STA 250: HW4: GPU MODULE

CHRISTOPHER CONLEY

Contents

1. Question 1: Truncated Normal Sampling

1.1. **Implementation.** Truncated normal sampling carried out identical algorithms for both the GPU and the CPU hardware. The CPU-based code was written first in R and then in Rcpp to enhance the speed. The GPU-based code was written in C and applied in the RCUDA framework.

For a finite number of tries, we sampled from the normal distribution and rejected until a sample fell in the truncated interval. If that did not succeed, then Robert one-sided truncation sampling was applied, where right or left-sided truncation was determined by whether the high truncation bound was less than the mean (right). The Robert sampling mechanism was written to handle in principle any Truncated Normal with arbitrary location, scale, or truncation interval; however, our testing revealed that some extreme tailing did reduce the accuracy of the method.

1.2. **Validating Truncation cases.** The following plots confirm the success of our sampling implementation in a variety of truncation cases. In black, the density of 10,000 randomly sampled Normal(μ, σ) random variables. In red, the density of 10,000 randomly sampled Truncated Normal(μ, σ, (a, b)) random variables.The sample mean of the truncated distribution is the thick vertical red line and the blue vertical line is the expected value of the truncated normal.

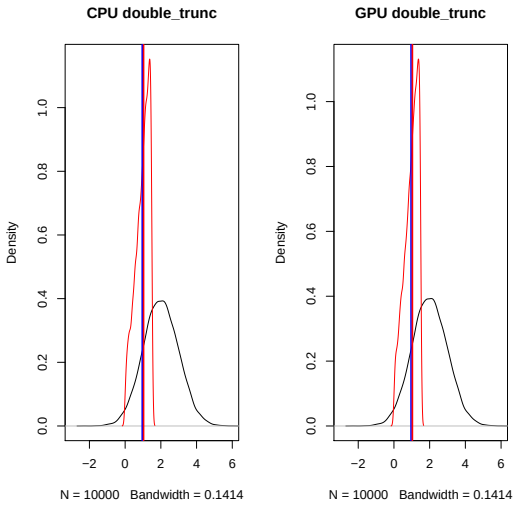

FIGURE 1. Double truncation sampling: $\mathrm{TN}\Big(2,\ 1,\ [0,\ 1.5]\ \Big)$. Sample median (CPU = 1.037119, GPU = 1.035505). Expected value = 0.9570067

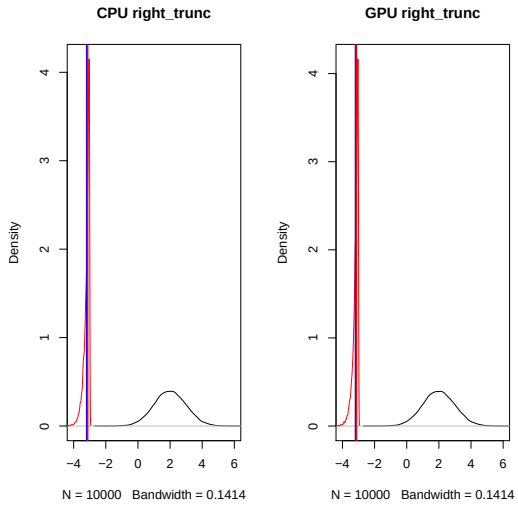

FIGURE 2. Right truncation sampling: $\text{TN}\Big(2,\ 1,\ (-\infty,\ \text{-}3]\ \Big)$. Sample median (CPU $= -3.129094$, GPU $= -3.13183$). Expected value $= -3.186504$

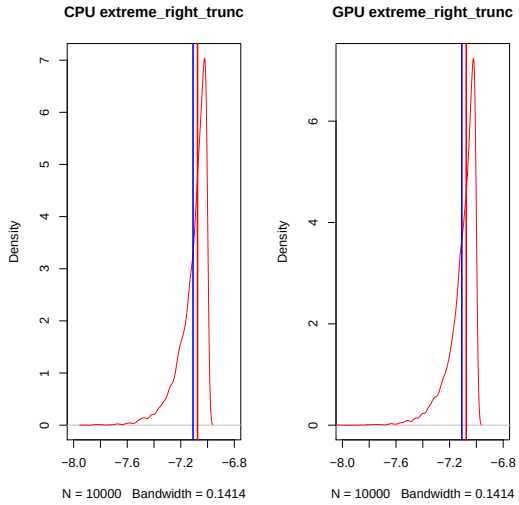

FIGURE 3. Extreme right truncation sampling: $\text{TN}\Big(2,\ 1,\ (-\infty,\ \text{-}7]\ \Big)$. Sample median (CPU $= -7.075001$, GPU $= -7.075756$). Expected value $= -7.108523$

1.3. **Comparing GPU to CPU performance.** Samples of $\{10^k : k = 1, \ldots, 8\}$ were obtained for the truncated normal under both GPU and CPU configurations. Because I wrote the CPU sampler in Rcpp

