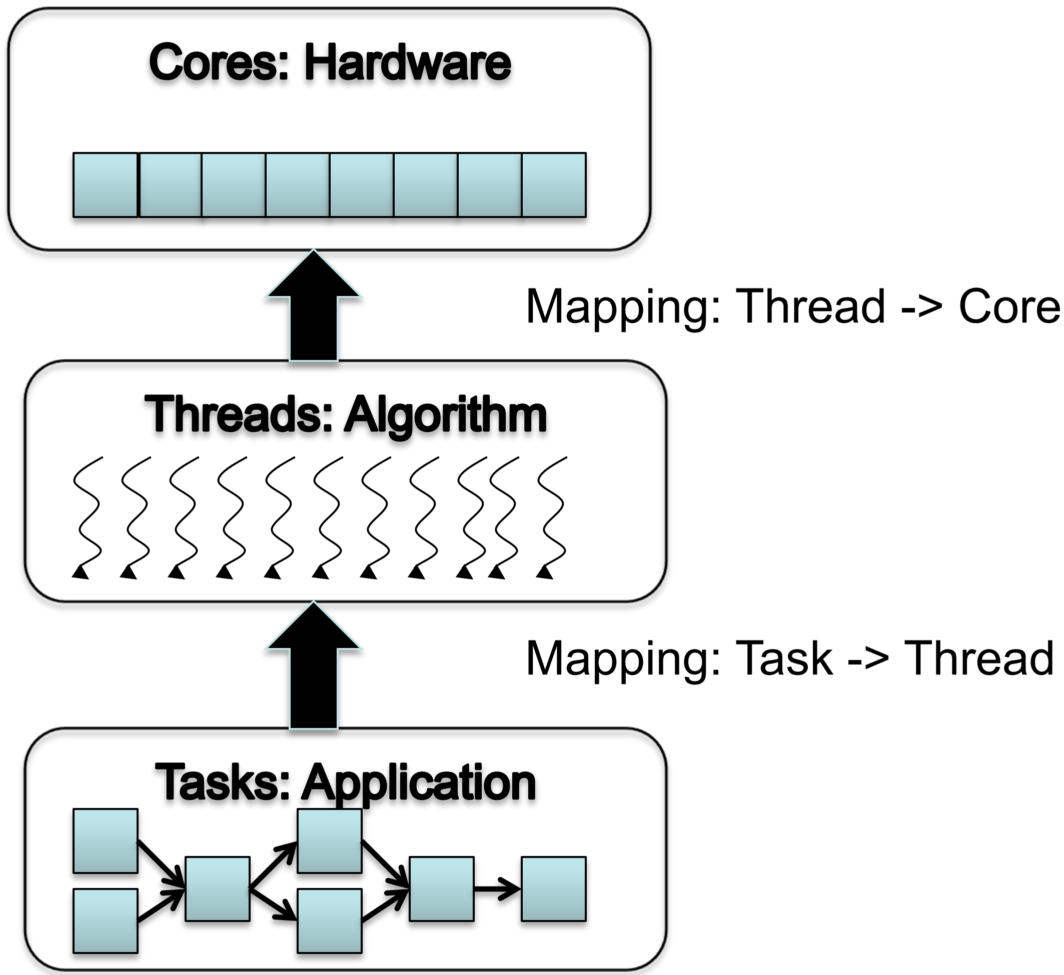
CUDA Programming Model

Parallel Algorithm Design Concepts



CUDA "Compute Unified Device Architecture"

- **General purpose parallel programming model**
  - Support "Zillions" of threads
- **Much easier to use**
  - C language, No shaders, No Graphics APIs
  - Shallow learning curve: tutorials, sample projects, forum
- **Key features**
  - Simple management of threads
  - Simple execution model
  - Simple synchronization
  - Simple communication

Goal:
Focus on parallel algorithms (kernels), rather than parallel management

CUDA "Compute Unified Device Architecture"

What we get?

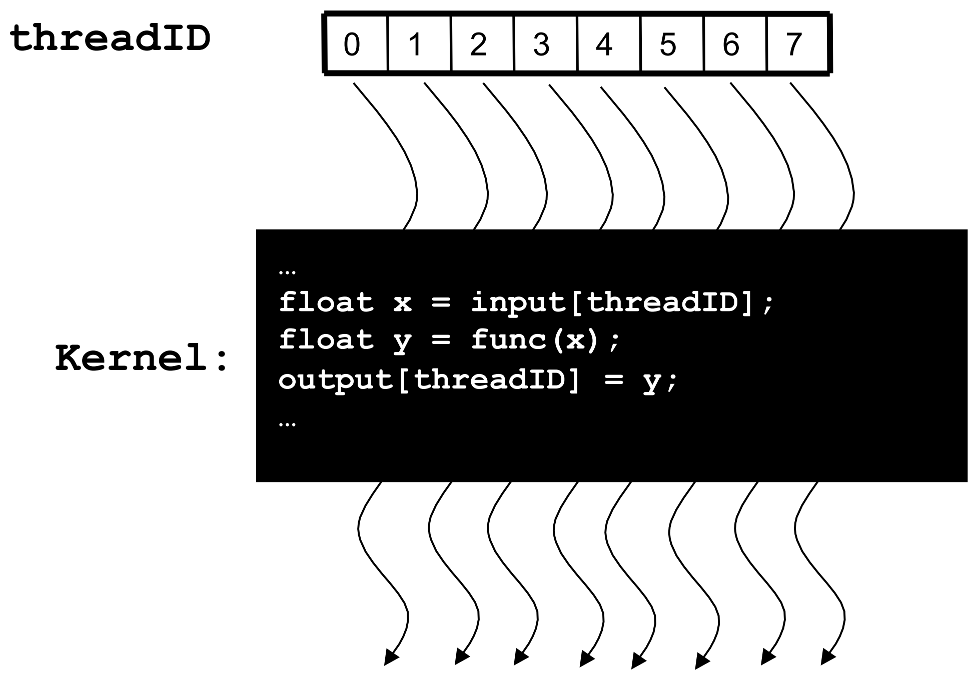
- Not enough controls
  - Only handle data-parallel application well
    - Easy to program
    - High performance
  - Not easy for some other applications (Large data dependency between threads)
- Easier than before, but not a fully general parallel programming model

