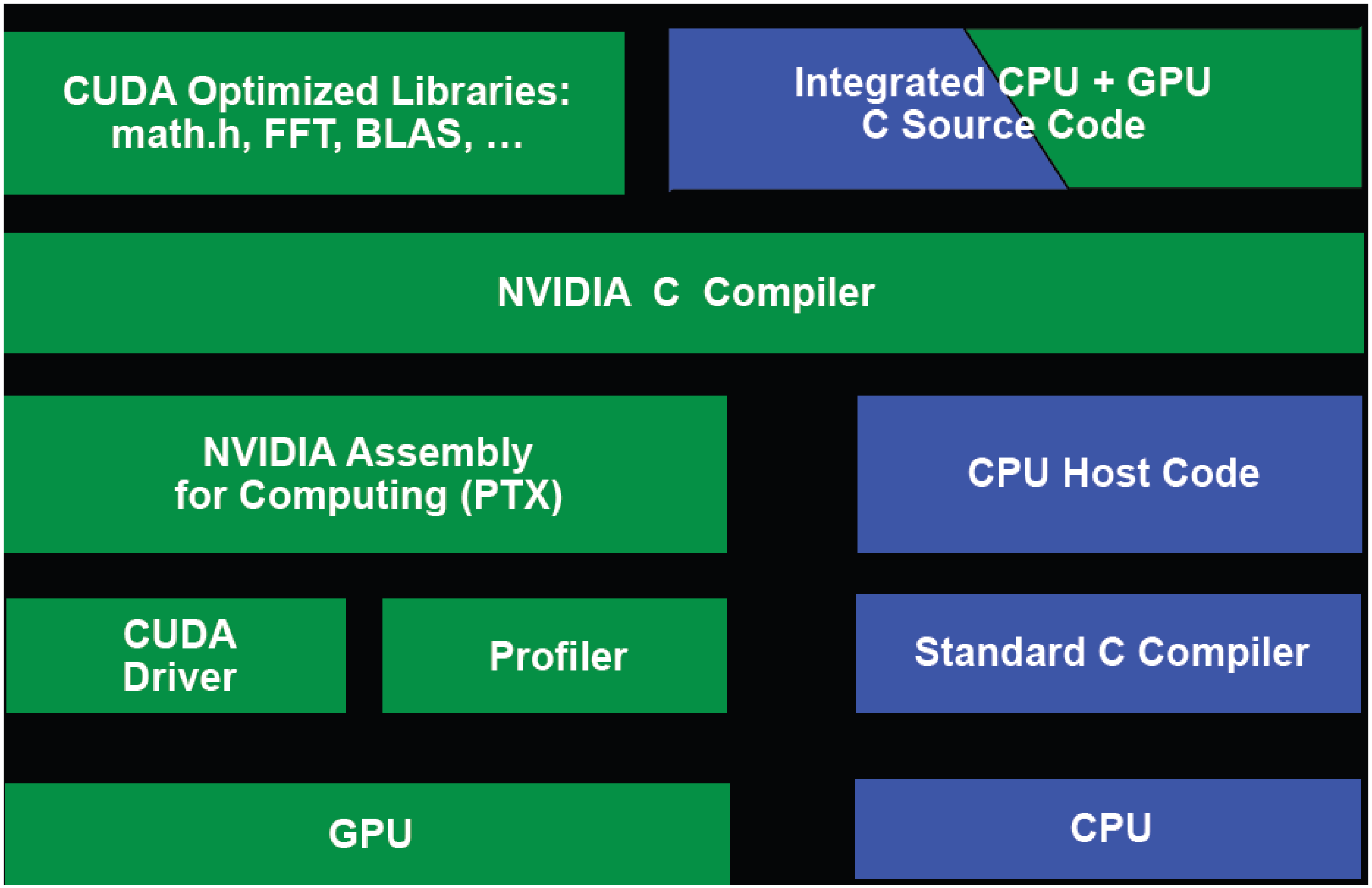
