

nVIDIA®

CUDA Programming Model Overview

C for CUDA



- **Subset of C with extensions**
- **C++ templates for GPU code**
- **CUDA goals:**
 - Scale code to 100s of cores and 1000s of parallel threads
 - Facilitate heterogeneous computing: CPU + GPU
- **CUDA defines:**
 - Programming model
 - Memory model

CUDA Kernels and Threads



- Parallel portions of an application are executed on the device as **kernels**
 - One **kernel** is executed at a time
 - Many threads execute each **kernel**
- Differences between CUDA and CPU threads
 - CUDA threads are extremely lightweight
 - Very little creation overhead
 - Fast switching
 - CUDA uses 1000s of threads to achieve efficiency
 - Multi-core CPUs can only use a few

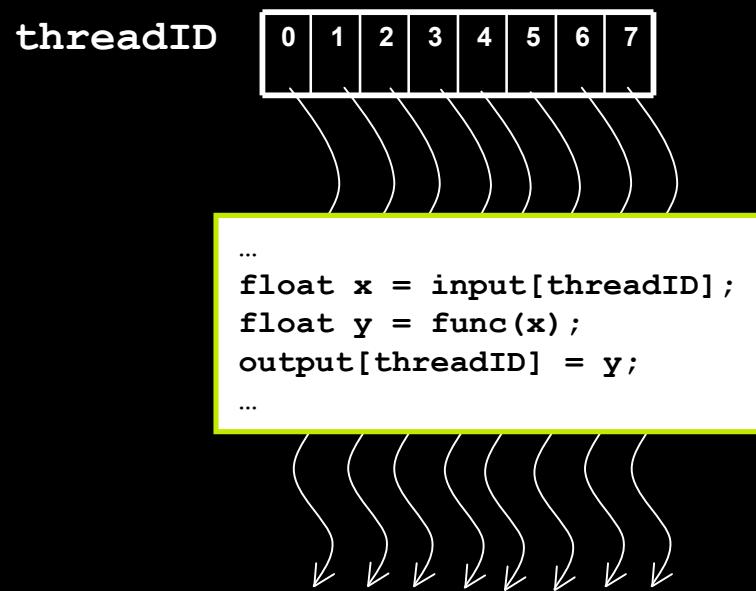
Definitions:

Device = GPU; *Host* = CPU

Kernel = function that runs on the device

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
 - All threads run the same code
 - Each thread has an ID that it uses to compute memory addresses and make control decisions



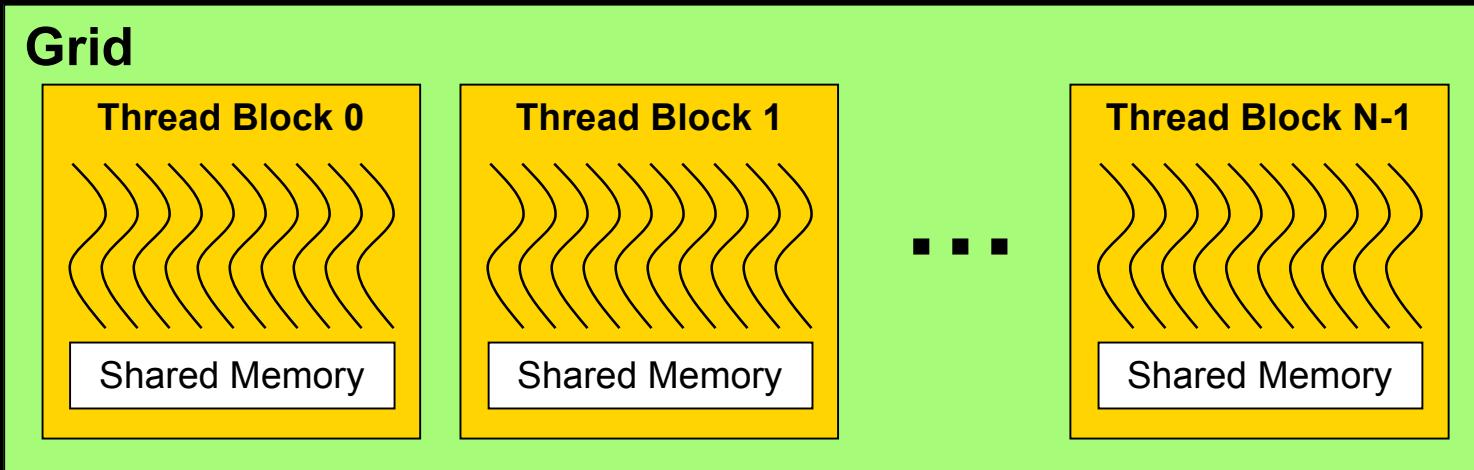
Thread Cooperation



- Threads may need or want to cooperate
- Thread cooperation is a powerful feature of CUDA
- Thread cooperation is valuable because threads can
 - Cooperate on memory accesses
 - Bandwidth reduction for some applications
 - Share results to avoid redundant computation
- Cooperation between a monolithic array of threads is not scalable
 - Cooperation within smaller batches of threads is scalable

