STA250, Homework 4
Christopher Aden
December 7, 2013

1 Generating Truncated Normals on the GPU

The objective of this section to sample from a truncated normal distribution on the GPU. This is necessary in some survival models, and, as we'll see later, probit regression. A random variable X having a truncated normal distribution with mean μ and standard deviation σ and truncation points (a, b) can be thought of as a $N(\mu, \sigma^2)$, with support defined instead on (a, b), and re-normalized to make it a valid PDF.

Sampling is accomplished several ways, each with pros and cons. I will detail all the implementations I use in my code. To ease notation, let $a' = (a - \mu)/\sigma$ and similar for b'. This will allow us to work on standard normals and rescale at the end.

A. Inverse-CDF:

- Draw $u \sim U[\Phi(a'), \Phi(b')]$.
- Compute $z = \Phi^{-1}(u)$.
- $x = \sigma * z + \mu \sim TN(\mu, \sigma^2, (a, b))$.

B. Naive Rejection Sampling

- Draw $x \sim N(\mu, \sigma)$
- If $x \in (a, b)$, return x. Else, repeat.

C. One-Sided Truncation using Truncated Exponential

- Require either $a = -\infty$ or $b = \infty$. Call $T = a'$ if $b = \infty$, or $T = -b'$ if $a = \infty$.
- Draw $z \sim Exp(\alpha) + T$, a truncated exponential, where $\alpha = \frac{T+\sqrt{T^2+4}}{2}$.
- If $T < \alpha$, let $\log \rho = -(z - \alpha)^2/2$ Else, $\log \rho = \frac{(T-\alpha)^2-(\alpha-z)^2}{2}$.
- Draw $u \sim U(0, 1)$. If $\log u \leq \log \rho$ and $T = a'$, return $x = \mu + \sigma * z$. Else if $\log u < \log \rho$ and $T = -b'$, return $x = -\mu + \sigma * z$. Else, repeat.

D. Robert (2009) Rejection Sampling

- Draw $z \sim U(a', b')$, $u \sim U(0, 1)$.
- If $a' < 0$ and $b' > 0$, let $\log \rho = -z^2/2$. Else if $a' > 0$, $\log \rho = -(a'^2 - z^2)/2$. Else, $\log \rho = -(b'^2 - z^2)/2$.
- If $\log u \leq \log \rho$, return $x = \mu + \sigma * z$. Else, repeat.

The inverse-CDF method is extremely fast and returns samples every single time. This means there is no looping and no throwing away samples. It, however, has very poor behavior when the truncation region is very far away from the mean, because of difficulty in calculating the CDF and inverse-CDF in single-precision floating point math. Through heuristics, I found this method unstable when the closest truncation point point was more than four standard deviations away from the mean. Luckily, that means this method is appropriate the vast majority of the time.

The Naive rejection sampling scheme was easy to write, but suffers a similar problem as the inverse-CDF method. It is numerically more stable for extreme truncation, but what it has in numerical stability it makes an unbelievable sacrifice in speed for. This method only exists as a fallback for inverse-CDF. I was more willing to make a sacrifice in stability in exchange for speed, as we would expect a $N(0, 1, (1.645, \infty))$ to require 20 repetitions before getting even one value!

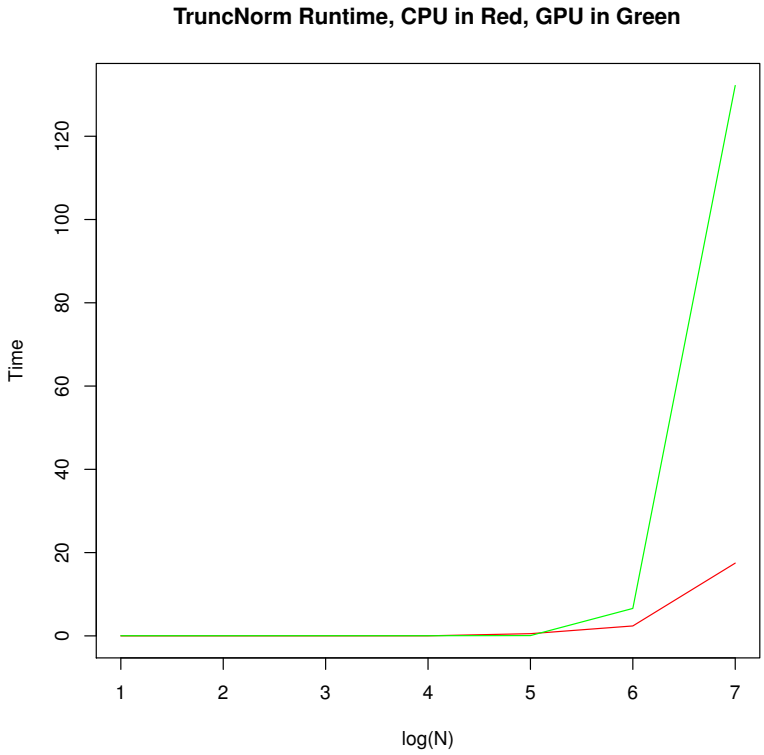
The one-sided truncation method is extremely effective and is able to beat the two naive methods in both speed and precision handily, provided there is only one point of truncation. Logarithms were used to make it more stable in the tails, and theoretical results show that as the truncation point approaches the unbounded side of the support, our acceptance probability goes to 1. This makes for efficient sampling in the tails.

