GPU-optimized math routines in Stan

Rok Češnovar, Davor Sluga, Jure Demšar, Steve Bronder, Erik Štrumbelj

Apr 15, 2018

Introduction

???

GPU implementation

Currently, we aim at speeding up the biggest computational bottlenecks on the GPU, while the remaining (non-parallelized) Stan code is executed on the CPU. Therefore, we move the data to and from the GPU for each GPU-parallelized function. Removing this often unnecessary data transfer to and from the GPU is one of the main priorities of future work.

The biggest bottlenecks identified and parallelized so far are the Cholesky decomposition, its derivative and the derivative of solving $Ax = B$, where A is a lower triangular matrix. In order to implement all three bottlenecks on the GPU, the following GPU functions were implemented (some are extensions of our previous work in (Češnovar and Štrumbelj 2017)):

- matrix transpose,
- multiplication of a matrix or its diagonal with a scalar,
- subtraction of matrices,
- copying submatrices,
- matrix multiplication,
- lower triangular matrix inverse, and
- Cholesky decomposition.

Our focus was on the latter three computational building blocks, the rest either have trivial parallelizations for GPU architectures. For instance, in multiplication of a $m \times n$ matrix with a scalar, we create $m \times n$ threads, where each thread is assigned a single multiplication. No further optimizations were done for these functions.

Matrix multiplication

Our matrix multiplication implementation is based on implementations in cuBLAS (NVIDIA 2017) and clBLAST (Nugteren 2017). The only optimizations that are used is assigning additional work to threads in large matrix multiplications and the use of tiling in local memory. Both are widely known optimization techniques and are known to boost performance on all GPU architectures. We separately deal with matrix multiplication $A \times A^T$. In this case, we know that the resulting matrix is symmetric and can reduce the number of multiplications by half.

Inverting a lower triangular matrix

The mostly widely used CPU algorithms for inverting a lower triangular matrix are not suitable for many-core architectures such as a GPU. The solution proposed in (Mahfoudhi, Mahjoub, and Nasri 2012) replaces most of the sequential code with matrix multiplications, which are more GPU-suitable operations.

The basic idea is to split the input matrix in blocks as shown in Figure 1. We first calculate the matrix inversion of the smaller matrices $A1$ and $A2$. These inverses are done using the basic sequential algorithms, with small amounts of parallelism. The final step is the calculation of $C3 = -C2 \times A3 \times C1$. In order to fully utilize the GPU, the number of blocks in the first step should be larger power of 2. The exact number of blocks depends on the input matrix size and the GPU used.

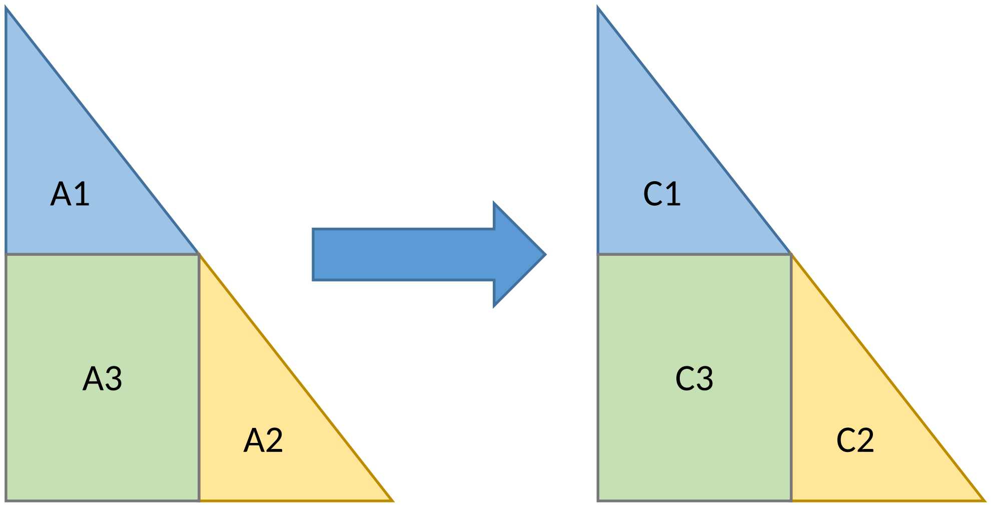

Figure 1: Blocked version of the lower triangualar matrix inverse.