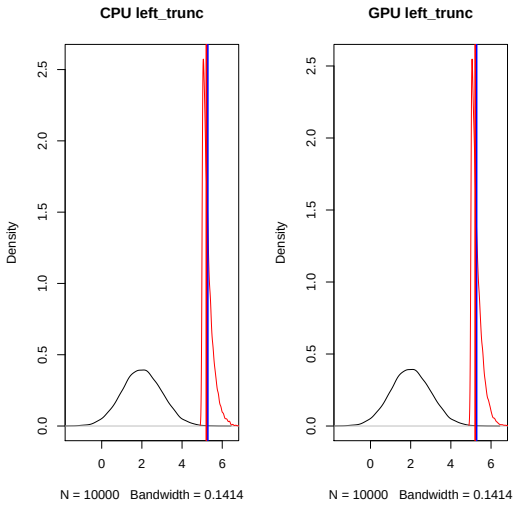

FIGURE 4. Left truncation sampling: $\mathrm{TN}\Big(2,\ 1,\ [5,\ -\infty)\ \Big)$. Sample median (CPU = 5.207818, GPU = 5.209829). Expected value = 5.283099

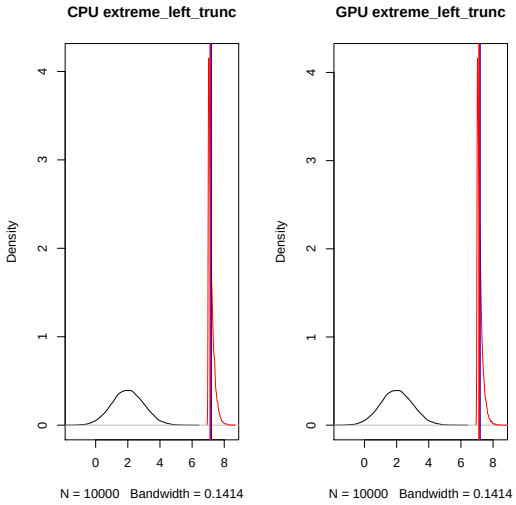

FIGURE 5. Extreme Left truncation sampling: $\mathrm{TN}\Big(2,\ 1,\ [7,\ -\infty)\ \Big)$. Sample median (CPU = 7.129094, GPU = 7.131956). Expected value = 7.186504

and kept the *maxtries* argument equal to the C kernel, the comparison is a little more fair than comparing pure R to C kernel sampling of the GPU. I anticipated that the GPU would scale dramatically better on

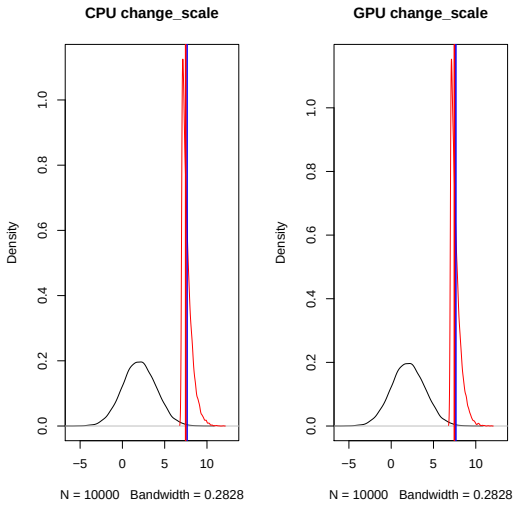

FIGURE 6. New scale, Left truncation sampling: $\mathrm{TN}\Big(2,\, 2,\, [-\infty,\, \infty)\ \Big)$. Sample median (CPU = 7.480974, GPU = 7.468181). Expected value = 7.64549

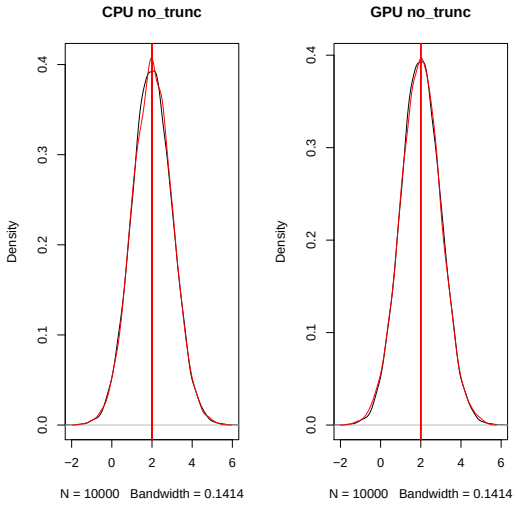

FIGURE 7. No truncation sampling: $\mathrm{TN}\Big(2,\, 1,\, [-\infty,\, \infty)\ \Big)$. Sample median (CPU = 2.008967, GPU = 1.998455). Expected value = 2

the last case of $k = 8 : 10^8$ observations, but not on the first seven cases. The improvement of the GPU

was not that impressive on the largest data sample. Figure 8 below shows their respective performance at sampling from the Truncated Normal $\Big(2,\ 1,\ [0,\ 1.5]\ \Big)$.