CUDA Programming Model

- **Executing *kernel* functions within *threads***
- **Threads organization**
  - Blocks and Grids
- **Hardware mapping of threads**
  - Computation-to-core mapping
    - Thread -> Core
    - Thread blocks -> Multi-processors

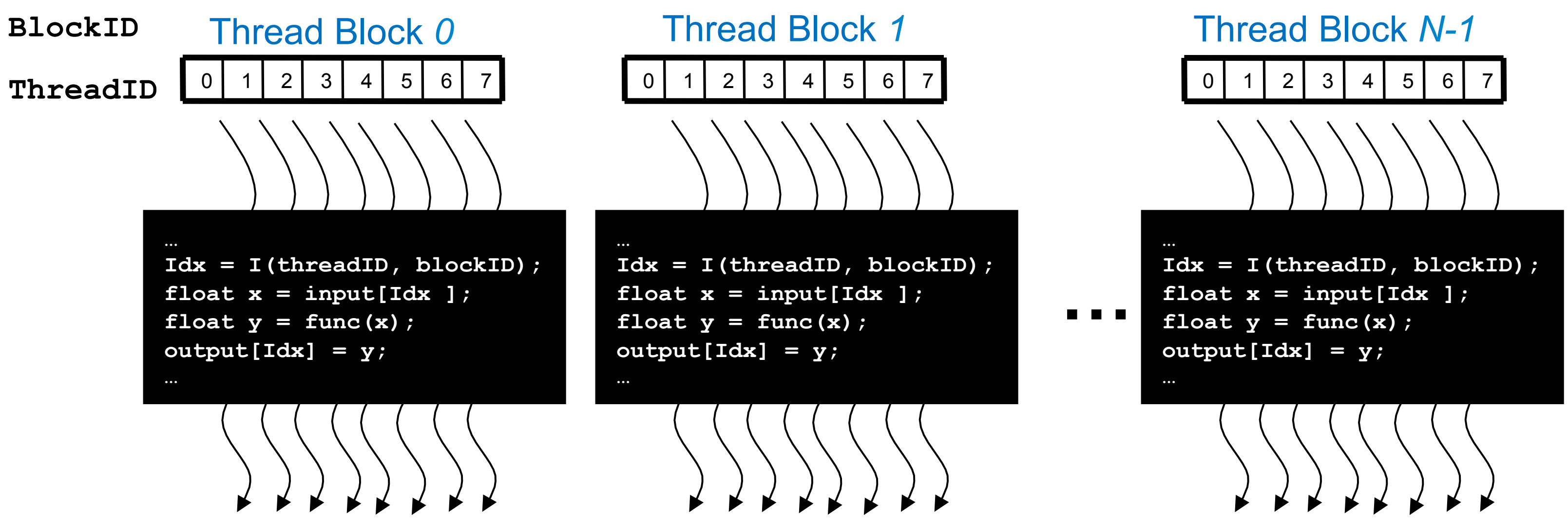
CUDA Threads and Functional Kernels

- **Many *threads* are executing a single *kernel* function**
  - Same Code (SIMD)
  - Different Data (using Thread ID)



