# CUDA
# Programming Model

© NVIDIA Corporation 2008

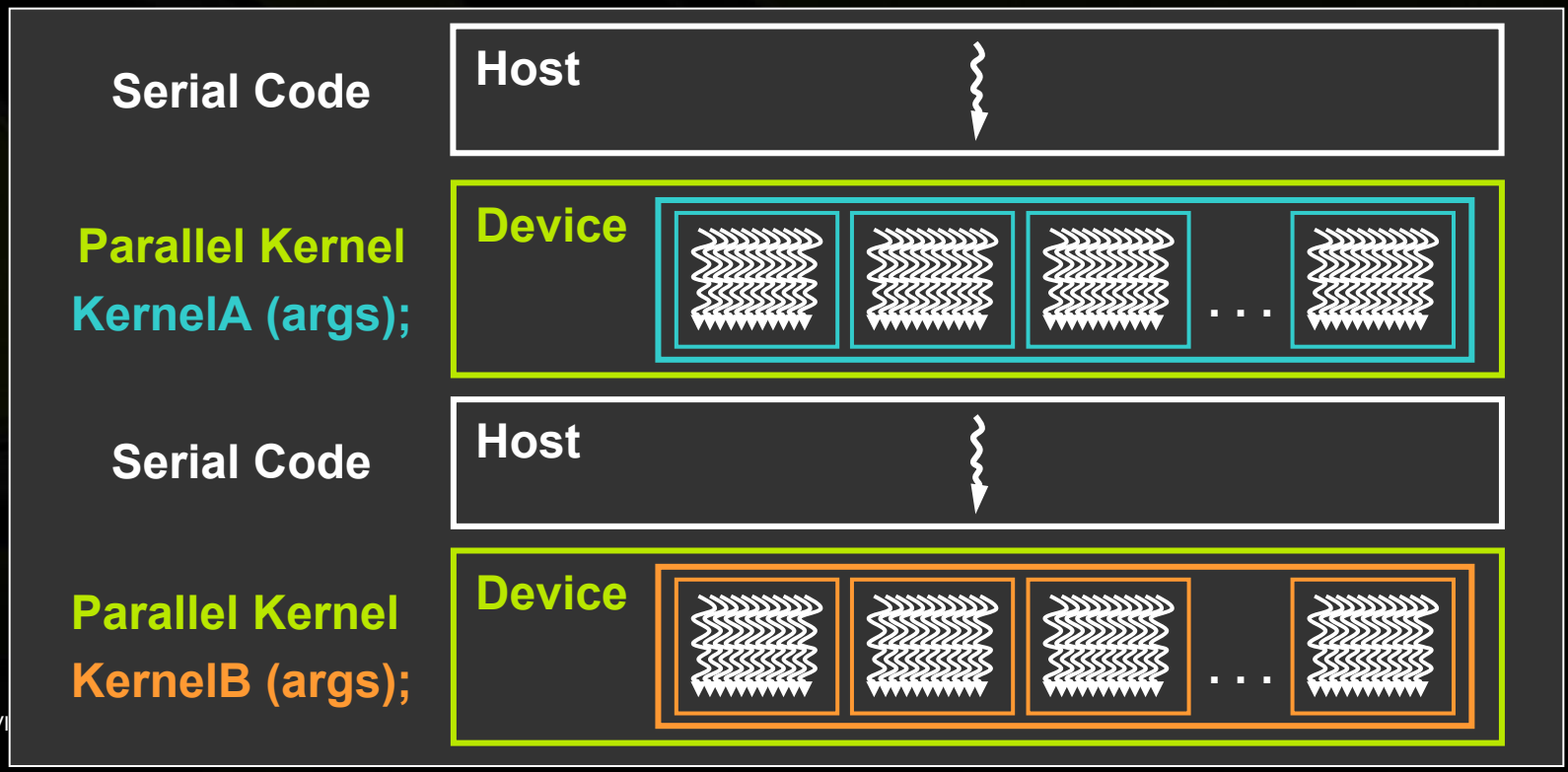
# Some Design Goals

- **Enable heterogeneous systems** (i.e., CPU+GPU)
  - CPU & GPU are separate devices with separate DRAMs
- **Scale** to 100's of cores, 1000's of parallel threads
- Let programmers focus on parallel algorithms
  - *not* mechanics of a parallel programming language
  - **Use C/C++** with minimal extensions

