Michael Bissell

STA 250

Advanced Statistical Computing

Assignment 4

GPU Computing

1) Implement a kernel to obtain samples from a truncated normal random variable

a) Write a kernel in CUDA C to obtain samples from a truncated normal random variable of the form:

$$X \sim TN(\mu, \sigma^2; (a, b)) \equiv N(\mu, \sigma^2)1_{\{X \in (a,b)\}}$$

Answer: See code appendices at end of report.

b) Compile your CUDA kernel using nvcc and check it can be launched properly

Answer: If only I could count the number of times I had to compile and re-compile this kernel... See below.

c) Sample 10,000 random variables from TN(2,1;(0,1.5)), and verify the expected value (roughly) matches the theoretical value (see class notes for details).

Answer: From the class notes, Lecture 13, if $W \sim TN(\mu, \sigma; (a, b))$, then the expected value of W is:

$$\mathbb{E}[W] = \mu + \sigma \frac{\phi\left(\frac{a-\mu}{\sigma}\right) - \phi\left(\frac{b-\mu}{\sigma}\right)}{\Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\left(\frac{a-\mu}{\sigma}\right)} = 0.9570.$$

The observed mean from the 10,000 samples is 0.958, which is extremely close to the theoretical value and the density plots virtually completely overlap when plotted as vertical lines, as shown in Figure 1 below.

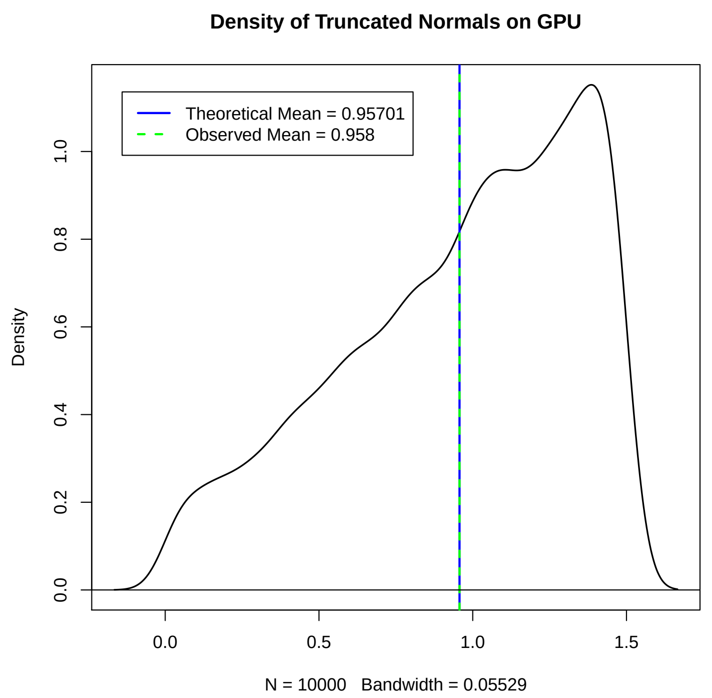

Figure 1: Density Plot of 10,000 Truncated Normals

d) Write an R function for sampling truncated normal random variables (possibly using a different algorithm). You may also use the code provide in the GitHub repo. Sample 10,000 random variables from this function and verify the mean (roughly) matches the theoretical values.

Answer: We simply use rtruncnorm() function in the R package truncnorm. The observed mean from the 10,000 samples is 0.94819, which is extremely close to the theoretical value of 0.95701 and the density plots virtually completely overlap when plotted as vertical lines, as shown in Figure 2 below.

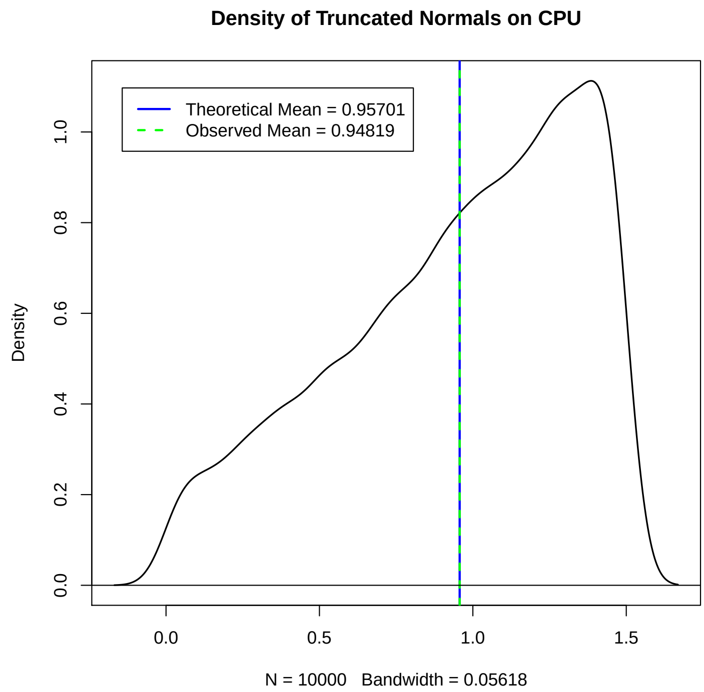

Figure 2: Density Plot of 10,000 Truncated Normals

e) Time your RCUDA function and pure R function for $n = 10^k$ for $k = 1, 2, \ldots, 8$. Plot the total runtimes for both functions on the y-axis as a function of n (on the log-scale as the x-axis). At what point did/do you expect the GPU function to outperform the non-GPU function? You may also want to decompose the GPU runtimes into copy to/kernel/copy back times for further detailed analysis.