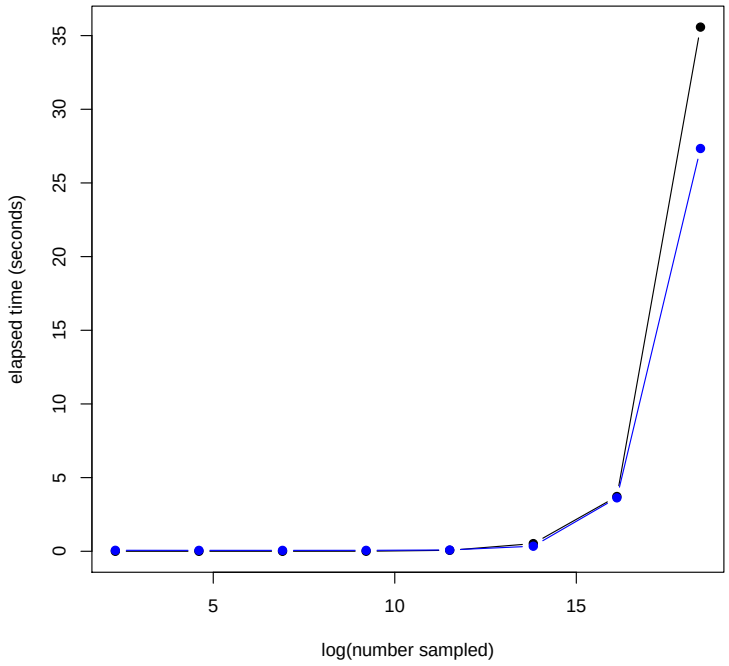

FIGURE 8. The run time (seconds) per increasing sample size. GPU (blue) performance scales to the largest sampling case the best. CPU (blue) is effectively as fast in all other cases.

2. PROBIT MCMC

2.1. **Gibbs Sampler & Full Conditional Derivations.** The following derivation details how we arrived at the full conditional distributions employed in the gibbs sampler.

Probit MCMC

$$Y_i \mid z_i, \beta \sim 1_{\{z_i > 0\}} = \begin{cases} 1 & \text{if } z_i > 0 \\ 0 & \text{if } z_i \leq 0 \end{cases}$$

$$z_i \mid \beta \sim N(x_i^T \beta^{(t)}, 1)$$

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

Gibbs Sampler

(1) Initialize $(z, \beta) = (z_0^{(0)}, \beta_0^{(0)})$ for $t = 0$

(2) Sample $z_i^{(t+1)}$ from $p(z_i \mid \beta^{(t)}, y) \sim$ Truncated Normal (?)

(3) Sample $\beta^{(t+1)}$ from $p(\beta \mid z_i^{(t+1)}, y) \sim$ Normal (?)

(4) Increment $t \longmapsto t+1$ and return to (2).

$$p(z_i \mid \beta^{(t)}, y) = \frac{p(y_i, \beta^{(t)}, z_i)}{\int p(y_i, \beta^{(t)}, z)\, d\beta}$$

$$= \frac{p(y_i \mid z_i, \beta)\, p(z_i \mid \beta)}{\int p(y_i \mid z_i, \beta)\, p(z_i \mid \beta)\, d\beta}$$

$$\propto p(y_i \mid z_i, \beta)\, p(z_i \mid \beta)$$

$$\propto 1_{\{z_i > 0\}} \cdot \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(z_i - x_i^T \beta^{(t)})^2}{2}\right)$$

This is a truncated normal density

$$z_i \mid y_i = 0, \beta^{(t)} \sim TN(x_i^T \beta^{(t)}, 1, (-\infty, 0])$$

$$z_i \mid y_i = 1, \beta^{(t)} \sim TN(x_i^T \beta^{(t)}, 1, [0, \infty))$$

$$p(\beta \mid \underline{z}^{(t+1)}, \underline{y}) \propto p(\underline{z}^{(t+1)} \mid \beta)\, p(\beta \mid \beta_0, V_0)$$

$$\propto \prod_{i=1}^{n} \left[\frac{1}{\sqrt{2\pi}} \exp\left\{ -\frac{1}{2}\left(z_i^{(t+1)} - x_i^T \beta^{(t)} \right)^2 \right] \frac{1}{\sqrt{2\pi\,|\Sigma_0|^{-1}}} \exp\left\{ \frac{-1}{2}(\beta - \beta_0)^T V_0^{-1} (\beta - \beta_0) \right\}$$

$$\propto \exp\left[-\frac{1}{2} \left\{ \sum_{i=1}^{n} \left(z_i^{(t+1)} - x_i^T \beta^{(t)} \right)^2 + (\beta - \beta_0)^T V_0^{-1} (\beta - \beta_0) \right\} \right]$$

Now expand and drop all constants that do not involve β

$$\propto \exp\left[-\frac{1}{2} \left\{ \sum_{i=1}^{n} \left(z_i^{(t+1)^2} + (x_i^T \beta)^2 - 2 (x_i^T \beta) z_i \right) + \beta^T V_0^{-1} \beta - 2 \beta^T V_0^{-1} \beta_0 + \beta_0^T V_0^{-1} \beta_0 \right\} \right]$$

$$\propto \exp\left[-\frac{1}{2} \left\{ \sum_{i=1}^{n} (x_i^T \beta)^2 - 2 \sum_{i=1}^{n} (x_i^T \beta) z_i + \beta^T V_0^{-1} \beta - 2 \beta^T V_0^{-1} \beta_0 \right\} \right]$$