© NVIDIA Corporation 2008

# Heterogeneous Programming

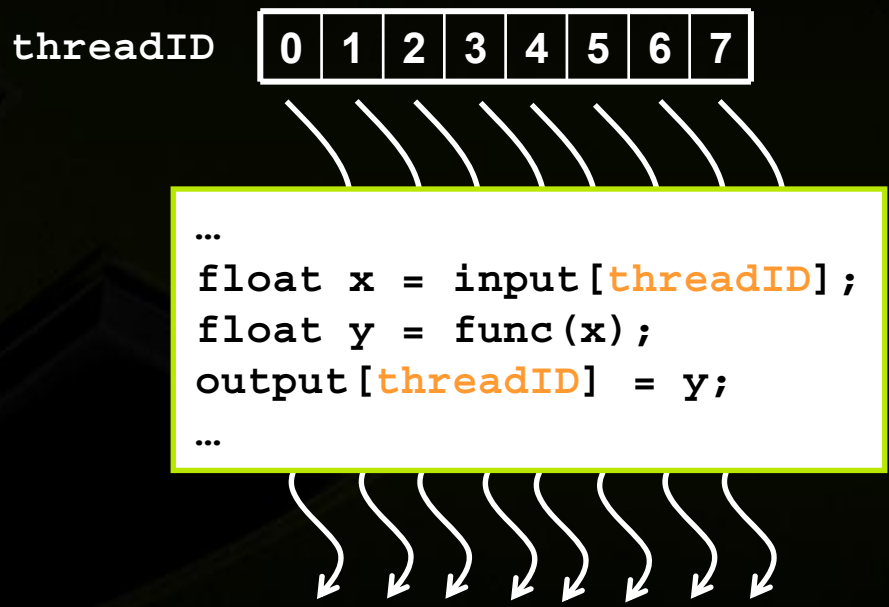
- CUDA = serial program with parallel kernels, all in C
  - Serial C code executes in a host thread (i.e. CPU thread)
  - Parallel kernel C code executes in many device threads across multiple processing elements (i.e. GPU threads)




© NVI

# Kernel = Many Concurrent Threads

- One kernel is executed at a time on the device
- Many threads execute each kernel
  - Each thread executes the same code…
  - … on different data based on its threadID

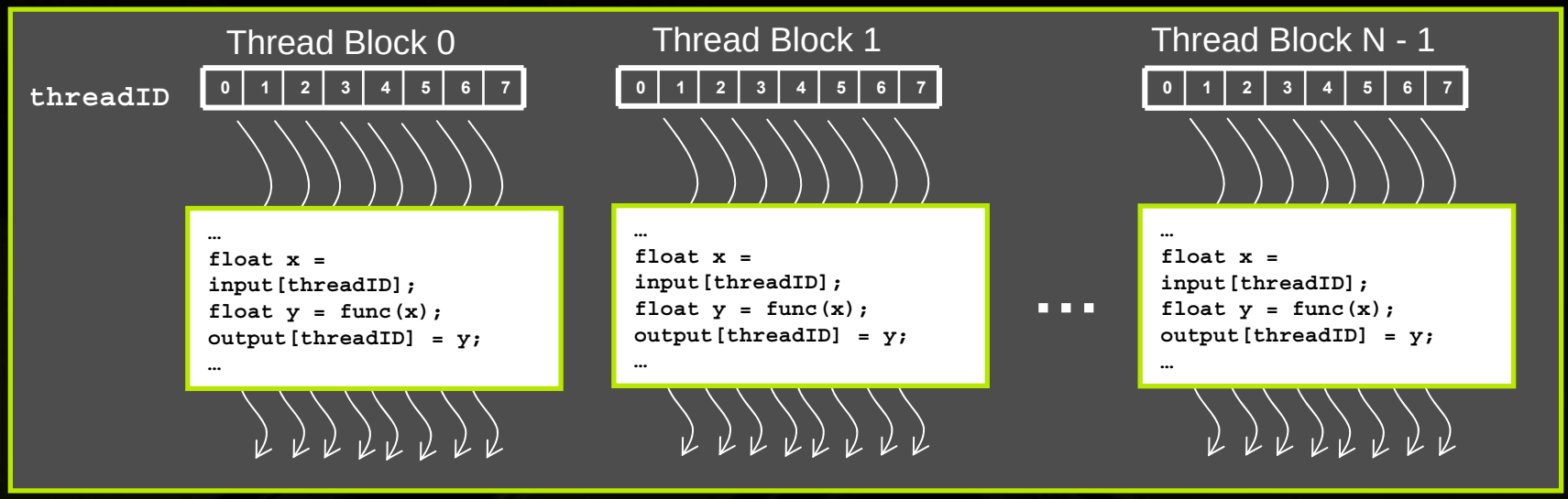



- CUDA threads might be
  - Physical threads
    - As on NVIDIA GPUs
    - GPU thread creation and context switching are essentially free
  - Or virtual threads
    - E.g. 1 CPU core might execute multiple CUDA threads