Cholesky decompostion

Our GPU implementation for the Cholesky decompostion is based on the blocked algorithm, proposed in (Louter-Nool 1992). Similar to the implementation of the lower triangular matrix inverse, we first split the input matrix into blocks, as shown in Figure 2. We then calculate the Cholesky decomposition of A_{11} with a basic algorithm with less parallelism. We also calculate the inverse of L_{11}^T. L_{21} and L_{22} are calculated as follows:

$$L_{21} = A_{21}(L_{11}^T)^{(-1)}$$

$$L_{22} = A_{22} - L_{21}(L_{21})^T$$

Matrix multiplications are implemented as discussed in the previous subsection. For larger matrices ($n > 1000$), the algorithm is executed in 2 levels. For example, when $n = 2000$, the size of the block A_{11} is $m = 400$. Using the basic sequential algorithm for such a large m would be slow, so we apply the initial algorithm to A_{11}, as if its the input matrix, with $m = 100$.

The implementation of the derivative of the Cholesky decomposition is based on the blocked version presented in (Murray 2016). This algorithm is cache friendly and uses GPU-suitable matrix operations. Similar to the

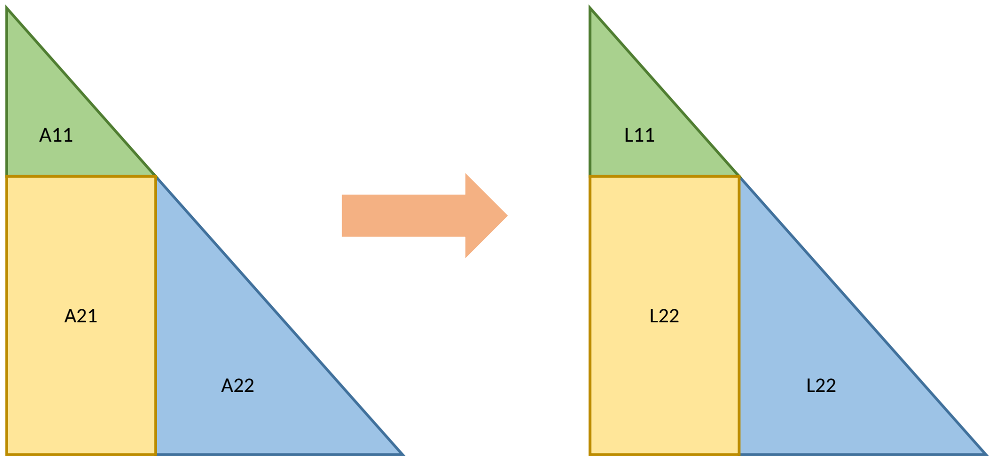

Figure 2: Blocked version of the Cholesky decomposition.

inversion and cholesky decomposition, the input matrix is split into smaller blocks on which we then perform various already discussed matrix operations: transpose, multiplication, lower triangular matrix inversion and subtraction. For details on the algorithm, refer to (Murray 2016).

Example 1: Gaussian process regression

The models that would benefit the most from the currently parallelized computational building blocks are models large covariance matrices. A good example of such a model are Gaussian processes (GP). We used 1D GP regression with hyperpriors example from the case study (Betancourt 2017) (see appendix).

We generated a toy dataset based on a simple functional relationship between x and y with added Gaussian noise:

$$x_i \sim_{\text{iid}} U(-10, 10)$$
$$y_i | x_i \sim_{\text{iid}} N\left(f(x), \frac{1}{10}\right), i = 1..n,$$

where $f(x) = \beta(x + x^2 - x^3 + 100 \sin 2x - \alpha)$. Parameters β and α were set so that the $E[f] = 0$ and $Var[f] = 1$. Figure 3 shows that there is no difference between GPU and CPU fits.