We need to re-express this as a quadratic form in β.

$$-\frac{1}{2} (\beta - \mu_\beta)^T V_\beta^{-1} (\beta - \mu_\beta) + \text{Constant}$$

Notice $\sum_{i=1}^{n} (x_i^T \beta)^2 = \sum_{i=1}^{n} (\beta^T x_i)(x_i^T \beta) = \sum_{i=1}^{n} \beta^T (x_i x_i^T) \beta = \beta^T X^T X \beta$

Then $\sum_{i=1}^{n} (x_i^T \beta)^2 + \beta^T V_0^{-1} \beta = \beta^T (X^T X + V_0^{-1}) \beta$

$$\Rightarrow V_\beta^{-1} = (X^T X + V_0^{-1}) \Rightarrow V_\beta = (X^T X + V_0^{-1})^{-1}$$

The expanded quadratic form is proportional to the following (dropping $-\frac{1}{2}$)

$$\beta^T V_\beta^{-1} \beta - 2 \mu_\beta^T V_\beta^{-1} \beta + \mu_\beta^T V_\beta^{-1} \mu_\beta + \text{Constant}$$

$\beta^T V_\beta^{-1} \beta$: identified as $\beta^T (X^T X + V_0^{-1}) \beta$

$-2 \mu_\beta^T V_\beta^{-1} \beta$ and $\mu_\beta^T V_\beta^{-1} \mu_\beta$: need to identify the μ_β and the constant

The following steps help identify the quantities μ_β and the constant $= C$

Notice $\sum_{i=1}^{n} (x_i^T \beta) z_i = \sum_{i=1}^{n} \beta^T x_i z_i = \beta^T \sum_{i=1}^{n} x_i z_i$

$(1 \times p) \quad (p \times 1)$

$$-2 \beta^T \left(V_0^{-1} \beta_0 + \sum_{i=1}^{n} x_i z_i \right) = -2 \beta^T V_0^{-1} \beta_0 - \sum_{i=1}^{n} (x_i^T \beta) z_i$$

Now we need to complete the square because we have a form

$$\beta^T (X^T X + V_0^{-1}) \beta - 2 \left(V_0^{-1} \beta_0 + \sum_{i=1}^{n} x_i z_i \right)^T \beta$$

and we need

$$(\beta - \mu_\beta) V_\beta^{-1} (\beta - \mu_\beta) + C$$

$$(\beta-\mu_\beta)^T V_\beta^{-1}(\beta-\mu_\beta) + C \qquad (5)$$

$$= \beta^T V_\beta^{-1}\beta - 2\mu_\beta^T V_\beta^{-1}\beta + \mu_\beta^T V_\beta^{-1}\mu_\beta + C = \beta^T V_\beta^{-1}\beta - 2\left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right)^T\beta$$

Then we need $C = \mu_\beta^T V_\beta^{-1}\mu_\beta$

and $$2\mu_\beta^T V_\beta^{-1} = 2\left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right)^T$$

$$\Leftrightarrow \quad \mu_\beta^T V_\beta^{-1} = \left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right)^T$$

$$\Leftrightarrow \quad V_\beta^{-1}\mu_\beta = \left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right)$$

$$\Leftrightarrow \quad \mu_\beta = V_\beta\left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right)$$

$(p\times p)$ $(p\times 1)$

$$\Leftrightarrow \quad C = \left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right)^T V_\beta V_\beta^{-1} V_\beta \left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right)$$

$$C = \left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right)^T V_\beta \left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} (x_i z_i)\right)$$

$$\Rightarrow \beta \mid z, y \sim \mathcal{N}\left(\left(X^TX + V_0^{-1}\right)\left(V_0^{-1}\beta_0 + \sum_{i=1}^{n} x_i z_i\right), \left(X^TX + V_0^{-1}\right)^{-1}\right)$$

2.2. **Verifying estimates on β.** We noticed that the standard errors for the mini data chunk are quite large. Table 1 shows that the posterior distribution for $\beta_i, i = 1, \ldots, 8$ almost covers the true beta in every case. Var1 and var5 are not exactly covered. The burn in time was very long (5×10^5) to get pretty good trace plots. Although var3 did struggle to fully converge. The trace plots and posterior density are not shown here.

For data chunk 01, the posterior density covers β in every case with a $burnin = 8 \times 10^4$ and $niter = 1 \times 20^4$. The reason why $niter$ was so low was to be able to show a trace plot with a small enough file size, but even that produced too large of a file and so it was excluded. The trace plots were not convincing of convergence in several cases and if more time was allowed, we would have run the MCMC longer.

