CISC 372: Parallel Computing
CUDA, part 1

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

CUDA Overview

- graphical processors (GPUs) are massively parallel machines
  - very different architecture from CPUs
  - multiple "streaming multiprocessors"
  - designed for doing large vector computations in parallel
  - not so good for algorithms with complex logic and branching
- ~ 2007: researchers realized that GPUs could be used for purposes other than graphics
  - birth of the GPGPU: general purpose GPU
- what was missing was a nice general purpose programming language targeting GPGPUs
- NVIDIA developed CUDA-C to fill this niche
  - based on C/C++
  - a few new language primitives; runtime library
  - for writing heterogeneous programs: mix of GPU and CPU code
  - kernel: a compute-intensive function to be run on the GPU
  - primitives for launching kernels, copying data between CPU and GPU
  - many algorithms can see orders of magnitude performance improvements over CPU

NVIDIA Tesla K80

K80 properties

```
Device name: Tesla K80
Compute capability: 3.7
Number of SMPs: 13
Max threads per block: 1024
Registers per block: 65536
Warp size: 32
Total global memory: 11996954624
Total constant memory: 65536
Shared memory per block: 49152
Memory Clock Rate (KHz): 2505000
Memory Bus Width (bits): 384
Peak Memory Bandwidth (GB/s): 240.480000
```

Price: $\sim \$500$

CUDA Background

- CUDA C is an extension of C for writing programs targeting NVIDIA's GPUs
- goal is to use GPUs for general purpose computing
- introduced in 2007, updated regularly
- some scientific problems can see enormous performance gains
- see `https://developer.nvidia.com/about-cuda`

References:

- the CUDA Programming Guide
  - in our repo under `docs/`
  - `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`
- CUDA by Example
  - `https://developer.nvidia.com/cuda-example`
  - pay for the book, examples are free

CUDA Programming Model

- program execution starts as a single thread as usual
- a *kernel* function is a function declared with specifier `__global__`
- invoking a kernel function instantiates a grid executing on a device
- the grid consists of a collection of blocks
  - organized in a 1d, 2d, or 3d Cartesian geometry
- each block consists of a collection of threads
  - organized in a 1d, 2d, or 3d Cartesian geometry
- the blocks must be completely independent
  - they can execute concurrently or sequentially or anything in between
  - in any order
- the threads within a block execute concurrenly and may coordinate
  - barriers
  - shared memory (shared by all threads in the block)

hello1.cu: Hello, world

```
#include <stdio.h>

__global__ void kernel(void) {
  printf("Hello from the GPU!\n");
}

int main (void) {
  kernel<<<1,1>>>();  // launch kernel with 1 block, 1 thread per block
  printf("Hello from the CPU!\n");
  cudaDeviceSynchronize();  // wait for kernel to return
}
```

- `__global__` indicates a function is a kernel: to be run on GPU
- `f<<<blocks,threadsPerBlock>>>(...)`
  - launch the kernel with `blocks` blocks and `threadsPerBlock` threads per block
  - returns immediately as kernel runs concurrently on GPU

Compiling and running a CUDA program on Beowulf

```
siegel@grendel:~/372/code/src/cuda/hello$ nvcc -o hello1.exec hello1.cu
siegel@grendel:~/372/code/src/cuda/hello$ srun  -n 1 --gres=gpu:1 ./hello1.exec
srun: job 172804 queued and waiting for resources
srun: job 172804 has been allocated resources
Hello from the CPU!
Hello from the GPU!
siegel@grendel:~/372/code/src/cuda/hello$
```

- `nvcc`: similar to `cc`, different options
- `--gres=gpu:1` requests one GPU (in addition to the one CPU core)

hello2.cu: multiple blocks, threads per block

```
#include <stdio.h>

__global__ void kernel(void) {
    int bid = blockIdx.x; // block ID number
    int tid = threadIdx.x; // thread ID number (within its block)
    printf("Hello from block %d, thread %d of the GPU\n", bid, tid);
}

int main (void) {
    kernel<<<3,4>>>(); // 3 blocks, 4 threads per block
    printf("Hello, World\n");
    cudaDeviceSynchronize();
}
```

