1 Lecture 17

Monitoring GPU usage on Pearson and Lipschitz:

- `ssh username@pearson.ucdavis.edu nvidia-smi -l`
- `ssh username@lipschitz.ucdavis.edu nvidia-smi -l`

Who's online?

- `finger`
- `who`

Arrays of Threads: The kernel launches a grid of thread blocks.

- Threads in different blocks **cannot** cooperate with other blocks.
- Threads within blocks can sync and share memory. This allows for GPUs to transparently scale.

Kernel Memory Access:

- Per-Thread: A. Registers a. VERY fast b. very small amount of memory B. Local thread memory a. Off-chip b. uncached
- Per-Block: Shared Block memory (still fast–not register fast)
- Per-device: Global device memory (accessed from any block, off-chip, large)

Types:

- int var (thread-scoped, in-register, fast)
- int array_var[10] (thread-scoped, local memory)
- `__shared__` int (block-scoped, cached)
- `__device__` (device-scoped, globally-scoped)
- `__constant__` int (constant memory, fast)

Note: We take a 100x penalty for using global variables (int array_var, `__device__`)

1.1 How many variables do we store?

- 100K's of per-thread variables, R/W by only that one thread
- 100s of shared variables, R/W by hundreds of threads
- 1 global/constant variable, 100K's of R/W's

1.2 When are GPUs good?

- Numerical Integration (give it tons of points, calculate a function at each point, return function value)
- MCMC where each iteration is **very** slow.
- Simple bootstraps (calculate estimator on each subset in parallel)
- Particle Filtering / Sequential Monte Carlo (take a parameter, get samples for that parameter all at once, repeat on other parameters–somewhat parallel)
- Brute force optimization and grid search
- Large matrix calculations (if you're a ninja)
- When you don't care if other people can read your code

1.3 When are GPUs a poor choice?

- Fast iteration MCMC
- "Difficult" bootstraps (where the estimator is difficult because of too much data)
- Sequential optimization problems
- Methodological work (GPUs lack portability)

In addition to writing CUDA C, we can write in Python or R, and use a binding to PyCUDA or RCUDA

1.4 RCUDA

- Full bindings to the NVIDIA CUDA API for R (mechanism to call any function within the CUDA API from within R)
- Have to write the kernel in C, then compile an intermediate representation using NVCC `nvcc --ptx`.
- As with most C calls from R, you have to wrap your C code with an extern, like so: `extern "C" { put code here }`
- To load in R, assign `loadModule("location/to/your/kernel.ptx")` to a variable (ex: `m`)
- All kernels written in the kernel.ptx file will be in the variable m, so assign them to the default environment for ease.

Accessing memory:

- Copying memory to the GPU: `mem = copyToDevice(x)`
- Call CUDA code: `.cuda(kernel, kernelArguments, gridDim, blockDim)`
- Copy from GPU to Host: `cu_ret = copyFromDevice()` or `cu_ret = mem[]`

Note:

- GPUs work in single-precision floating points! R works with double precision. If you want double precision, GPUs are not a great choice.
- Grid dimensions **need** to be make integers, ie: `dim=c(100L, 1L, 5L)`.

1.5 RNG on GPU

- You have to be careful about the state of the RNG. Setting the RNG seed is vital, within each thread.
- Solution: Set up each random number state in each thread
- More tricky: Set up the states outside of the thread and malloc them over to the device.

1.6 General Algorithm for GPUs

1. Copy memory from CPU to GPU
2. Run the code on the GPU
3. Copy Results from GPU to CPU

1.7 Thrust of the Homework

1. Write a Kernel to generate truncated random normals (gene)

   a. Generate regular normal and use acceptance-rejection algorithm (simple to write, inefficient algorithm)
   b. qnorm/pnorm (not in CUDA standard library or cuRAND)

2. Call from R, do tests and timings of truncated normals.
3. Probit MCMC

   - $y_i | z_i = I_{\{z_i > 0\}}, z_i | \beta \sim N(x_i^T \beta, 1)$
   - EM: Want to find $argmax_\beta P(y|\beta) = \int p(y|z) p(z|\beta) dz$
   - Prior on β: $\beta \sim N(\beta_0, \Sigma_0)$
   - Sample from $p(\beta|y)$ using Gibbs
   - Then $P(\beta|z,y) \sim Normal$ and $P(z|\beta,y) \sim TruncNormal$