© NVIDIA Corporation 2008

# Hierarchy of Concurrent Threads

- Threads are grouped into thread blocks
  - Kernel = grid of thread blocks




- By definition, threads in the same block may synchronize with barriers

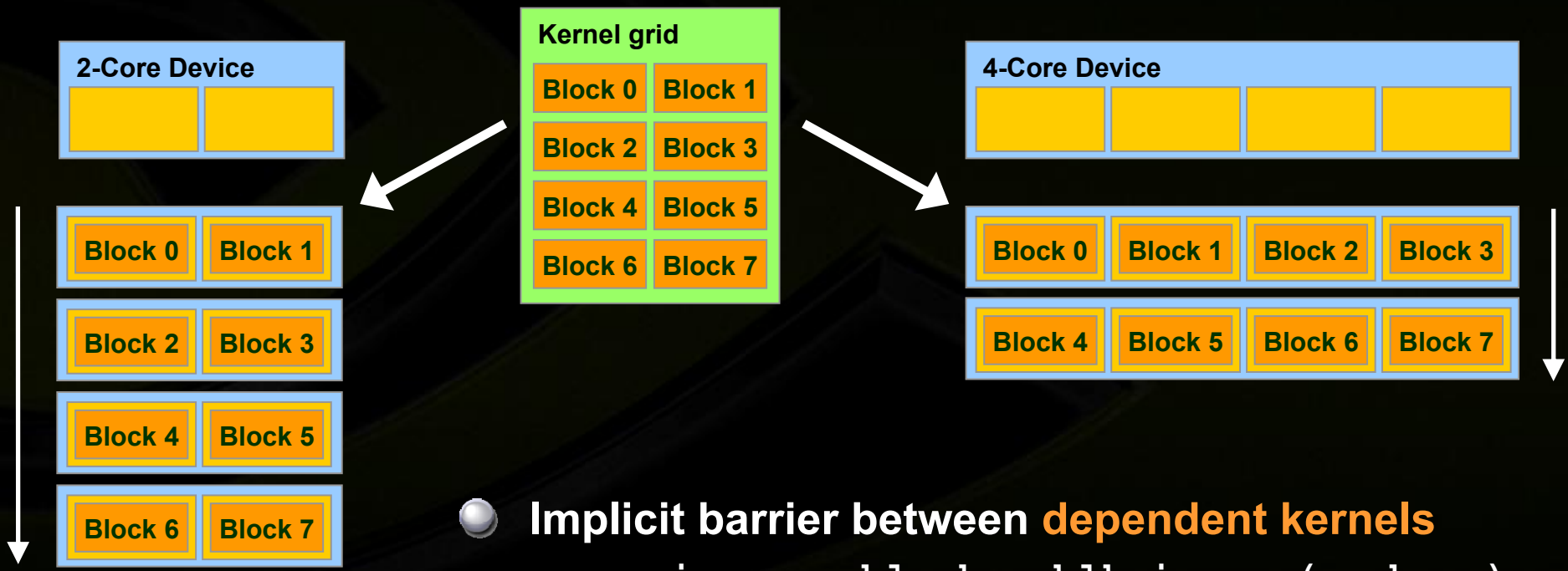
```
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID - 1];
```

Threads wait at the barrier until all threads in the same block reach the barrier

© NV

# Transparent Scalability

- Thread blocks cannot synchronize
  - So they can run in any order, concurrently or sequentially
- This independence gives scalability:
  - A kernel scales across any number of parallel cores