Answer:

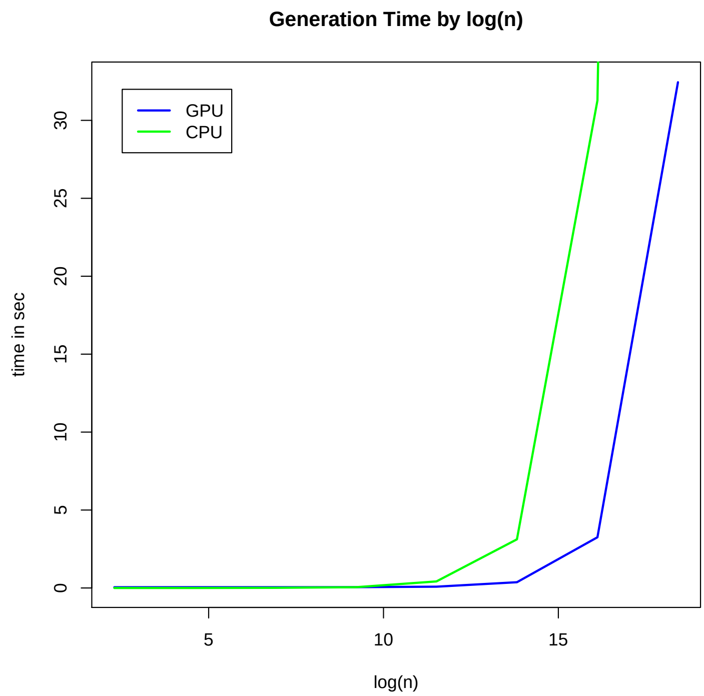

Figure 3: Generation Time by log(n)

	n	copyToDevice	kernel	copyFromDevice
1	10	0.00	0.04	0.00
2	100	0.00	0.05	0.00
3	1000	0.00	0.04	0.00
4	10000	0.00	0.05	0.00
5	100000	0.01	0.08	0.00
6	1000000	0.03	0.33	0.01
7	10000000	0.26	2.92	0.07
8	100000000	2.64	29.15	0.66

f) Verify that both your GPU and CPU code work for $a = -\infty$ and/or $b = +\infty$.

Answer: We use $N(0,1) = TN(0,1;(-\infty,\infty))$, where we directly initialize the vectors 'lo' and 'hi' with values '-Inf' and 'Inf', which are correctly handled by R and RCUDA. The theoretical mean is obviously zero by symmetry and the observed mean from the 10,000 samples is -0.00147, which is extremely close and the density plots virtually completely overlap when plotted as vertical lines, as shown in Figure 4 below. This along with part g) below, demonstrates that the code *appears* to works correctly.

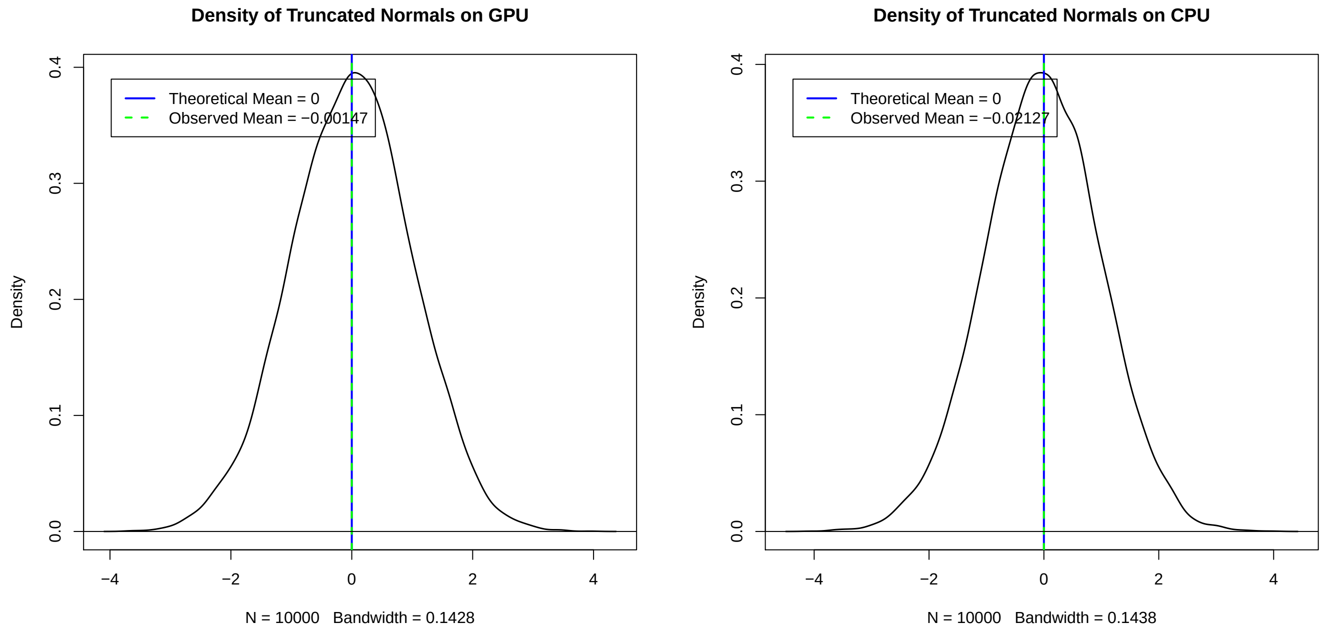

Figure 4: Density Plot of 10,000 Truncated Normals on GPU and CPU

g) Verify that both your GPU and CPU code work for truncation regions in the tail of the distribution e.g., $a = -\infty, b = -10, \mu = 0, \sigma = 1$.

Answer: From the class notes, Lecture 13, if $U \sim TN(\mu, \sigma; (-\infty, b))$, then the expected value of U is:

$$\mathbb{E}[U] = \mu - \sigma \frac{\phi\left(\frac{b-\mu}{\sigma}\right)}{\Phi\left(\frac{b-\mu}{\sigma}\right)} = -10.09809.$$

The observed mean from the 10,000 samples is 10.1, which is extremely close to the theoretical value and the density plots virtually completely overlap when plotted as vertical lines, as shown in Figure 5 below.