Thread Batching

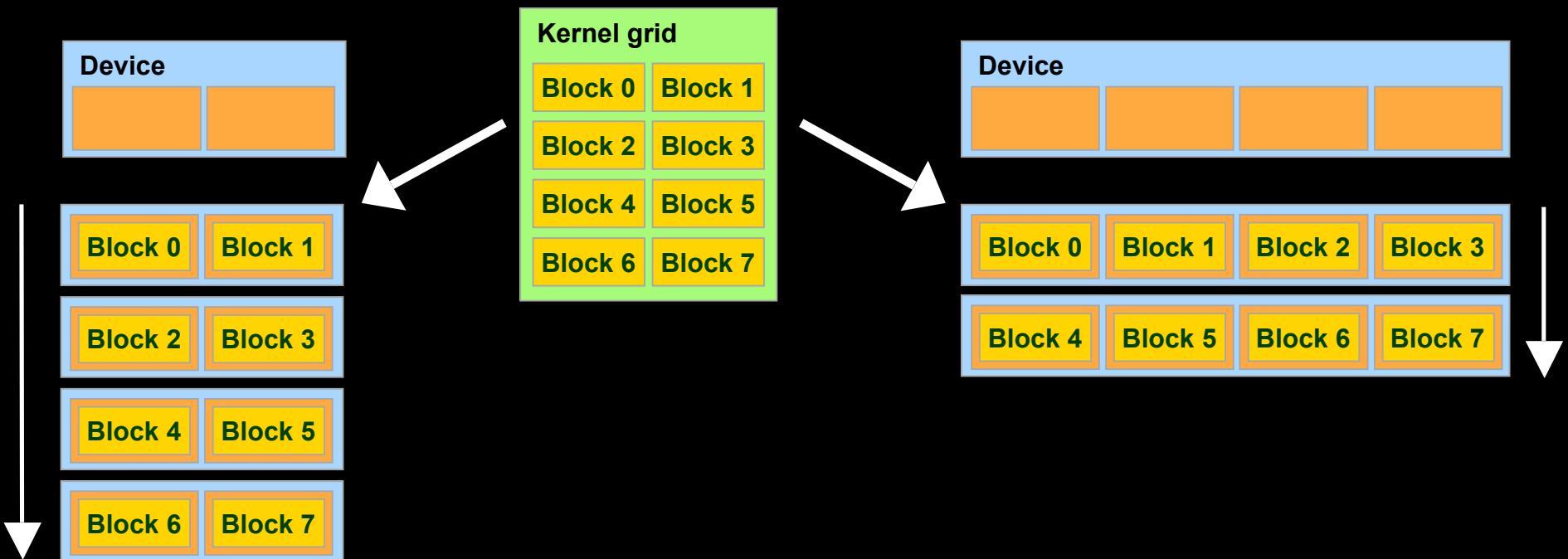
- Kernel launches a grid of **thread blocks**



- Threads within a block cooperate via **shared memory**
- Threads in different blocks cannot cooperate
- Allows programs to ***transparently scale*** to different GPUs

Transparent Scalability

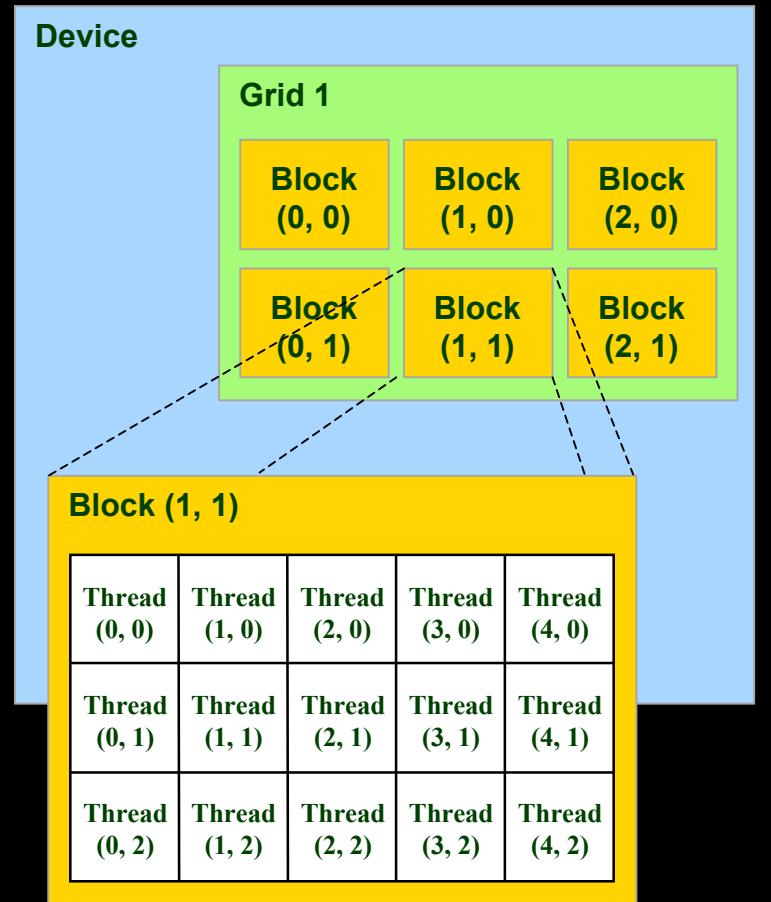
- Hardware is free to schedule thread blocks on any processor



