1. CUDA overview
   - Kernels
   - Thread Hierarchy
   - Memory Hierarchy
   - Heterogeneous Programming
   - Compute Capability

2. Programming Interface
   - NVCC and PTX
   - CUDA C runtime
Acknowledgements

Material used in creating these slides comes from

- NVIDIA’s **CUDA C Programming Guide**
- **Course on CUDA Programming** by Mike Giles, Oxford University Mathematical Institute
Vendors like NVIDIA already have significant experience developing 3-D graphics software to scale with available 3-D hardware.

CUDA designed to similarly scale with available GPU hardware:
- GPU devices at different price points and generations have different numbers of cores; ideally a GPU application can take best advantage of available GPU device
- multiple GPU devices can be present in a single system

Three key abstractions made available to programmers:
1. hierarchy of threads
2. hierarchy of memories
3. barrier synchronization
The CUDA programming model assumes a heterogeneous environment consisting of a *host* and at least one *device*.

- **The host** is usually a traditional computer or workstation or a compute node in a parallel cluster.

- A **device** is a GPU, which may be located directly on the motherboard (e.g. integrated graphics) or on an add-on card. Regardless, it is connected via the PCIe bus.

- As already noted, a single host may have access to multiple devices.
Scalability in practice

- An NVIDIA GPU device has one or more *streaming multiprocessors* (SMs) that each execute a block of threads.

- Thread blocks are scheduled on available SMs
The CUDA development environment

- is based on C with some extensions
- has extensive C++ support
- has lots of example code with good documentation
A CUDA installation consists of

- driver
- toolkit (locally installed in /usr/local/cuda-5.0)
  - `nvcc`, the CUDA compiler
  - profiling and debugging tools
  - libraries
- Samples (locally installed in /usr/local/cuda-5.0/samples)
  - lots of demonstration examples
  - almost no documentation
At the host code level there are library routines for:

- memory allocation/deallocation on the device
- data transfer to/from the device memory, including
  - ordinary data
  - constants
  - texture arrays (read-only, useful for look-ups)
- error checking
- timing
Outline

1 CUDA overview
   - Kernels
     - Thread Hierarchy
     - Memory Hierarchy
     - Heterogeneous Programming
     - Compute Capability

2 Programming Interface
   - NVCC and PTX
   - CUDA C runtime
A *kernel* is a function or routine that executes on the GPU device.

Multiple instances of the kernel are run, each carrying out the work of a single thread.

Kernel definitions begin with `__global__` and must be declared to be `void`.

Kernels accept a small number of parameters, usually pointers to locations in device memory and scalar values passed by value.
A first kernel

Suppose we have two vectors (arrays) of size \( N \) and need to add corresponding elements to create a vector containing the sum.

- In C this looks like
  
  ```c
  for (i = 0; i < N; i++)
      C[i] = A[i] + B[i];
  ```

- The corresponding CUDA kernel could be
  
  ```c
  // Kernel definition
  __global__ void VecAdd(float* A, float* B, float* C) {
      int i = threadIdx.x;
      C[i] = A[i] + B[i];
  }
  ```