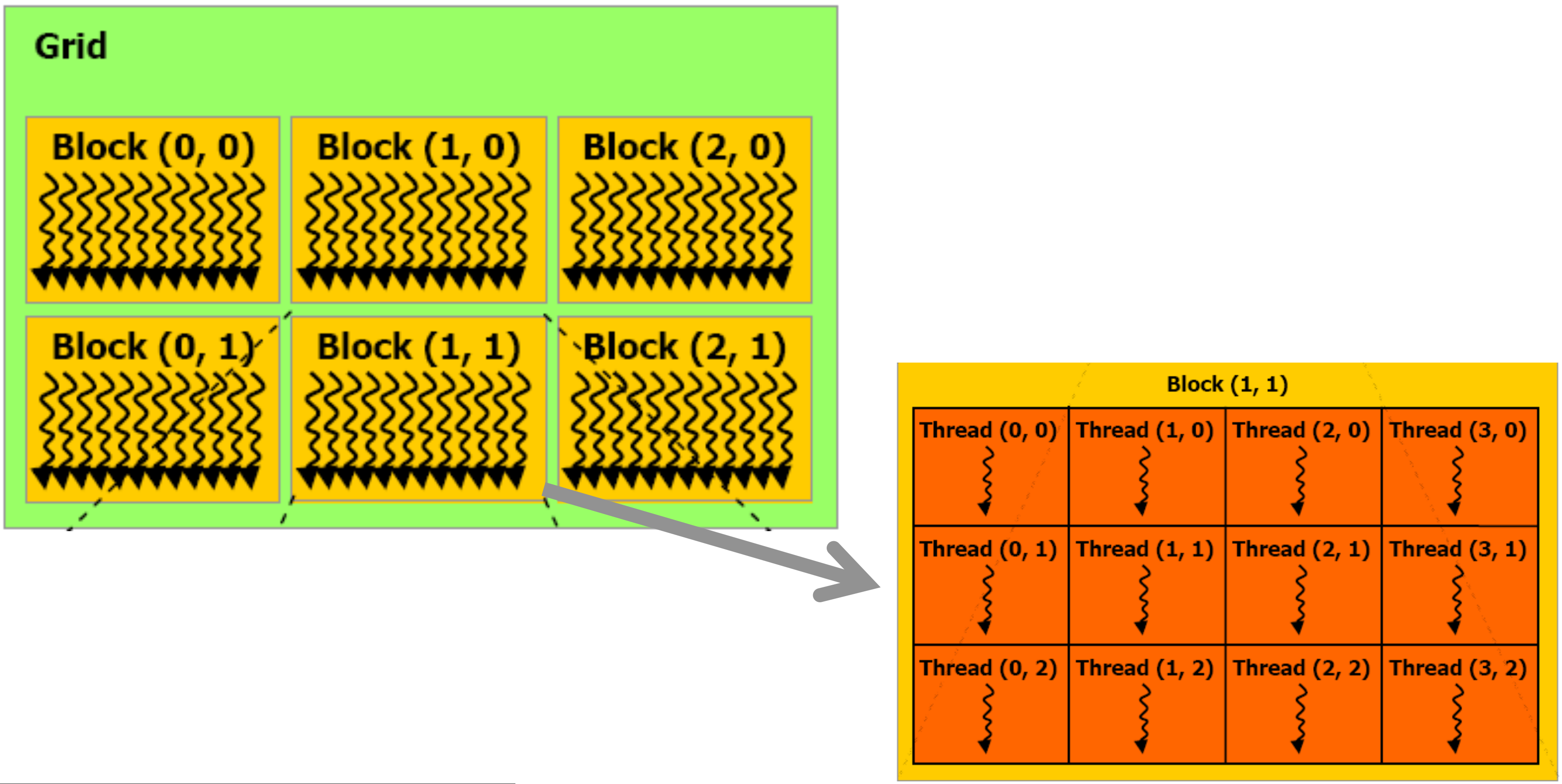
Thread Blocks

- Threads are grouped into multiple ***blocks***



Grid

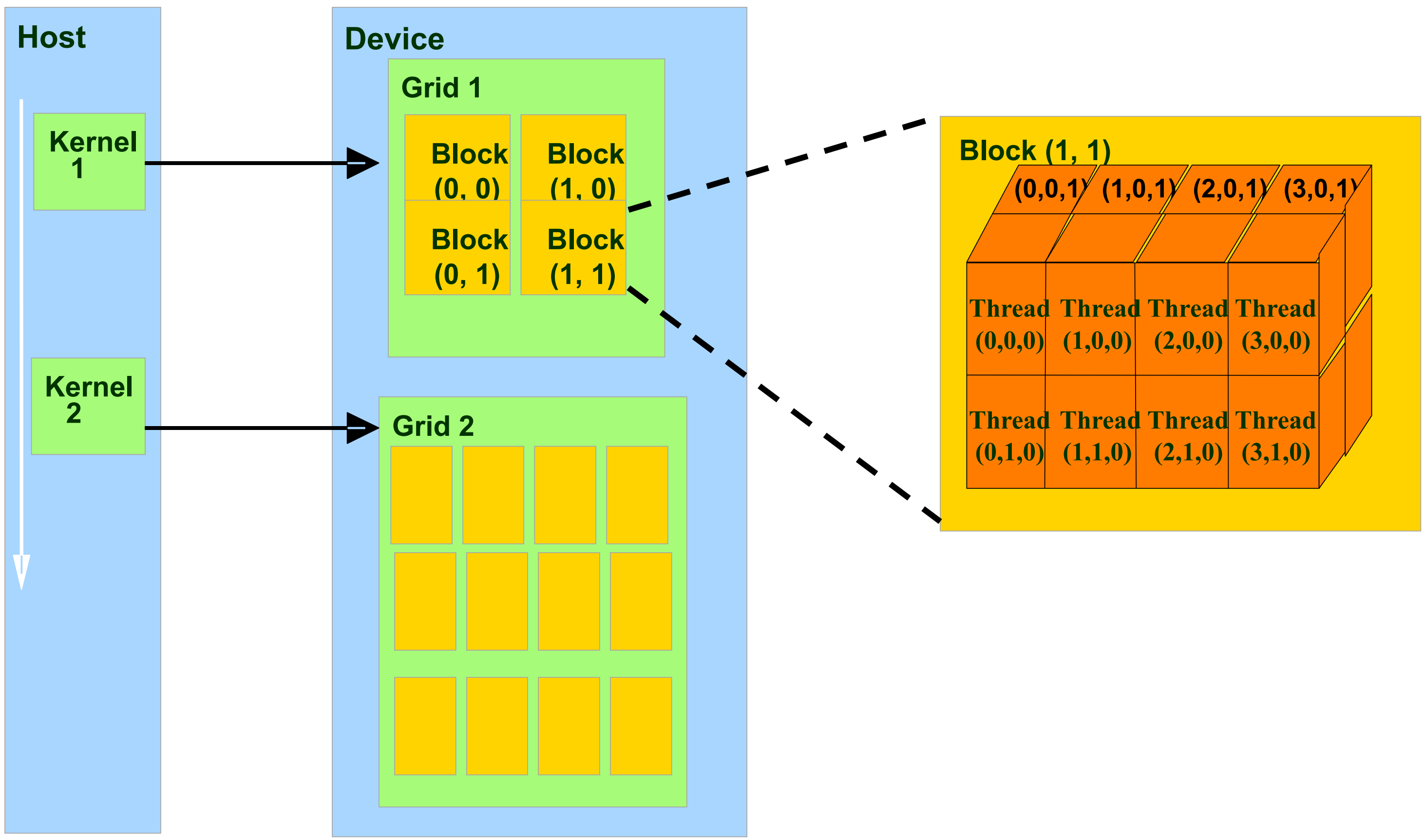
- A number of blocks are grouped into ***Grid***.