Multidimensional IDs



- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
 - Image processing
 - Solving PDEs on volumes

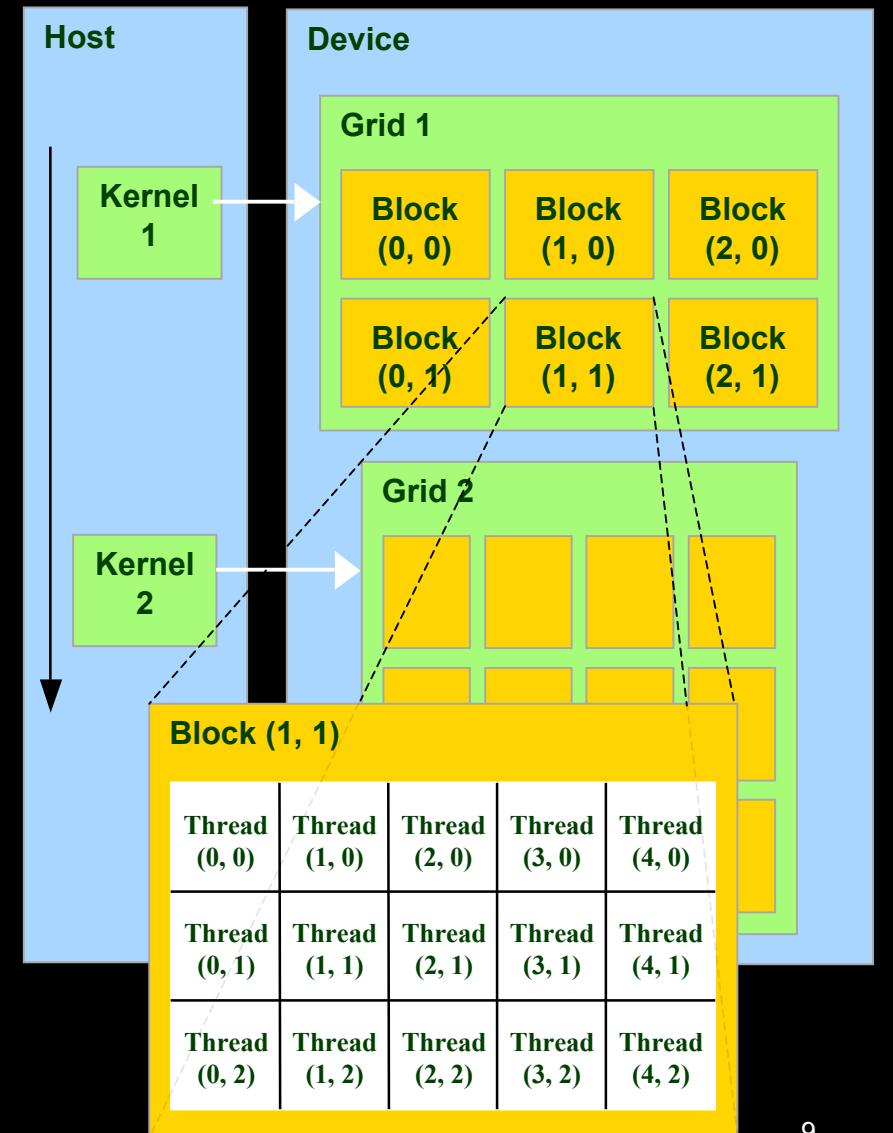


CUDA Programming Model



A kernel is executed by a grid of **thread blocks**

- A **thread block** is a batch of threads that can cooperate with each other by:
 - Sharing data through shared memory
 - Synchronizing their execution
- Threads from different blocks cannot cooperate





Memory Model

- **Registers**
 - Per thread
 - Data lifetime = thread lifetime
- **Local memory**
 - Per thread off-chip memory (physically in device DRAM)
 - Data lifetime = thread lifetime
- **Shared memory**
 - Per thread block on-chip memory
 - Data lifetime = block lifetime
- **Global (device) memory**
 - Accessible by all threads as well as host (CPU)
 - Data lifetime = from allocation to deallocation
- **Host (CPU) memory**
 - Not directly accessible by CUDA threads

