STA250 Homework 4 Report

Shuo Li

December 8, 2013

1 Question 1

1. a)
   The kernel in CUDA C is in the Appendix Code part. The program first simulate random numbers from $N(\mu, \sigma^2)$ and check if it is in the truncated range. After 3000 trials, if the number is still not within the range, the kernel will use the Algorithm in C. P. Robert's paper and identify if it is a two sided truncated normal distribution tail case or a one sided truncated normal distribution tail case and deal with the two cases respectively.

2. b)
   The RCUDA function is in the Appedix Code part.

3. c)
   Using the mtmvnorm() function in R, we can get the theoretical mean of $TN(2, 1; (0, 1.5))$, which is 0.9570067. And the result from my GPU code is 0.951362. They are roughly the same, which proves the correctness of my CUDA kernal and RCUDA code.
   And there is a Scatter-plot and density plot in next page.
   From the plot we can see that all the GPU simulated points are within (0, 1.5), and the points are denser around 1.5 than the points around 0. The red line shows the theoretical mean; the blue dotted line shows the sample mean, they are very close, which also proves the correctness of my CUDA kernel function.

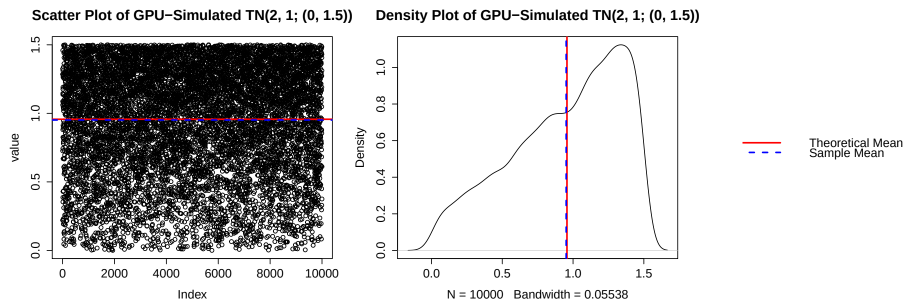

Figure 1:

4. d)

The R function for sampling truncated normal random variables are in the Appendix Code part. And I write two version, the first one is based on inverse CDF algorithm. The second one is of the same algorithm with the GPU one. All the following results are based on the calculation from the inverse-CDF algorithm. Here, the mean of the samples from the truncated normal $TN(2, 1; (0, 1.5))$ is 0.9635237. From part c), the theoretical mean of the truncated normal $TN(2, 1; (0, 1.5))$ is 0.9570067. The sample mean is very close to the true mean, as a result, the effectiveness of the algorithm can be verified. And here is another Scatter-plot and density plot:

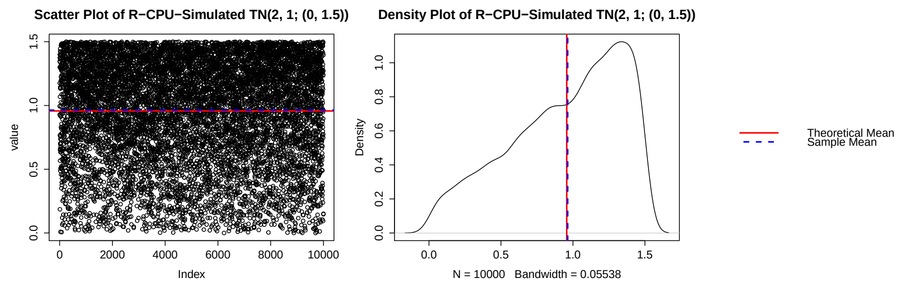

Figure 2:

From the plot we can see that all the CPU simulated points are within (0, 1.5), and the points are denser around 1.5 than the points around 0. The red line and the blue dotted line are very close, which also proves the correctness of my R function.

5. e)
Here, I make a chart to show the processing time of my CPU and GPU code. (The running time are based on "Lipschitz")
For GPU:

	$N=10^1$	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Time: user.self	0.0001313	0.00095	0.0097	0.109	1.064	14.388	186.420	2147.276
Time: sys.self	0.0000001	0.00000	0.0000	0.000	0.003	0.037	0.386	8.186
Time: elapsed	0.0001315	0.00097	0.0097	0.108	1.068	14.445	187.122	2158.895

For CPU:

	$N=10^1$	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Time: user.self	0.032	0.030	0.031	0.032	0.05	0.288	2.439	20.683
Time: sys.self	0.000	0.001	0.001	0.002	0.01	0.041	0.550	7.850
Time: elapsed	0.032	0.030	0.032	0.034	0.06	0.329	2.993	28.533