We measured the computation times of the CPU and GPU implementations of our model in Stan for different input sizes. The measurements include sampling and warmup iterations, but not model compilation time. The experiments were executed on a desktop computer with a Intel Xeon CPU running at $2.3GHz$ and a NVIDIA GTX1070 GPU. Results are shown in Figure 4. Due to unnecessary data transfers, the GPU implementation is not faster than the CPU version for smaller input sizes ($n < 600$). For larger n, the data transfer become negligible and we can observe a speedup of 13 for $n = 3000$. Speedup measurements for larger n were infeasible due to large CPU computation times, however, as the speedups do not yet taper off, we expect that the maximum speedup is even larger than 13.

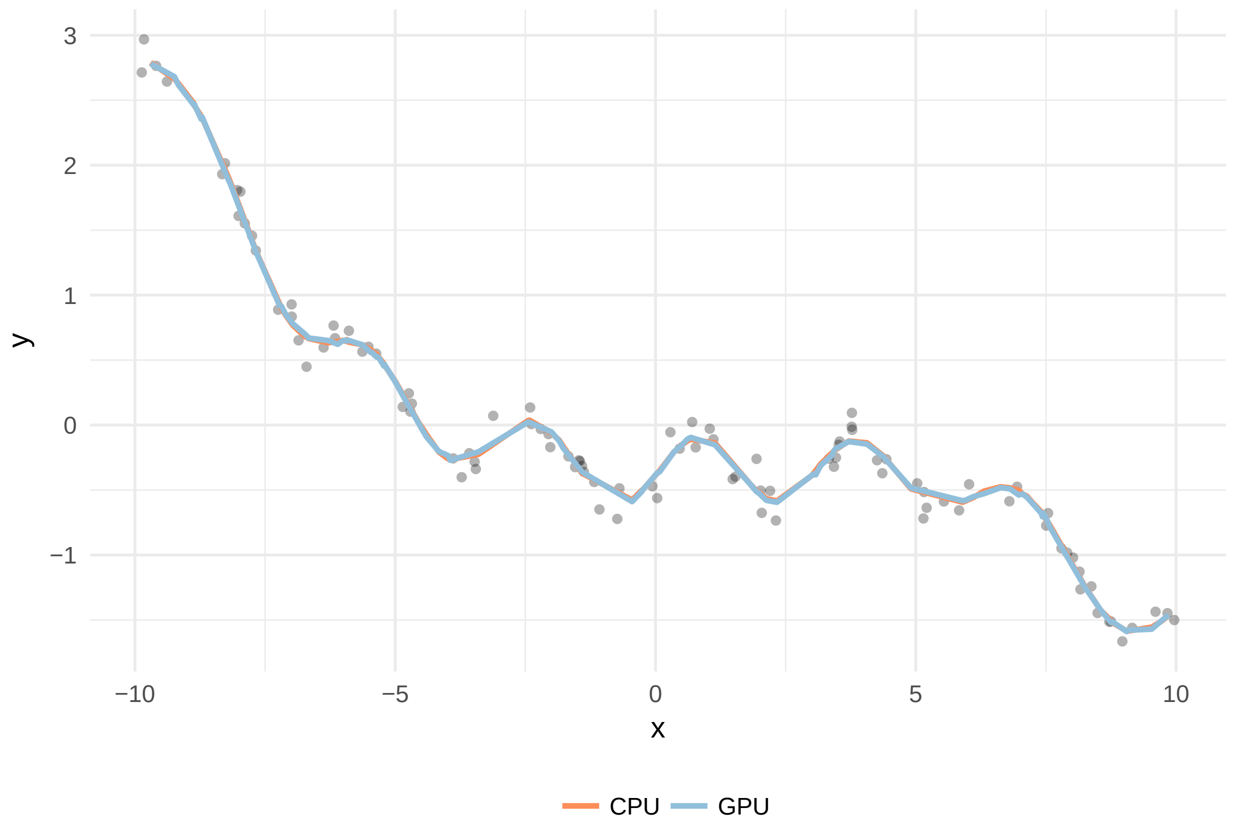

Figure 3: Comparison of CPU and GPU fits.