The two-sided method proposed by Christian Robert is slower than the one-sided method in many cases, but is still more effective than the naive rejection sampling algorithm unless the truncation points are close to the mean. For this reason, we use it when we have two-sided truncation in the tails to similar success as the previous method.

Implementing this on the GPU was a near-triviality, given the embarrassingly parallel nature of the task. In the interest of not blowing my stack, arguments could be passed in as much smaller arrays than the total number of samples, and their arguments were recycled. This means I only needed one really large memory copy–the one from the device to the host with all the samples.

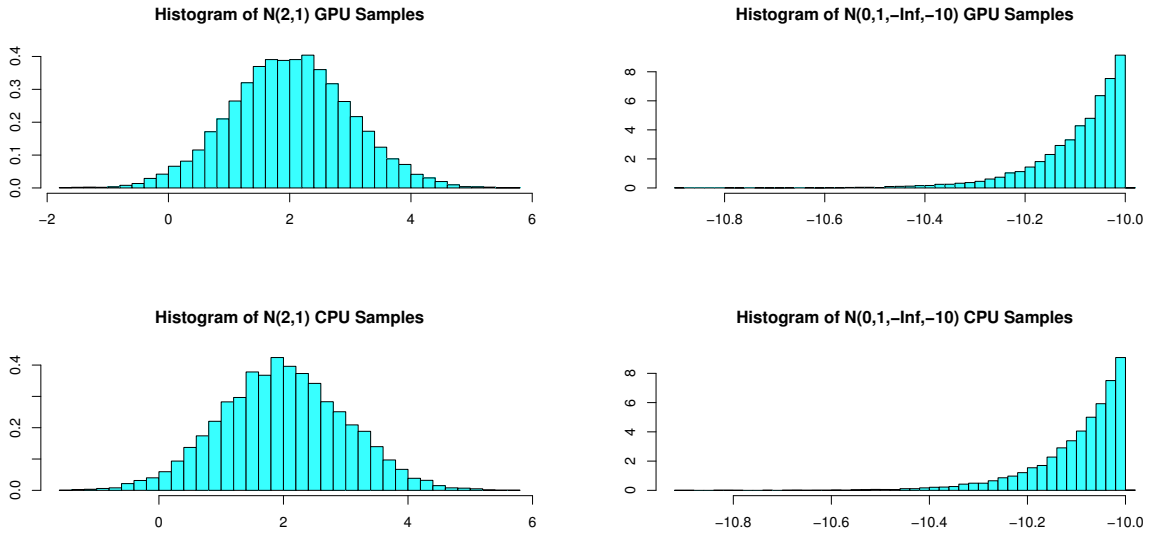
Confirming the behavior of the sampler, the sample mean of 10,000 samples was not super close to the mean (sample mean: 0.962903. True Mean: 0.957007), but the approximation did get better with more samples. There is also the added handicap of the GPU using single-precision floating point numbers. The CPU implementation was able to get much closer (sample mean: 0.959399, true mean: 0.957007).

An implementation in the `msm` package written mostly in serial C was still drastically faster than my CUDA implementation for sampling from $TN(2, 1, (0, 1.5))$.