Thread organization Overview

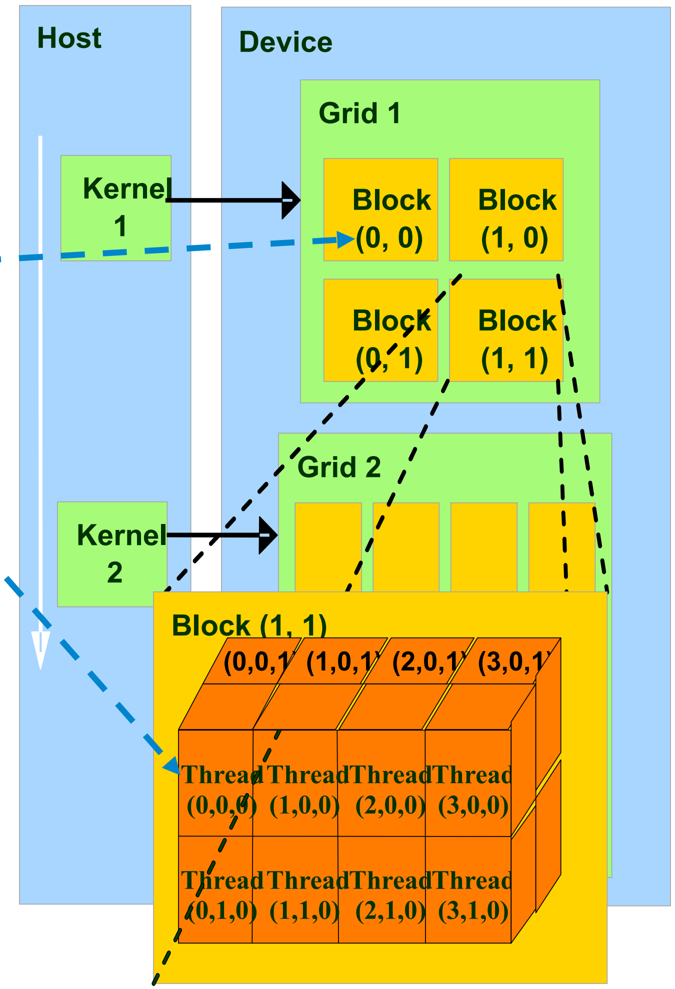
- An array of threads -> block
- An array of blocks -> grid
- All threads in one grid execute the same kernel
- Grids are executed sequentially.

Thread organization Overview

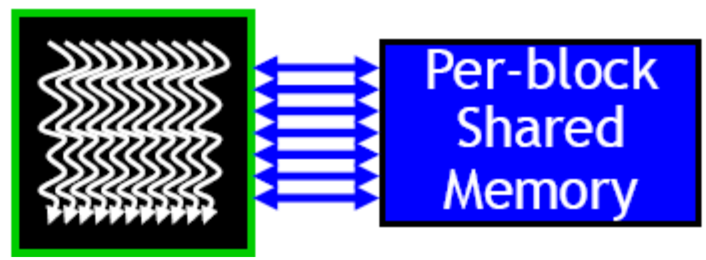


Thread Identification

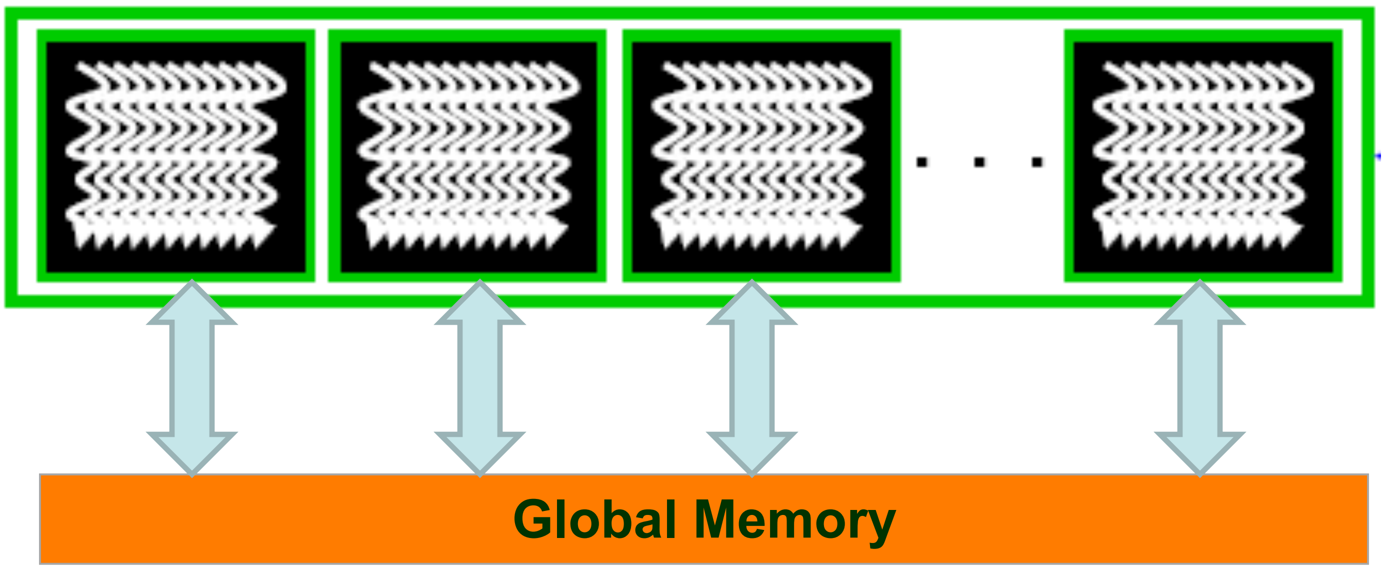
- **Block IDs and Thread IDs**
  - Threads use IDs to decide which data to operation on.
  - Block ID: 1D or 2D or 3D array
  - Thread ID: 1D, 2D, or 3D array
- **Advantage: Easy for data parallel processing with rigid grid data organization**