- Where did the loop go?
- What is threadIdx.x?
Invoking the kernel

- A simple kernel invocation code fragment:

```c
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

- $N$ instances of the kernel will be executed, each with a different value of `threadIdx.x` between 0 and $N - 1$. 

CPS343 (Parallel and HPC) The CUDA Programming Model Spring 2013 13 / 42
Outline

1. **CUDA overview**
   - Kernels
   - Thread Hierarchy
   - Memory Hierarchy
   - Heterogeneous Programming
   - Compute Capability

2. **Programming Interface**
   - NVCC and PTX
   - CUDA C runtime
CUDA defines threadIdx as a C struct (type \textit{dim3}) with three elements; threadIdx.x, threadIdx.y, and threadIdx.z.

- Provides access to a \textit{one}, \textit{two}, or \textit{three}-dimensional thread block.

- Each thread has a unique ID

<table>
<thead>
<tr>
<th>Dim</th>
<th>Thread Index</th>
<th>Block Size</th>
<th>Thread ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>$D_x$</td>
<td>$x$</td>
</tr>
<tr>
<td>2</td>
<td>(x, y)</td>
<td>((D_x, D_y))</td>
<td>$x + yD_x$</td>
</tr>
<tr>
<td>3</td>
<td>(x, y, z)</td>
<td>((D_x, D_y, D_z))</td>
<td>$x + (y + zD_y)D_x$</td>
</tr>
</tbody>
</table>

- The thread ID calculation is the same as the offset from the start of a linear array where $x$ elements are contiguous.
Thread ID diagram

one-dimensional

\[ \begin{array}{cccc}
  & & & \\
  & & & \\
  & & & \\
  \end{array} \]

two-dimensional

\[ \begin{array}{cccc}
  & & & \\
  & & & \\
  & & & \\
  \end{array} \]

three-dimensional

\[ \begin{array}{cccc}
  & & & \\
  & & & \\
  & & & \\
  \end{array} \]
The following code adds two $N \times N$ matrices $A$ and $B$ to produce the $N \times N$ matrix $C$:

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                        float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    ...
    // Invoke kernel with one block of $N \times N \times 1$ threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```
Specifying grid and block dimensions

The simplest form of a kernel invocation is

```c
kernel_name<<<gridDim, blockDim>>>(args,...);
```

where

- `gridDim` is the number of thread blocks that will be executed
- `blockDim` is the number of threads within each block
- `args,...` is a limited number of arguments, usually constants and pointers to arrays in the device memory

Both `gridDim` and `blockDim` can be declared as `int` or `dim3`.

Number of threads in a block is limited; current implementations allow up to 1024. In practice, 256 is often used.
Two-dimensional thread block example

Blocks are organized into a one, two, or three-dimensional grid of thread blocks. Here is a two-dimensional example:
Two-dimensional thread block example

\[ x = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} = 1 \times 4 + 3 = 7 \]
\[ y = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y} = 2 \times 4 + 1 = 9 \]
Thread index example with multiple blocks

This version of the matrix addition code uses $16 \times 16$ blocks, and assumes $N$ is a multiple of 16. Each thread corresponds to a single matrix element.

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                        float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y);
    MatAdd<<<num Blocks, threadsPerBlock>>>(A, B, C);
    ...
}
```
Final thread hierarchy notes

- Thread blocks are required to execute independently.

- Threads within the same block can cooperate using shared memory and synchronization.

- Synchronization is achieved using the `__syncthreads()` function; this produces a *barrier* at which all threads in the block wait before any is allowed to proceed.

- GPU devices are expected to have fast, low-latency shared memory and a lightweight implementation of `__syncthreads()`.
1. CUDA overview
   - Kernels
   - Thread Hierarchy
   - Memory Hierarchy
   - Heterogeneous Programming
   - Compute Capability

2. Programming Interface
   - NVCC and PTX
   - CUDA C runtime
CUDA device memory

CUDA threads may access multiple memory spaces:

- *private local memory* - available only to the thread
- *shared memory* - available to all threads in block
- *global memory* - available to all threads

Two additional read-only global memory spaces optimized for different memory access patterns:

- *constant memory*
- *texture memory*

The global, constant, and texture memory address spaces are persistent across kernel launches by the same application.
Memory Hierarchy

- Thread
- Per-thread local memory
- Thread Block
- Per-block shared memory
- Grid 0
- Global memory
- Grid 1
Outline

1. CUDA overview
   - Kernels
   - Thread Hierarchy
   - Memory Hierarchy
   - Heterogeneous Programming
   - Compute Capability

2. Programming Interface
   - NVCC and PTX
   - CUDA C runtime
Heterogeneous Programming: *Host* and *device*

- host and device have different memory spaces
- main program runs on host
- host is responsible for memory transfer to/from device
- host launches kernels
- device executes kernels
- host executes all remaining code
Heterogeneous Programming

The CUDA Programming Model
Outline

1. CUDA overview
   - Kernels
   - Thread Hierarchy
   - Memory Hierarchy
   - Heterogeneous Programming
   - Compute Capability

2. Programming Interface
   - NVCC and PTX
   - CUDA C runtime
Each CUDA device has a compute capability in the form major.minor.

major is a integer that refers to the device architecture: 1 for Tesla, 2 for Fermi, and 3 for Kepler.

minor is an integer that corresponds to incremental improvements over the core architecture.