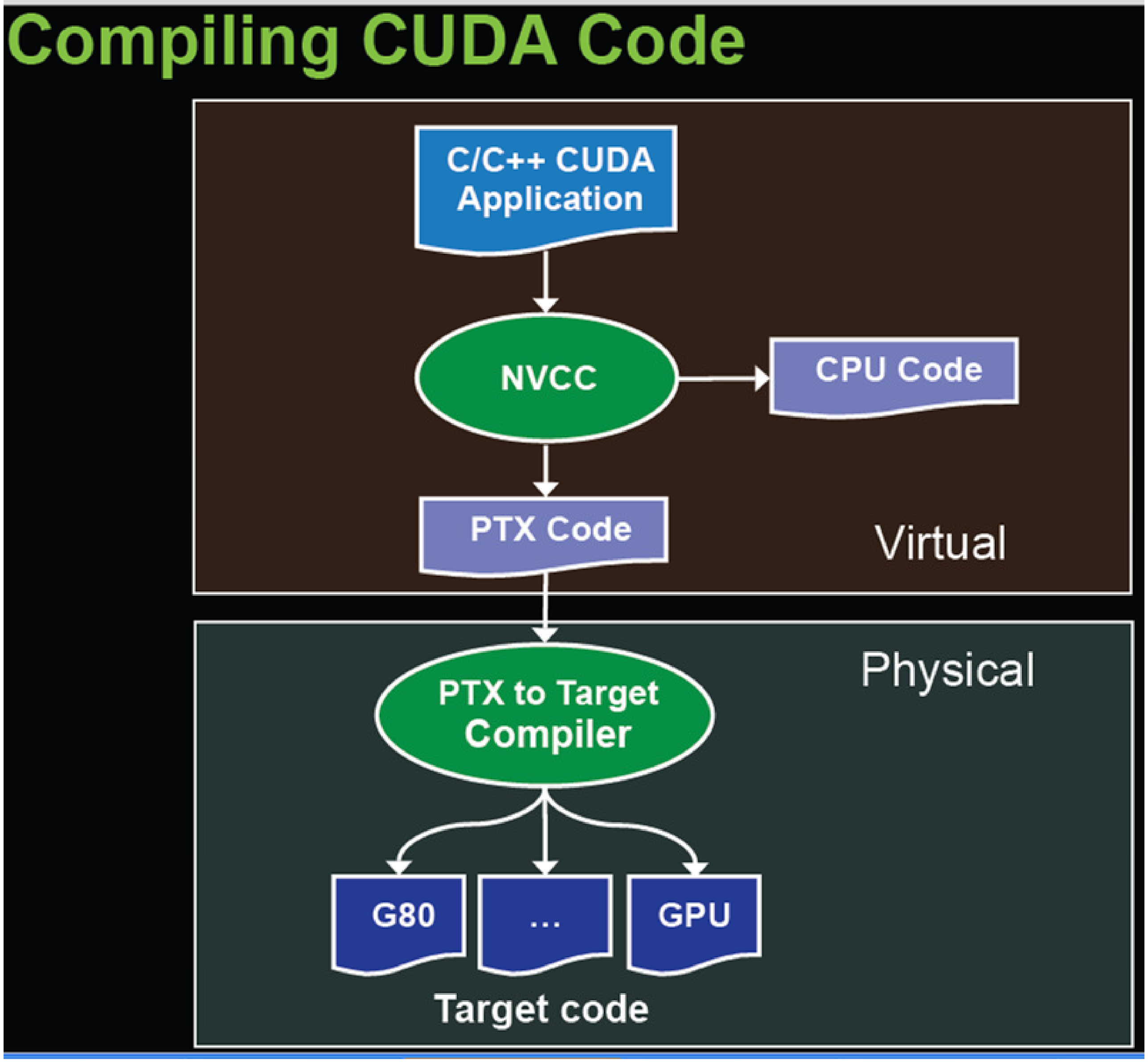
CUDA Programming