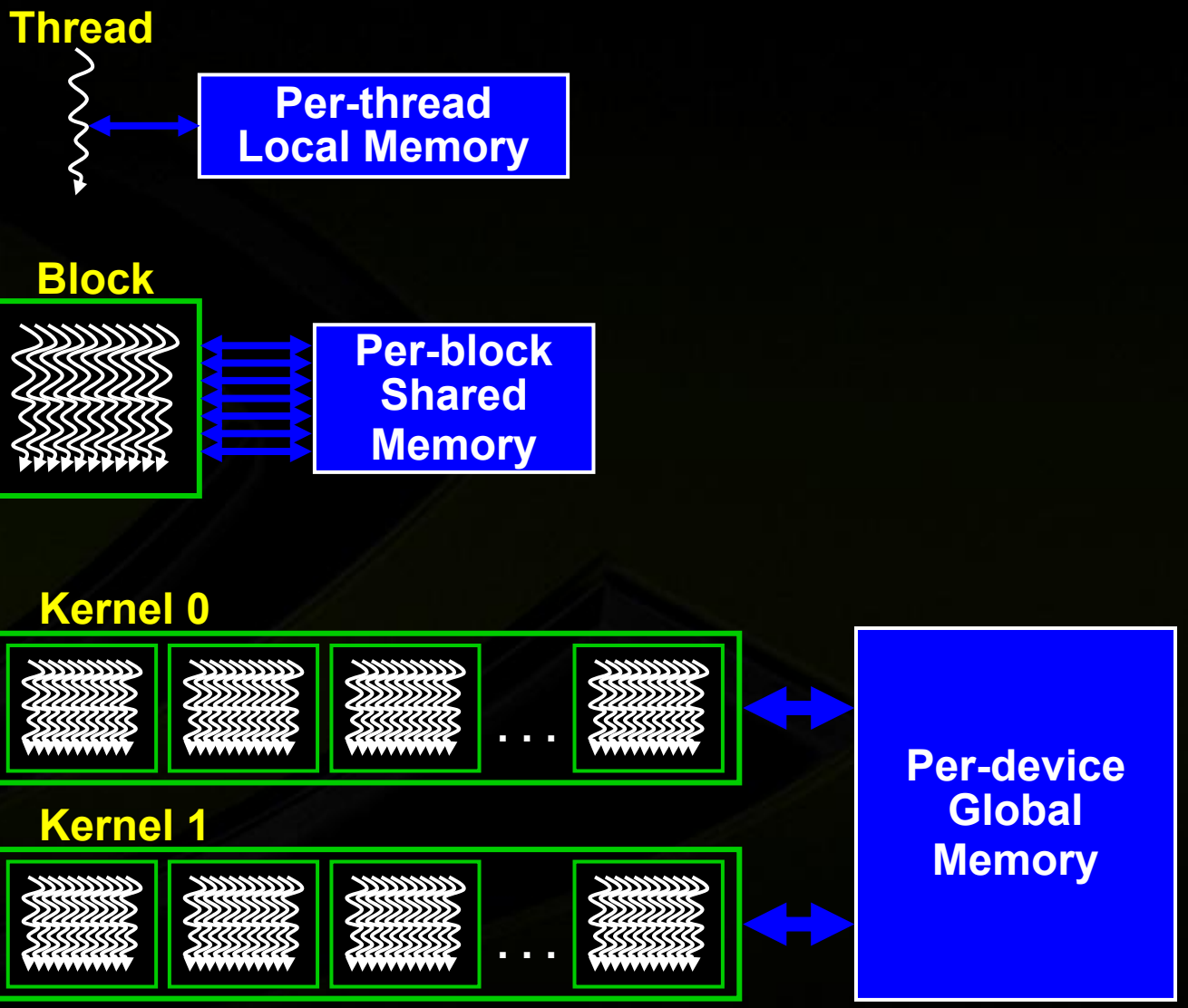



- Implicit barrier between dependent kernels

```
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
```

© NVIDIA Corporation 2008

# Memory Hierarchy




© NVIDIA Corporation 2008

# Heterogeneous Memory Model


© NVIDIA Corporation 2008

# CUDA Language: C with Minimal Extensions

- Philosophy: provide minimal set of extensions necessary to expose power
- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   // kernel function, runs on device
__device__ int  GlobalVar;         // variable in device memory
__shared__ int  SharedVar;         // variable in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);     // launch 500 blocks w/ 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;  dim3 gridDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                   // barrier synchronization within kernel
```

© NVIDIA Corporation 2008

# CUDA Runtime

- **Device management:**
  `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`
- **Device memory management:**
  `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- **Graphics interoperability:**
  `cudaGLMapBufferObject()`, `cudaD3D9MapResources()`
- **Texture management:**
  `cudaBindTexture()`, `cudaBindTextureToArray()`

© NVIDIA Corporation 2008

# 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;
}


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

## CUDA program

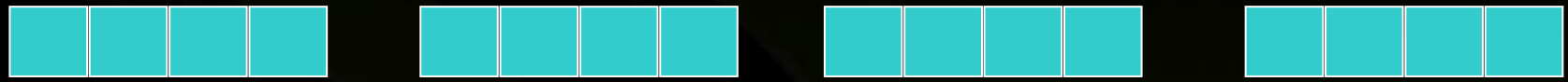
```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}