TABLE 1. Estimates for $\hat{\beta}$ from the **data chunk mini**

	var1	var2	var3	var4	var5	var6	var7	var8
post. $\hat{\beta}$ 1%	0.655	-0.188	-9.216	-0.636	1.112	-0.528	0.866	-1.107
post. $\hat{\beta}$ 99%	2.807	1.241	-2.908	0.551	3.809	0.415	3.410	0.435
post. mean $\hat{\beta}$	1.564	0.482	-5.295	-0.029	2.179	-0.044	1.936	-0.291
true β	0.568	-0.106	-2.059	0.121	1.053	-0.102	1.233	-0.027

TABLE 2. Estimates for $\hat{\beta}$ from the **data chunk 01** show that β is covered by the posterior in every case.

	var1	var2	var3	var4	var5	var6	var7	var8
post. $\hat{\beta}$ 1%	-0.002	-1.243	0.180	1.788	1.396	-1.268	2.080	0.649
post. $\hat{\beta}$ 99%	0.363	-0.776	0.549	2.554	2.044	-0.791	2.927	1.092
post. mean $\hat{\beta}$	0.183	-1.000	0.362	2.159	1.708	-1.018	2.488	0.858
true.beta	0.140	-0.973	0.308	1.868	1.486	-0.948	2.416	0.803

2.3. **MCMC performance as the data size scales up.** We were able finish running the code on all data chunks (01-04) but not the last (05). We ran the largest data chunk for over 16 hours but it did not even finish for the CPU. We started the GPU on data chunk 05, but it was terminated due to lack of time and because we know did not sample any faster on the order of 10^7—the size of data chunk 05—in the comparative performance of question 1 (e) above. A small caveat is that the *maxtries* argument is only 25 instead of 2000 for the Rcpp code. In the future, I would like to have kept them equal, but I forgot to change it and now it is too late.

What is interesting about Figure 9 is that the performance between the two hardware configurations seem to differ by a constant in the Probit MCMC case. This is likely the cost of copying the result from the device back to the host, but we did not confirm this suspicion. We speculate that the GPU code would begin to outperform the CPU in the Probit MCMC example when the number of truncated normal samples is on the order of 10^8 and above. This conjecture is based on the first performance evaluation of question 1.

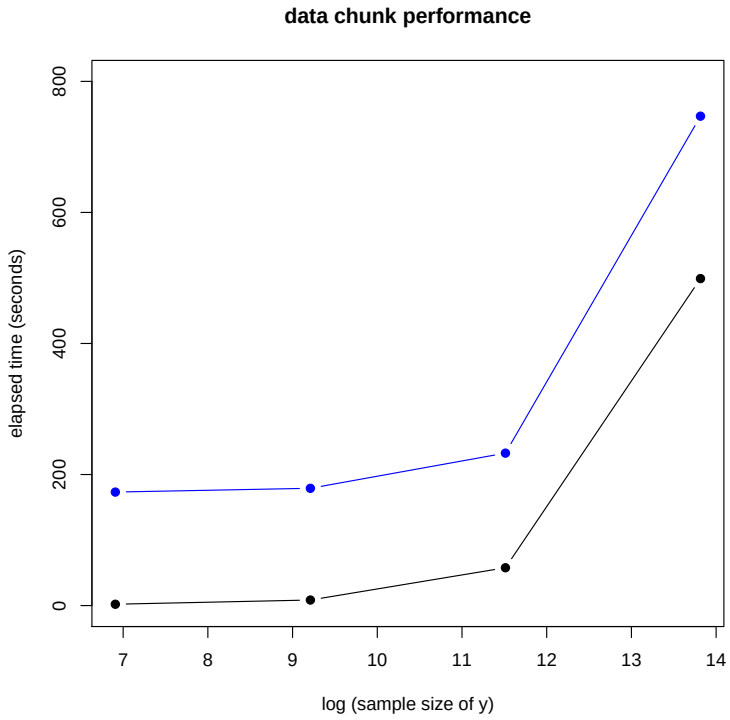

FIGURE 9. Probit MCMC comparative performance of CPU (black), GPU (blue).