Many slides adapted from the slides of Hwu & Kirk at UIUC; and NVIDIA CUDA tutorials

CUDA Software Development

- Is done on the host (CPU)
  - programming environment, compilers and libraries
  - Profiler, emulator





Source code is on CPU

It can be mixed, with parts meant for the CPU and other parts for the GPU

NVCC separates the CPU code and passes it to the system compiler (Visual studio or gcc)

CPU environment is set up to call appropriate GPU libraries

GPU code is compiled to a GPU assembler

PTX is then compiled to the device
Can also be compiled to a CPU emulator/CPU debug emulator

Extensions to C

- **Declspecs**
  - **global, device, shared, local, constant**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...
```

- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**

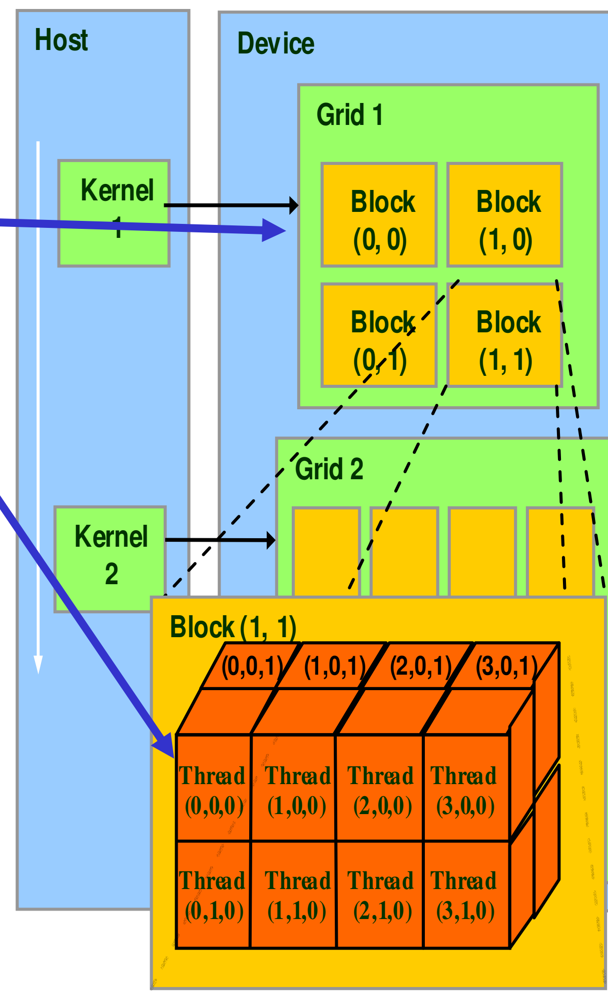
```
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)
```

- **Function launch**

```
// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

GPU Memory Allocation / Release

- **cudaMalloc(void ** pointer, size_t nbytes)**
- **cudaMemset(void * pointer, int value, size_t count)**
- **cudaFree(void* pointer)**

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d, nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

Data Copies

- **cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);**
  - direction specifies locations (host or device) of src and dst
  - Blocks CPU thread: returns after the copy is complete
  - Doesn’t start copying until previous CUDA calls complete
- **enum cudaMemcpyKind**
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

Data Movement Example

Host variables – h

Device variables – d

Allocate and get pointer on host and device

Copy the data from host to device (notice the order of arguments)
From device-to-device
from device-to-host

Free

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```

Cuda Kernels

- **Kernels are C functions with some restrictions**
  - **Cannot access host memory**
  - **Must have void return type**
  - **No variable number of arguments (“varargs”)**
  - **Not recursive**
  - **No static variables**
- **Function arguments automatically copied from host to device**

Function Qualifiers

- Kernels designated by function qualifier:
  __global__
  - Function called from host and executed on device
  - Must return void
- Other CUDA function qualifiers
  __device__
  - Function called from device and run on device
  - Cannot be called from host code
  __host__
  - Function called from host and executed on host (default)
- __host__ and __device__ qualifiers can be combined to generate both CPU and GPU code

