CUDA SHARED MEMORY

NVIDIA Corporation
Difference between host and device

- **Host**: CPU
- **Device**: GPU

Using `__global__` to declare a function as device code

- Executes on the device
- Called from the host (or possibly from other device code)

Passing parameters from host code to a device function
REVIEW (2 OF 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch \(N\) copies of `add()` with `add<<<N,1>>>(...);
  - Use `blockIdx.x` to access block index
1D STENCIL

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:
IMPLEMENTING WITHIN A BLOCK

- Each thread processes one output element
  - `blockDim.x` elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times
SHARING DATA BETWEEN THREADS

- Terminology: within a block, threads share data via shared memory

- Extremely fast on-chip memory, user-managed

- Declare using __shared__, allocated per block

- Data is not visible to threads in other blocks
IMPLEMENTING WITH SHARED MEMORY

- Cache data in shared memory
  - Read \((\text{blockDim} \cdot x + 2 \times \text{radius})\) input elements from global memory to shared memory
  - Compute \(\text{blockDim} \cdot x\) output elements
  - Write \(\text{blockDim} \cdot x\) output elements to global memory
- Each block needs a halo of \(\text{radius}\) elements at each boundary

\[\text{halo on left} \quad \rightarrow \quad \text{blockDim} \cdot x \text{ output elements} \quad \rightarrow \quad \text{halo on right}\]
__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];
    }
STENCIL KERNEL

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

// Store the result
out[gindex] = result;
}
The stencil example will not work...

Suppose thread 15 reads the halo before thread 0 has fetched:

```c
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    int result = 0;
    result += temp[lindex + 1];
```

- **Store at temp[18]**
- **Load from temp[19]**
- **Skipped, threadIdx > RADIUS**
void __syncthreads();

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block
__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();
}
STENCIL KERNEL

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

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

- Use __shared__ to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

- Use __syncthreads() as a barrier
  - Use to prevent data hazards
LOOKING FORWARD

**Cooperative Groups:** a flexible model for synchronization and communication within groups of threads.

At a glance

- Scalable Cooperation among groups of threads
- Flexible parallel decompositions
- Composition across software boundaries
- Deploy Everywhere

Benefits **all** applications

Examples include:
- Persistent RNNs
- Physics
- Search Algorithms
- Sorting
FOR EXAMPLE: THREAD BLOCK

Implicit group of all the threads in the launched thread block

Implements the same interface as `thread_group`:

```c
void sync(); // Synchronize the threads in the group
unsigned size(); // Total number of threads in the group
unsigned thread_rank(); // Rank of the calling thread within [0, size)
bool is_valid(); // Whether the group violated any API constraints
```

And additional `thread_block` specific functions:

```c
dim3 group_index(); // 3-dimensional block index within the grid
dim3 thread_index(); // 3-dimensional thread index within the block
```
NARROWING THE SHARED MEMORY GAP

with the GV100 L1 cache

**Cache:** vs shared
- Easier to use
- 90%+ as good

**Shared:** vs cache
- Faster atomics
- More banks
- More predictable

Directed testing: shared in global

- Pascal: 70% benefit
- Volta: 93% benefit

Average Shared Memory Benefit
FUTURE SESSIONS

- CUDA GPU architecture and basic optimizations
- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups
FURTHER STUDY

- Shared memory:

- CUDA Programming Guide:

- CUDA Documentation:
  - https://docs.nvidia.com/cuda/index.html
HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming
QUESTIONS?