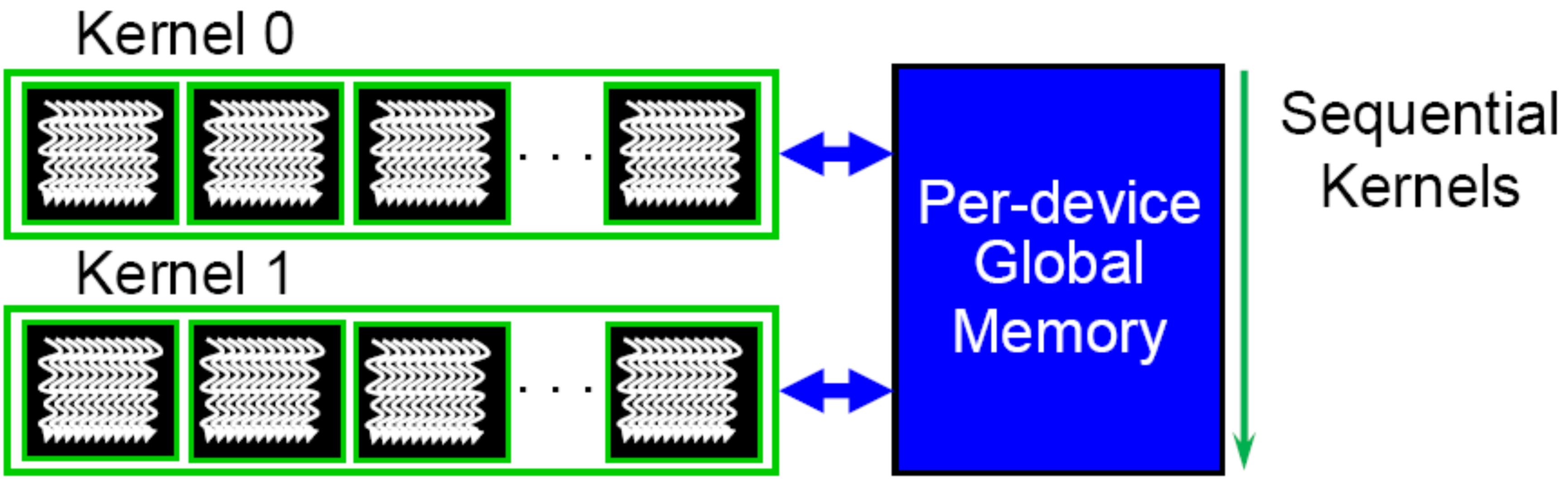
Memory Model: Thread and Block



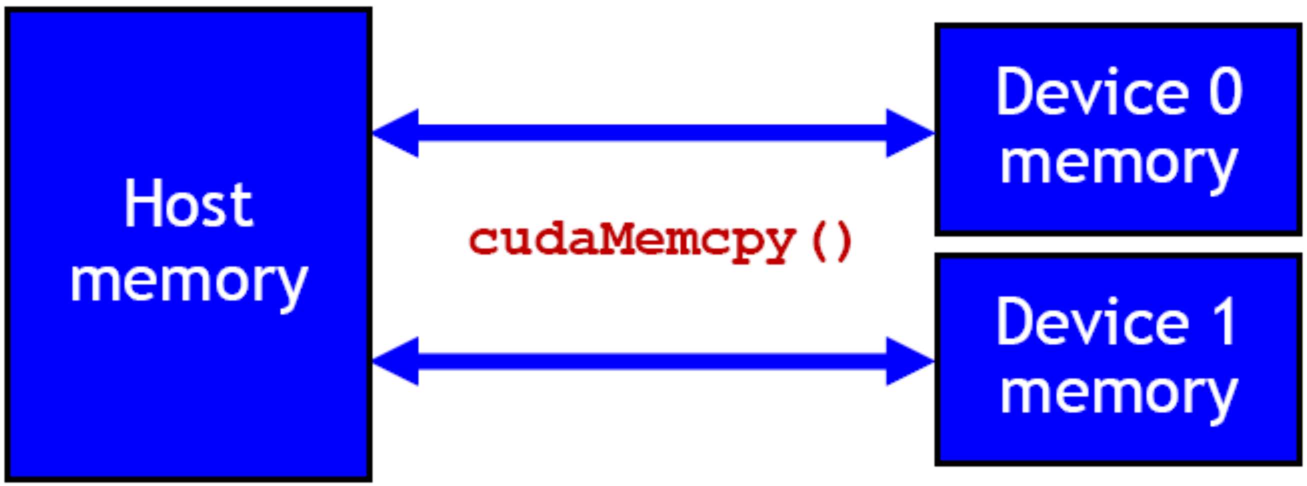
Memory Model: Between Blocks



Memory Model: Between Grids (Kernels)