Here, we can use the elapsed time to compare. We can see that for small N, CPU code is faster. The CPU code begin to be faster than the CPU one for some N in $(1000, 10000)$. And GPU code begin to be far more efficient especially when N is significantly large. In particular, when $N = 10^8$, the processing time for GPU is about 1.3% of the CPU code. I make a series of plot to illustrate the results:

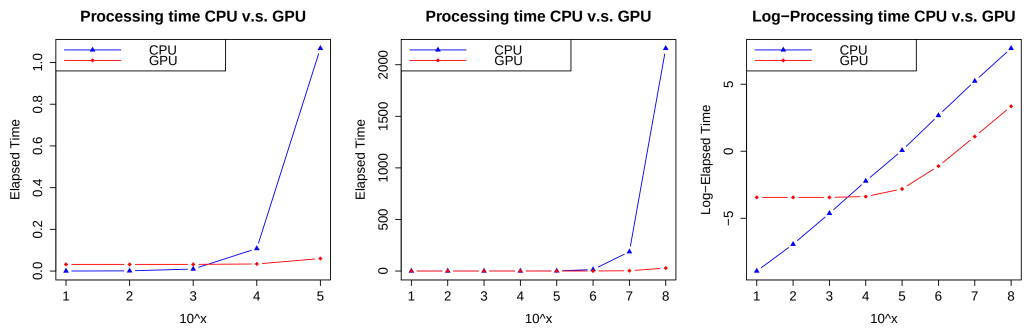

Figure 3:

For CPU code, from the log-transformed elapsed time plot, the time increase is almost linear for $N = 10 \sim 10^8$. When N is multiplied by

10, the running time is about $10 \sim 15$ times the former time.

For GPU, we can find that when N is within $10 \sim 10000$, the running time for GPU code is almost the same as 0.032 second. Then it begin to increase linearly.

6. f)
When $a = -\infty$, the theoretical mean of a truncated normal $TN(2, 1; (-\infty, 1.5))$ is 0.8589222.
For the CPU code, the sample mean is 0.8559228, For the GPU code, the sample mean is 0.8612098. They are both very close to the true mean, which verify the correctness of the code.

   When $b = +\infty$, the theoretical mean of a truncated normal $TN(2, 1; (0, +\infty))$ is 2.055248.
For the CPU code, the sample mean is 2.050537, For the GPU code, the sample mean is 2.052933. They are both very close to the true mean, which verify the correctness of the code.

7. g)
For the truncated regions in the tail of the distribution case, the theoretical mean of $TN(0, 1; (-\infty, -10))$ is -10.09809.
For the CPU case, the sample mean is -10.09953, For the GPU case, the sample mean is -10.09885. They are both very close to the true mean, which verify the correctness of the code.

2 Question 2

1. a)
The Bayesian Probit regression model:
After algebraic derivation, we can see that:
The β prior is from:

$$\beta_{prior} \sim N(\beta_0, \Sigma_0)$$

   The z_i's are from truncated normal distribution:

$$z_i | y_i = 0, \beta \sim TN(x_i^T \beta, 1, (-\infty, 0])$$

$$z_i | y_i = 1, \beta \sim TN(x_i^T \beta, 1, [0, +\infty))$$

And the β posterior is from multivariate normal distribution:

$$\beta_{posterior} \sim N((\Sigma_0^{-1} + X'X)^{-1}(\Sigma_0\beta_0 + X'z), (\Sigma_0^{-1} + X'X)^{-1})$$

For the MCMC algorithm, we set up a start value for $\overrightarrow{\beta}^0$. For each iteration, we sample $z_i^{(t+1)}$ from the truncated normal distribution with $\mu = x_i^T\beta^{(t)}$, then sample $\overrightarrow{\beta}^{(t+1)}$ from multivariate normal $N((\Sigma_0^{-1} + X'X)^{-1}(\Sigma_0\beta_0 + X'z^{(t+1)}), (\Sigma_0^{-1} + X'X)^{-1})$ and repeat the procedure for iteration+burnin times. The R code for the algorithm is in the Appendix code part. (And in this function, I use the rtruncnorm() function in the "truncnorm" package to sample z_i's from truncated normal distribution, instead of the inverse CDF function which I implemented for question 1, to improve the efficiency of CPU computation and save time.)

2. b)
   The RCUDA function is in the Appendix Code part.

