McGill

# ECSE 420
# CUDA

TA: Loren Lugosch

November 2016

# Outline

- CUDA
- Lab 3

# CUDA

- CUDA is a shared-memory/data-parallel programming model for computers with GPUs
- CUDA C = C with extensions
- Compile using `nvcc`
- If you don’t have an NVIDIA GPU, you can emulate using gpuOcelot

# CUDA

- A typical CUDA program looks like this:

1) CPU allocates storage on GPU
(`cudaMalloc()`)

2) CPU copies input data from CPU → GPU
(`cudaMemcpy()`)

3) CPU launches kernel(s) on GPU to process the data

4) CPU copies the results from GPU → CPU
(`cudaMemcpy()`)

# CUDA

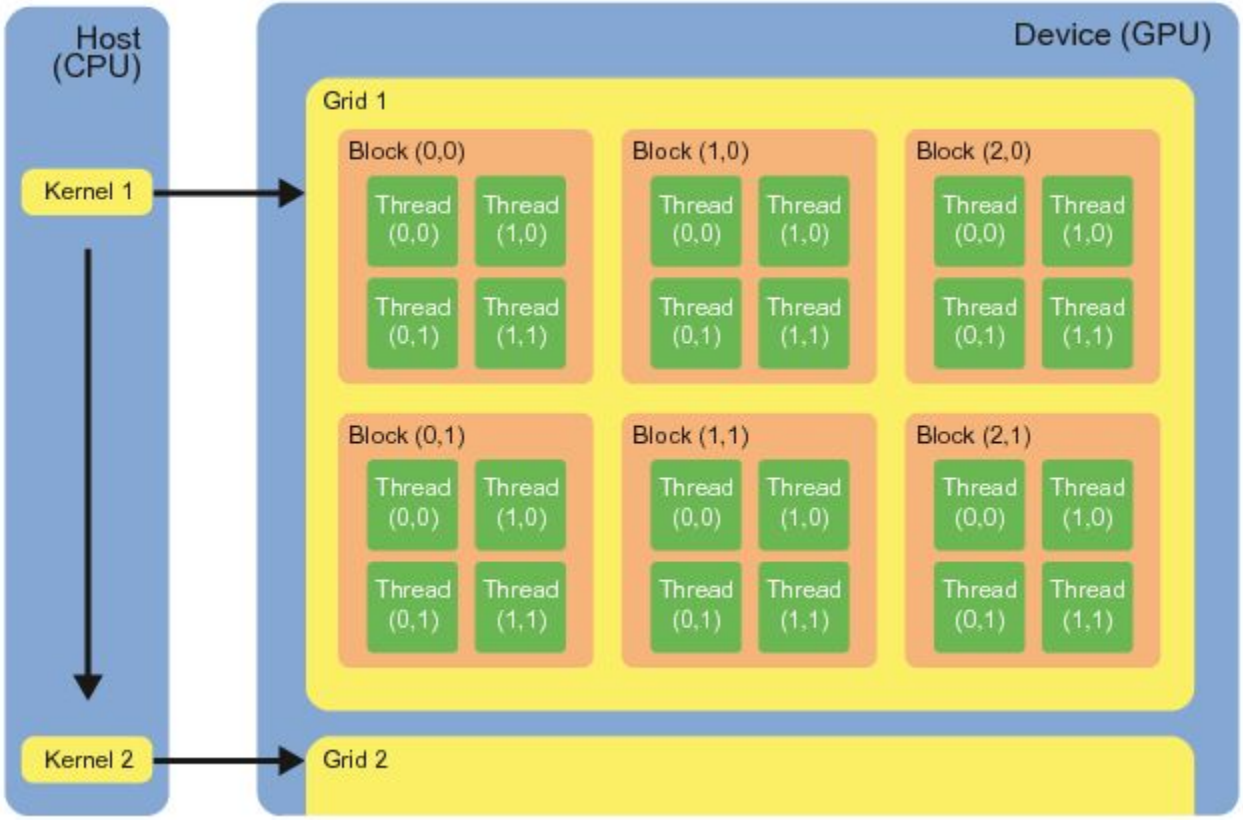
- Example: squaring every element of an array using the GPU

# CUDA

- **Kernels** run on **threads**
- **Blocks** are composed of many threads
- All threads in a block run in parallel
- Blocks are mapped to “streaming multiprocessors”
- A block is scheduled as soon as a multiprocessor is available
- Blocks run in parallel on as many multiprocessors as are available