3. Appendix A: Truncated Normal Sampler Code

3.1. **GPU Kernel.**

```
#include <stdio.h>
#include <stdlib.h>

#include <cuda.h>
#include <curand_kernel.h>
#include <math_constants.h>
#include <math.h>
//for boolean functionality
#include <stdbool.h>
```

```
extern "C"
{

__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 max_tries,
                  int rng_a, int rng_b, int rng_c)
{
    /* Usual block/thread indexing. Note this code, according to Prof. Baines,
       is only robust when the grid and block dims have the structure like;
            (x, 1,1)
       */
    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;

    //make sure index is not overrunning the number of threads
    if (idx < n) {

            // Setup the RNG:
            curandState rng_state;
            curand_init(rng_a + idx*rng_b, rng_c, 0, &rng_state);
```

```
            /*
               Rejection Sampling:
               (i) First try two-sided truncation naive approach. It works for cases
where: |high - low| >> 0. which will be true for this homework
assignment. This code is not robust to really tricky two-sided
truncation. because the probability of sampling from that region
is high.
(ii) When that fails for the one-sided tail regions, that have a small
probability of being sampled, then apply the Robert approach.
             */

            int accepted = 0;
            int iter_count = 0;
            while (accepted == 0 && iter_count < max_tries) {
                    iter_count = iter_count + 1;
                    vals[idx] = mu[idx] + sigma[idx] * curand_normal(&rng_state);
                    //accepted or not?
                    if (vals[idx] > lo[idx] && vals[idx] <= hi[idx]) {
                            accepted = 1;
                            return;
                    }
            }

            /*If it never accepted, then for this assignment we can assume that
              we have a case of heavy right or left truncation where we are
              trying to sample from only one of the tails.
             */
            if (accepted == 0) {
```

```
/*right truncation requires adaptation because the Robert-rejection
  sampling for one-sided truncation defaults to left truncation*/

//indicate whether it is right truncated to flip the sign of the
//sampled value if right_trunc = 1.
int right_trunc;

float mu_tmp = mu[idx];
float lo_tmp = lo[idx];

if (hi[idx] <  mu_tmp) {
        right_trunc = 1;
        mu_tmp = -1 * mu_tmp;
        lo_tmp = -1 * hi[idx];
} else {
        //left truncation
        right_trunc = 0;
}

//see Appendix A below
int mu_minus = ( lo_tmp  - mu_tmp ) / sigma[idx];

/***************************************/
/* left truncation, right tail sampling*/

/*step 0: set the optimal rate parameter for the exponential
  distribution*/
float alpha =  ( mu_minus + sqrtf(mu_minus*mu_minus + 4) ) / 2;
```

```
        while (accepted == 0) {
                /*step 1:generate: z ~ Expo(\alpha, \mu_minus)
                  by the inv-cdf transform (since z is continous)
                 */
                float z = mu_minus - log (curand_uniform(&rng_state)) / alpha;

                /*step 2: compute ratio h(z) / ( M * g(z) ) */
                float psi;
                float offset1 = alpha - z;
                float offset2;
                if (mu_minus < alpha) {
                        psi  = exp( -0.5 * offset1*offset1 );
                } else {
                        offset2 = mu_minus - alpha;
                        psi = exp( -0.5 * ( offset1*offset1 + offset2*offset2 )
                }

                //accepted the sample
                if (curand_uniform(&rng_state) <= psi) {
                        if (right_trunc == 1) {
                                vals[idx] = -1 * (mu_tmp + sigma[idx]*z);
                        } else {
                                vals[idx] = mu_tmp + sigma[idx]*z;
                        }
                        accepted = 1;
                        return;
                }
        } // END while loop
} // END Robert rejection-sampling.
```

```
    }
    return;
} // END rtruncnorm_kernel

} // END extern "C"

/*Appendix A*/
/*Need to adjust the truncation boundary of X, which is
          lo[idx] to the truncation boundary of a standard normal (Z):

          Std Normal (left truncation):
          Z ~ N(\mu = 0, \sigma = 1, low_z = \mu_minus, hi_z = \Inf)

          Standardized location-scale relation:
          Z = (X - \mu_x) / \sigma_x
          X = \mu_x + \sigma_x * Z
          X ~ N(\mu = \mu_x,
                \sigma = \sigma_x,
                lo[idx] = \sigma_x*\mu_minus + \mu_x,
                \Inf)

          Then:
          \mu_minus = ( lo[idx] - \mu_x ) / sigma_x;

*/
```

3.2. **CPU Truncated Normal Sampler.** This is virtually the same code as the GPU kernel, but written in Rcpp.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector rtruncnorm_cpu_rcpp(int n,
        NumericVector mu,
        NumericVector sigma,
        NumericVector lo,
        NumericVector hi,
        int max_tries) {




    //init return value
    NumericVector vals(n);
    //init state of RNG
    RNGScope scope;

    NumericVector uniform_input(1,1.0);
    for (int idx = 0; idx < n; idx++) {
        bool accepted = false;
        int iter_count = 0;
        while (!accepted && iter_count < max_tries) {
            iter_count = iter_count + 1;
            //NumericVector z_val = qnorm(runif(uniform_input));
            vals[idx] = mu[idx] + sigma[idx] * R::rnorm(0, 1);
            if (vals[idx] > lo[idx] && vals[idx] <= hi[idx]) {
                accepted = true;
            }
```

```
}

if (!accepted) {
    double mu_tmp = mu[idx];
    double lo_tmp = lo[idx];
    bool right_trunc;
    if (hi[idx] <  mu_tmp) {
        right_trunc = true;
        mu_tmp = -1 * mu_tmp;
        lo_tmp = -1 * hi[idx];
    } else {
        //left truncation
        right_trunc = false;
    }

    double mu_minus = ( lo_tmp  - mu_tmp ) / sigma[idx];
    double alpha =  ( mu_minus + sqrt(mu_minus*mu_minus + 4) ) / 2;
    while (!accepted) {

        double z = mu_minus - log (R::runif(0,1)) / alpha;
        double psi = 0;
        double offset1 = alpha - z;
        double offset2 = 0;
        if (mu_minus < alpha) {
            psi  = exp( -0.5 * offset1*offset1 );
        } else {
            offset2 = mu_minus - alpha;
            psi = exp( -0.5 * ( offset1*offset1 + offset2*offset2 ) );
        }
```

```
                if (R::runif(0,1) <= psi) {
                    if (right_trunc) {
                        vals[idx] = -1 * ( mu_tmp + sigma[idx]*z );
                    } else {
                        vals[idx] = mu_tmp + sigma[idx]*z;
                    }
                    accepted = true;
                }

            }  //END while loop
        } // END Robert rejection-sampling.
    }
    return vals;
} // END rtruncnorm_cpu
```

4. Appendix B: Validate Sampler & Profile Sampler

4.1. **Validate samplers.**