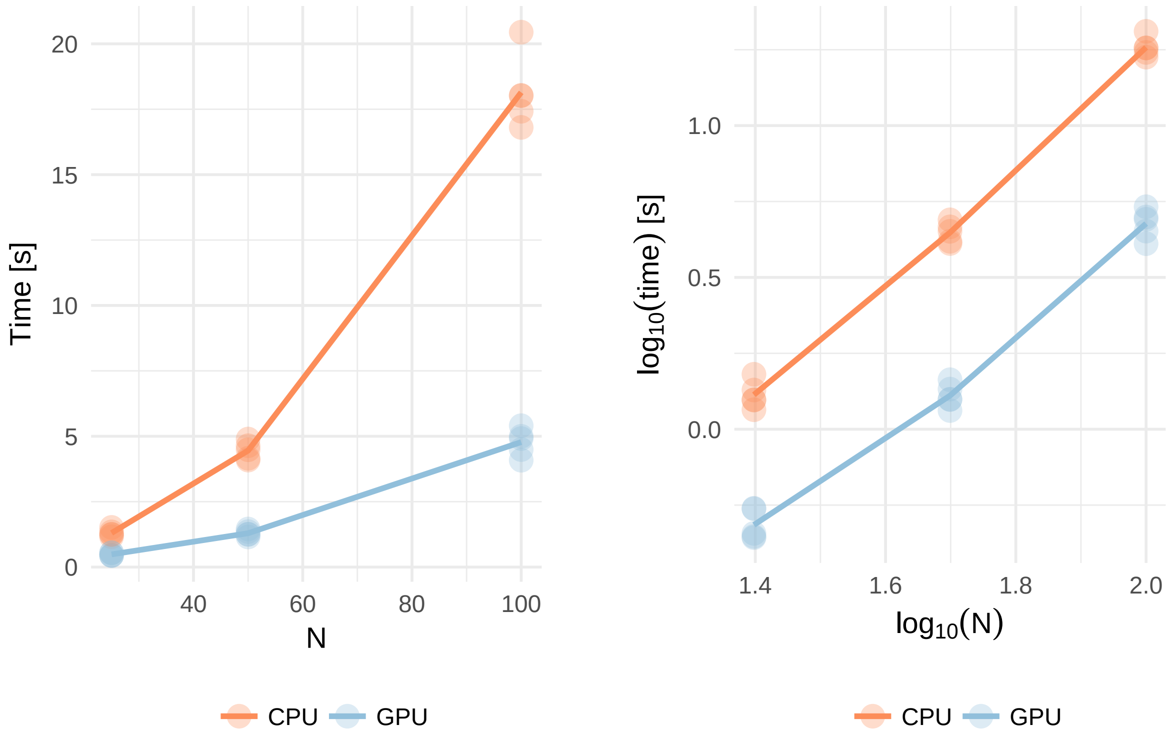

Figure 4: Visualizations of speedup when using our GPU approach compared to default CPU implementation.

Example 2: Spatial Gaussian predictive process

TODO: Description of the model.

Figure 5 shows that there is no difference in fitter parameters between the GPU and CPU approaches.