# CUDA

- A **grid** is a collection of thread blocks
- Each kernel has its own grid



# CUDA

- Grids and blocks are 3-dimensional
- The size of the grid is set using the arguments of the kernel launch, which are either:
  - `int`
  - `dim3`
- Example:

# CUDA

- Use lots of threads!
- **Note:** There is a maximum number of threads which can run in a block (usually 512 or 1024)
- But there is no maximum number of blocks
- One strategy: use blocks of maximum size, then use as many blocks as needed to give each input its own thread
- Another strategy: use blocks of a size which makes array index calculations convenient

# CUDA

- Cheat sheet for indexing with different block/grid dimensions:
  - http://www.martinpeniak.com/index.php?option=com_content&view=article&catid=17:updates&id=288:cuda-thread-indexing-explained
- Note that functions in this cheat sheet have `__device__` identifier
  - Functions on GPU called from CPU are declared using `__global__`
  - Functions on GPU called from GPU are declared using `__device__`

# CUDA

- `__syncthreads()` creates a barrier within a block of threads
  - Similar to the functionality of `pthread_barrier_wait()` in Pthreads, `#pragma omp barrier` in OpenMP, `MPI_Barrier()` in MPI
- Need barrier after a write, if other threads use the written values
- Example: shifting an array with and without synchronization

# CUDA

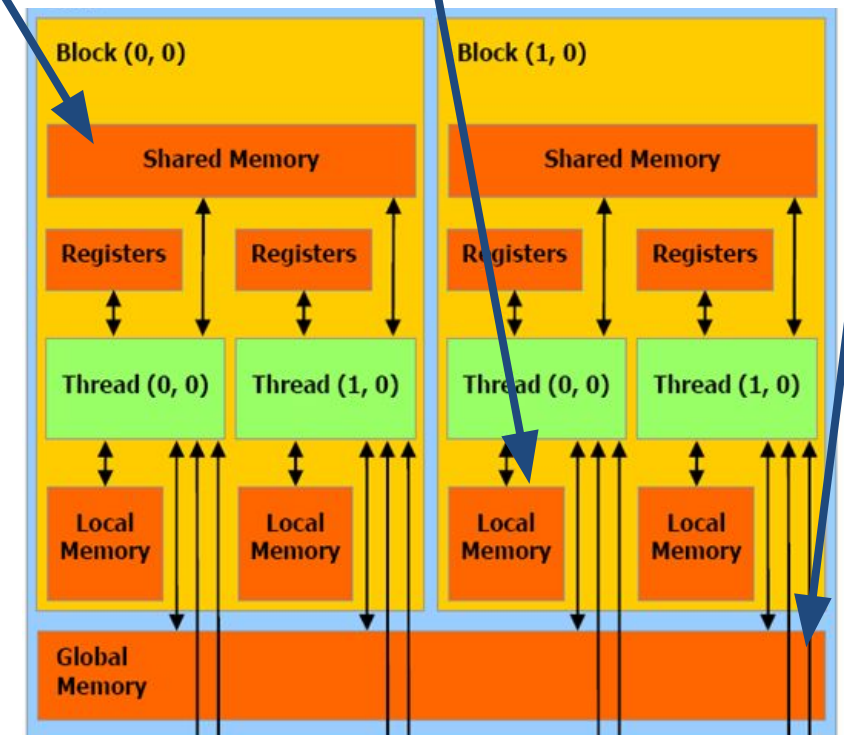
- `__syncthreads()` does not synchronize across blocks!
- But kernel calls do not return until all blocks in the grid have finished
  → Waiting for a kernel to complete can be used for synchronization between blocks of threads

# CUDA

- Atomic memory operations:
  - atomicAdd()
  - atomicMin()
  - atomicCAS()
- Example:

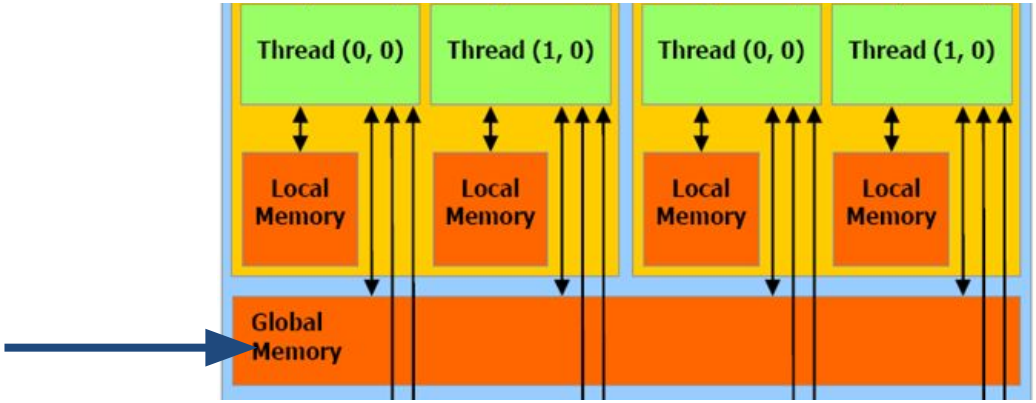
# CUDA

- Three main types of memory: global memory, shared memory, local memory



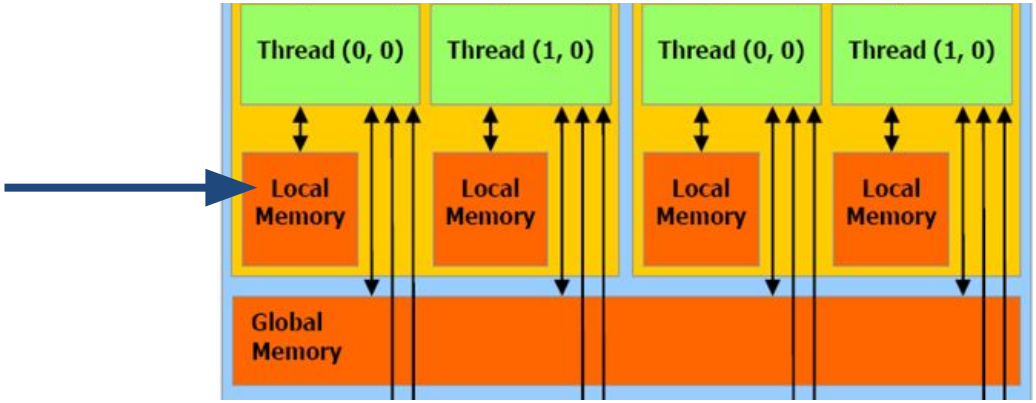
# CUDA

- Global memory
  - Accessible by all threads in all blocks
  - Variables declared in host code and allocated using `cudaMalloc()`



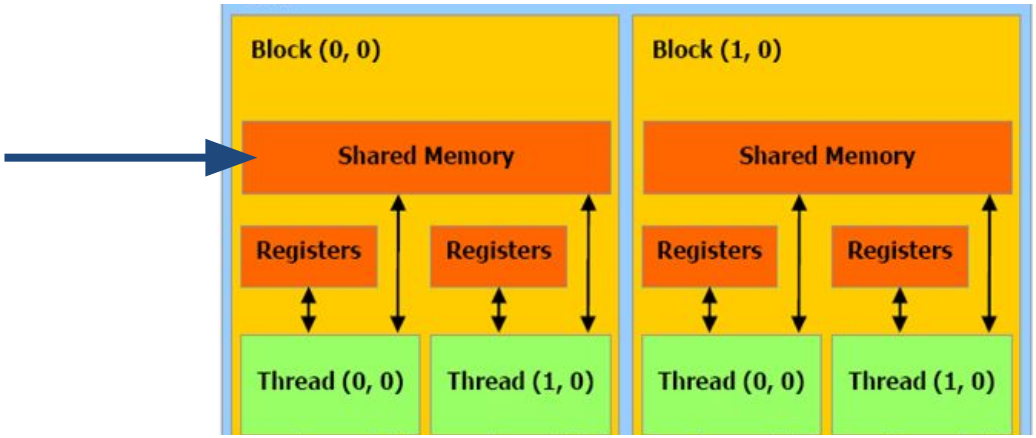
# CUDA

- Local memory
  - Accessible only by one thread
  - Local memory is actually just global memory
  - Variables declared in a kernel without the `__shared__` identifier (but sometimes these variables will end up in registers instead)



# CUDA

- Shared memory
  - Accessible by all threads in the same block
  - On-chip, and thus faster than global memory or local memory
  - Variables declared in a kernel with the `__shared__` identifier



# CUDA

- To statically allocate shared memory, just use `__shared__` identifier
- To dynamically allocate shared memory,
  - add third argument (number of bytes for shared memory) to the kernel launch
  - add `extern` to shared memory variable declaration
- Copying global memory into shared memory can make kernel faster
- But remember to `__syncthreads()` after copying into shared memory

# CUDA

- Use `nvidia-smi` to check state of GPU, processes using the GPU, memory utilization, temperature, etc.

# Lab 3

- Redo everything from Lab 1 and Lab 2 in CUDA!
- To help you get started, let's do rectify