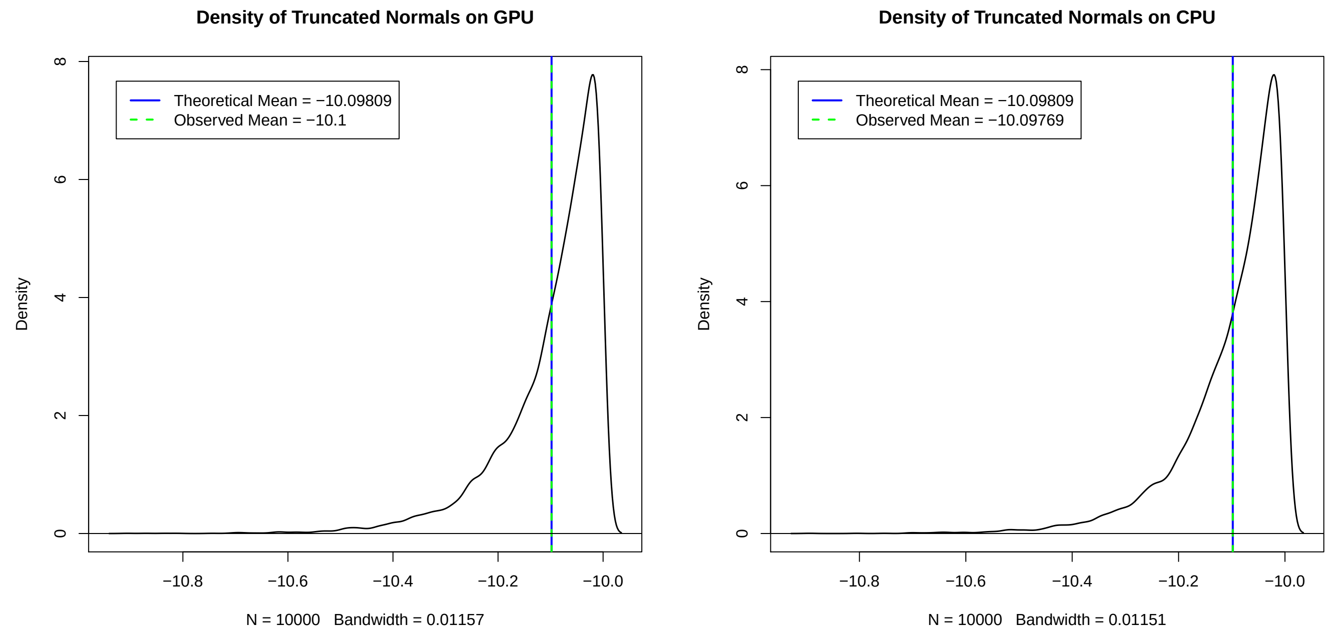

Figure 5: Density Plot of 10,000 Truncated Normals on GPU and CPU

---

2) In this question you will implement Probit MCMC i.e., fitting a Bayesian Probit regression model using MCMC. This model turns out to be computationally nice and simple, lending itself to a Gibbs sampling algorithm with each distribution available in sample-able form. The model is as follows:

$$Y_i|Z_i \sim 1_{\{Z_i>0\}}$$

$$Z_i|\beta \sim N(x_i^T\beta, 1)$$

$$\beta \sim N(\beta_0, \Sigma_0),$$

where β_0 is a $p \times 1$ vector corresponding to the prior mean, and Σ_0^{-1} is the prior precision matrix. Note that for convenience we supply Σ_0^{-1} as an argument to the probit MCMC function, to allow for flat priors for β.

Answer: What we want are the estimates of β's and thus we derive the posterior distribution of the β's and take their means as the estimates. We have:

$$\pi(\beta) \propto \exp\left\{-\frac{1}{2}(\beta-\beta_0)^T \Sigma_0^{-1} (\beta-\beta_0)\right\}$$

$$p(\vec{z}|x_i,\beta) \propto \prod_{i=1}^{n} p(z_i|x_i,\beta) \propto \exp\left\{-\frac{1}{2}\sum_{i=1}^{n}\left(z_i - x_i^T\beta\right)^2\right\} \propto p(Z|X,\beta) \propto \exp\left\{-\frac{1}{2}(Z-X\beta)^2\right\}$$

$$p(Z|X,Y,\beta) \sim \begin{cases} TN_p(X\beta, I_p; [0,\infty)) & y=1 \\ TN_p(X\beta, I_p; (-\infty, 0]) & y=0 \end{cases}$$

Thus to get estimates of our parameters β given the data, we find the posterior distribution of $\beta|X,Y,Z$ as follows:

$$p(\beta|X,Y,Z) \propto \pi(\beta)p(Z|X,Y,\beta) \propto \exp\left\{-\frac{1}{2}\left[(\beta-\beta_0)^T \Sigma_0^{-1} (\beta-\beta_0) + (Z-X\beta)^2\right]\right\}$$

$$\propto \exp\left\{-\frac{1}{2}\left[(\beta-\beta_0)^T\Sigma_0^{-1}(\beta-\beta_0)+(Z-X\beta)^T(Z-X\beta)\right]\right\}$$
$$\propto \exp\left\{-\frac{1}{2}\left[\beta^T\Sigma_0^{-1}\beta-\beta^T\Sigma_0^{-1}\beta_0-\beta_0^T\Sigma_0^{-1}\beta+\beta_0^T\Sigma_0^{-1}\beta_0+Z^TZ-Z^TX\beta-\beta^TX^TZ+\beta^TX^TX\beta\right]\right\}$$
$$\propto \exp\left\{-\frac{1}{2}\left[\beta^T\Sigma_0^{-1}\beta-\beta^T\Sigma_0^{-1}\beta_0-\beta_0^T\Sigma_0^{-1}\beta-Z^TX\beta-\beta^TX^TZ+\beta^TX^TX\beta\right]\right\}$$
$$\propto \exp\left\{-\frac{1}{2}\left[\beta^T\left(\Sigma_0^{-1}+X^TX\right)\beta-\beta^T\left(\Sigma_0^{-1}\beta_0+X^TZ\right)-\left(\beta_0^T\Sigma_0^{-1}+Z^TX\right)\beta\right]\right\}$$

From here we can identify the posterior multivariate normal mean and variance by comparing to standard quadratic forms. For a standard quadratic form, we would have:

$$\propto \exp\left\{-\frac{1}{2}\left[(X-\mu)^T\Sigma^{-1}(X-\mu)\right]\right\}\propto \exp\left\{-\frac{1}{2}\left[X^T\Sigma^{-1}X-X^T\Sigma^{-1}\mu-\mu^T\Sigma^{-1}X+\mu^T\Sigma^{-1}\mu\right]\right\}$$