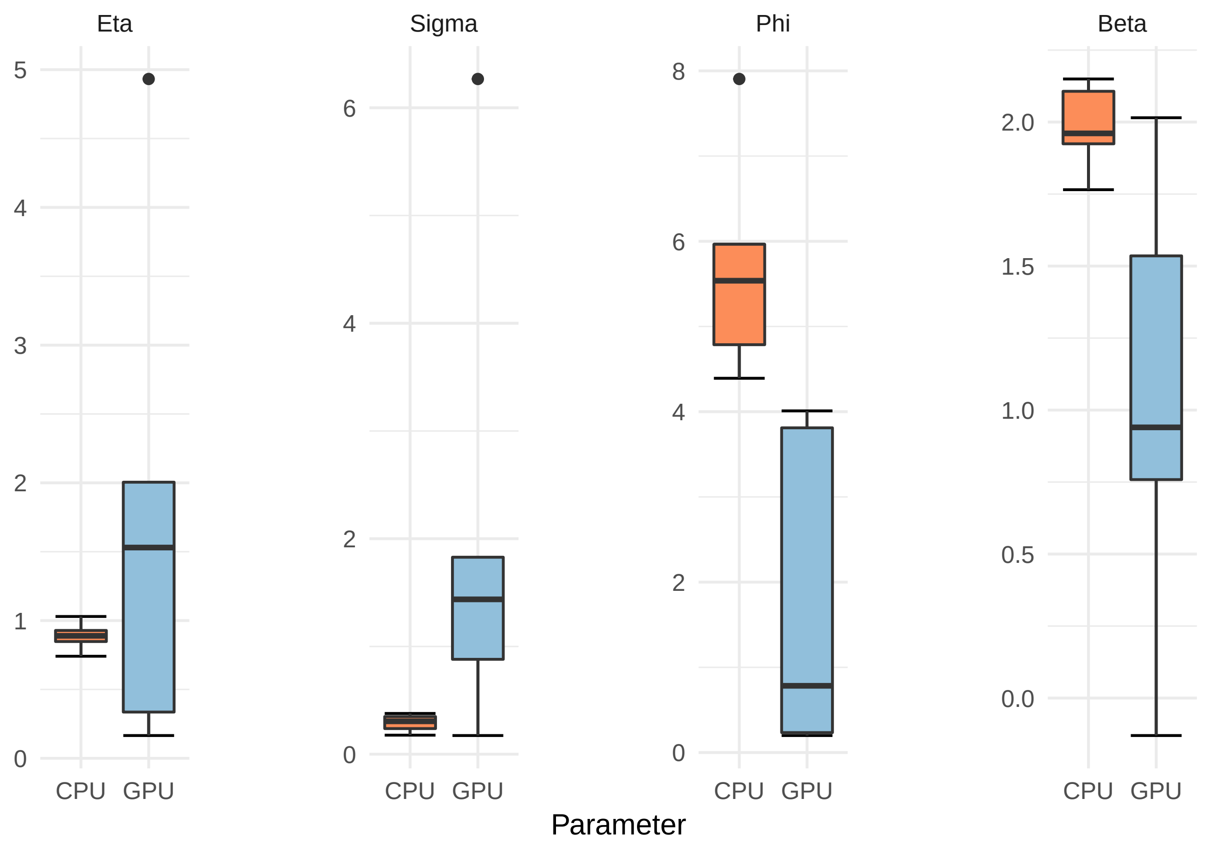

Figure 5: Comparison of calculated parameters between CPU and GPU.

TODO: Text.

Results are shown in Figure 6.

Conclusion

We have shown that practically meaningful speedups of inference in Stan can be achieved by paralellizing certain computational building blocks that are computational bottlenecks.

Main directions for future work include (a) removing unnecessary data transfers to and from the GPU, which should improve GPU performance, especially for smaller input size problems, but would require more fundamental changes to the Stan `math` library, (b) providing a R interface to `cmdstan` or access to GPU parallelized computation through `rstan`, and (c) adding GPU-parallel implementations for other computational building blocks (other matrix methods, density calculation, and generating random variates).