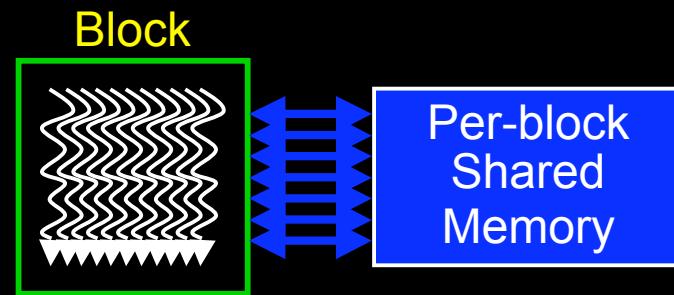
Memory model



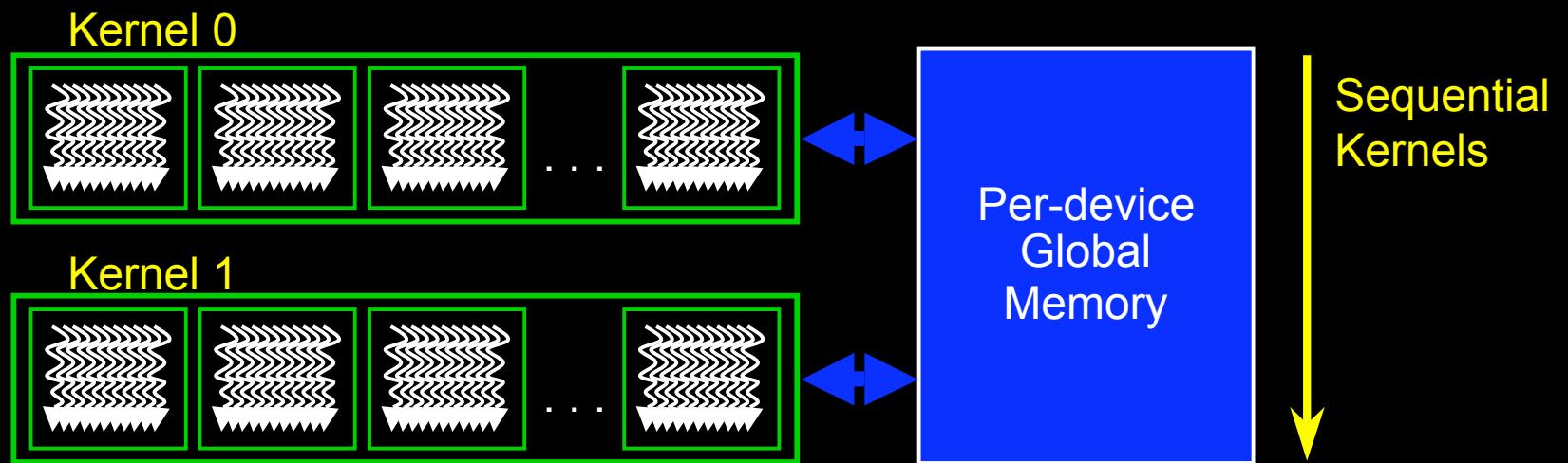
Memory model



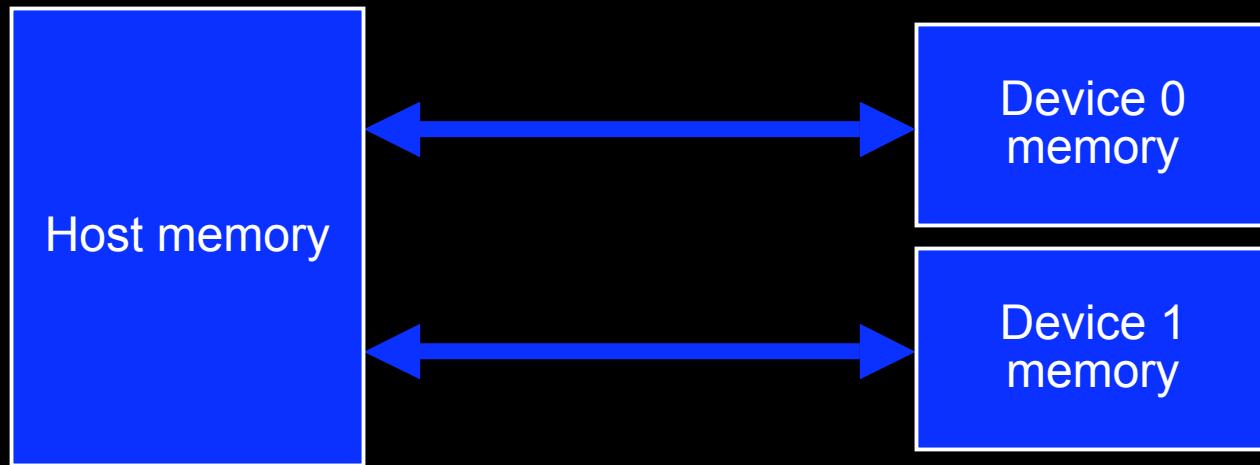
Memory model



Memory model

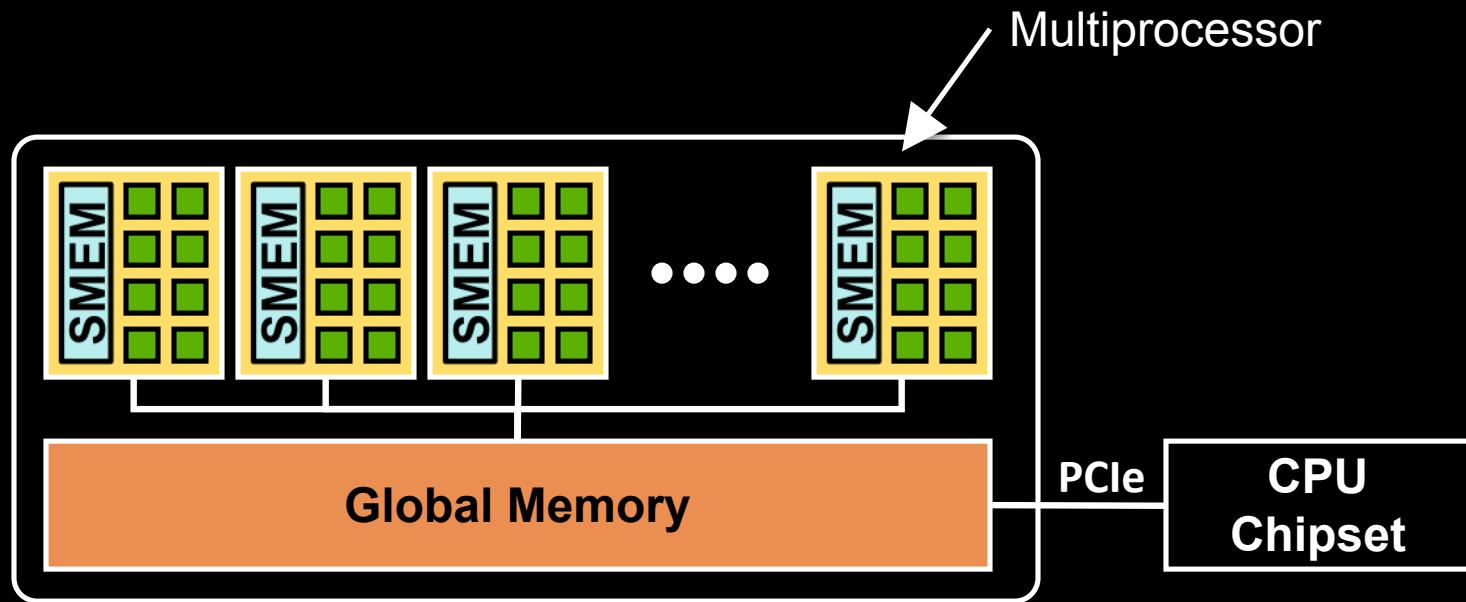


Memory model





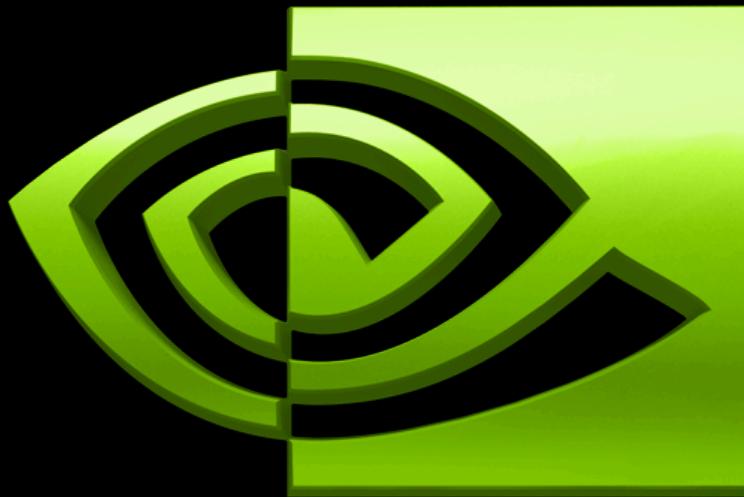
Simple Hardware View



Execution Model



- **Kernels are launched in grids**
 - One kernel executes at a time
- **A thread block executes on one multiprocessor**
 - Does not migrate
- **Several blocks can reside concurrently on one multiprocessor**
 - Number is limited by multiprocessor resources
 - Registers are *partitioned* among all resident threads
 - Shared memory is *partitioned* among all resident thread blocks



nVIDIA®

CUDA Programming

The Basics



Outline of CUDA Basics

- **Basics to set up and execute GPU code:**
 - GPU memory management
 - GPU kernel launches
 - Some specifics of GPU code
- **Some additional features:**
 - Vector types
 - Synchronization
 - Checking CUDA errors
- **Note: only the basic features are covered**
 - See the Programming Guide and Reference Manual for more information



Managing Memory

- CPU and GPU have separate memory spaces
- Host (CPU) code manages device (GPU) memory:
 - Allocate / free
 - Copy data to and from device
 - Applies to *global* device memory (DRAM)

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 *d_a = 0;  
cudaMalloc( (void**) &d_a, nbytes );  
cudaMemset( d_a, 0, nbytes );  
cudaFree(d_a);
```



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**



Executing Code on the GPU

- **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
- **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



Launching kernels

- Modified C function call syntax:

```
kernel<<<dim3 grid, dim3 block>>>(...)
```

- Execution Configuration (“<<< >>>”):
 - grid dimensions: **x** and **y**
 - thread-block dimensions: **x**, **y**, and **z**

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

CUDA Built-in Device Variables

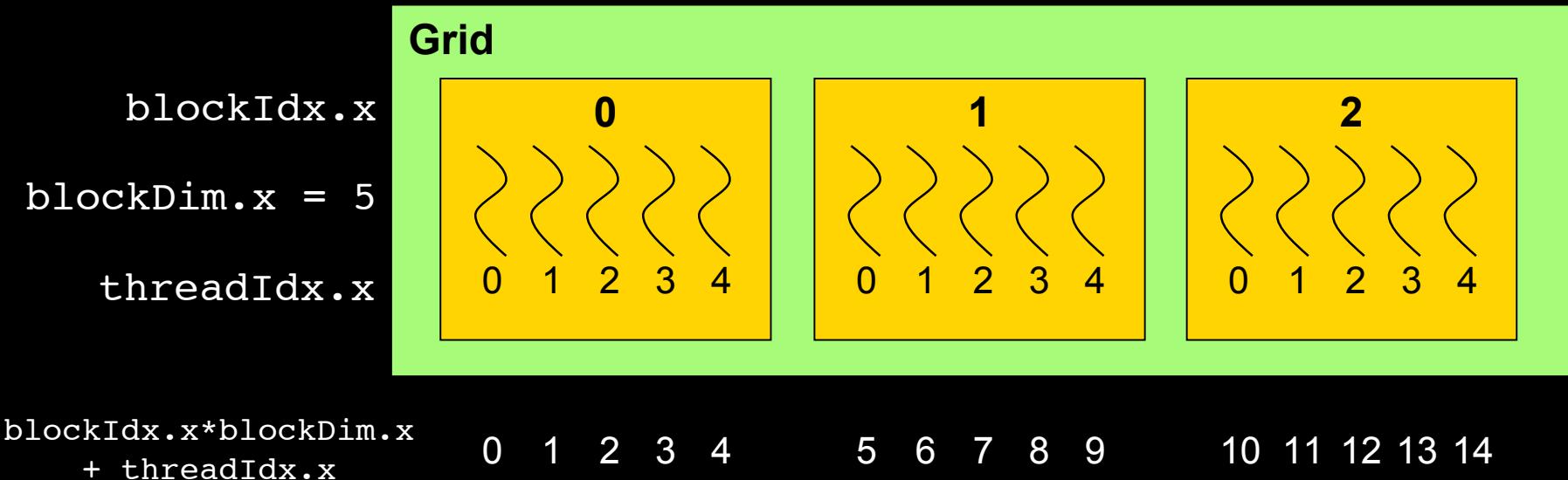


- All **`_global_`** and **`_device_`** functions have access to these automatically defined variables
 - **`dim3 gridDim;`**
 - Dimensions of the grid in blocks (at most 2D)
 - **`dim3 blockDim;`**
 - Dimensions of the block in threads
 - **`dim3 blockIdx;`**
 - Block index within the grid
 - **`dim3 threadIdx;`**
 - Thread index within the block

Data Decomposition



- Often want each thread in kernel to access a different element of an array





Minimal Kernels

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

A brace diagram highlights the calculation of `idx` and the assignment statement `d_a[idx] = value;`. A callout box labeled "Common Pattern!" is positioned below the assignment statement.

Example: Increment Array Elements



Increment N-element vector a by scalar b



Let's assume N=16, blockDim=4 → 4 blocks



blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

`int idx = blockDim.x * blockIdx.x + threadIdx.x;`
will map from local index `threadIdx` to global index

NB: `blockDim` should be ≥ 32 in real code, this is just an example

Example: Increment Array Elements



CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}
```

```
void main()
{
    ....
    increment_cpu(a, b, N);
}
```

```
void main()
{
    ....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize ) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```



Minimal Kernel for 2D data

```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;

    d_a[idx] = value;
}

...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```



Host Synchronization

- All kernel launches are ***asynchronous***
 - control returns to CPU immediately
 - kernel executes after all previous CUDA calls have completed
- ***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

Example: Host Code



```
// allocate host memory
int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```



Variable Qualifiers (GPU code)

- **__device__**
 - Stored in device memory (large capacity, high latency, uncached)
 - Allocated with `cudaMalloc` (**__device__** qualifier implied)
 - Accessible by all threads
 - Lifetime: application
- **__shared__**
 - Stored in on-chip shared memory (SRAM, low latency)
 - Allocated by execution configuration or at compile time
 - Accessible by all threads in the same thread block
 - Lifetime: duration of thread block
- **Unqualified variables:**
 - Scalars and built-in vector types are stored in registers
 - Arrays may be in registers or local memory (*registers are not addressable*)

Using shared memory



Size known at compile time

```
__global__ void kernel(...)  
{  
    ...  
    __shared__ float sData[256];  
    ...  
}  
  
int main(void)  
{  
    ...  
    kernel<<<nBlocks,blockSize>>>(...);  
    ...  
}
```

Size known at kernel launch

```
__global__ void kernel(...)  
{  
    ...  
    extern __shared__ float sData[];  
    ...  
}  
  
int main(void)  
{  
    ...  
    smBytes = blockSize*sizeof(float);  
    kernel<<<nBlocks, blockSize,  
    smBytes>>>(...);  
    ...  
}
```



Built-in Vector Types

Can be used in GPU and CPU code

- **[u]char[1..4], [u]short[1..4], [u]int[1..4],
[u]long[1..4], float[1..4]**
 - Structures accessed with x, y, z, w fields:

```
uint4 param;  
int y = param.y;
```
- **dim3**
 - Based on uint3
 - Used to specify dimensions
 - Default value (1,1,1)

GPU Thread Synchronization



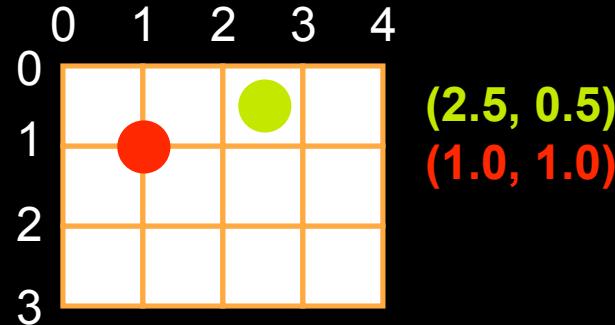
-
- `void __syncthreads();`
- **Synchronizes all threads in a block**
 - Generates barrier synchronization instruction
 - No thread can pass this barrier until all threads in the block reach it
 - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- **Allowed in conditional code only if the conditional is uniform across the entire thread block**



Textures in CUDA

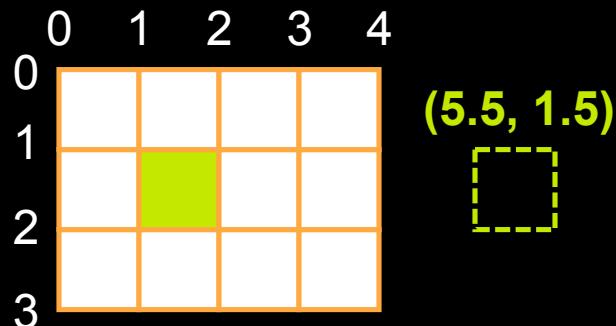
- **Texture is an object for *reading* data**
- **Benefits:**
 - Data is cached (optimized for 2D locality)
 - Helpful when coalescing is a problem
 - Filtering
 - Linear / bilinear / trilinear
 - dedicated hardware
 - Wrap modes (for “out-of-bounds” addresses)
 - Clamp to edge / repeat
 - Addressable in 1D, 2D, or 3D
 - Using integer or normalized coordinates
- **Usage:**
 - CPU code binds data to a texture object
 - Kernel reads data by calling a *fetch* function

Texture Addressing



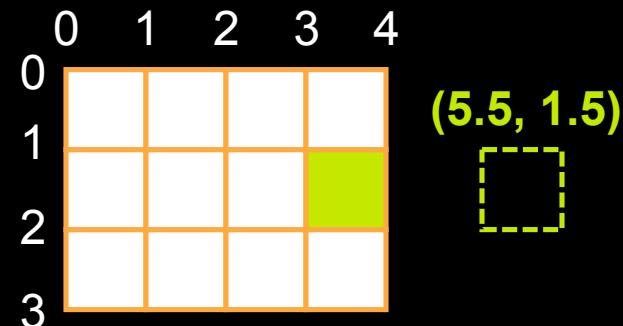
Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



Clamp

- Out-of-bounds coordinate is replaced with the closest boundary





Two CUDA Texture Types

- **Bound to linear memory**
 - Global memory address is bound to a texture
 - Only 1D
 - Integer addressing
 - No filtering, no addressing modes
- **Bound to CUDA arrays**
 - CUDA array is bound to a texture
 - 1D, 2D, or 3D
 - Float addressing (size-based or normalized)
 - Filtering
 - Addressing modes (clamping, repeat)
- **Both:**
 - Return either element type or normalized float



CUDA Texturing Steps

- **Host (CPU) code:**
 - Allocate/obtain memory (global linear, or CUDA array)
 - Create a texture reference object
 - Currently must be at file-scope
 - Bind the texture reference to memory/array
 - When done:
 - Unbind the texture reference, free resources
- **Device (kernel) code:**
 - Fetch using texture reference
 - Linear memory textures:
 - `tex1Dfetch()`
 - Array textures:
 - `tex1D()` or `tex2D()` or `tex3D()`

GPU Atomic Integer Operations



- **Requires hardware with compute capability >= 1.1**
 - G80 = Compute capability 1.0
 - G84/G86/G92 = Compute capability 1.1
 - GT200 = Compute capability 1.3
- **Atomic operations on integers in global memory:**
 - Associative operations on signed/unsigned ints
 - add, sub, min, max, ...
 - and, or, xor
 - Increment, decrement
 - Exchange, compare and swap



Blocks must be independent

- Any possible interleaving of blocks should be valid
 - presumed to run to completion without pre-emption
 - can run in any order
 - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
 - shared queue pointer: OK
 - shared lock: **BAD** ... can easily deadlock

Shared memory atomics & warp voting



- Requires hardware with compute capability ≥ 1.2
- Adds atomic integer operations for shared memory (in addition to global memory)
- Warp vote functions:
 - `_all(int predicate)` returns true iff the predicate evaluates as true for all threads of a warp
 - `_any(int predicate)` returns true iff the predicate evaluates as true for any threads of a warp
 - A warp is a groups of 32 threads within a block (more on this later)



CUDA Error Reporting to CPU

- All CUDA calls return error code:
 - Except for kernel launches
 - `cudaError_t` type
- `cudaError_t cudaGetLastError(void)`
 - Returns the code for the last error (no error has a code)
 - Can be used to get error from kernel execution
- `char* cudaGetString(cudaError_t code)`
 - Returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetString( cudaGetLastError() ) );
```