By equating the first term in the posterior derivation above and the first term in the standard form above, we identify $X^T\Sigma^{-1}X=\beta^T\left(\Sigma_0^{-1}+X^TX\right)\beta$. Thus the posterior variance is $\left(\Sigma_0^{-1}+X^TX\right)$. Similarly, equating the second terms, we see that $X^T\Sigma^{-1}\mu=\beta^T\left(\Sigma_0^{-1}+X^TX\right)$ giving $\Sigma^{-1}\mu=\left(\Sigma_0^{-1}+X^TX\right) \implies \left(\Sigma_0^{-1}+X^TX\right)\mu=\left(\Sigma_0^{-1}+X^TX\right)$

$\implies \mu=\left(\Sigma_0^{-1}+X^TX\right)^{-1}\left(\Sigma_0^{-1}\beta_0+X^TZ\right)$

Thus we have:

$$p(\beta|X,Y,Z)\propto N_p\Big(\ \left(\Sigma_0^{-1}+X^TX\right)^{-1}\left(\Sigma_0^{-1}\beta_0+X^TZ\right)\ ,\ \left(\Sigma_0^{-1}+X^TX\right)\ \Big)$$

a) Write a R function ‘probit_mcmc’ to sample from the posterior distribution of β using the CPU only. Your function should return the posterior samples of β as a matrix/array or ‘mcmc’ object (if using R). The posterior samples of Z do not need to be returned (and should not be stored – they will take up too much memory!).

Answer: See code appendix for “probit_mcmc_cpu” below and results from problems below.

b) Write a RCUDA function ‘probit_mcmc’ to sample from the posterior distribution of β using the CPU and GPU. You can also compute the block and grid dimensions within your function if preferred. Note that the GPU should only be used for the sampling of the Z_i vector. Your function should return the posterior samples of β as a matrix/array or ‘mcmc’ object (if using R). The posterior samples of Z do not need to be returned (and should not be stored – they will take up too much memory!).

Answer: See code appendix for “probit_mcmc_gpu” below and results from problems below.

c) Test your code by fitting the mini dataset ‘mini_test.txt’. This dataset can be generated by running the file ‘sim_probit.R’, supplied in the course GitHub repo. The first column of the dataset corresponds to ‘y’, with all other columns corresponding to ‘X’. Assume prior parameters $\beta_0=\vec{0}$ and $\Sigma_0^{-1}=\mathbf{0}$. Verify that both functions give posterior means/medians that are at least relatively close to the true values (in ‘mini_pars.txt’, also generated when you run ‘sim_probit.R’) and the estimates produced by standard GLM functions.

Answer: We can see that the GPU runtime is considerably longer than the CPU runtime as expected when the copyToDevice overhead is not offset by the speed of the GPU since the number of threads in the mini dataset is so small and the β’s are all over the place for the CPU but roughly on the correct scale for the GPU:

	X	user.self	sys.self	elapsed	user.child	sys.child
1	gpu_time	103.98	3.12	107.07	0	0
2	cpu_time	1.68	0.00	1.68	0	0

	Source	Beta0	Beta1	Beta2	Beta3	Beta4	Beta5	Beta6	Beta6
1	GPU	17.73	5.00	-66.90	-3.05	27.17	-0.66	21.85	-6.87
2	CPU	1.62	0.47	-5.50	-0.02	2.25	-0.08	2.05	-0.31
3	PARS	0.57	-0.11	-2.06	0.12	1.05	-0.10	1.23	-0.03

d) Run 'sim_probit.R' to create each of the following datasets. Then analyze as many as possible, using both your CPU and GPU code:

'data_01.txt': n=1000 for 'niter=2000', 'burnin=500'

	X	user.self	sys.self	elapsed	user.child	sys.child
1	gpu_time	103.62	1.71	105.34	0	0
2	cpu_time	3.57	0.00	3.57	0	0

	Source	Beta0	Beta1	Beta2	Beta3	Beta4	Beta5	Beta6	Beta6
1	GPU	-0.05	-0.91	0.19	1.75	1.71	-1.18	2.30	0.74
2	CPU	0.17	-1.01	0.38	2.19	1.73	-1.03	2.53	0.87
3	PARS	0.14	-0.97	0.31	1.87	1.49	-0.95	2.42	0.80

'data_02.txt': n=10000 for 'niter=2000', 'burnin=500'

	X	user.self	sys.self	elapsed	user.child	sys.child
1	gpu_time	115.88	4.31	120.20	0	0
2	cpu_time	24.04	0.00	24.05	0	0

	Source	Beta0	Beta1	Beta2	Beta3	Beta4	Beta5	Beta6	Beta6
1	GPU	0.12	0.17	-0.65	0.12	1.20	0.31	0.57	-0.49
2	CPU	0.11	0.13	-0.58	0.11	1.10	0.28	0.49	-0.45
3	PARS	0.09	0.16	-0.58	0.12	1.10	0.27	0.48	-0.44

'data_03.txt': n=100000 for 'niter=2000', 'burnin=500'

	X	user.self	sys.self	elapsed	user.child	sys.child
1	gpu_time	181.76	28.61	210.37	0	0
2	cpu_time	211.76	0.06	211.81	0	0

	Source	Beta0	Beta1	Beta2	Beta3	Beta4	Beta5	Beta6	Beta6
1	GPU	2.44	1.38	0.39	-0.16	-0.69	-1.33	0.44	-1.31
2	CPU	2.45	1.37	0.42	-0.16	-0.69	-1.34	0.45	-1.33
3	PARS	2.45	1.38	0.41	-0.15	-0.69	-1.33	0.45	-1.35

'data_04.txt': n=1000000 for 'niter=2000', 'burnin=500'

	X	user.self	sys.self	elapsed	user.child	sys.child
1	gpu_time	792.26	283.64	1075.87	0	0
2	cpu_time	2147.28	36.90	2184.05	0	0

	Source	Beta0	Beta1	Beta2	Beta3	Beta4	Beta5	Beta6	Beta6
1	GPU	-1.21	0.23	0.30	1.02	1.18	1.79	1.22	-2.87
2	CPU	-1.21	0.23	0.30	1.01	1.18	1.78	1.22	-2.87
3	PARS	-1.21	0.24	0.30	1.01	1.18	1.78	1.22	-2.87

'data_05.txt': n=10000000 for 'niter=2000', 'burnin=500'
data_05.txt did actually complete for me after several runs of fixing bugs with a total runtime of approximately 4 hours for the GPU and 7.5 hours for the CPU. However, while attempting to run these jobs in the background on AWS using an '&', I introduced one final mistake in the write.csv step of the output, so I have no proof other than my word.