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

© NVIDIA Corporation 2008

# 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 * blockId.x + threadIdx.x;
will map from local index threadIdx to global index

**Common Pattern!**

NB: blockDim should be >= 32 in real code, this is just an example

© NVIDIA Corporation 2008

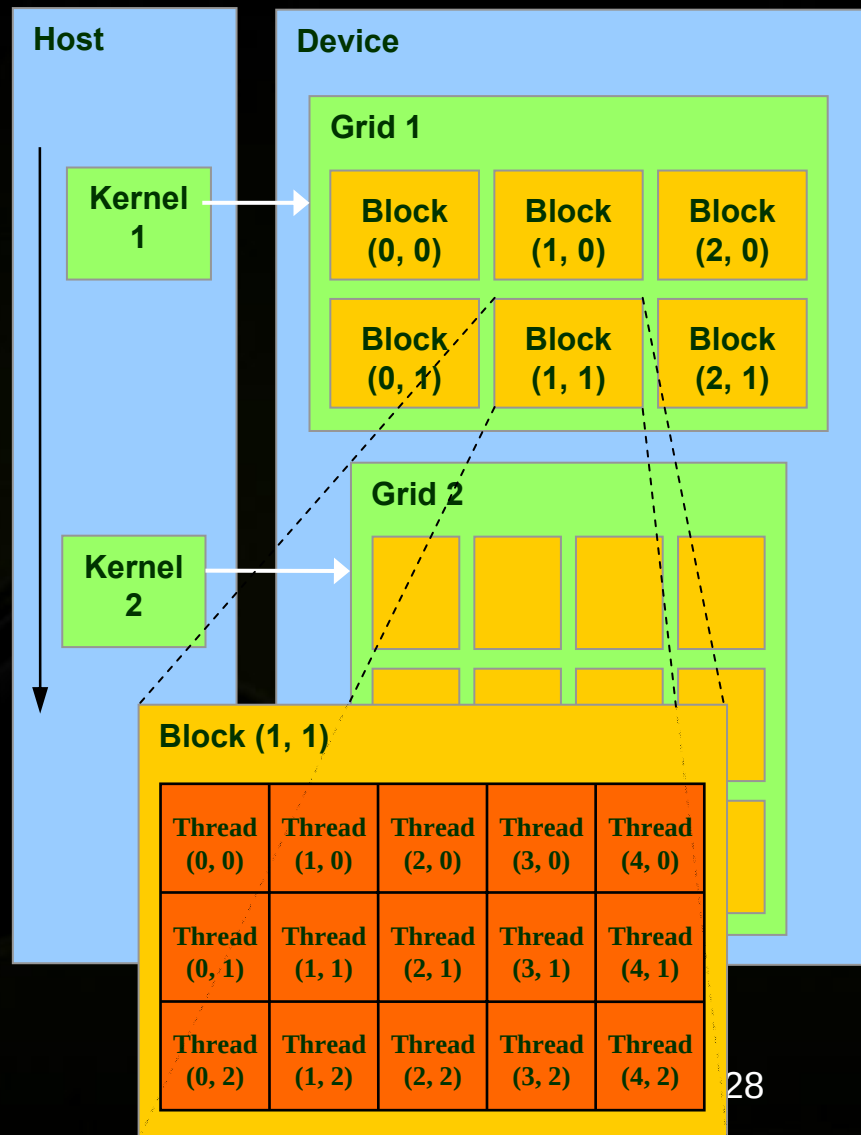
# Example: Host Code


```
// allocate host memory
unsigned 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);
```