3. c)

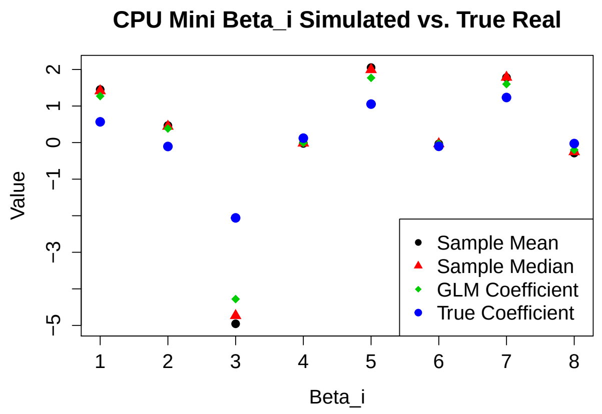

Figure 4:

For the mini data, I make two plot (one for CPU and the other for GPU simulated β's) to show the sample mean, sample median, true

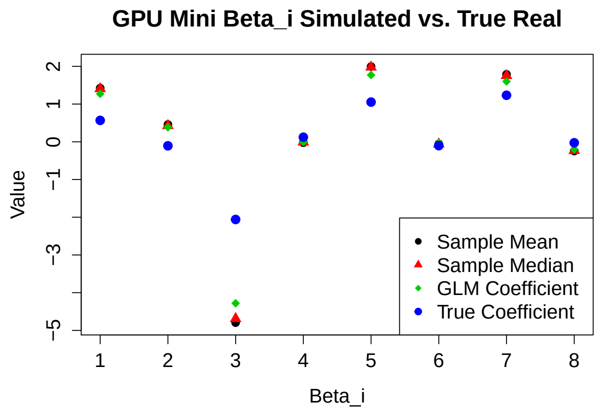

Figure 5:

mean and the estimated regression coefficient from GLM.

We can see from the plots that the CPU & GPU sample mean, sample median and GLM estimated regression coefficient are approximately overlap each other for β_1 to β_8. Meanwhile, they are also close to the true parameters. (Since there are only 100 observations, a bias is usual). These prove the correctness of my CPU and GPU functions. Further, when I simulate β's for the data_01 to data_05 and the there are more observations, the results shows that the sample mean, sample median, GLM estimators and true parameters are all most the same and overlap each other on the plots (the plots for data_01 to data_05 are omitted here), which prove the correctness of my functions and the algorithm.

4. d)
   For the first four data, I run 2000 iterations and the burnin times is set to be 500. For the last data set with $N = 10^7$, the iteration number is set to be 200 and with 50 burnins. The CPU and GPU code are run on the AWS. And the results are in the following charts:

When running the MCMC with sampling z_i's with CPU:

	data_01	data_02	data_03	data_04	data_05
Time: user.self	2.216	15.001	132.784	1336.352	1351.114
Time: sys.self	0.000	0.128	7.152	86.761	125.568
Time: elapsed	2.218	15.130	139.929	1423.043	1476.647

When running the MCMC with sampling z_i's with CPU:

	data_01	data_02	data_03	data_04	data_05
Time: user.self	90.794	106.043	239.363	1568.786	1363.042
Time: sys.self	1.688	4.189	38.822	284.786	346.389
Time: elapsed	92.483	110.229	278.178	1853.455	1708.822

*Note the processing time for data_05 is based on 200 iterations and 50 burnins. And here is a plot to show the CPU and GPU probit regression MCMC code running time for data_01 to data_04 with $N = 1000$ to $N = 10000000$:

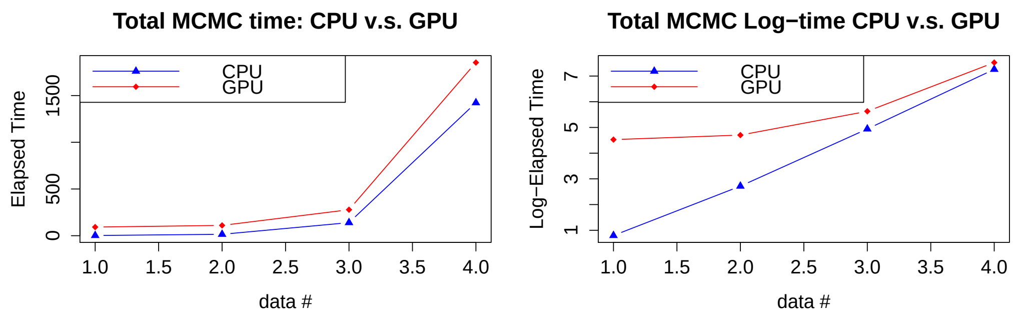

Figure 6:

5. e)
From the charts and plots above, we can see that for data_01 to data_04, the running time for MCMC with GPU sampled truncated normal z_i's are all longer than the CPU one. This may be due to the fact that I use the rtruncnorm() function in the truncnorm package in the CPU code and the computation of the rtruncnorm() function is vectorized, leading to a super efficient result. Another explanation of the result is that the example here requires copying variables to the device and from device for every iteration, this takes much time to finish.