We notice that the GPU code has overhead from copyToDevice that dominates runtime until the speed of multiple threads can overtake the copy time. By looking at data_03.txt, it seems that the break even point maybe in the neighborhood of 100,000 threads, after which, the GPU code creams the CPU code. As seen in data_04.txt, the runtime for the CPU is approxiately twice as long as the GPU code.

e) Discuss the relative performance of your CPU and GPU code. At what point do you think your GPU code would become competitive with the CPU code?

Answer: The first thing we notice is that the GPU code has overhead from copyToDevice that dominates runtime until the speed of multiple threads can overtake the copy time. By looking at data_03.txt, it seems that the break even point maybe in the neighborhood of 100,000 threads. In additona, as noted in the assignment, "the number of iterations used here is not sufficient to obtain reliable posterior estimates. This exercise is for illustration and learning purposes."

Additonally, we notice that the GPU estimates themselves seem to be a bit off for small datasets relative to built-in R functions. This could indicate that there is a bug in our code, or perhaps the built-in R function is more efficient or accurate in computation. As the size of the datasets increase, both estimates from the GPU and the CPU match very closely to the true parameter values.

Lastly, as noted in the assignment, "the number of iterations used here is not sufficient to obtain reliable posterior estimates. This exercise is for illustration and learning purposes."

Code Appendix

"rtruncnorm.cu"

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>
#include <math.h>

extern "C"
{

    __device__ float rand_expon(float a, curandState *state)
    {
        return -log(curand_uniform(state))/a;  // x is now random expon by inverse CDF
    } // END rand_expo


    __device__ float psi_calc(float mu_minus, float alpha, float z)
    {
        float psi;
            // Compute Psi
                if(mu_minus < alpha){
                        psi = expf( -1/2*pow(alpha-z,2));
                }
                else {
                        psi = expf(  1/2*( pow(mu_minus-alpha,2) - pow(alpha-z,2) ) );
                }
        return psi;
    }

    __global__ void rtruncnorm_kernel(float *vals, int n,
            float *mu, float *sigma,
                float *lo, float *hi,
                int mu_len, int sigma_len,
                int lo_len, int hi_len,
                int rng_seed_a, int rng_seed_b, int rng_seed_c,
                int maxtries)
    {
            int accepted = 0;
            int numtries = 0;
                float x;
            float u;
            float alpha;
                float psi;
            float z;
            float a;
            float mu_minus;
            int left_trunc = 0;

            // Figure out which thread and block you are in and map these to a single index, "idx"
            // Usual block/thread indexing...
            int myblock = blockIdx.x + blockIdx.y * gridDim.x;
            int blocksize = blockDim.x * blockDim.y * blockDim.z;
            int subthread = threadIdx.z*(blockDim.x * blockDim.y) + threadIdx.y*blockDim.x + threadIdx.x;
            int idx = myblock * blocksize + subthread;

            // Check: if index idx < n generate a sample, else in unneeded thread
            if(idx<n){

                        // Setup the RNG:
              curandState rng;
              curand_init(rng_seed_a + idx*rng_seed_b, rng_seed_c, 0, &rng);

                 // Sample the truncated normal
              // i.e. pick off mu and sigma corresponding to idx and generate a random sample, x
              // if that random sample, x, is in the truncation region, update the return value to x, i.e. vals[idx]=
       x
              // if x is not in the trunc region, try again until you get a sample in the trunc region or if more
      than maxtries,
              // move on to Robert's approx method
              while(accepted == 0 && numtries < maxtries){
              numtries++;  // Increment numtries
              x = mu[idx] + sigma[idx]*curand_normal(&rng);
              if(x >= lo[idx] && x <= hi[idx]){
                 accepted = 1;
                 vals[idx] = x;
              }
              }

              // Robert's approx method
              // We don't want to write both trunc algos for left and right tail truncations, just use
            // right tail trancation.  If we want to sample from Y~N(mu, sigma, -Inf, b), we transform
            // first X~N(mu, sigma, -b+2*mu, Inf), use only right truncation, sample from the right
            // tail to get a X, then transform back Y=2*mu-X to get left truncation sample if needed in Robert.
              if(lo[idx] < mu[idx]) {        // then left truncation
              left_trunc = 1;
              a = -1*hi[idx] + 2*mu[idx];       // flip up to right tail
              }
              else {
              a = lo[idx];              // right truncation from a=lo[idx] to infinity
              }
              mu_minus = (a-mu[idx])/sigma[idx];

                    // need to find mu_minus but that depends on if lower trunc or upper trunc
                    alpha = (mu_minus + sqrtf(pow(mu_minus,2) + 4))/2;
            numtries = 1;  //  If couldn't get sample naively, reset and try Robert
              while(accepted == 0 && numtries < maxtries){

              numtries++;  // Increment numtries
```

```
            // Need random expon for Robert no curand_expon function so do inverse CDF
            // F(x) = 1-exp(-alpha*x) --> F^1(x) = -log(U)/alpha where U~Unif[0,1]
            // u = curand_uniform(&rng);
            // x = -1 * log(u)/alpha;  // x is now random expon by inverse CDF
            z = mu_minus + rand_expon(alpha, &rng);

            // Compute Psi = probability of acceptance
            psi = psi_calc(mu_minus, alpha, z);