© NVIDIA Corporation 2008

# More on Thread and Block IDs

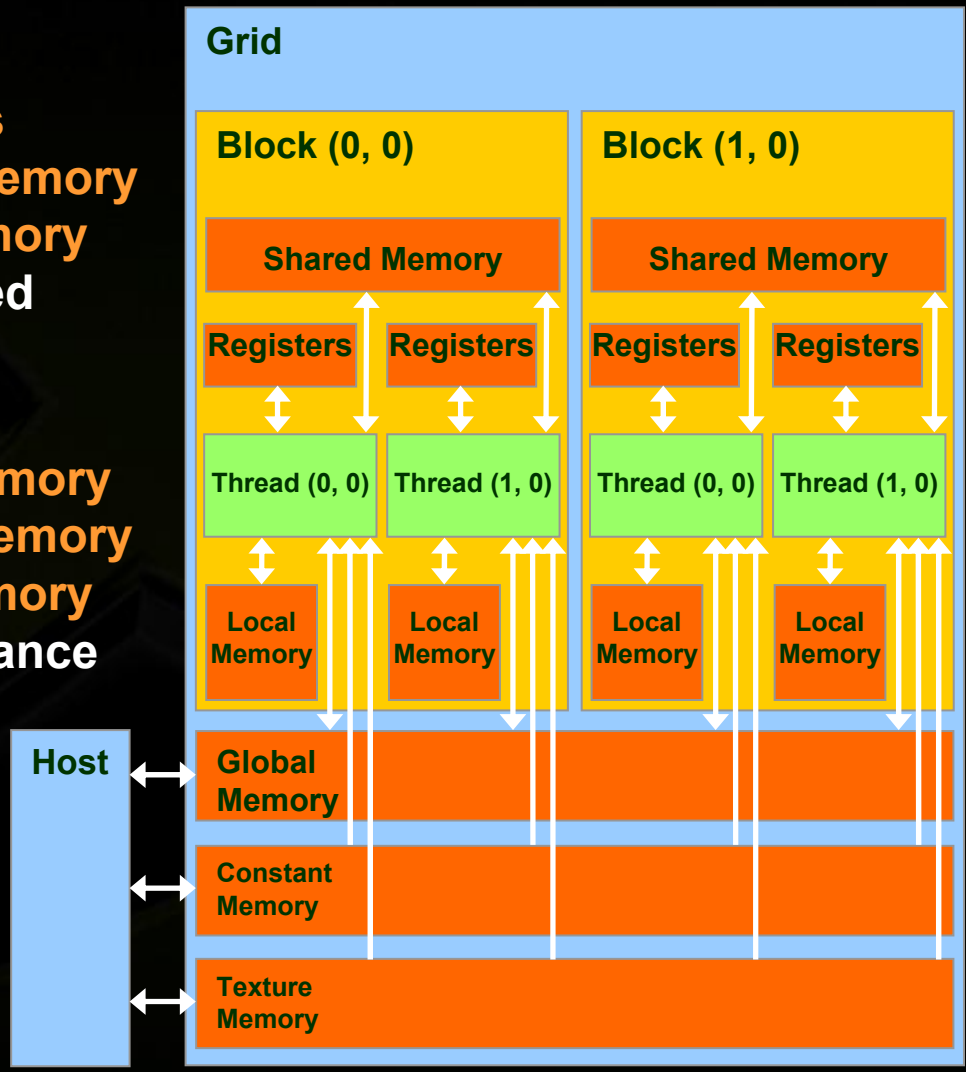
- Threads and blocks have IDs
  - So each thread can decide what data to work on
- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes




© NVIDIA Corporation 2008

# More on Memory Spaces

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Most important, commonly used
- Each thread can also:
  - Read/write per-thread local memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
  - Used for convenience/performance
    - More details later
- The host can read/write global, constant, and texture memory (stored in DRAM)




© NVIDIA Corporation 2008

# Features Available in Device Code

- Standard mathematical functions

  `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.

- Texture accesses in kernels

```
texture<float,2> my_texture;   // declare texture reference
float4 texel = texfetch(my_texture, u, v);
```