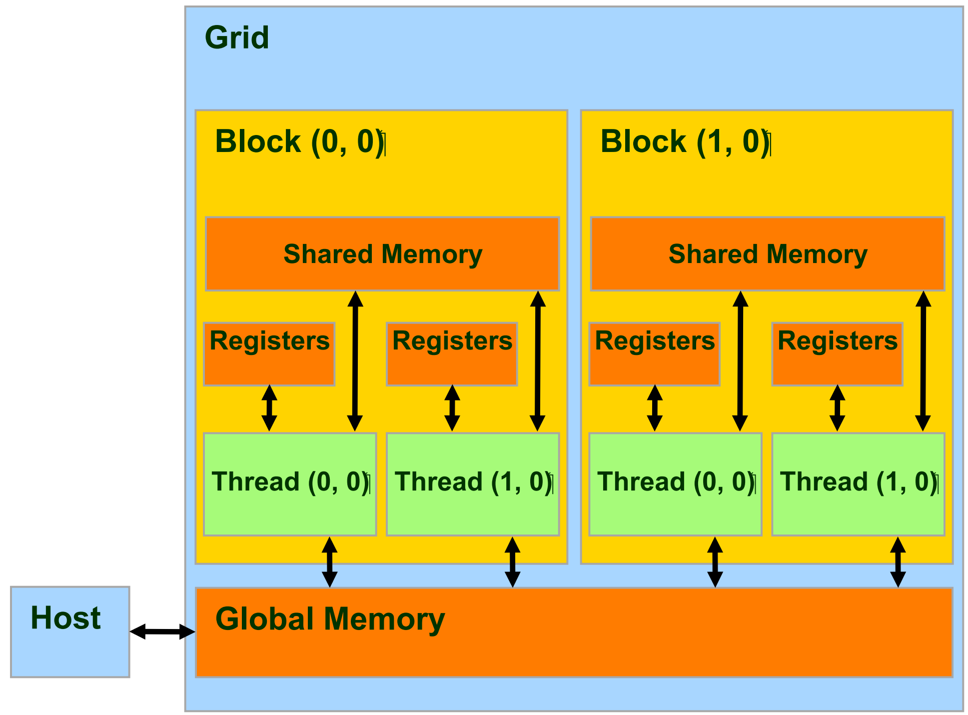


Memory Model: Between Devices



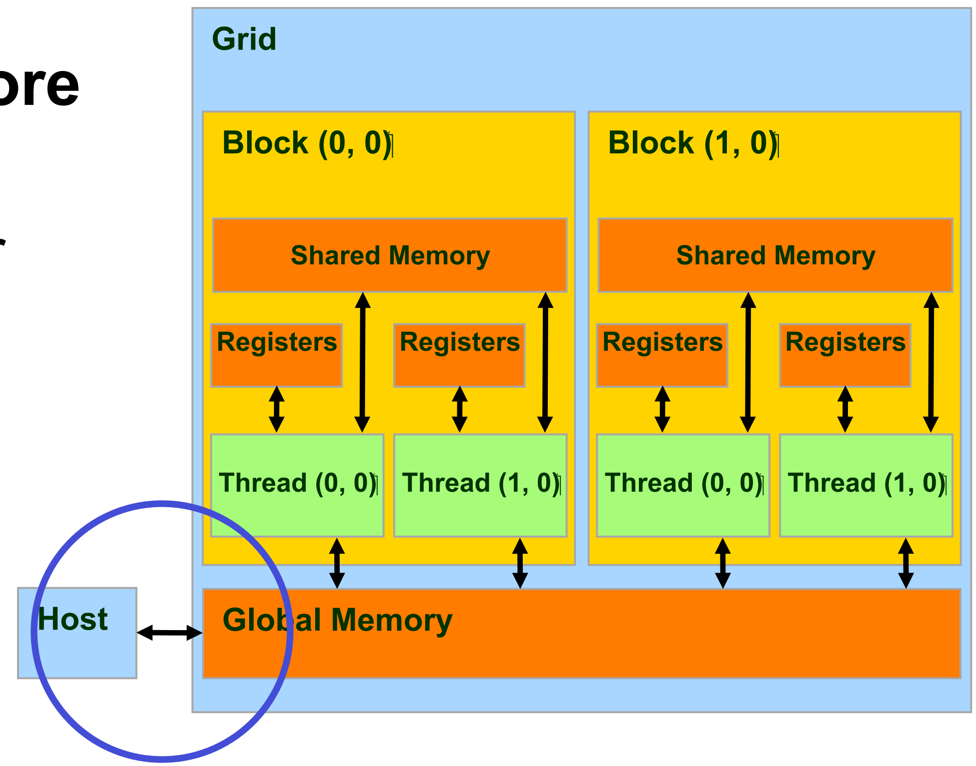
Threads Cooperation

- **Threads within a block**
  - Shared memory
  - Atomic operation
    - Share memory
    - Global memory
  - Barrier
- **Threads between blocks**
  - Atomic operation
    - Global memory
- **Threads between grids**
  - No way!