There are likely a few factors at play here. The algorithms were just fine–I originally wrote my GPU code entirely in C (including the memory management) and the speed was fantastic. Using RCUDA, I imported the kernel's LLVM code, specified the number of blocks and grids, copied the parameters to the device, then let R run the kernel. Perhaps it was my initialization of the results vector. I'm about 90% sure the reason the GPU code was much slower was inappropriate allocation of the device and host memory, and incorrect copies. Smarter programming could've drastically brought the run time down, since the GPU handles the computation drastically faster than the serial CPU implementation. I was unable to generate 100,000,000 truncated normal samples using either GPU or CPU, as R was unable to hold them in memory, and CUDA couldn't allocate an array that large. Potentially, I could've run the code to the maximum amount of usable memory, then stored the results on disk until they were needed.

At least it managed to correctly sample from $TN(2, 1, (-\infty, \infty))$ and $TN(0, 1, (-\infty, -10))$, so while it's slow, at least it gets the answers right.



2 Bayesian Probit Regression

Now that we have a (somewhat) working truncated normal sampler, let's direct our attention to an actual problem: Bayesian probit regression. The idea is simple: We observe an indicator function y of an unobserved latent variable, z. To assess a relationship between y and some covariates, we suppose that our z's have a linear relationship to the covariates. If we impose normality on the latent variables, we can say that the probability of $y = 1$ is $p = \Phi(X^T\beta)$. Probit regression is not a huge leap of the imagination if one is familiar with logistic regression–the difference is simply the choice of the link function. The goal is to estimate the influence of the covariates β on the probability of observed y being 1 or 0. Being Bayesians, we can also place prior belief on the values for β.

For this analysis, we place a completely diffuse, conditionally conjugate prior on β. Let n denote the sample size and p denote the number of covariates (including intercept).

$$\begin{aligned}
\beta &\sim N(\beta_0 = 0_p, \Sigma_0^{-1} = 0_{p\times p}) \\
z_i|\beta &\sim N(X^T\beta_i, 1) \\
y_i|z_i &= 1_{(z_i>0)}
\end{aligned}$$

We sample from the posterior of β, given y, using Gibbs Sampling.

$$\begin{aligned}
z_i^t|\beta^t, y_i = 0 &\sim N(X^T\beta_i^t, 1)1_{(-\infty,0)} \\
z_i^t|\beta^t, y_i = 1 &\sim N(X^T\beta_i^t, 1)1_{(0,\infty)} \\
\beta^{t+1}|z^t, y &\sim N(b_z, \Sigma),
\end{aligned}$$

where the posterior variance Σ is a combination of the prior variance and the variance of the OLS estimator,

$$\Sigma = (\Sigma_0^{-1} + X'X)^{-1},$$

and the posterior mean is similar to the OLS estimate with an additional term to account for prior mean,

$$b_z = \Sigma(\Sigma_0^{-1}\beta_0 + X'z)$$

It is worth noticing that at each step, with a completely diffuse prior as we have, the posterior mean and variance are identical to the distribution of the classical least squares estimate if we regressed z on our covariates. Our algorithm for getting posterior samples will thus be:

A Set $\beta^1 = (X'X)^{-1}X'y$, the least squares solution.

B Determine truncation points: $L_i = -\infty 1_{y_i=0}$, $H_i = \infty 1_{y_i=1}$, for $i = 1, \ldots, n$.

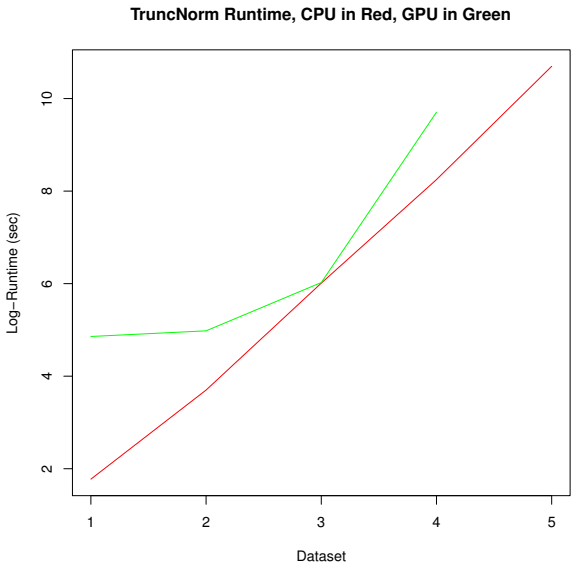
For each $i = 1, \ldots,$ maxiter:

1. Sample z_i from $TN(X'\beta^i, 1, (L_i, H_i))$
2. Sample β^{t+1} from Multivariate $N(b_z, \Sigma)$.

A faux-CUDA version is implemented by swapping out the truncated normal sampler for a GPU-version. Ideally, a GPU implementation would do the multivariate normal, the truncated normal, and possibly some of the matrix math on the GPU. This proved unwieldy for memory management reasons. A more intelligent method of manipulating memory might alleviate some of this strain. Not having to transfer the latent variable z's off the GPU could drastically improve performance.