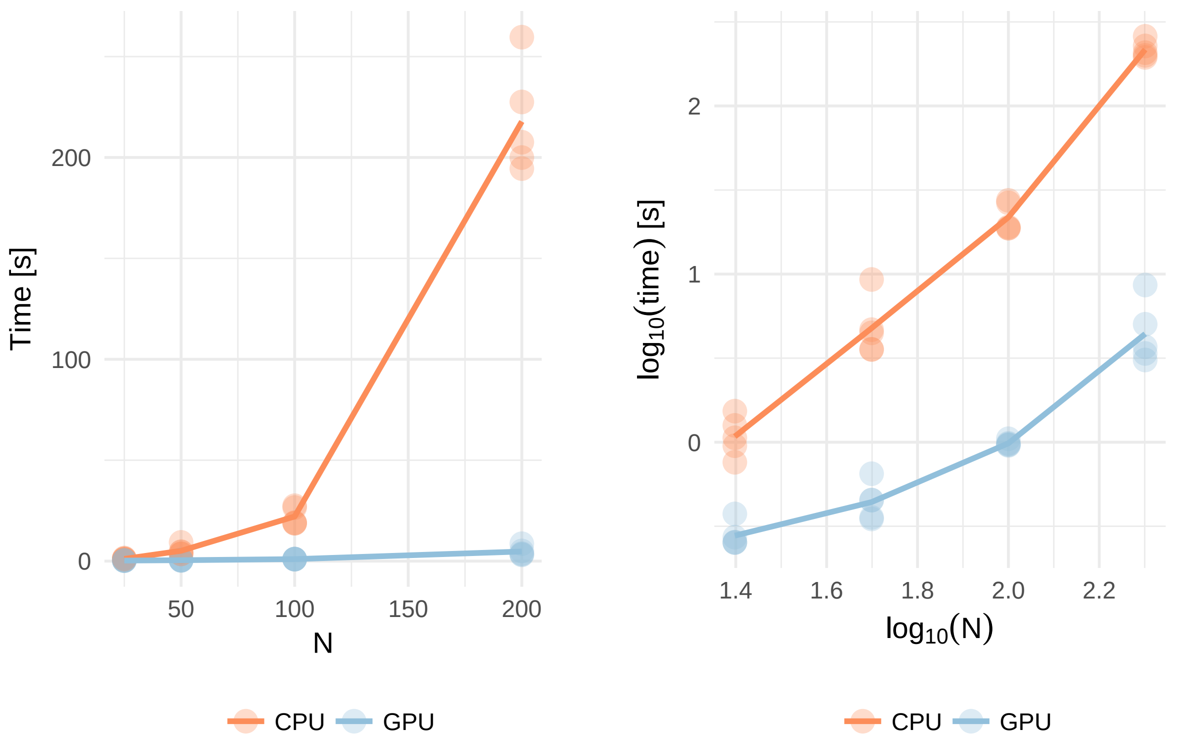

Figure 6: Visualizations of speedup when using our GPU approach compared to default CPU implementation.

Appendix

Stan model for Gaussian process regression

```
functions {
  vector gp_pred_rng(real[] x2,
                     vector y1, real[] x1,
                     real alpha, real rho, real sigma, real delta) {
    int N1 = rows(y1);
    int N2 = size(x2);
    vector[N2] f2;
    {
      matrix[N1, N1] K =   cov_exp_quad(x1, alpha, rho)
                         + diag_matrix(rep_vector(square(sigma), N1));
      matrix[N1, N1] L_K = cholesky_decompose(K);

      vector[N1] L_K_div_y1 = mdivide_left_tri_low(L_K, y1);
      vector[N1] K_div_y1 = mdivide_right_tri_low(L_K_div_y1', L_K)';
      matrix[N1, N2] k_x1_x2 = cov_exp_quad(x1, x2, alpha, rho);
      vector[N2] f2_mu = (k_x1_x2' * K_div_y1);
      matrix[N1, N2] v_pred = mdivide_left_tri_low(L_K, k_x1_x2);
      matrix[N2, N2] cov_f2 =   cov_exp_quad(x2, alpha, rho) - v_pred' * v_pred
                              + diag_matrix(rep_vector(delta, N2));
      f2 = multi_normal_rng(f2_mu, cov_f2);
    }
    return f2;
  }
}

data {
  int<lower=1> N;
  real x[N];
  vector[N] y;

  int<lower=1> N_predict;
  real x_predict[N_predict];
}

parameters {
  real<lower=0> rho;
  real<lower=0> alpha;
  real<lower=0> sigma;
}

model {
  matrix[N, N] cov =   cov_exp_quad(x, alpha, rho)
                     + diag_matrix(rep_vector(square(sigma), N));
  matrix[N, N] L_cov = cholesky_decompose(cov);

  // P[rho < 2.0] = 0.01
  // P[rho > 10] = 0.01
  rho ~ inv_gamma(8.91924, 34.5805);
  alpha ~ normal(0, 2);
  sigma ~ normal(0, 1);

  y ~ multi_normal_cholesky(rep_vector(0, N), L_cov);
}

generated quantities {
  vector[N_predict] f_predict = gp_pred_rng(x_predict, y, x, alpha, rho, sigma, 1e-10);
  vector[N_predict] y_predict;
  for (n in 1:N_predict)
    y_predict[n] = normal_rng(f_predict[n], sigma);
}
```