            // Check if Random Unif[0,1] < Psi, if so accept, else reject and try again
            u = curand_uniform(&rng);
            if (u < psi){
               accepted = 1;  // we now have our vals[idx]
               if (left_trunc == 1){  // since originally left trunc, and flip back to left tail and final transform
                  vals[idx] = mu[idx] - sigma[idx]*z;
               }
               else {   // right truncation originally so we're done after final transform
                  vals[idx] = mu[idx] + sigma[idx]*z;
               }
            }
            }
                  if(accepted == 0){   // Just in case both naive and Roberts fail
                     vals[idx] = -999;
                  }

         } // END if (idx<n)
         return;
    } // END rtruncnorm_kernel
} // END extern "C"
```

"rtruncnorm_driver.R"

```
library(RCUDA)

cat("\nSetting cuGetContext(TRUE)...\n ")
cuGetContext(TRUE)
cat("done. Profiling CUDA code.\n \n")

cat("Loading module...\n")
m = loadModule("rtruncnorm.ptx")
cat("done. Loading module.\n \n")

cat("Extracting kernelkernel...\n")
k = m$rtruncnorm_kernel
cat("done. Extracting kernelkernel.\n \n")

cat("Setting up input params...\n")

#t_k = 1:8
t_k = 8
#i=1
for (i in 1:length(t_k)){
  N = as.integer(10^t_k[i])

  vals = rep(0,N)
  mu = rep(2, N)
  sigma = rep(1, N)
  lo = rep(0, N)
  hi = rep(1.5, N)
  mu_len = N
  sigma_len = N
  lo_len = N
  hi_len = N
  rng_seed_a = 1234L
  rng_seed_b = 1423L
  rng_seed_c = 1842L

  maxtries = 2000L
  cat("done. Setting input params.\n \n")

  # Fix block dims:
  threads_per_block <- 512L
  block_dims <- c(threads_per_block, 1L, 1L)
  grid_d1 <- as.integer(floor(sqrt(N/threads_per_block)))
  grid_d2 <- as.integer(ceiling(N/(grid_d1*threads_per_block)))
  grid_dims <- c(grid_d1, grid_d2, 1L)

#   "compute_grid" <- function(N,sqrt_threads_per_block=16L,grid_nd=1)
#   {
#     # if...
#     # N = 1,000,000
#     # => 1954 blocks of 512 threads will suffice
#     # => (62 x 32) grid, (512 x 1 x 1) blocks
#     # Fix block dims:
#     block_dims <- c(as.integer(sqrt_threads_per_block), as.integer(sqrt_threads_per_block), 1L)
#     threads_per_block <- prod(block_dims)
#     if (grid_nd==1){
#       grid_d1 <- as.integer(max(1L,ceiling(N/threads_per_block)))
#       grid_d2 <- 1L
#     } else {
#       grid_d1 <- as.integer(max(1L, floor(sqrt(N/threads_per_block))))
#       grid_d2 <- as.integer(ceiling(N/(grid_d1*threads_per_block)))
#     }
#     grid_dims <- c(grid_d1, grid_d2, 1L)
#     return(list("grid_dims"=grid_dims,"block_dims"=block_dims))
#   }
#   grid = compute_grid(N)
#   grid_dims = grid$grid_dims
#   block_dims = grid$block_dims

  cat("Grid size:\n")
  print(grid_dims)
  cat("Block size:\n")
  print(block_dims)
```

```
    nthreads <- prod(grid_dims)*prod(block_dims)
    cat("Total number of threads to launch = ",nthreads,"\n \n")
    if (nthreads < N){
      stop("Grid is not large enough...!")
    }

    cat("*********************** \n")
    cat(paste(" SYSTEM TIMES for N = 10^", t_k[i], sep=""), "\n")
    cat("*********************** \n")

    cat("Running CUDA kernel...\n \n")
    ##### total_cuda_time <- system.time({
      copy_to_time <- system.time({
        cat("Copying to device...\n")
        vals_dev = copyToDevice(vals)
        mu_dev = copyToDevice(mu)
        sigma_dev = copyToDevice(sigma)
        hi_dev = copyToDevice(hi)
        lo_dev = copyToDevice(lo)
      })
      print(copy_to_time)
      cat("done. Copying to device...\n \n")

      kernel_time <- system.time({
        cat("Call the kernel...\n")
        .cuda(k, vals_dev, N, mu_dev, sigma_dev, lo_dev, hi_dev, mu_len, sigma_len,
              lo_len, hi_len, rng_seed_a, rng_seed_b, rng_seed_c, maxtries,
              gridDim = grid_dims, blockDim = block_dims)
      })
      print(kernel_time)
      cat("done. Calling the kernel...\n \n")

      copy_from_time <- system.time({
        cat("Copying result back from device...\n")
        vals = copyFromDevice(obj=vals_dev,nels=vals_dev@nels,type="float")
      })
      print(copy_from_time)
      cat("done. Copying result back from device...\n \n")
    ##### })
    ##### print(total_cuda_time)
    cat("done. Running CUDA kernel.\n \n")

     five_num = summary(vals)
     #write.csv(five_num, paste0("five_num_", i, ".csv"))
     print(five_num)

    times = rbind(copy_to_time, kernel_time, copy_from_time)
    write.csv(times, paste0("times_", i, ".csv"))

    # Two sided truncation
    mean_theory = round(mu[1] +
      sigma[1]*(dnorm((lo[1]-mu[1])/sigma[1])-dnorm((hi[1]-mu[1])/sigma[1]))/
      (pnorm((hi[1]-mu[1])/sigma[1])-pnorm((lo[1]-mu[1])/sigma[1])), 5)

    # Left-sided truncation
    # mean_theory = round(mu[1] -
    #     dnorm((hi[1]-mu[1])/sigma[1])/(pnorm((hi[1]-mu[1])/sigma[1])), 5)

    mean_obs = round(five_num["Mean"], 5)

    label_mean_theory = paste0("Theoretical Mean = ", mean_theory)
    label_mean_obs = paste0("Observed Mean = ", mean_obs)

    pdf("density.pdf")
      plot(density(vals), main="Density of Truncated Normals on GPU", lwd=1.5)
      abline(v=mean_theory, lwd=2, col="blue")
      abline(v=mean_theory, lty="dashed", lwd=2, col="green")
      abline(h=0)
      legend("topleft", inset=.05, legend=c(label_mean_theory, label_mean_obs), lty=c("solid", "dashed"), lwd=c(2,2),
        col=c("blue","green"))
    dev.off()
}