Device Management

- CPU can query and select GPU devices
 - `cudaGetDeviceCount(int* count)`
 - `cudaSetDevice(int device)`
 - `cudaGetDevice(int *current_device)`
 - `cudaGetDeviceProperties(cudaDeviceProp* prop,
int device)`
 - `cudaChooseDevice(int *device, cudaDeviceProp* prop)`
- Multi-GPU setup:
 - device 0 is used by default
 - one CPU thread can control one GPU
 - multiple CPU threads can control the same GPU
 - calls are serialized by the driver



Multiple CPU Threads and CUDA

- CUDA resources allocated by a CPU thread can be consumed only by CUDA calls from the same CPU thread (more specifically, the same *context*)
- **Violation Example:**
 - CPU **thread 2** allocates GPU memory, stores address in ***p***
 - **thread 3** issues a CUDA call that accesses memory via ***p***

CUDA Event API



- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
 - measure elapsed time for CUDA calls (clock cycle precision)
 - query the status of an asynchronous CUDA call
 - block CPU until CUDA calls prior to the event are completed
 - asyncAPI sample in CUDA SDK

```
cudaEvent_t start, stop;  
cudaEventCreate(&start);           cudaEventCreate(&stop);  
cudaEventRecord(start, 0);  
kernel<<<grid, block>>>(...);  
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);  
float elapsedTime;  
cudaEventElapsedTime(&elapsedTime, start, stop);  
cudaEventDestroy(start);  cudaEventDestroy(stop);
```

Driver API



- Up to this point the host code we've seen has been from the *runtime API*
 - Runtime API: `cuda*` () functions
 - Driver API: `cu*` () functions
- **Advantages:**
 - No dependency on runtime library
 - More control over devices
 - One CPU thread can control multiple GPUs
 - No C extensions in host code, so you can use something other than the default host CPU compiler (e.g. `icc`, etc.)
 - PTX Just-In-Time (JIT) compilation
 - Parallel Thread eXecution (PTX) is our virtual ISA (more on this later)
- **Disadvantages:**
 - No device emulation
 - More verbose code
- **Device code is identical whether you use the runtime or driver API**

Initialization and Device Management



- Must initialize with **cuInit()** before any other call
- Device management:

```
int deviceCount;
cuDeviceGetCount(&deviceCount);
int device;
for (int device = 0; device < deviceCount; ++device)
{
    CUdevice cuDevice;
    cuDeviceGet(&cuDevice, device);
    int major, minor;
    cuDeviceComputeCapability(&major, &minor, cuDevice);
}
```



Context Management

- CUDA context analogous to CPU process
 - Each context has its own address space
- Context created with `cuCtxCreate()`
- A host CPU thread can only have one context current at a time
- Each host CPU thread can have a stack of current contexts
- `cuCtxPopCurrent()` and `cuCtxPushCurrent()` can be used to detach and push a context to a new thread
- `cuCtxAttach()` and `cuCtxDetach()` increment and decrement the usage count and allow for interoperability of code in the same context (e.g. libraries)



Module Management

- Modules are dynamically loadable pieces of device code, analogous to DLLs or shared libraries
- For example to load a module and get a handle to a kernel:

```
CUmodule cuModule;
cuModuleLoad(&cuModule, "myModule.cubin");
CUfunction cuFunction;
cuModuleGetFunction(&cuFunction, cuModule, "myKernel");
```

- A module may contain binaries (a .cubin file) or PTX code that will be Just-In-Time (JIT) compiled

Execution control



```
cuFuncSetBlockShape(cuFunction, blockWidth, blockHeight, 1);
int offset = 0;
int i;
cuParamSeti(cuFunction, offset, i);
offset += sizeof(i);
float f;
cuParamSetf(cuFunction, offset, f);
offset += sizeof(f);
char data[32];
cuParamSetv(cuFunction, offset, (void*)data, sizeof(data));
offset += sizeof(data);
cuParamSetSize(cuFunction, offset);
cuFuncSetSharedSize(cuFunction, numElements * sizeof(float));
cuLaunchGrid(cuFunction, gridWidth, gridHeight);
```



Memory management

- Linear memory is managed using **cuMemAlloc()** and **cuMemFree()**

```
CUdeviceptr devPtr;  
cuMemAlloc(&devPtr, 256 * sizeof(float));
```

- Synchronous copies between host and device

```
cuMemcpyHtoD(devPtr, hostPtr, bytes);  
cuMemcpyDtoH(hostPtr, devPtr, bytes);
```

Summary of Runtime and Driver API



- The runtime API is probably the best place to start for virtually all developers
- Easy to migrate to driver API if/when it is needed
- Anything which can be done in the runtime API can also be done in the driver API, but not vice versa (e.g. migrate a context)
- Much, much more information on both APIs in the ***CUDA Reference Manual***



New Features in CUDA 2.2

- “Zero copy”
 - CUDA threads can directly read/write host (CPU) memory
 - Requires “pinned” (non-pageable) memory
 - Main benefits:
 - More efficient than small PCIe data transfers
 - May be better performance when there is no opportunity for data reuse from device DRAM
- Texturing from pitch linear memory
 - Reduces or eliminates the need for device to device memory copies
 - Can read from it via texture fetches
 - Can also read (uncached, unfiltered) and write