NVIDIA’s list of CUDA devices with their compute capability can be found at http://developer.nvidia.com/cuda-gpus

The features available for each compute capability can be found in the CUDA C Programming Guide

Workstations have Quadra 2000 devices, compute capability 2.1

LittleFe nodes have ION devices, compute capability 1.2
Outline

1 CUDA overview
   - Kernels
   - Thread Hierarchy
   - Memory Hierarchy
   - Heterogeneous Programming
   - Compute Capability

2 Programming Interface
   - NVCC and PTX
   - CUDA C runtime
Compiling with NVCC

NVCC is a compiler driver. In its usual mode of operation it

1. compiles the device code into PTX (device assembly code) and/or binary form,

2. modifies host code, replacing the `<<<...>>>` syntax by the necessary CUDA C runtime function calls, and

3. invokes the native C or C++ compiler to compile and link the modified host code.

At runtime any PTX code is compiled by the device driver at runtime (just-in-time compilation).

This slows application load time, but allows for performance improvements due to updated drivers and/or device hardware.
Outline

1. CUDA overview
   - Kernels
   - Thread Hierarchy
   - Memory Hierarchy
   - Heterogeneous Programming
   - Compute Capability

2. Programming Interface
   - NVCC and PTX
   - CUDA C runtime
Initialization

- No explicit initialization required
- Done automatically when first runtime function is called
- Creates a new context for each device (opaque to user applications)
Using device memory

- Recall that there are several types of device memory available to all threads in an application: *global*, *constant*, and *texture*.

- Device memory is allocated as either a *linear array* or a *CUDA array* (used for texture fetching).

- Global memory is typically allocated with `cudaMalloc()` and released with `cudaFree()`, both called from the host program.

- Data is transferred between the host and the device using `cudaMemcpy()`.

- Example: Add two vectors to produce third vector...
/* Kernel that executes on CUDA device */
__global__ void add_vectors(float *c, float *a, float *b, int n)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) c[idx] = a[idx] + b[idx];
}
int main(int argc, char* argv[])
{
    const int n = 10;
    size_t size = n * sizeof(float);

    int num_blocks; /* number of blocks */
    int block_size; /* threads per block */
    int i; /* counter */

    float *a_h, *b_h, *c_h; /* ptrs to host memory */
    float *a_d, *b_d, *c_d; /* ptrs to device memory */
Vector addition example: Memory allocation

/* allocate memory for arrays on host */
float *a_h = malloc(size);
float *b_h = malloc(size);
float *c_h = malloc(size);

/* allocate memory for arrays on device */
cudaMalloc((void**)&a_d, size);
cudaMalloc((void**)&b_d, size);
cudaMalloc((void**)&c_d, size);
/* initialize arrays and copy them to device */
for (i = 0; i < n; i++)
{
    a_h[i] = 1.0 * i;
    b_h[i] = 100.0 * i;
}
cudamemcpy(a_d, a_h, size, cudamemcpyHostToDevice);
cudamemcpy(b_d, b_h, size, cudamemcpyHostToDevice);
/* do calculation on device */
block_size = 256;
num_blocks = (n + block_size - 1) / block_size;
add_vectors<<<num_blocks, block_size>>>(c_d, a_d, b_d, n);
/* retrieve result from device and store on host */
cudamemcpy(c_h, c_d, size, cudamemcpyDeviceToHost);
/* print results */
for (i = 0; i < n; i++)
{
    printf("%8.2f + %8.2f = %8.2f\n",
            a_h[i], b_h[i], c_h[i]);
}

/* cleanup and quit */
free(a_h); free(b_h); free(c_h);
cudaFree(a_d); cudaFree(b_d); cudaFree(c_d);

return 0;
}
The CUDA C Programming Guide contains information on these and other topics, including:

- Page-locked host memory: provides non-paged host memory that can be mapped to device memory to permit asynchronous host-device memory transfers.
- Asynchronous concurrent execution: Kernel launches and certain other CUDA runtime commands provide for concurrent host and device execution.
- Multi-device system: multiple GPU devices may be present. Kernels are launched on the current device. The current device can be changed using the cudaMemcpyDevice() function.
- Error checking: All runtime functions return an error code.