# Free memory...
rm(list=ls())

q("no")
```

"probit_mcmc_driver.R"

```
# Clear out everything in memory for a nice clean run and easier debugging
rm(list=ls())

# Load necessary libraries
library(RCUDA)
library(mvtnorm)
library(truncnorm)

########################
# Function Definitions
########################

"compute_grid" <- function(N,sqrt_threads_per_block=16L,grid_nd=1){
  # if...
  # N = 1,000,000
  # => 1954 blocks of 512 threads will suffice
  # => (62 x 32) grid, (512 x 1 x 1) blocks
  # Fix block dims:
  block_dims <- c(as.integer(sqrt_threads_per_block), as.integer(sqrt_threads_per_block), 1L)
  threads_per_block <- prod(block_dims)
  if (grid_nd==1){
    grid_d1 <- as.integer(max(1L,ceiling(N/threads_per_block)))
    grid_d2 <- 1L
  } else {
    grid_d1 <- as.integer(max(1L, floor(sqrt(N/threads_per_block))))
```

```r
    grid_d2 <- as.integer(ceiling(N/(grid_d1*threads_per_block)))
  }
  grid_dims <- c(grid_d1, grid_d2, 1L)
  return(list("grid_dims"=grid_dims,"block_dims"=block_dims))
} # end compute grid

"probit_mcmc_gpu" = function(y, X, beta_0, Sigma_0_inv, niter, burnin, n, p){

  z = rep(0,n)
  beta_mat = matrix(0, nrow=(burnin+niter), ncol=p)
  beta_t = beta_0
  lo = ifelse(y>0,0,-Inf)
  hi = ifelse(y>0,Inf,0)
  sigma = matrix(1,nrow=n, ncol=1)
  mu_len = n
  sigma_len = n
  lo_len = n
  hi_len = n
  rng_seed_a = 1234L
  rng_seed_b = 1423L
  rng_seed_c = 1842L
  maxtries = 2000L

  # Setup static pieces of posterior mean and variance that do not need to be inside the loop
  #     Beta^(t+1)|X,Y,Z^(t)
  #          ~ MVN(  (Sigma_0_inv+X'X)^(-1)*(Sigma_0_inv*beta_0+X'Z^(t))  ,   Sigma_0_inv+X'X )
  SXX_inv = solve(Sigma_0_inv + t(X)%*%X)      # Static so compute here
  SXX = Sigma_0_inv + t(X)%*%X                 # Static so compute here
  Sb0 = Sigma_0_inv %*% beta_0                 # Static so compute here

  # Setup grid
  grid = compute_grid(n)
  grid_dims = grid$grid_dims
  block_dims = grid$block_dims

  cat("Grid size:\n")
  print(grid_dims)
  cat("Block size:\n")
  print(block_dims)

  nthreads <- prod(grid_dims)*prod(block_dims)
  cat("Total number of threads to launch = ",nthreads,"\n \n")
  if (nthreads < n){
    stop("Grid is not large enough...!")
  }

  cat("Copying to device...\n")
  z_dev = copyToDevice(z)
  sigma_dev = copyToDevice(sigma)
  hi_dev = copyToDevice(hi)
  lo_dev = copyToDevice(lo)
  cat("done. Copying to device...\n \n")

  cat("Main loop of Gibbs Sampler on GPU...\n")
  for (i in 1:(burnin+niter)){

    if (i %% 500 == 0){
      cat("           GPU iteration =", i, "\n")
    }

    # Get Z^(t)
    #     Z^(t)|X,Y,Beta^(t) ~ TN(X*Beta^(t), I; [0,Inf)) if y=1
    #                        ~ TN(X*Beta^(t), I; (-Inf,0]) if y=0
    mu = X%*%beta_t
    mu_dev = copyToDevice(mu)

    # Get Z^(t) from the GPU
    .cuda(k, z_dev, n, mu_dev, sigma_dev, lo_dev, hi_dev, mu_len, sigma_len,
          lo_len, hi_len, rng_seed_a, rng_seed_b, rng_seed_c, maxtries,
          gridDim = grid_dims, blockDim = block_dims)

    z = copyFromDevice(obj=z_dev,nels=z_dev@nels,type="float")

    # Get Beta^(t+1)
    #     Beta^(t+1)|X,Y,Z^(t)
    #          ~ MVN(  (Sigma_0_inv+X'X)^(-1)*(Sigma_0_inv*beta_0+X'Z^(t))  ,   Sigma_0_inv+X'X )

    # Calc the dynamic piece of the posterior mean which depends on Z^(t)
    mu = SXX_inv%*%(Sb0+t(X)%*%as.matrix(z))

    # sample from MVN to get Beta^(t+1)
    beta_t = t(rmvnorm(1, mu, SXX_inv))
    beta_mat[i,] = beta_t
  } # end burnin+niter for loop
  cat("done. Main loop of Gibbs Sampler on GPU...\n")

  return(beta_mat)

} # end probit_mcmc_gpu

"probit_mcmc_cpu" = function(y, X, beta_0, Sigma_0_inv, niter, burnin, n, p){

  beta_mat = matrix(0,nrow=(burnin+niter), ncol = p)
  beta_t = beta_0

  # Setup static pieces of posterior mean and variance that do not need to be inside the loop
  #     Beta^(t+1)|X,Y,Z^(t)
  #          ~ MVN(  (Sigma_0_inv+X'X)^(-1)*(Sigma_0_inv*beta_0+X'Z^(t))  ,   Sigma_0_inv+X'X )
  SXX_inv = solve(Sigma_0_inv + t(X)%*%X)      # Static so compute here
  SXX = Sigma_0_inv + t(X)%*%X                 # Static so compute here
  Sb0 = Sigma_0_inv %*% beta_0                 # Static so compute here

  cat("Main loop of Gibbs Sampler on CPU...\n")
  for (i in 1:(burnin+niter)){

    if (i %% 500 == 0){
      cat("           CPU iteration =", i, "\n")
    }
```

```
    # Get Z^(t)
    #     Z^(t)|X,Y,Beta^(t) ~ TN(X*Beta^(t), I; [0,Inf)) if y=1
    #                        ~ TN(X*Beta^(t), I; (-Inf,0]) if y=0
    # Get Z^(t) from the CPU using rtruncnorm
    z = ifelse(y>0, rtruncnorm(1,0,Inf,X%*%beta_t,1), rtruncnorm(1,-Inf,0,X%*%beta_t,1))