Besides, we notice that the time increase for the CPU one is linear; the time increase for the GPU truncated normal MCMC increase slightly from data_01 to data_02 and then also begin to increase linearly. However, for $N = 10^7$, the slope of the CPU line is steeper than the GPU line, indicating a faster processing-time increase of the CPU code. Running 200 iteration and 50 burnins for data_05, the CPU code is still faster than the GPU one, however it is foreseeable that when the data size is even larger, the efficiency of the GPU MCMC code may surpass the CPU MCMC code.

The example also enlighten us the applicability of GPU programming. If we need to handle big data with simple parallel computation and there are not a lot of data copying/transmission procedure during the process, GPU programming will show better efficiency.

3 Appendix Code

```
##Question 1
##a
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>
#include <math_constants.h>
extern "C"
{
__global__ void
rtruncnorm_kernel(float *vals, int n,
                  float *mu, float *sigma,
                  float *lo, float *hi,
                  int rng_a, int rng_b,
                  int rng_c,
                  int maxtries)
{
    // 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;
if (idx < n)
{
        // Setup the RNG:
        curandState rng;
        curand_init (rng_a+idx*rng_b, rng_c, 0, &rng);
        // Sample:
        int accept=0;
        int numtries=0;
        while (!accept && numtries < maxtries)
        {
              numtries ++;
              vals[idx]=mu[idx]+sigma[idx]*curand_normal(&rng);
              if (vals[idx]>=lo[idx] && vals[idx]<=hi[idx])
              { accept=1; }
              else {}
        }
        if (numtries==maxtries)
        {
              // Handle the One Sided Truncated Normal,
              Distribution Tail Case:
              if (isfinite(hi[idx])==0)
              {
                  float u_lo=(lo[idx]-mu[idx])/sigma[idx];
                  float alpha=(u_lo+sqrt((u_lo*u_lo)+4))/2;
                  int accept2=0;
                  while (!accept2)
                  {
                      float z=u_lo-log(curand_uniform(&rng))/alpha;
                      float u1=curand_uniform(&rng);
                      float qz;
                      if (u_lo < alpha)
                      {qz=exp(-((alpha-z)*(alpha-z))/2);}
                      else {qz=exp(-((u_lo-alpha)*
            (u_lo-alpha))/2-((alpha-z)*(alpha-z))/2);}
            if (u1<qz)
            {
                accept2=1;
                vals[idx]=sigma[idx]*z+mu[idx];
            }
            else {}
        }
    }

    else if  (isfinite(lo[idx])==0)
    {
        float u_lo=-(hi[idx]-mu[idx])/sigma[idx];
        float alpha=(u_lo+sqrt((u_lo*u_lo)+4))/2;
        int accept3=0;
        while (!accept3)
        {
              float z=u_lo
                    -log(curand_uniform(&rng))/alpha;
              float u1=curand_uniform(&rng);
              float qz;
              if (u_lo < alpha)
              {qz=exp(-((alpha-z)*(alpha-z))/2);}
              else {qz=exp(-((u_lo-alpha)*(u_lo-alpha))
              /2-((alpha-z)*(alpha-z))/2);}
              if (u1<qz)
              {
                  accept3=1;
                  vals[idx]=-sigma[idx]*z+mu[idx];
              }
              else {}
        }
    }
    // Handle the Two Sided Truncated Normal,
    Distribution Tail Case:
    else
    {
        int accept4=0;
```

```
                        while (! accept4)
                        {