- Integer atomic operations in global memory

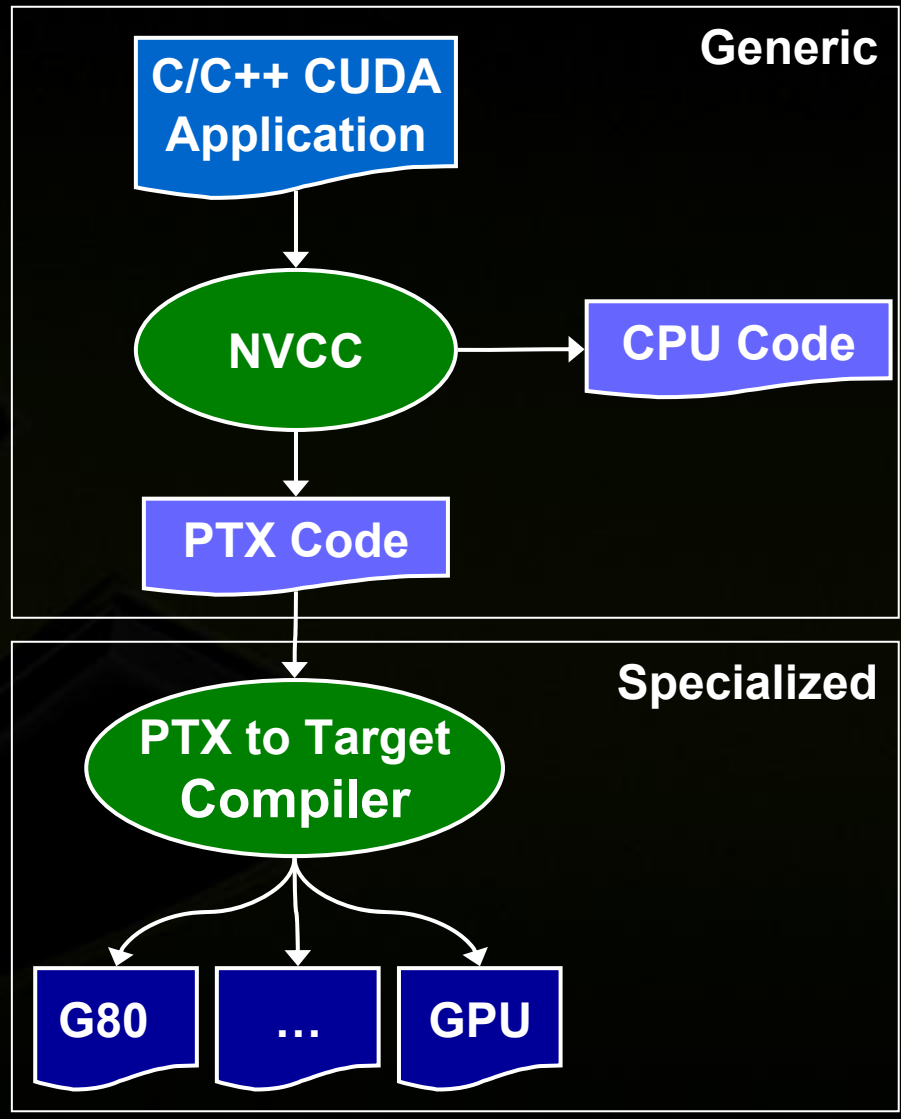
  `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.

  - e.g., increment shared queue pointer with `atomicInc()`
  - Only for devices with compute capability 1.1
    - 1.0 = Tesla, Quadro FX5600, GeForce 8800 GTX, etc.
    - 1.1 = GeForce 8800 GT, etc.

© NVIDIA Corporation 2008

# Compiling CUDA for NVIDIA GPUs

- Any source file containing CUDA language extensions must be compiled with NVCC
  - NVCC separates code running on the host from code running on the device
- Two-stage compilation:
  1. Virtual ISA
     - Parallel Thread eXecution
  2. Device-specific binary object




© NVIDIA Corporation 2008

# Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

© NVIDIA Corporation 2008

# Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads potentially produce different results
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs
  - Different instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

© NVIDIA Corporation 2008

# Reduction Example

- Reduce N values to a single one:
  - Sum($v_0$, $v_1$, … , $v_{N-2}$, $v_{N-1}$)
  - Min($v_0$, $v_1$, … , $v_{N-2}$, $v_{N-1}$)
  - Max($v_0$, $v_1$, … , $v_{N-2}$, $v_{N-1}$)
- Common primitive in parallel programming
- Easy to implement in CUDA
  - Less so to get it right
- Divided into 5 exercises throughout the day
  - Each exercise illustrates one particular optimization strategy

© NVIDIA Corporation 2008

# Reduction Exercise

- At the end of each exercise, the result of the reduction computed on the device is checked for correctness
  - "Test PASSED" or "Test FAILED" is printed out to the console
- The goal is to replace the "`TODO`" words in the code by the right piece of code to get "test PASSED"