CUDA Built-in Device Variables

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

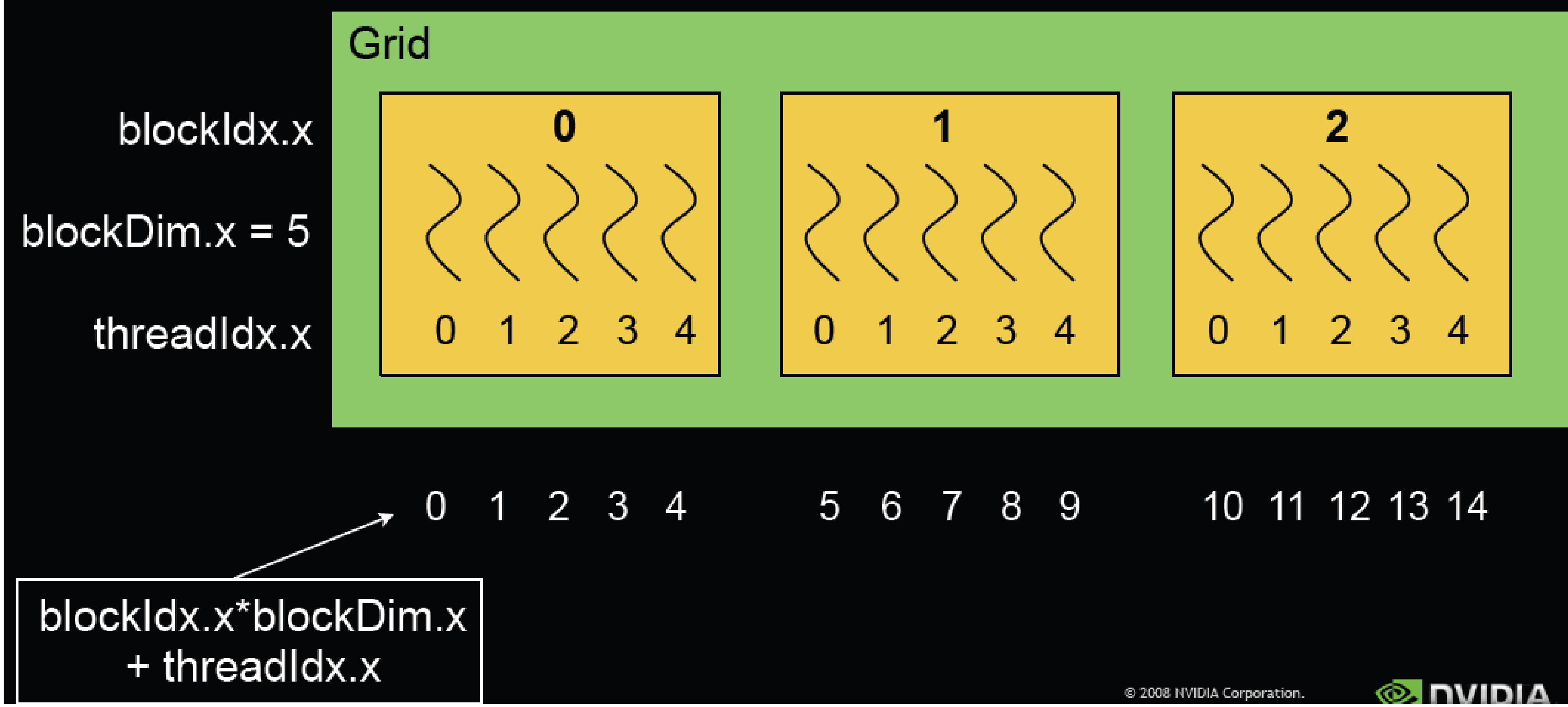
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

Calling a kernel function

- **kernel<<<dim3 dG, dim3 dB>>>(…)**
  - **Execution Configuration ("<<< >>>")**
  - **dG - dimension and size of grid in blocks**
    - **Two-dimensional: x and y**
    - **Blocks launched in the grid: dG.x * dG.y**
  - **dB - dimension and size of blocks in threads:**
    - **Three-dimensional: x, y, and z**
  - **Threads per block: dB.x * dB.y * dB.z**
- **Unspecified dim3 fields initialize to 1**

Unique Thread ID

- **Built-in variables are used to determine unique thread IDs**
  - **Map from local thread ID (threadIdx) to a global ID which can be used as array indices**



Host synchronization

- **All kernel launches are asynchronous**
  - **control returns to CPU immediately**
- **cudaMemcpy() is synchronous**
  - **control returns to CPU after copy completes**
  - **copy starts after all previous CUDA calls have completed**
- **cudaThreadSynchronize()**
  - **blocks until all previous CUDA calls complete**

Host Sync example

- **// copy data from host to device**

cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);

- **// execute the kernel**

inc_gpu<<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);

// run independent CPU code

run_cpu_stuff();

// copy data from device back to host

cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);