Testing the CPU and GPU implementations on $n = 1000$ dataset, the posterior matches up nicely with the true underlying values of β. Setting the initial value at the OLS estimator was smart–technically, we did not need a 500-sample burn-in, and burning 500 observations didn't do much for us.

What did kill us, however, was the number of iterations. Having to pass around the latent variables device the device and host was devastating for a larger data set. The actual computation was inconsequential in comparison. If I had managed my own memory, I wouldn't have passed the latent variables back and forth between device and host. Ideally, level 2 BLAS functionality could've done the math on the GPU itself, as the only time I needed the latent variables was in calculating the posterior mean of β.



The serial implementation increased in runtime linearly with the sample size. From my pure-C implementation of the truncated normal when I was developing it, I noticed the computing time was relatively constant, or at least scaled far better than the computing time of the CPU implementation. There is no runtime available for the final dataset on the GPU–it took too long to run.

If the memory copies and allocation could be done once and written over when needed, I am certain the GPU would beat the CPU implementation after the initial cost of allocating memory on the GPU took the same amount of time as computing samples on the CPU. This is problem-dependent, but it is possible we could see a significant speedup as soon as a sample size of 100,000 or 1,000,000.

R Source Code

The CUDA truncated normal kernel and helper functions.

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

extern "C"
{
  __device__ float sim_inv_cdf( float mu, float sigma, float phi_a, float phi_b, curandState localState)
  { //Generate trunc StdNorm by inverse CDF if conditions are stable.
    float u = ((phi_b - phi_a) * curand_uniform(&localState)) + phi_a;
    float x = sigma * normcdfinvf(u) + mu;
    return x;
  } //Gets valid sample every time--so speed, very wow.

  __device__ float simple_rejection( float mu, float sigma, float stdA, float stdB, int maxtries,
      curandState localState)
  {
    float res = 1.0f/0.0f; //set result to inf. For error checking.
    float z;

    for(int i=0; i < maxtries; i++)
    {
      z = curand_normal(&localState);
      if ( z <= stdB && z >= stdA)
      {
        res = mu + (sigma*z);
        return res;
      }
    } //end maxtry for loop
    return res;
  } //end simple rejection-sampler.

  __device__ float one_sided_trunc( float mu, float sigma, float stdA, float stdB, curandState localState)
  {
    float logRho, z, logU, res, alpha, trunc;

    if (isinf(stdA)) trunc=-stdB; else trunc=stdA; //If a is -Inf, then b is truncated. Else, a is
        truncated.

    alpha = (trunc + sqrtf(trunc*trunc + 4))/2; //Optimal alpha
    do { //Start rejection sample loop.
      z = (-logf( curand_uniform(&localState) )/ alpha) + trunc; //Truncated Exponential
      if (trunc < alpha) logRho = -((z-alpha) * (z-alpha)) / 2;
      else logRho = ((trunc-alpha)*(trunc-alpha) -(alpha-z)*(alpha-z)) / 2;
      logU = logf(curand_uniform(&localState));
    } while (logU > logRho);

    //If left trunc, do as usual. If right trunc, reflect the z value, then add mu.
    if (isinf(stdB)) res = mu + sigma * z;
    else res = mu - sigma*z;
    return res;
  } //end one-sided sampler.

