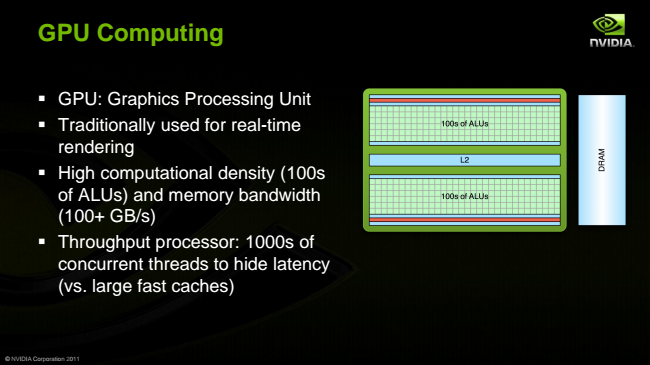


**An Introduction to GPU Computing and CUDA Architecture**

Sarah Tariq, NVIDIA Corporation

© NVIDIA Corporation 2011



**GPU Computing**

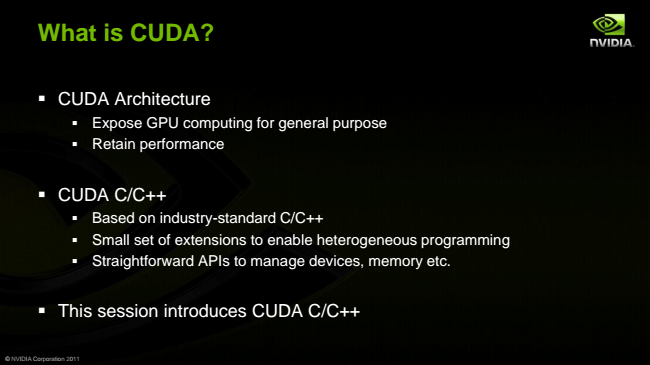
▪ GPU: Graphics Processing Unit

▪ Traditionally used for real-time rendering

▪ High computational density (100s of ALUs) and memory bandwidth (100+ GB/s)

▪ Throughput processor: 1000s of concurrent threads to hide latency (vs. large fast caches)

© NVIDIA Corporation 2011



**What is CUDA?**

▪ CUDA Architecture

▪ Expose GPU computing for general purpose

▪ Retain performance

▪ CUDA C/C++

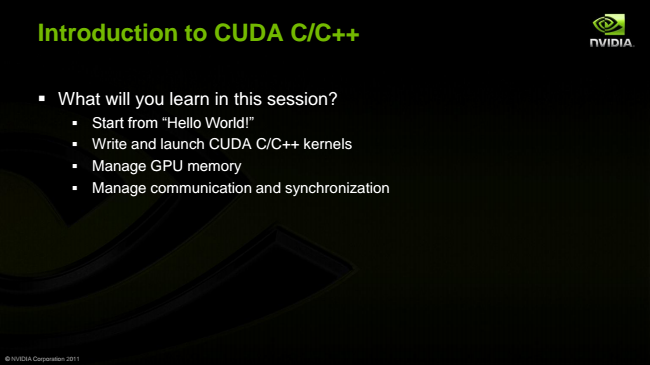
▪ Based on industry-standard C/C++

▪ Small set of extensions to enable heterogeneous programming

▪ Straightforward APIs to manage devices, memory etc.

▪ This session introduces CUDA C/C++

© NVIDIA Corporation 2011



**Introduction to CUDA C/C++**

▪ What will you learn in this session?

▪ Start from “Hello World!”

▪ Write and launch CUDA C/C++ kernels

▪ Manage GPU memory

▪ Manage communication and synchronization

© NVIDIA Corporation 2011



**Prerequisites**

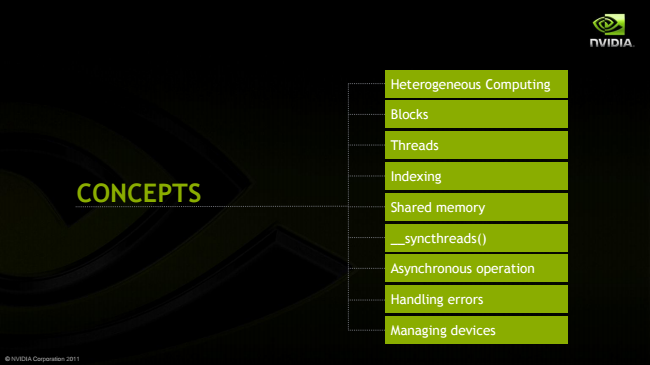
▪ You (probably) need experience with C or C++

▪ You don’t need GPU experience

▪ You don’t need parallel programming experience

▪ You don’t need graphics experience

© NVIDIA Corporation 2011



© NVIDIA Corporation 2011

**CONCEPTS**

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

\_\_syncthreads()