```
rm(list = ls())
source("rtruncnorm_cpu.r")
library(Rcpp)
sourceCpp("rtruncnorm_cpu.cpp")

################################################################################
#expected value of right truncation on (-Inf, b)
```

```
Eright <- function(mu, sigma, b) {
    stopifnot(is.finite(b))
    z_b <- (b - mu) / sigma
    mu - sigma * ( dnorm(z_b) / pnorm(z_b) )
}

#expected value of left truncation on (a, Inf)
Eleft <- function(mu, sigma, a) {
    stopifnot(is.finite(a))
    z_a <- (a - mu) / sigma
    mu + sigma * ( dnorm(z_a) / (1 -  pnorm(z_a) ) )
}

#expected value of two-sided truncation on (a, b)
Edouble <- function(mu, sigma, a, b) {
    z_a <- (a - mu) / sigma
    z_b <- (b - mu) / sigma
    numer <- dnorm(z_a) - dnorm(z_b)
    denom <- pnorm(z_b) - pnorm(z_a)
    mu + sigma * ( numer / denom )
}

#key parameters to simulation
max_tries_r <- 25L
N <- 1e4L
mu <- rep(2,N)
sd <- rep(1,N)

################################################################################
```

```
#plot the sampled output to graphically verify
#cpu
verify_rtruncnorm_cpu <- function(mu, sd, lo, hi, max_tries_r, test_type,
                                  theoretical_value, xlimits) {
    x <- rtruncnorm_cpu_rcpp(N, mu, sd, lo, hi, max_tries_r)
    #pdf(test_type)
    dx <- density(x)
    plot(density(rnorm(n = N, mean = mu, sd = sd)), type = 'l',
         ylim = c(0, max(dx$y)),
         xlim = xlimits,
         main = test_type)
    lines(dx, col = "red")
    abline(v = theoretical_value, col = "blue", lwd = 2)
    abline(v = median(x), col = "red", lwd = 2)
    #dev.off()
}

#Set up the rcuda environment and corresponding functions
source("launch_rcuda_env.r")

#both cpu & gpu together
verify_rtruncnorm <- function(mu, sd, lo, hi, max_tries_r, test_type,
                                  theoretical_value, xlimits) {

  cpu_x <- rtruncnorm_cpu_rcpp(N, mu, sd, lo, hi, max_tries_r)
  gpu_x <- rtruncnorm_gpu(kernel, x = runif(N), N, mu, sd, lo, hi,
                          compute_grid(N))

  plot_trunc <- function(x, hardware) {
```

```
    dx <- density(x)
    plot(density(rnorm(n = N, mean = mu, sd = sd)), type = 'l',
         ylim = c(0, max(dx$y)),
         xlim = xlimits,
         main = paste(hardware, test_type))
    lines(dx, col = "red")
    abline(v = theoretical_value, col = "blue", lwd = 2)
    abline(v = median(x), col = "red", lwd = 2)
    cat("median of sample:\t", median(x), "\t expected value:\t", theoretical_value, "\n");
  }


  pdf(paste0(test_type, ".pdf"))
  par(mfrow = c(1,2))
  #cpu
  set.seed(34)
  plot_trunc(cpu_x, "CPU")
  set.seed(34)
  plot_trunc(gpu_x, "GPU")
  dev.off()
}

##############################################################################

#double truncation
lo <- rep(0,N)
hi <- rep(1.5, N)
verify_rtruncnorm(mu, sd, lo, hi,
                  max_tries_r, "double_trunc",
```

```
                    Edouble(mu[1], sd[1], lo[1], hi[1]),
                    c(-3,6))

#right truncation
lo <- rep(-Inf,N)
hi <- rep(-3,N)
verify_rtruncnorm(mu, sd, lo, hi,
                    max_tries_r, "right_trunc",
                    Eright(mu[1], sd[1], hi[1]),
                    c(-4, 6))

#extreme right truncation
lo <- rep(-Inf,N)
hi <- rep(-7,N)
verify_rtruncnorm(mu, sd, lo, hi,
                    max_tries_r, "extreme_right_trunc",
                    Eright(mu[1], sd[1], hi[1]),
                    c(-8, -6.8))

#left truncation
lo <- rep(5,N)
hi <- rep(Inf,N)
verify_rtruncnorm(mu, sd, lo, hi,
                    max_tries_r, "left_trunc",
                    Eleft(mu[1], sd[1], lo[1]),
                    c(-1.5, 6.5))

#extreme left truncation
lo <- rep(7,N)
```

```
hi <- rep(Inf,N)
verify_rtruncnorm(mu, sd, lo, hi,
                  max_tries_r, "extreme_left_trunc",
                  Eleft(mu[1], sd[1], lo[1]),
                  c(-1.5, 8.5))

#No truncation
lo <- rep(-Inf,N)
hi <- rep(Inf,N)
verify_rtruncnorm(mu, sd, lo, hi,
                  max_tries_r, "no_trunc",
                  Edouble(mu[1], sd[1], lo[1], hi[1]),
                  c(-2, 6))