  __device__ float robert( float mu, float sigma, float stdA, float stdB, curandState localState)
  { //Condns unstable for inv-CDF. Do Robert (2009).
    float logrho, z, logu, res;
    do {
      z = (stdB-stdA) * curand_uniform(&localState) + stdA;
      logu = logf( curand_uniform(&localState) );
      if ( stdA<=0 && stdB>= 0 ) logrho = -(z*z)/2;
      else if (stdA > 0) logrho = -((stdA*stdA)-(z*z))/2;
      else logrho = -((stdB*stdB)-(z*z))/2;
    } while(logu > logrho);

    res = sigma*z + mu;
    return res;
```

```
67    } //end Robert sampler.
68
69    __global__ void truncnormal_kernel(float *result, int n, float *mu,
70              float *sigma, float *a, float *b, int maxtries,
71              int mu_len, int sigma_len, int a_len, int b_len)
72    {
73      int myblock = blockIdx.x + blockIdx.y * gridDim.x;
74      int blocksize = blockDim.x * blockDim.y * blockDim.z;
75      int subthread = threadIdx.z*(blockDim.x * blockDim.y) + threadIdx.y*blockDim.x + threadIdx.x;
76      int idx = myblock * blocksize + subthread;
77
78      if (idx >= n) return; //Index is larger than sample size--do no calculation here.
79      curandState localState;
80      curand_init(idx, idx, 0, &localState);
81
82      //Declare vars in thread-local memory.
83      float t_a = a[idx % a_len];
84      float t_b = b[idx % b_len];
85      float t_mu = mu[idx % mu_len];
86      float t_sigma = sigma[idx % sigma_len];
87      float res = 1.0f/0.0f;
88
89      float stdA = (t_a - t_mu)/t_sigma; //Standardize truncation points. Done in-thread.
90      float stdB = (t_b - t_mu)/t_sigma;
91
92      float phi_a = normcdff(stdA); //Calculate CDF of trunc points from StdNormal.
93      float phi_b = normcdff(stdB);
94
95      if ( phi_b - phi_a > 0.02f )
96      { //If stable conditions, use inverse-CDF.
97        result[idx] = sim_inv_cdf(t_mu, t_sigma, phi_a, phi_b, localState);
98        return;
99      }
100
101     else if ( isinf(stdA) || isinf(stdB) )
102     { //One-sided truncation.
103       res = one_sided_trunc( t_mu, t_sigma, stdA, stdB, localState);
104       result[idx] = res;
105       return;
106     }
107
108     else if ( stdB-stdA >= sqrtf(6.2831853f) && phi_b-phi_a > 0.0001f)
109     { //Two-sided trunc with truncs far away--do the naive rejection sampler
110       res = simple_rejection(t_mu, t_sigma, stdA, stdB, maxtries, localState);
111       if (!isinf(res))
112       {
113         result[idx] = res;
114         return;
115       }
116     }
117     else{ //Do the Robert method.
118       res = robert(t_mu, t_sigma, stdA, stdB, localState);
119       result[idx] = res;
120       return;
121     } //End truncation regions on same side of mean.
122   } //end truncnorm kernel
123 } //end extern C.
```

Functions to run the make file, load the CUDA kernel into R, and make it interface nicer with R.

```
library(RCUDA)
library(MASS)
library(msm)
library(coda)
library(compiler)
library(mvtnorm)

cuGetContext(TRUE)
system("make k_tnorm.ptx") #Run ptx makefile.
m = loadModule("k_tnorm.ptx")
k_tnorm = m$truncnormal_kernel #Get TNorm kernel

tnorm.gpu = function(N, mu=0.0, sigma=1.0, a=-Inf, b=Inf, maxtries=5000, threads_per_block=512L, verbose=
    FALSE)
{
    a_len = length(a)
    b_len = length(b)
    mu_len = length(mu)
    sigma_len = length(sigma)

    # N = 10,000 < 32*512 => 32 blocks of 512 threads
    block_dims <- c(threads_per_block, 1L, 1L)
    grid_d1 <- max(floor(sqrt(N/threads_per_block)), 1)
    grid_d2 <- ceiling(N/(grid_d1*threads_per_block))
    grid_dims <- c(grid_d1, grid_d2, 1L)
    nthreads <- prod(grid_dims)*prod(block_dims)

    if (verbose) {
      print(paste("Grid size:", list(grid_dims), sep=" "))
      print(paste("Block size:", list(block_dims), sep=" "))
      print(paste("Total number of threads to launch = ", nthreads, sep=""))
  }

    d_mu <- copyToDevice(mu)
    d_sigma <- copyToDevice(sigma)
    d_a <- copyToDevice(a)
    d_b <- copyToDevice(b)
    d_results <- copyToDevice(rep(0.0, N))

    if (verbose) cat("Launching tnorm kernel.\n")
    results = .cuda(k_tnorm, "results"=d_results, as.integer(N), d_mu, d_sigma, d_a, d_b, as.integer(
        maxtries),
    as.integer(mu_len), as.integer(sigma_len), as.integer(a_len), as.integer(b_len),
    gridDim=grid_dims, blockDim=block_dims, outputs="results")
    if (verbose) cat("Done with CUDA tnorm kernel.\n")
    return(results)
}
```

Code to put the truncated normal sampler through its paces:

```
#setwd("~/Dropbox/STA250/Assignments/HW4")
source("helper_functions.R")

N=1E4
mu=2; sigma=1
lower=0; upper=1.5

#Part C
truth = mu + (dnorm(lower, mu, sigma) - dnorm(upper, mu, sigma)) * sigma / (pnorm(upper, mu, sigma)-pnorm(
    lower, mu, sigma))
samples.mean.gpu = mean(tnorm.gpu(N, mu, sigma, lower, upper))
print(paste0("GPU Sample Mean: ", round(samples.mean.gpu, 6), ". True Mean: ", round(truth, 6)))

#Part D
samples.mean.cpu = mean(rtnorm(N, mu, sigma, lower, upper))
print(paste0("CPU Sample Mean: ", round(samples.mean.cpu, 6), ". True Mean: ", round(truth, 6)))

#Part E
```

```
logn = 7
GPU_time = CPU_time = numeric(logn)

for(i in 1:logn){
  paste0("Working on Dataset i= ", i)
  GPU_time[i] = system.time( tnorm.gpu(as.integer(10^i), mu, sigma, lower, upper) )["elapsed"]
  CPU_time[i] = system.time( rtnorm(   as.integer(10^i), mu, sigma, lower, upper) )["elapsed"]
}

setEPS()
postscript('TruncNorm_Timings.eps')
plot(1:logn, CPU_time, xlab="log(N)", ylab="Time", main="TruncNorm Runtime, CPU in Red, GPU in Green",
     type='l', col='red',
      ylim=c(min(c(CPU_time, GPU_time)), max(c(CPU_time, GPU_time)) ) )
lines(1:logn, GPU_time, type='l', col='green')
dev.off()
save(GPU_time, CPU_time, file="Timings.Rdata")

#Part F:
GPU.samps = tnorm.gpu(N, mu, sigma, a=-Inf, b=Inf)
CPU.samps = rtnorm(N, mu, sigma, lower=-Inf, upper=Inf)

setEPS()
postscript('No_TruncNorm_Hist.eps')
par(mfrow=c(2,1))
truehist(GPU.samps, main="Histogram of N(2,1) GPU Samples", xlab="")
truehist(CPU.samps, main="Histogram of N(2,1) CPU Samples", xlab="")
dev.off()

#Part G:
GPU.samps = tnorm.gpu(N=1E4, mu=0.0, sigma=1.0, a=-Inf, b=-10)
CPU.samps = rtnorm(n=1E4, mean=0.0, sd=1.0, lower=-Inf, upper=-10)

setEPS()
postscript('Tail_TruncNorm_Hist.eps')
par(mfrow=c(2,1))
truehist(GPU.samps, main="Histogram of N(0,1,-Inf,-10) GPU Samples", xlab="")
truehist(CPU.samps, main="Histogram of N(0,1,-Inf,-10) CPU Samples", xlab="")
dev.off()
```

Code to do Bayesian probit regression:

```
#setwd("~/Dropbox/STA250/Assignments/HW4")

#Figure out if on AWS. If yes, do GPU comp, else CPU.
AWS = ifelse(Sys.info()["user"]=="ec2-user", TRUE, FALSE)

if (AWS){ #Do GPU stuff
  source("helper_functions.R")
  probit_mcmc_gpu = function(y, X, beta_0 = NA, Sigma_0_inv=NA, niter=2000, burnin=500, maxtries=250,
       threads_per_block=512L, verbose=FALSE, Datapath=NA)
  { #Noninformative prior--don't need to specify prior variance or mean.
    if( !is.na(Datapath) ){
      Data = read.table(Datapath, header=TRUE)
      y=as.numeric(Data[,1])
      X=as.matrix(Data[,2:ncol(Data)])
    }
    p = ncol(X)
    n = nrow(X)
    if (all(is.na(beta_0))) beta_0 = rep(0, p)
    if (all(is.na(Sigma_0_inv))) Sigma_0_inv = matrix(0, p, p)
    z = numeric(p)
    beta = matrix(NA, nrow=niter, ncol=p)
    beta[1, ] = lm.fit(X,y)$coefficients #Start with OLS.