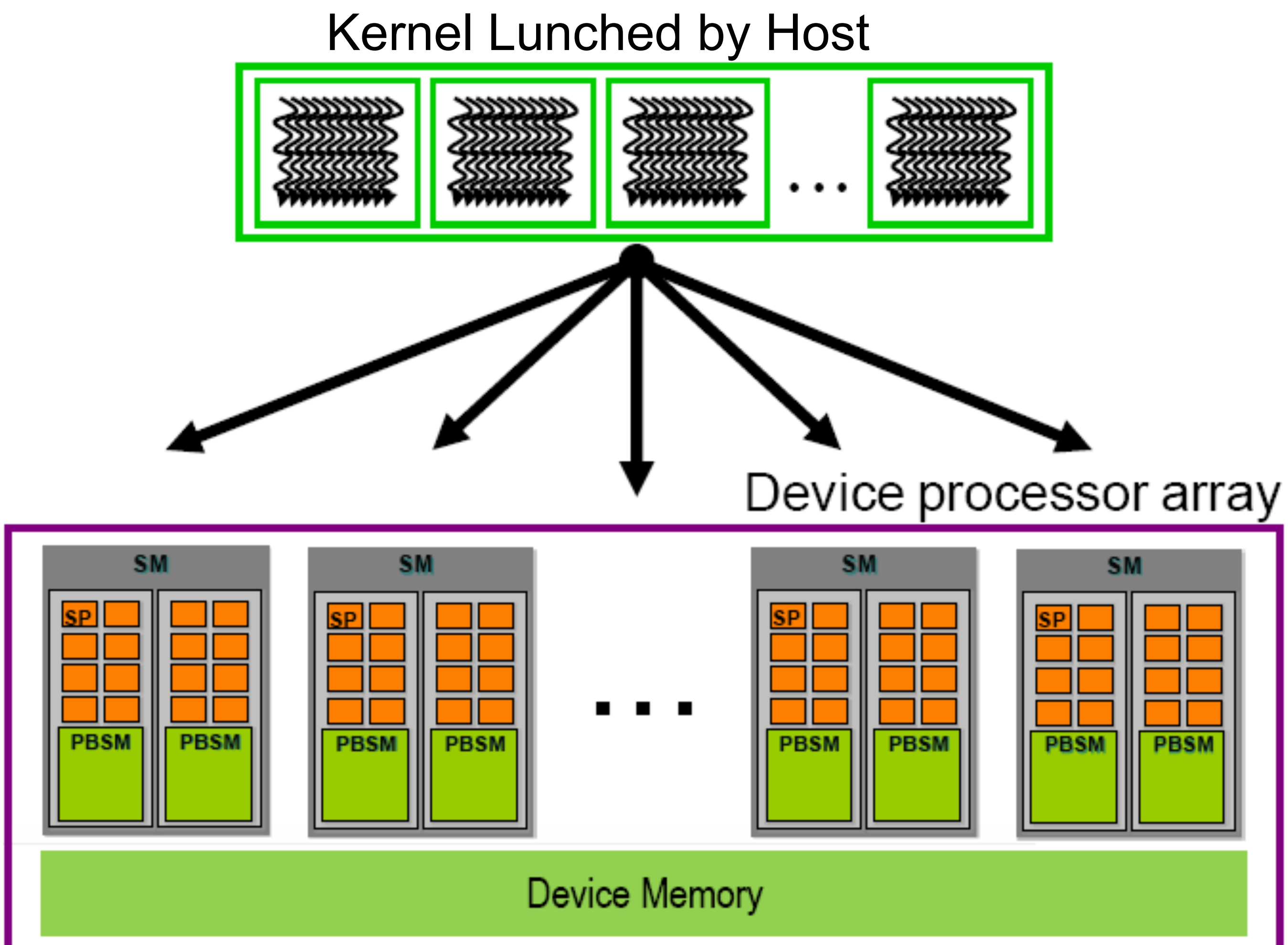


Thread Communication with Host (CPU)

- No communication when GPU kernel is running
- Use global memory before or after GPU kernel call
  - Host initializes transfer request
    - Async vs Sync transfer
  - Only host can allocate device memory
    - No runtime memory allocation on device