    # Get Beta^(t+1)
    #     Beta^(t+1)|X,Y,Z^(t)
    #          ~ MVN(  (Sigma_0_inv+X'X)^(-1)*(Sigma_0_inv*beta_0+X'Z^(t))  ,   Sigma_0_inv+X'X )

    # Calc the dynamic piece of the posterior mean which depends on Z^(t)
    mu = SXX_inv%*%(Sb0+t(X)%*%as.matrix(z))

    # sample from MVN to get Beta^(t+1)
    beta_t = t(rmvnorm(1, mu, SXX_inv))
    beta_mat[i,] = beta_t

  } # end burnin+niter for loop
  cat("done. Main loop of Gibbs Sampler on CPU...\n")

  return(beta_mat[(burnin+1):(burnin+niter),])
} # end probit_mcmc_cpu


########################
# Read in Data and Setup inputs
########################
extension = "05"
burnin = 500
niter = 2000

pars = read.table(paste0("pars_", extension, ".txt"), header=T, quote="\"")
names(pars) = "params"
pars = round(pars,5)
dat = read.table(paste0("data_", extension, ".txt"), header=T, quote="\"")
y = as.matrix(dat[,1])
X = as.matrix(dat[,-1])

n = dim(X)[1]
p = dim(X)[2]

# Setup priors given in problem
beta_0 = matrix(0, nrow=p, ncol=1)
Sigma_0_inv = matrix(0, nrow=p, ncol=p)

# Initialize matrix for beta estimate results
beta_est_gpu = matrix(0, nrow=(burnin+niter), ncol=p)

# Setup GPU
cat("\nSetting cuGetContext(TRUE)...\n ")
cuGetContext(TRUE)
cat("done. Profiling CUDA code.\n \n")

cat("Loading module...\n")
m = loadModule("rtruncnorm.ptx")
cat("done. Loading module.\n \n")

cat("Extracting kernelkernel...\n")
k = m$rtruncnorm_kernel
cat("done. Extracting kernelkernel.\n \n")


########################
# Do actual function calls and get results
########################
gpu_time = system.time({
  beta_est_gpu = probit_mcmc_gpu(y=y, X=X, beta_0=beta_0, Sigma_0_inv=Sigma_0_inv,
                                 niter=niter, burnin=burnin, n=n, p=p)
})

write.csv(gpu_time, paste0("times_", extension, "-gpu.csv"))
beta_summary = rbind(round(apply(beta_est_gpu, MARGIN=2,FUN=mean), 5),
                     round(t(pars), 5))
write.csv(beta_summary, paste0("beta_summary_", extension, "-gpu.csv"))


cpu_time = system.time({
  beta_est_cpu = probit_mcmc_cpu(y=y, X=X, beta_0=beta_0, Sigma_0_inv=Sigma_0_inv,
                                 niter=niter, burnin=burnin, n=n, p=p)
})
write.csv(cpu_time, paste0("times_", extension, "-cpu.csv"))
beta_summary = rbind(round(apply(beta_est_cpu, MARGIN=2,FUN=mean), 5),
                     round(t(pars), 5))
write.csv(beta_summary, paste0("beta_summary_", extension, "-cpu.csv"))


# times = rbind(gpu_time,cpu_time)
# write.csv(times, paste0("times_", extension, ".csv"))
#
# beta_summary = rbind(round(apply(beta_est_gpu, MARGIN=2,FUN=mean), 5),
#                      round(apply(beta_est_cpu, MARGIN=2,FUN=mean), 5),
#                      round(t(pars), 5))
# write.csv(beta_summary, paste0("beta_summary_", extension, ".csv"))

q("no")
```

"rtruncnorm_timer.R"

```
# Clear out everything in memory for a nice clean run and easier debugging
rm(list=ls())

# Load necessary libraries
library(truncnorm)
```

```r
mu = 2
sd = 1
lo = 0
hi = 1.5
rm(rtruncnorm_time)
t_k = 1:8
for (i in 1:length(t_k)){
  N = as.integer(10^t_k[i])
  rtruncnorm_time = system.time({
    rtnorm = rtruncnorm(N,lo,hi,mu,sd)
  })
  cat(10^i, " ", rtruncnorm_time, "\n")
  #write.csv(rtruncnorm_time, paste0("times_rtruncnorm", i, ".csv"))
  # cat(N, "\n")
}


CPU_time = matrix(c(10, 0, 0, 0, NA, NA, 100, 0, 0, 0, NA, NA, 1000, 0, 0, 0.01, NA, NA,
                    10000, 0.03, 0, 0.05, NA, NA, 100000,  0.33, 0, 0.42, NA, NA, 1000000,
                    3.01, 0, 3.12, NA, NA, 10000000,30.6, 0.16, 31.28, NA, NA, 1000000000,
                    308.2, 1.76, 324.29, NA, NA), nrow=8, ncol=6, byrow=TRUE )
GPU_time = c(0.047,0.05,0.049,0.053,0.083,0.369,3.253,32.446)
n = 10^(1:8)

pdf("plot-1e.pdf")
plot(x=log(n), GPU_time, type="l", col="blue", lwd=2, main="Generation Time by log(n)", ylab="time in sec")
lines(x=log(n), y=CPU_time[,4], type="l", col="green",lwd=2)
legend("topleft", inset=.05, legend=c("GPU", "CPU"), lty=c("solid", "solid"), lwd=c(2,2), col=c("blue","green"))
dev.off()

times = matrix(0, nrow=8, ncol=4)
for (i in 1:8){
  dat = read.csv(paste0("times_", i, ".csv"), header=T, quote="\"")
  times[i,] = cbind(10^i, t(dat[,4]))
}
xtable(times)
```