                                float z;
                                float q;
                                float u;
                                z=lo[idx]+(hi[idx]-lo[idx])*
                                  curand_uniform(&rng);

                                if (0>=lo[idx] && 0<=hi[idx])
                                {
                                     q=exp(-(z*z)/2)  ;
                                }
                                else if (hi[idx]<0)
                                {
                                     q=exp((hi[idx]*hi[idx]-z*z)/2);
                                }
                                else
                                {
                                     q=exp((lo[idx]*lo[idx]-z*z)/2);
                                }
                                u=curand_uniform(&rng);
                                if (u<=q)
                                {
                                     accept4=1;
                                     vals[idx]=z;
                                }
                                else {}
                        }
                   }
           }
     }
     return;
}
} // END extern "C"

##b
rcuda_trun=function(N, rng_a, rng_b, rng_c, maxtries)
{
```

```
library(RCUDA)
cat("Loading module...\n")
m = loadModule("truncnorm.ptx")

cat("done. Extracting kernels...\n")
k_rtruncnorm = m$rtruncnorm_kernel

cat("done. Setting up Parameters...\n")
##Maybe revised when mu, sigma, lo, hi are not the same
lo = rep(0, N)
hi = rep(1.5, N)
mu =rep(2, N)
sigma=rep(1, N)

cat("done. Calculating Grid/ Block dimensions...\n")
bg = compute_grid(N)
grid_dims = bg$grid_dims
block_dims = bg$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")

cat("done. Runing CUDA kernels...\n")
x = rep(0,N)
all_time=system.time({
  mem = copyToDevice(x)
  .cuda(k_rtruncnorm, mem, N, mu, sigma, lo, hi,
  rng_a, rng_b, rng_c, maxtries,
  gridDim=grid_dims, blockDim=block_dims)

  cat("Copying result back from device...\n")
  trun_num = copyFromDevice(obj=mem, nels=mem$nels, type="float")
})
```

```
  write.table(trun_num, "trun.txt", sep=" ")
  cat("Time: \n")
  print(all_time)
  cat("Mean: \n")
  print(mean(trun_num))
  return(all_time)
}

##c
#Set up parameters:
N = 10^4L
rng_a=37L
rng_b=46L
rng_c=23L
maxtries=2000L

#Run:
rcuda_trun(N, rng_a, rng_b, rng_c, maxtries)

##real mean
mu=2
sigma=1
a=0
b=1.5
library(tmvtnorm)
mtmvnorm(mu, sigma, lower=a, upper=b)

##d
#Inverse CDF
trun_2=function (mu, sigma, a, b, n)
{
  num=runif(n, min=0, max=1)
  alpha=(a-mu)/sigma
  beta=(b-mu)/sigma
  sam=sapply(num, function(x) qnorm(pnorm(alpha)+x*(pnorm(beta)
  -pnorm(alpha)))*sigma+mu)
  return(sam)
```

```
}
sam2=trun_2(2, 1, 0, 1.5, 10000)
mean(sam2)

#Alternative version (Same algorithm with the GPU one)
trun = function(mu, sigma, a, b, maxtries=2000)
{
  accept=FALSE
  numtries=0

  while(!accept && numtries < maxtries)
  {
    numtries=numtries+1
    x=rnorm(1, mu, sigma)
    if (x>=a && x<=b)
    {
      accept=TRUE
    }
    else {}
  }

  if (numtries==maxtries)
  {
    if (b==Inf)
    {
      u_lo=(a-mu)/sigma
      alpha=(u_lo+sqrt((u_lo^2)+4))/2
      accept2=FALSE
      while (!accept2)
      {
        z=u_lo+rexp(1, alpha)
        u=runif(1, 0, 1)
        if (u_lo < alpha) {qz=exp(-((alpha-z)^2)/2)}
        else {qz=exp(-((u_lo-alpha)^2)/2-((alpha-z)^2)/2)}
        if (u<qz)
        {accept2=TRUE
         x=sigma*z+mu
        }
        else
        {}
      }
    }

    if (a==-Inf)
    {
      u_lo=-(b-mu)/sigma
      alpha=(u_lo+sqrt((u_lo^2)+4))/2