Hardware Mapping of Threads

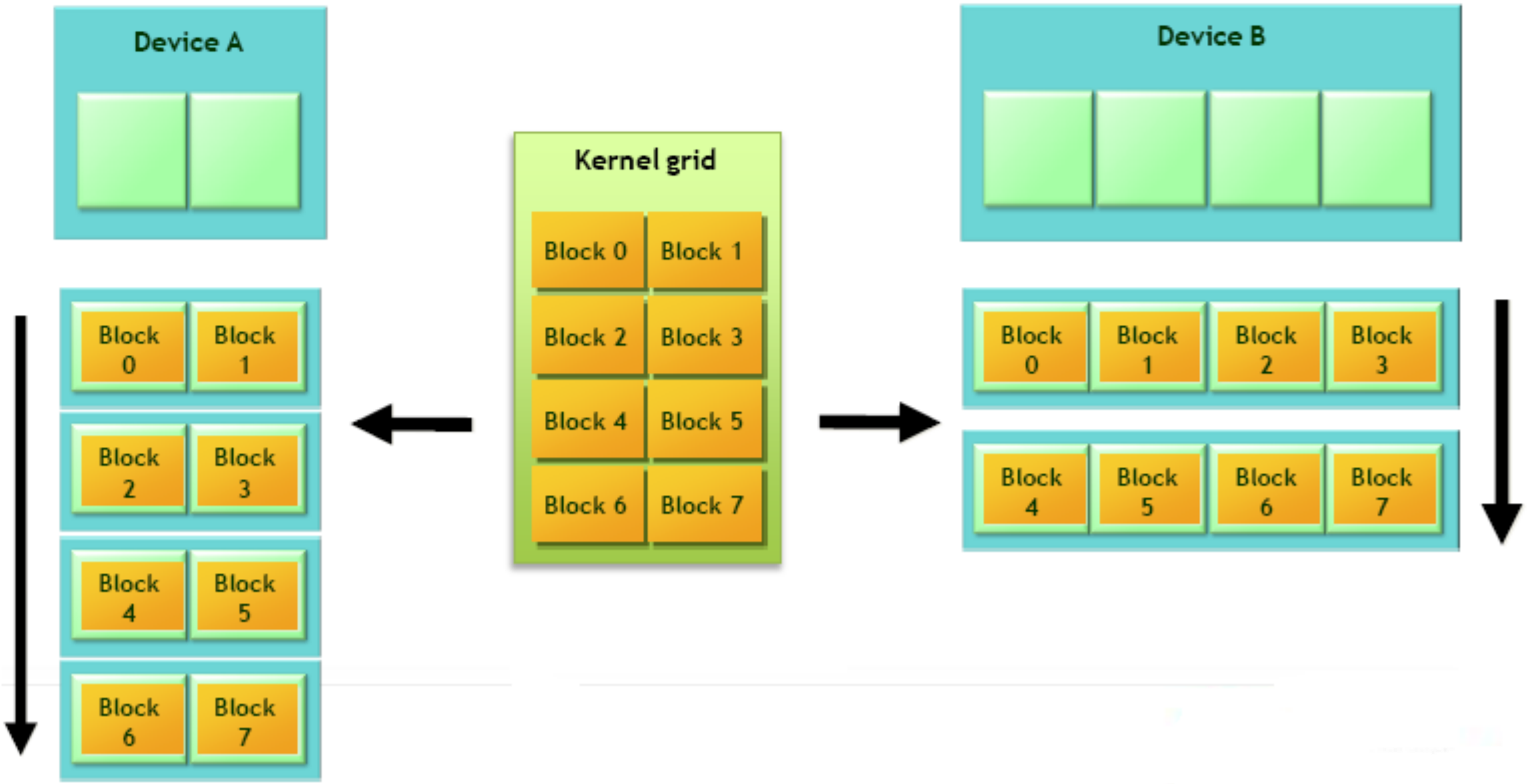


Thread Mapping and Scheduling

- **A grid of threads takes over the whole device.**
- **A block of threads is mapped on one multi-processor.**
  - A multi-processor can take more than one blocks. (Occupancy)
  - A block can not be preempted until finish.
- **Threads within a blocks are scheduled to run on the cores of multi-processor.**
  - Threads are grouped into warps (warp size is 32) as scheduling units.

Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - Kernels scale to any number of parallel multiprocessors



Lightweight Threads

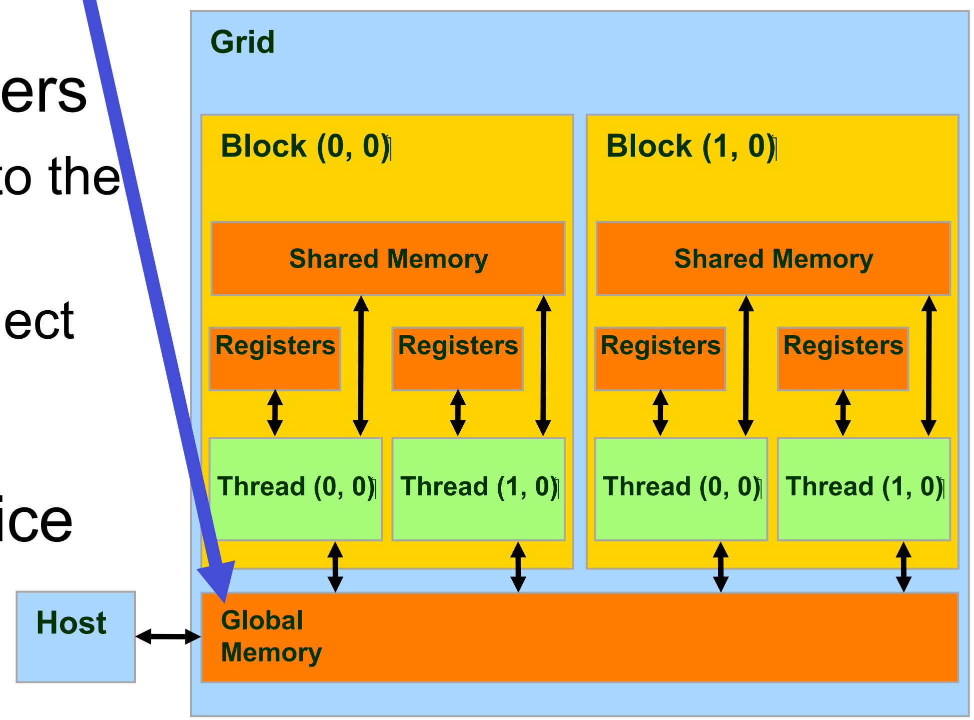
- **Easy to map to cores (Rigid Grid)**
- **Easy to schedule (One cycle)**
- **Therefore:**
  - + High performance (data parallel application)
  - - Hard to synchronize for applications with intensive data dependencies

CUDA Basics

- CUDA device memory allocation and transfer.
- CUDA specific language features.
- Our “Hello World!” CUDA example.

CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device <u>Global Memory</u>
    - <u>Global Memory is R/W</u>
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
- **cudaFree()**
  - Frees object from device Global Memory
    - Pointer to freed object



CUDA Host-Device Data Transfer

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```

- **Code example:**
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` **defines a kernel function**
  - Must return `void`
- **For functions executed on the device:**
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

"Hello World!" – Vector Addition

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

(Length of the vectors: N)

Device Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

"Hello World!" – Vector Addition

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Vector Addition – Host Code for Memory

```
// allocate host (CPU) memory
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
… initalize h_A and h_B …

// allocate device (GPU) memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),cudaMemcpyHostToDevice));
cudaMemcpy( d_B, h_B, N * sizeof(float),cudaMemcpyHostToDevice));
```

Reading

- Please read the second chapters of NIVIDA CUDA Programming Guide.