© NVIDIA Corporation 2008

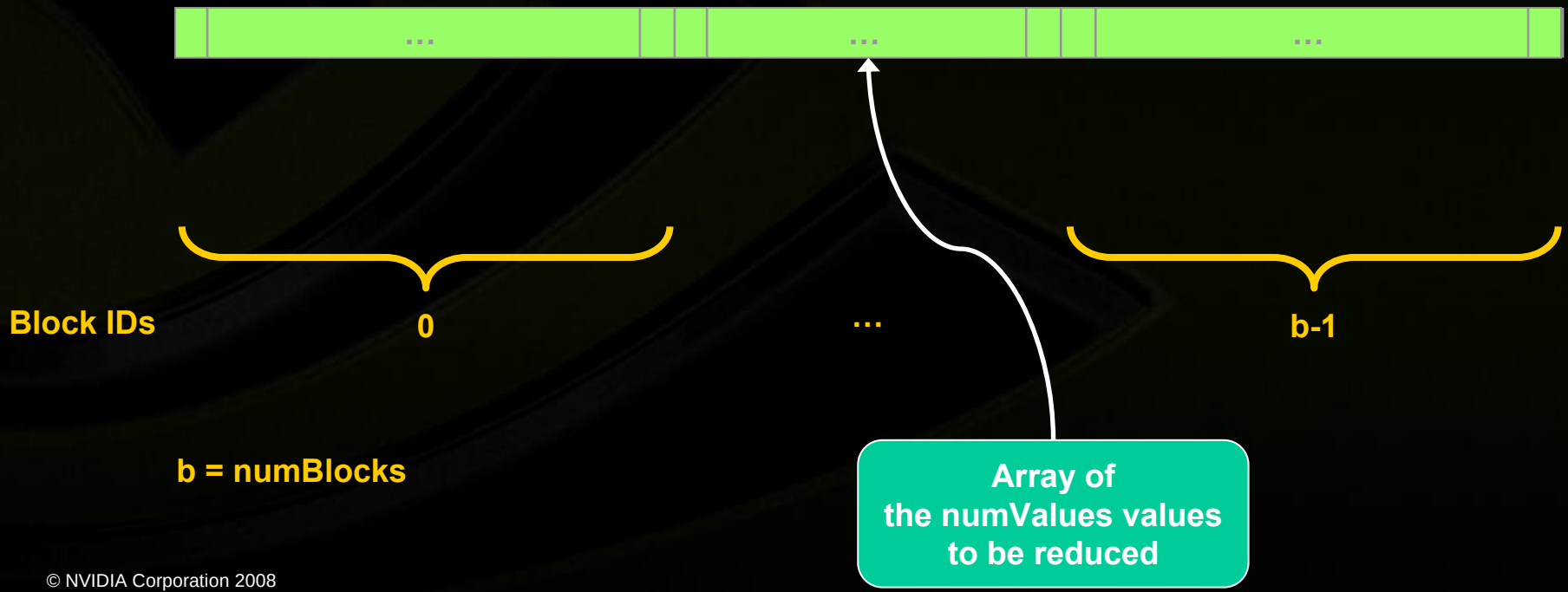
# Reduction Exercise 1

- Open up `reduce\src\reduce1.sln`
- Code walkthrough:
  - `main.cpp`
    - Allocate host and device memory
    - Call `reduce()` defined in `reduce1.cu`
    - Profile and verify result
  - `reduce1.cu`
    - CUDA code compiled with `nvcc`
    - Contains `TODO`s
- Device emulation compilation configurations: `Emu*`

© NVIDIA Corporation 2008

# Reduce 1: Blocking the Data

- Split the work among the N multiprocessors (16 on G80) by launching `numBlocks=N` thread blocks




© NVIDIA Corporation 2008

# Reduce 1: Blocking the Data

- Within a block, split the work among the threads
  - A block can have at most 512 threads
  - We choose `numThreadsPerBlock`=512 threads


t = numThreadsPerBlock

b = numBlocks

© NVIDIA Corporation 2008

# Reduce 1: Multi-Pass Reduction

- Blocks cannot synchronize so `reduce_kernel` is called multiple times:
  - First call reduces from `numValues` to `numThreads`
  - Each subsequent call reduces by half
  - Ping pong between input and output buffers (`d_Result[2]`)

© NVIDIA Corporation 2008

# Reduce 1: Go Ahead!

- Goal: Replace the `TODO`s in `reduce1.cu` to get "test PASSED"


© NVIDIA Corporation 2008

# CUDA Implementation on the GPU

© NVIDIA Corporation 2008