      accept3=FALSE
      while (!accept3)
      {
        z=u_lo+rexp(1, alpha)
        u=runif(1, 0, 1)
        if (u_lo < alpha) {qz=exp(-((alpha-z)^2)/2)}
        else {qz=exp(-((u_lo-alpha)^2)/2-((alpha-z)^2)/2)}
        if (u<qz)
        {accept3=TRUE
         x=-sigma*z+mu
        }
        else
        {}
      }
    }
  }
  return(x)
}

##e
n_vec=as.integer(c(1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8))

#System time for RCUDA function:
time2=sapply(n_vec, function(n)
      rcuda_trun(n, rng_a, rng_b, rng_c, maxtries))
write.table(time2, "time2.txt", sep=" ")

#System time for pure R code:
#Need to repeat a couple of times if the n is to small to calculate
```

```
#running time
time=sapply(n_vec,
       function(n){
         if(n<100)
           B=10000
         else if(n<1000)
           B=100
         else if (n<10000)
           B=10
         else
           B=1
         system.time(replicate(B,trun_2(2, 1, 0, 1.5, n)))/B
       })
write.table(time, "time.txt")

#Plot the Time for different N's
timecpu=read.table("time.txt", sep=" ")
timegpu=read.table("time3.txt", sep=" ")

par(mfrow=c(1, 3))
plot(x=1:5, y=timecpu[3, 1:5], type="b", col="blue", pch=17,
     xlab="10^x", ylab="Elapsed Time",
     main="Processing time CPU v.s. GPU")
points(x=1:5, y=timegpu[3, 1:5], type="b", col="red", pch=18)
legend("topleft", legend=c("CPU", "GPU"), pch=c(17,18),
       col=c("blue", "red"), lty=1)
plot(x=1:8, y=timecpu[3,], type="b", col="blue", pch=17,
     xlab="10^x", ylab="Elapsed Time",
     main="Processing time CPU v.s. GPU")
points(x=1:8, y=timegpu[3,], type="b", col="red", pch=18)
legend("topleft", legend=c("CPU", "GPU"), pch=c(17,18),
       col=c("blue", "red"), lty=1)
plot(x=1:8, y=log(timecpu[3,]), type="b", col="blue", pch=17,
     xlab="10^x", ylab="Log-Elapsed Time",
     main="Log-Processing time CPU v.s. GPU")
points(x=1:8, y=log(timegpu[3,]), type="b", col="red", pch=18)
legend("topleft", legend=c("CPU", "GPU"), pch=c(17,18),
       col=c("blue", "red"), lty=1)
```

```
##f
#lower bound=-infinity
#CPU:
mtmvnorm(mu, sigma, lower=-Inf, upper=b)
mean(trun_2(2, 1, -Inf, 1.5, 10000))
#GPU:
lo = rep(-Inf, N)
hi = rep(1.5, N)
rcuda_trun(N, rng_a, rng_b, rng_c, maxtries)

#upper bound=+infinity
mtmvnorm(mu, sigma, lower=a, upper=Inf)
mean(trun_2(2, 1, 0, Inf, 10000))
lo = rep(0, N)
hi = rep(Inf, N)
rcuda_trun(N, rng_a, rng_b, rng_c, maxtries)

##g
mtmvnorm(0, 1, lower=-Inf, upper=-10)
#CPU:
mean(trun_2(0, 1, -Inf, -10, 10000))
#GPU:
lo = rep(-Inf, N)
hi = rep(-10, N)
rcuda_trun(N, rng_a, rng_b, rng_c, maxtries)

##Question 2
##a
probit_mcmc_cpu = function(y, X,
                           p, ##length of beta
                           N, ##Number of Observations
                           sigma,##sigma for truncated normal z
                           lo, ##lower bound for truncated normal z
                           hi, ##upper bound for truncated normal z
                           beta.0, ##beta_prior mean
                           Sigma.0.inv, ## beta prior variance inverse
                           niter=2000, burnin=500)
```

```
{
  beta=matrix(numeric(p*(burnin+niter+1)), (burnin+niter+1), p)
  for (i in 1: (burnin+niter))
  {
    ##Sample from truncated normal for z
    u=as.vector(X%*%beta[i, ])
    z=rtruncnorm(1, lo, hi, u, sigma)

    ##Calculate beta_posterior mean and variance
    po_beta_mu=solve(Sigma.0.inv+t(X)%*%X)%*%
               (Sigma.0.inv%*%beta.0+t(X)%*%z)
    po_beta_sigma=solve(Sigma.0.inv+t(X)%*%X)

    ##Sample beta from multivariate normal