#left truncation different scale
lo <- rep(7,N)
hi <- rep(Inf,N)
sd <- rep(2, N)
verify_rtruncnorm(mu, sd, lo, hi,
                  max_tries_r, "change_scale",
                  Edouble(mu[1], sd[1], lo[1], hi[1]),
                  c(-6, 13))
```

4.2. **Profile samplers.**

```
rm(list = ls());
max_tries_r <- 2000L;
ntimes <- 8L;
```

```
log_n <- sapply(seq_len(ntimes), function(k) log(10^k));
source("rtruncnorm_cpu.r");
library(Rcpp);
sourceCpp("rtruncnorm_cpu.cpp");
cpu_times <- sapply(seq_len(ntimes), function(k) {
                system.time({
                        N <- as.integer(10^k);
                        mu <- rep(2,N);
                        sd <- rep(1,N);
                        lo <- rep(0,N);
                        hi <- rep(1.5, N);
                        rtruncnorm_cpu_rcpp(n = N, mu, sd,
                                lo, hi,
                                max_tries_r);
                        })
                })

source("launch_rcuda_env.r");
gpu_times <- sapply(seq_len(ntimes), function(k) {
        system.time({
                N <- as.integer(10^k);
                mu <- rep(2,N);
                sd <- rep(1,N);
                lo <- rep(0,N);
                hi <- rep(1.5, N);
                rtruncnorm_gpu(kernel, x = runif(N), N, mu, sd, lo, hi,
                                compute_grid(N));
                })
        })
```

```
pdf("q1e_run_time.pdf")
plot(log_n, cpu_times["elapsed",], type = 'b', pch = 19,
     xlab = "log(number sampled)", ylab = "elapsed time (seconds)" )
lines(log_n, gpu_times["elapsed",], type = 'b', pch = 19,
                main = "GPU based on Rcpp code", xlab = "log(number sampled)", ylab = "elapsed tim
        col = "blue")
dev.off()
```

5. Appendix C: Probit MCMC Code

5.1. **Probit MCMC.**

```
library(Rcpp)
sourceCpp("rtruncnorm_cpu.cpp")

probit_mcmc <- function(
  y,           # vector of length n
  X,           # (n x p) design matrix
  beta_0,      # (p x 1) prior mean
  Sigma_0_inv, # (p x p) prior precision
  niter,       # number of post burnin iterations
  burnin,      # number of burnin iterations
  GPU)         # logical: use the GPU for Z sampling
{

  library(mvtnorm)
  library(coda)
```

```
##############
#initialize
##############
N <- length(y)
p <- length(beta_0)
#storage
beta <- matrix(nrow = niter + burnin, ncol = p)
beta[1,] <- beta_0;
sigma_z <- rep(1, times = N)
z <- rnorm(N, mean = X %*% beta_0, sd = sigma_z)
y_equals_zero <- y == 0
lo <- ifelse(y_equals_zero, -Inf, 0)
hi <- ifelse(y_equals_zero, 0, Inf)
Sigma_beta_inv <- t(X) %*% X + Sigma_0_inv
Sigma_beta <- solve(Sigma_beta_inv)

if (GPU) {
  cat("Launching GPU Kernel\n")
  source("launch_rcuda_env.r")
  block_grid_dims <- compute_grid(N)
  mem = copyToDevice(runif(N))
}

#gibbs sampler: sample from full conditionals
for(t in seq_len(niter + burnin - 1)) {
  mu_z.t <- X %*% beta[t,]
  if (GPU) {
    z <- rtruncnorm_gpu_mcmc(kernel, mem, N, mu_z.t, sigma_z, lo, hi,
                             block_grid_dims)
```

```
    } else {
      z <- rtruncnorm_cpu_rcpp(N, mu_z.t, sigma_z, lo, hi, 25)
    }
    mu_beta <- Sigma_beta %*% (Sigma_0_inv%*%beta_0 + colSums(X*z) )
    beta[t + 1,] <- rmvnorm(1, mean = mu_beta, sigma =  Sigma_beta)
  }

  #return the betas
  mcmc(data = beta[(burnin + 1):(niter + burnin),])
}

# convenience wrapper function for multiple runs of different data chunks
# for this homework assigment.
run_mcmc <- function(data, niter, burnin, GPU) {
  # number of variables
  p <- dim(data)[2]  - 1
  #run the mcmc
  probit_mcmc(y = data$y, X = as.matrix(data[,2:dim(data)[2]]),
              beta_0 = rep(0, p), Sigma_0_inv = matrix(0,p,p),
              niter, burnin, GPU)
}

validate_mcmc <- function(mcmc_obj, par_file, plot.trace = FALSE) {
  library(coda)
  library(xtable)
  #estimates
  post.beta <- colMeans(mcmc_obj)
  true.beta <- read.table(par_file, header = TRUE)[,1]
```

```
  mat <- as.matrix(mcmc_obj)

  if (plot.trace) {
    pdf(paste0(par_file,".pdf"))
    plot(mcmc_obj)
    dev.off()
  }

  posterior.quantiles <- apply( mat, MARGIN = 2, FUN = quantile,
                                probs = c(0.01, 0.99))
  xtable(rbind(posterior.quantiles, post.beta, true.beta), digits = 3)
}
```