Asynchronous operation

Handling errors

Managing devices



**HELLO WORLD!**

© NVIDIA Corporation 2011

**CONCEPTS**

Heterogeneous Computing

Blocks

Threads

Indexing

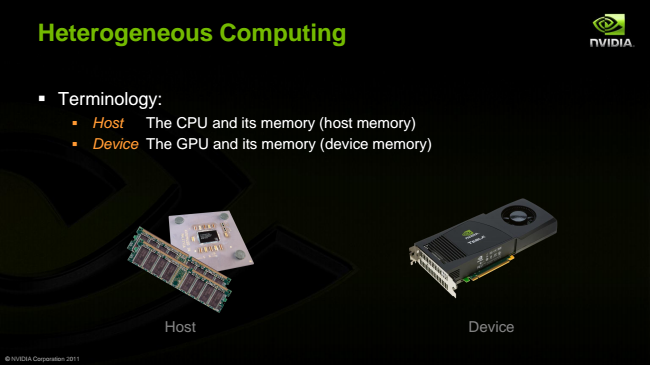
Shared memory

\_\_syncthreads()

Asynchronous operation

Handling errors

Managing devices



**Heterogeneous Computing**

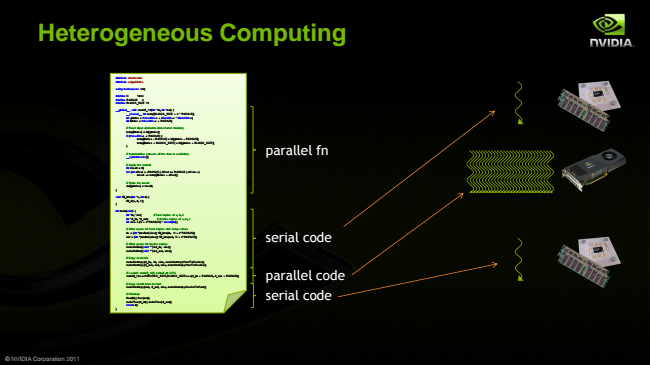
▪ Terminology:

▪ Host The CPU and its memory (host memory)

▪ Device The GPU and its memory (device memory)

© NVIDIA Corporation 2011

Host Device



**Heterogeneous Computing**

© NVIDIA Corporation 2011

#include <iostream> #include <algorithm>

using namespace std;

#define N 1024 #define RADIUS 3 #define BLOCK\_SIZE 16

\_\_global\_\_ void stencil\_1d(int \*in, int \*out) {

\_\_shared\_\_ int temp[BLOCK\_SIZE + 2 \* RADIUS]; int gindex = threadIdx.x + blockIdx.x \* blockDim.x; int lindex = threadIdx.x + RADIUS;

// Read input elements into shared memory temp[lindex] = in[gindex]; if (threadIdx.x < RADIUS) {

temp[lindex - RADIUS] = in[gindex - RADIUS]; temp[lindex + BLOCK\_SIZE] = in[gindex + BLOCK\_SIZE]; }

// Synchronize (ensure all the data is available) \_\_syncthreads();

// Apply the stencil int result = 0; for (int offset = -RADIUS ; offset <= RADIUS ; offset++)

result += temp[lindex + offset];

// Store the result out[gindex] = result; }

void fill\_ints(int \*x, int n) {

fill\_n(x, n, 1); }

int main(void) {

int \*in, \*out; // host copies of a, b, c int \*d\_in, \*d\_out; // device copies of a, b, c int size = (N + 2\*RADIUS) \* sizeof(int);

// Alloc space for host copies and setup values in = (int \*)malloc(size); fill\_ints(in, N + 2\*RADIUS); out = (int \*)malloc(size); fill\_ints(out, N + 2\*RADIUS);

// Alloc space for device copies cudaMalloc((void \*\*)&d\_in, size); cudaMalloc((void \*\*)&d\_out, size);

// Copy to device cudaMemcpy(d\_in, in, size, cudaMemcpyHostToDevice); cudaMemcpy(d\_out, out, size, cudaMemcpyHostToDevice);

// Launch stencil\_1d() kernel on GPU stencil\_1d<<<N/BLOCK\_SIZE,BLOCK\_SIZE>>>(d\_in + RADIUS, d\_out + RADIUS);

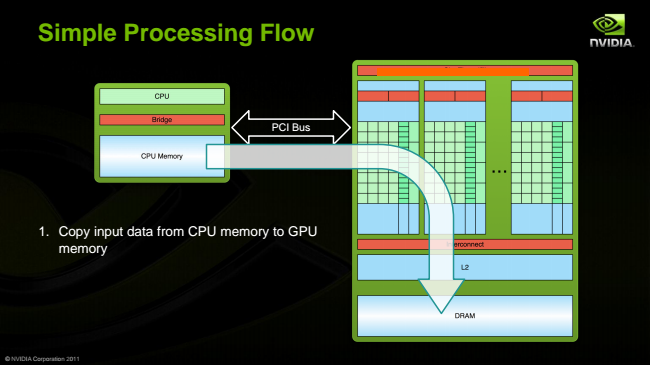
// Copy result back to host cudaMemcpy(out, d\_out, size, cudaMemcpyDeviceToHost);

// Cleanup free(in); free(out); cudaFree(d\_in); cudaFree(d\_out); return 0; }

parallel fn

serial code

parallel code serial code



**Simple Processing Flow**

1. Copy input data from CPU memory to GPU

memory

© NVIDIA Corporation 2011

PCI Bus