    beta[i+1,]=mvrnorm(n = 1, mu=po_beta_mu, Sigma=po_beta_sigma)

    ##Print results for every 250 iterations
    if(i%%250==0)
    {
      cat("Finished", i, "Iterations...\n")
    }
  }
  cat("Work Completed...\n")
  return(beta[(burnin+2):(burnin+niter+1),])
}

##b
probit_mcmc_gpu=function(y, X,
                         p, ##length of beta
                         N, ##Number of Observations
                         sigma, ##sigma for truncated normal z
                         val, ##parameter to store the simulated z
                         lo, ##lower bound for truncated normal z
                         hi, ##upper bound for truncated normal z
                         rng_a, rng_b, ##RNG parameter
                         maxtries,
                         m, k_rtruncnorm,
                         beta_0, ##beta_prior mean
                         Sigma_0_inv, ##beta_prior variance inverse
                         block_dims, grid_dims,
                         niter=2000, burnin=500)
{
  beta=matrix(numeric(p*(burnin+niter+1)), (burnin+niter+1), p)
  for (i in 1: (burnin+niter))
  {
    ##rng_c related to the current iteration time
    rng_c=as.integer(i)

    ##calculate mean for z
    mu=as.vector(X%*%beta[i, ])

    ##sample z using CUDA kernel
    z= .cuda(k_rtruncnorm, "x"=val, N, mu, sigma, lo, hi,
             rng_a, rng_b, rng_c, maxtries, m, k_rtruncnorm,
             gridDim=grid_dims, blockDim=block_dims, outputs="x")

    ##Calculate beta posterior mean and variance
    po_beta_mu=solve(Sigma.0.inv+t(X)%*%X)%*%
               (Sigma.0.inv%*%beta.0+t(X)%*%z)
    po_beta_sigma=solve(Sigma.0.inv+t(X)%*%X)

    ##Sample beta from multivariate normal
    beta[i+1,]=mvrnorm(n = 1, mu=po_beta_mu, Sigma=po_beta_sigma)

    ##Print results for every 250 iterations
    if(i%%250==0)
    {
      cat("Finished", i, "Iterations...\n")
    }
  }
  cat("Work Completed...\n")
  return(beta[(burnin+2):(burnin+niter+1),])
}

##c
#For CPU:
```

```
#Set up parameters:
data=read.table("mini_data.txt", header=TRUE, sep=" ")
y=data$y
X=as.matrix(data[,-1])
p=ncol(X)
N=length(y)
sigma=rep(1, N)
lo=numeric(N)
lo[which(y==0)]=-Inf
hi=numeric(N)
hi[which(y==1)]=Inf
beta.0=rep(0,p)
Sigma.0.inv=matrix(rep(0, p*p), p, p)
#Get simulated betas:
betas=probit_mcmc_cpu(y, X, p, N, sigma, lo, hi, beta.0, Sigma.0.inv)
#Plot the CPU result:
par=read.table("mini_pars.txt", header=TRUE)$V1
fit=glm(y ~ -1+X_1+X_2+X_3+X_4+X_5+X_6+X_7+X_8, data,
        family = binomial(link = "probit"))
glmcoef=fit$coefficient
max=max(c(colMeans(betas)), c(apply(betas, 2, median)),
        c(glmcoef), c(par))
min=min(c(colMeans(betas)), c(apply(betas, 2, median)),
        c(glmcoef), c(par))
plot(colMeans(betas), ylim=c(min-0.05, max+0.05), pch=16,
     col=1, cex=1.3, xlab="Beta_i", ylab="Value",
     main="CPU Mini Beta_i Simulated vs. True Real")
points(apply(betas, 2, median), pch=17, col=2, cex=1.3)
points(glmcoef, pch=18, col=3, cex=1.3)
points(par, cex=1.3, col=4, pch=19)
col=c("green", "red", "blue", "yellow")
legend ("bottomright", legend=c("Sample Mean", "Sample Median",
        "GLM Coefficient", "True Coefficient"),
        col=c(1:4), pch=c(16:19), cex=1)

#For GPU:
#Set up parameters:
data=read.table("mini_data.txt", header=TRUE, sep=" ")
```

```
y=data$y
X=as.matrix(data[,-1])
p=ncol(X)
N=length(y)
sigma=rep(1, N)
val=rep(0, N)
lo=numeric(N)
lo[which(y==0)]=-Inf
hi=numeric(N)
hi[which(y==1)]=Inf
beta.0=rep(0,p)
Sigma.0.inv=matrix(rep(0, p*p), p, p)

bg = compute_grid(N)
grid_dims = bg$grid_dims
block_dims = bg$block_dims