    uppers = ifelse(y==1, Inf, 0) #If y=1, then z>0, so trunc points were (0, Inf)
    lowers = ifelse(y==1, 0, -Inf) #If y=0, z<=0, so trunc points were (-Inf, 0)

    #If Flat Prior, posterior variance is (X'X)^-1 .
    beta_var = solve(Sigma_0_inv + t(X) %*% X)

    for (i in 1:(niter-1)) {
      #Sample (z|y,beta)
```

```
      z_mean = X %*% beta[i,]
      z = tnorm.gpu(as.integer(n), z_mean, 1, lowers, uppers)

      #Sample (beta | z, y)
      if (all(Sigma_0_inv==0)) {beta_mean = lm.fit(X, z)$coefficients} else{
        beta_mean = beta_var %*% (Sigma_0_inv %*% beta_0 + t(X) %*% z)}
      beta[i+1, ] = rmvnorm(1, mean=beta_mean, sigma=beta_var)
    }
    return(mcmc(beta[(burnin+1):niter, ]))
  }
  probit_mcmc_gpu = cmpfun(probit_mcmc_gpu)
} else {
  library(msm)
  library(coda)
  library(compiler)
  library(mvtnorm)
  probit_mcmc_cpu = function(y, X, beta_0 = NA, Sigma_0_inv=NA, niter=2000, burnin=500, Datapath=NULL)
  { #Noninformative prior--don't need to specify prior variance or mean.
    if( !is.na(Datapath) ){
      Data = read.table(Datapath, header=TRUE)
      y=as.numeric(Data[,1])
      X=as.matrix(Data[,2:ncol(Data)])
    }
    p = ncol(X)
    n = nrow(X)
    if (all(is.na(beta_0))) beta_0 = rep(0, p)
    if (all(is.na(Sigma_0_inv))) Sigma_0_inv = matrix(0, p, p)
    z = numeric(p)
    beta = matrix(NA, nrow=niter, ncol=p)
    beta[1, ] = lm.fit(X,y)$coefficients #Start with OLS.

    uppers = ifelse(y==1, Inf, 0) #If y=1, then z>0, so trunc points were (0, Inf)
    lowers = ifelse(y==1, 0, -Inf) #If y=0, z<=0, so trunc points were (-Inf, 0)

    #If Flat Prior, posterior variance is (X'X)^-1 .
    beta_var = solve(Sigma_0_inv + t(X) %*% X)

    for (i in 1:(niter-1)) {
      #Sample (z|y,beta)
      z_mean = X %*% beta[i,]
      z = rtnorm(n, mean=z_mean, sd=rep(1, n), lower=lowers, upper=uppers)

      #Sample (beta | z, y)
      if (all(Sigma_0_inv==0)) {beta_mean = lm.fit(X, z)$coefficients} else{
        beta_mean = beta_var %*% (Sigma_0_inv %*% beta_0 + t(X) %*% z)}
      beta[i+1, ] = rmvnorm(1, mean=beta_mean, sigma=beta_var)
    }
    return(mcmc(beta[(burnin+1):niter, ]))
  }
  probit_mcmc_cpu = cmpfun(probit_mcmc_cpu)
}

source("sim_probit.R") #Generate the data
GPU_times = CPU_times = numeric(5)
filenames = paste0("data_0", 1:5, ".txt")

if(AWS){
  for (i in 1:5){ #Times saved seperately so I could write the paper asyncronously.
    GPU_times[i] = system.time(probit_mcmc_gpu(Datapath=filenames[i]))["elapsed"]
    GPU = GPU_times[1:i]
    save(GPU, file=paste0("GPU_probit_times_0",i,".RData"))
  }
} else{
  for (i in 1:5){ #Times saved seperately so I could write the paper asyncronously.
    CPU_times[i] = system.time(probit_mcmc_cpu(Datapath=filenames[i]))["elapsed"]
    CPU = CPU_times[1:i]
    save(CPU, file=paste0("CPU_probit_times_0",i,".RData"))
  }
}
```