Output of hello2.cu

```
$ nvcc  -o hello2.exec hello2.cu
$ srun --unbuffered -n 1 --gres=gpu:1 ./hello2.exec
srun: job 172815 queued and waiting for resources
srun: job 172815 has been allocated resources
Hello, World
Hello from block 0, thread 0 of the GPU
Hello from block 0, thread 1 of the GPU
Hello from block 0, thread 2 of the GPU
Hello from block 0, thread 3 of the GPU
Hello from block 2, thread 0 of the GPU
Hello from block 2, thread 1 of the GPU
Hello from block 2, thread 2 of the GPU
Hello from block 2, thread 3 of the GPU
Hello from block 1, thread 0 of the GPU
Hello from block 1, thread 1 of the GPU
Hello from block 1, thread 2 of the GPU
Hello from block 1, thread 3 of the GPU
```

Compiling and running CUDA programs on Bridges-2

Create a batch script like this:

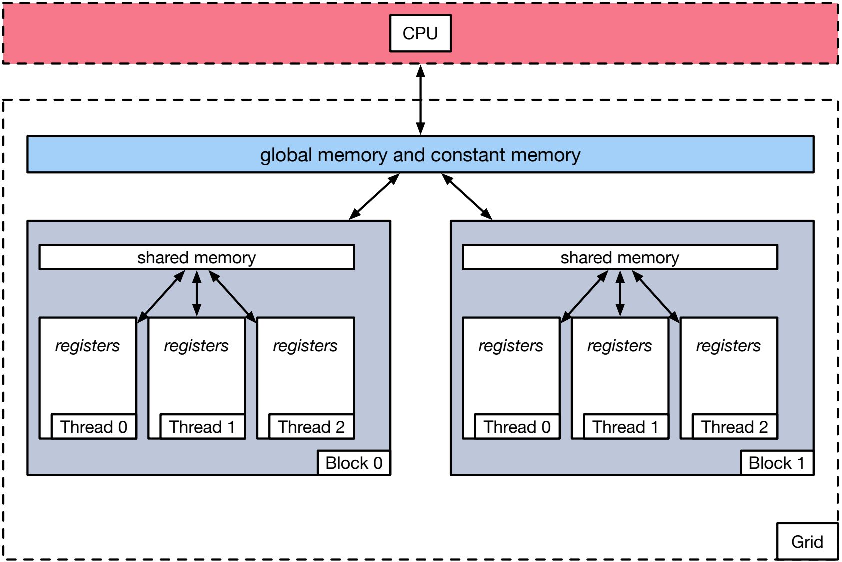
```
#!/bin/bash
#SBATCH -p GPU-shared
#SBATCH -t 00:01:00
#SBATCH -N 1
#SBATCH --gpus=1
# echo commands to stdout
set -x
./hello1.exec
```

- GPU partitions: `GPU-shared`, `GPU-small`, `GPU`
  - you will get charged for a full node (all CPUs and GPUs) unless you use `GPU-shared`
- each GPU node has multiple CPUs and GPUs
  - 8 NVIDIA Tesla V100-32GB SXM2 GPUs
  - 2 Intel Xeon Gold 6248 “Cascade Lake” CPUs: 40 cores total; 2.50–3.90 GHz
  - `https://portal.xsede.org/psc-bridges-2`: *GPU Nodes*, *GPU & GPU-shared partitions*

CUDA C Language Elements: Function type qualifiers

- `__device__`: executed on device, callable from device only
- `__global__`: kernel, executed on device, callable from host
  - function must return `void`
  - callable from device for compute quality $> 3.x$
  - calls must specify execution configuration
  - asynchronous
- `__host__`: executed on host, callable on host only
  - default
  - may be used with `__device__`
    - function is duplicated
    - use preprocessor macro `__CUDA_ARCH__` in body to include code that may be just for device or just for host version of function; if this is defined you are in CUDA version, else host version

CUDA memory hierarchy



Variable type qualifiers

- `__device__`
  - may be used in conjunction with `__constant__` or `__shared__`
  - when alone, variable resides in global memory space on device
  - has lifetime of application
  - is accessible from all threads within the grid
  - is accessible from host through library functions
  - may be used with `__managed__` to be directly referenced from host code
- `__constant__`
  - variable resides on device in constant memory
  - has lifetime of application
  - accessible from all threads in grid
  - accessible from host through library functions
- `__shared__`
  - variable resides in shared memory space of one block
  - has lifetime of the block
  - only accessible by all threads in that block
- nothing: thread-local variable