rng_a=37L
rng_b=46L
maxtries=2000L

m = loadModule("truncnorm.ptx")
k_rtruncnorm = m$rtruncnorm_kernel

betas=probit_mcmc_gpu(y, X, p, N, sigma, val, lo, hi,
                        rng_a, rng_b, maxtries, m, k_rtruncnorm,
                        beta.0, Sigma.0.inv, block_dims, grid_dims)
#Plot the GPU result:
betas=read.table("betas.txt", sep=" ")
max=max(c(colMeans(betas)), c(apply(betas, 2, median)),
         c(glmcoef), c(par))
min=min(c(colMeans(betas)), c(apply(betas, 2, median)),
         c(glmcoef), c(par))
plot(colMeans(betas), ylim=c(min-0.05, max+0.05), pch=16,
      col=1, cex=1.3, xlab="Beta_i", ylab="Value",
      main="GPU Mini Beta_i Simulated vs. True Real")
points(apply(betas, 2, median), pch=17, col=2, cex=1.3)
points(glmcoef, pch=18, col=3, cex=1.3)
points(par, cex=1.3, col=4, pch=19)
legend ("bottomright", legend=c("Sample Mean", "Sample Median",
        "GLM Coefficient", "True Coefficient"),
        col=c(1:4), pch=c(16:19), cex=1)

##d
names=c("data_01.txt", "data_02.txt", "data_03.txt",
      "data_04.txt", "data_05.txt")
#Compute CPU time for data_01 to data_05
compute_cpu_time=function(names)
{
  library(truncnorm)
  library(MASS)
  cat("Begin Computing: ", names, " \n")

  ##Set up Parameters:
  data=read.table(names, header=TRUE, sep=" ")
  y=data$y
  X=as.matrix(data[,-1])
  p=ncol(X)
  N=length(y)
  sigma=rep(1, N)
  lo=numeric(N)
  lo[which(y==0)]=-Inf
  hi=numeric(N)
  hi[which(y==1)]=Inf
  beta.0=rep(0,p)
  Sigma.0.inv=matrix(rep(0, p*p), p, p)

  ##Record computing time:
  time=system.time({betas=probit_mcmc_cpu(y, X, p, N, sigma, lo, hi,
       beta.0, Sigma.0.inv)})
  cat("Time for ", names, " is ", time, "...\n")
  return(time)
}
cpu=sapply(names, function(x) compute_cpu_time(x))

#Compute GPU time for data_01 to data_05
```

```
compute_gpu_time=function(names)
{
   cat("Begin Computing: ", names, " \n")

   ##Set up Parameters:
   data=read.table(names, header=TRUE, sep=" ")
   y=data$y
   X=as.matrix(data[,-1])
   p=ncol(X)
   N=length(y)
   sigma=rep(1, N)
   val=rep(0, N)
   lo=numeric(N)
   lo[which(y==0)]=-Inf
   hi=numeric(N)
   hi[which(y==1)]=Inf
   beta.0=rep(0,p)
   Sigma.0.inv=matrix(rep(0, p*p), p, p)
   rng_a=37L
   rng_b=46L
   maxtries=2000L

   ##Calculate Block, Grid dimensions
   bg = compute_grid(N)
   grid_dims = bg$grid_dims
   block_dims = bg$block_dims

   m = loadModule("truncnorm.ptx")
   k_rtruncnorm = m$rtruncnorm_kernel

   ##Calculate running time
   time=system.time({betas=probit_mcmc_gpu(y, X, p, N, sigma, val,
                          lo, hi, rng_a, rng_b, maxtries, m, k_rtruncnorm,
                          beta.0, Sigma.0.inv, block_dims, grid_dims)})
   cat("Time for ", names, " is ", time, "...\n")
   return(time)
}
gpu=sapply(names, function(x) compute_gpu_time(x))
```

```
#Plot the time:
par(mfrow=c(1, 2))
plot(x=1:4, y=cpu[3, ], ylim=c(min(gpu[3, ], cpu[3, ]),
     max(gpu[3, ], cpu[3, ])), type="b", col="blue", pch=17,
     xlab="data␣#", ylab="Elapsed␣Time",
     main="Total␣MCMC␣time:␣CPU␣v.s.␣GPU")
points(x=1:4, y=gpu[3, ], type="b", col="red", pch=18)
legend("topleft", legend=c("CPU", "GPU"), pch=c(17,18),
       col=c("blue", "red"), lty=1)
plot(x=1:4, y=log(cpu[3,]), ylim=c(log(min(gpu[3, ], cpu[3, ])),
     log(max(gpu[3, ], cpu[3, ]))), type="b", col="blue", pch=17,
     xlab="data␣#", ylab="Log-Elapsed␣Time",
     main="Total␣MCMC␣Log-time␣CPU␣v.s.␣GPU")
points(x=1:4, y=log(gpu[3,]), type="b", col="red", pch=18)
legend("topleft", legend=c("CPU", "GPU"), pch=c(17,18),
       col=c("blue", "red"), lty=1)
```