Stan model for spatial Gaussian predictive process

```
data {
  int<lower = 1> N;
  int<lower = 1> p;
  matrix[N, p] X;
  int<lower = 0> y[N];
  matrix[N, N] D;
}

parameters {
  vector[N] z;
  real<lower=0> eta;
  real<lower=0> phi;
  real<lower=0> sigma;
  vector[p] beta;
}

transformed parameters {
  cov_matrix[N] Sigma;
  vector[N] w;
  real<lower = 0> eta_sq;
  real<lower = 0> sig_sq;

  eta_sq = pow(eta, 2);
  sig_sq = pow(sigma, 2);

  for (i in 1:(N - 1)) {
    for (j in (i + 1):N) {
      Sigma[i, j] = eta_sq * exp(-D[i, j] * phi);
      Sigma[j, i] = Sigma[i, j];
    }
  }

  for (k in 1:N) Sigma[k, k] = eta_sq + sig_sq;
  w = cholesky_decompose(Sigma) * z;
}

model {
  eta ~ normal(0, 1);
  sigma ~ normal(0, 1);
  phi ~ normal(0, 5);
  beta ~ normal(0, 1);
  z ~ normal(0, 1);
  y ~ poisson_log(X * beta + w);
}
```

Original Computing Environment

```r
sessionInfo()
```

```
## R version 3.3.3 (2017-03-06)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 8.1 x64 (build 9600)
##
## locale:
## [1] LC_COLLATE=Slovenian_Slovenia.1250  LC_CTYPE=Slovenian_Slovenia.1250
## [3] LC_MONETARY=Slovenian_Slovenia.1250 LC_NUMERIC=C
## [5] LC_TIME=Slovenian_Slovenia.1250
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] reshape_0.8.7      cowplot_0.9.2      rstan_2.17.3
## [4] StanHeaders_2.17.2 plyr_1.8.4         ggplot2_2.2.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.16     knitr_1.20       magrittr_1.5     munsell_0.4.3
##  [5] colorspace_1.3-2 rlang_0.2.0      highr_0.6        stringr_1.3.0
##  [9] tools_3.3.3      grid_3.3.3       gtable_0.2.0     htmltools_0.3.6
## [13] yaml_2.1.18      lazyeval_0.2.1   rprojroot_1.3-2  digest_0.6.15
## [17] tibble_1.4.2     gridExtra_2.3    inline_0.3.14    evaluate_0.10.1
## [21] rmarkdown_1.9    labeling_0.3     stringi_1.1.7    pillar_1.2.1
## [25] scales_0.5.0     backports_1.1.2  stats4_3.3.3
```

References

Betancourt, Michael. 2017. "Robust Gaussian Processes in Stan, Part 3."

Češnovar, R, and E Štrumbelj. 2017. "Bayesian Lasso and multinomial logistic regression on GPU." *PLoS ONE* 12 (6): e0180343.

Louter-Nool, Margreet. 1992. "Block-Cholesky for Parallel Processing." *Appl. Numer. Math.* 10 (1). Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V.: 37–57. doi:10.1016/0168-9274(92)90054-H.

Mahfoudhi, R., Z. Mahjoub, and W. Nasri. 2012. "Parallel Communication-Free Algorithm for Triangular Matrix Inversion on Heterogenoues Platform." In *2012 Federated Conference on Computer Science and Information Systems (Fedcsis)*, 553–60.

Murray, Iain. 2016. "Differentiation of the Cholesky Decomposition." https://arxiv.org/abs/1602.07527.

Nugteren, Cedric. 2017. "CLBlast: A Tuned Opencl BLAS Library." *CoRR* abs/1705.05249. http://arxiv.org/abs/1705.05249.

NVIDIA. 2017. "cuBLAS library."