GPU Optimized Math Routines in the Stan Math Library

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

Apr 16, 2018

Introduction

The Stan Math library's Hamilton Monte Carlo (HMC) sampler has computationally expensive draws while usually searching the target distribution more effectivly than alternative MCMC methods with less iterations. The bottleneck within draws makes Stan a prime candidate for GPU optimizations within samples. This project implements GPU optimizations for the cholesky decompostion and it's derivative in the Stan Math library (Carpenter et al. 2015). Several modeling languages exist which implement MCMC sampling using GPUs, however this is the first known open source implementation of the cholesky decomposition with a GPU in a HMC setting. Furthermore, the GPU kernels are written in OpenCL which allows the methods to be implemented across any type of GPU. While results show that GPU optimizations are not optimal for small $N \times M$ matrices, large matrices can see speedups of 13 fold while retaining the same precision as models run purely on a CPU.

GPU implementation

The largest linear algebra bottlenecks in stan tends to come from the cholesky decomposition, it's derivative, and the derivative of solving $Ax = B$. In order to reduce these bottlenecks, the GPU methods implemented in the Stan Math library include:

1. Matrix transpose
2. Multiplication of matrices with a diagonal and scalar
3. Subtraction of matrices
4. Copying submatrices
5. Matrix multiplication
6. Lower triangular matrix inverse
7. Cholesky decomposition
8. First derivative of Cholesky Decomposition

Where [what numbers are from (Češnovar and Štrumbelj 2017)?] come from (Češnovar and Štrumbelj 2017). Some of the optimizations used here are simple on a GPU. 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. In the next section we will cover how the methods in (6-8) utilize the methods of (1-5).

The OpenCL (Stone, Gohara, and Shi 2010) context which manages the devices, platforms, memory, and kernels is built into stan using the `C++11` Meyer's singleton pattern called `opencl_context_base::getInstance()`. Developers are able to access the context through an friend adapter class called `opencl_context` which provides a simple wrapper for accessing the base context.

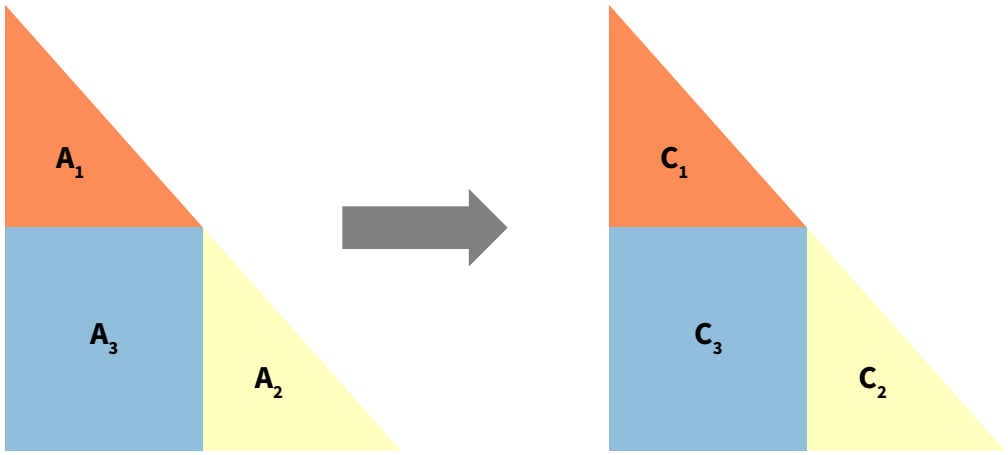

Figure 1: Blocked version of the lower triangualar matrix inverse.

Matrix multiplication

Our matrix multiplication implementation is based on routines in cuBLAS (NVIDIA 2017) and clBLAST (Nugteren 2017). The matrix multiplication routines are optimized through two standard methods, assigning additional work to threads in large matrix multiplications and the use of tiling in local memory. In addition, optimizations for special cases are created such as $A \times A^T$, since the result is symmetric and the number of multiplications can be reduced 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. Figure 1 gives a graphical illustration of the solution proposed in (Mahfoudhi, Mahjoub, and Nasri 2012) which replaces most of the sequential code with matrix multiplications and is more suited for many-core systems.

The input matrix is split into blocks as shown in Figure 1. The first step is to 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 $C3 = -C2 \times A3 \times C1$. The optimal number of blocks depends on the input matrix size and the GPU used[1].

Cholesky decompostion

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

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

[1]Generally thread blocks and warps will grouped in powers of two so the optimal block size is recommended to be a power of two such as 32x32

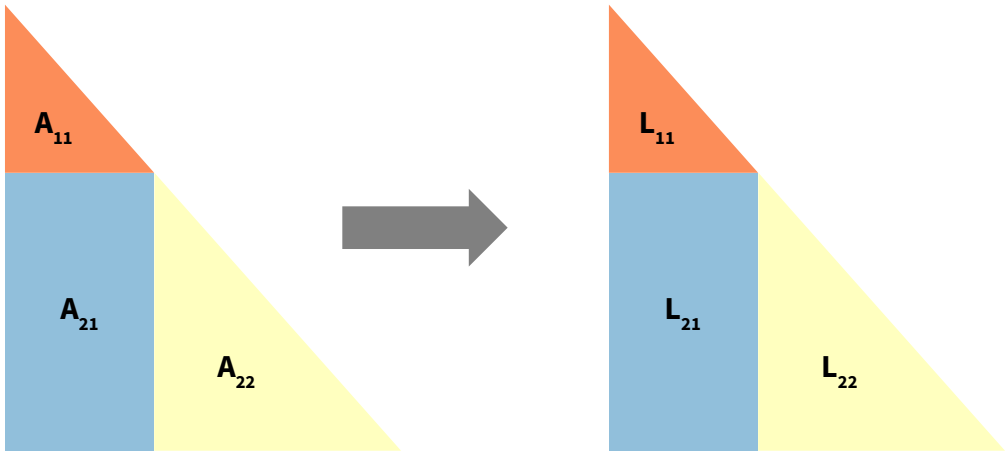

Figure 2: Blocked version of the Cholesky decomposition.

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

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 $m = 400$. Because the sequential algorithm would be slow for a large A_{11} block, the routine is run recursively on A_{11} until m reaches a reasonable size.

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 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).

Users can access the Cholesky GPU routines by calling `cholesky_decompose_gpu()` in the stan language. In the future all GPU methods will be implimented in the same way so that users can make their code access the GPU routines through calling `<func_name>_gpu()`.

Example 1: Gaussian process regression

Models that use large covariance matrices will tend to benefit the most from the cholesky GPU routines. The example below uses 1D GP regression with hyperpriors as exampled in the case study (Betancourt 2017) (see appendix).

This example uses 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.

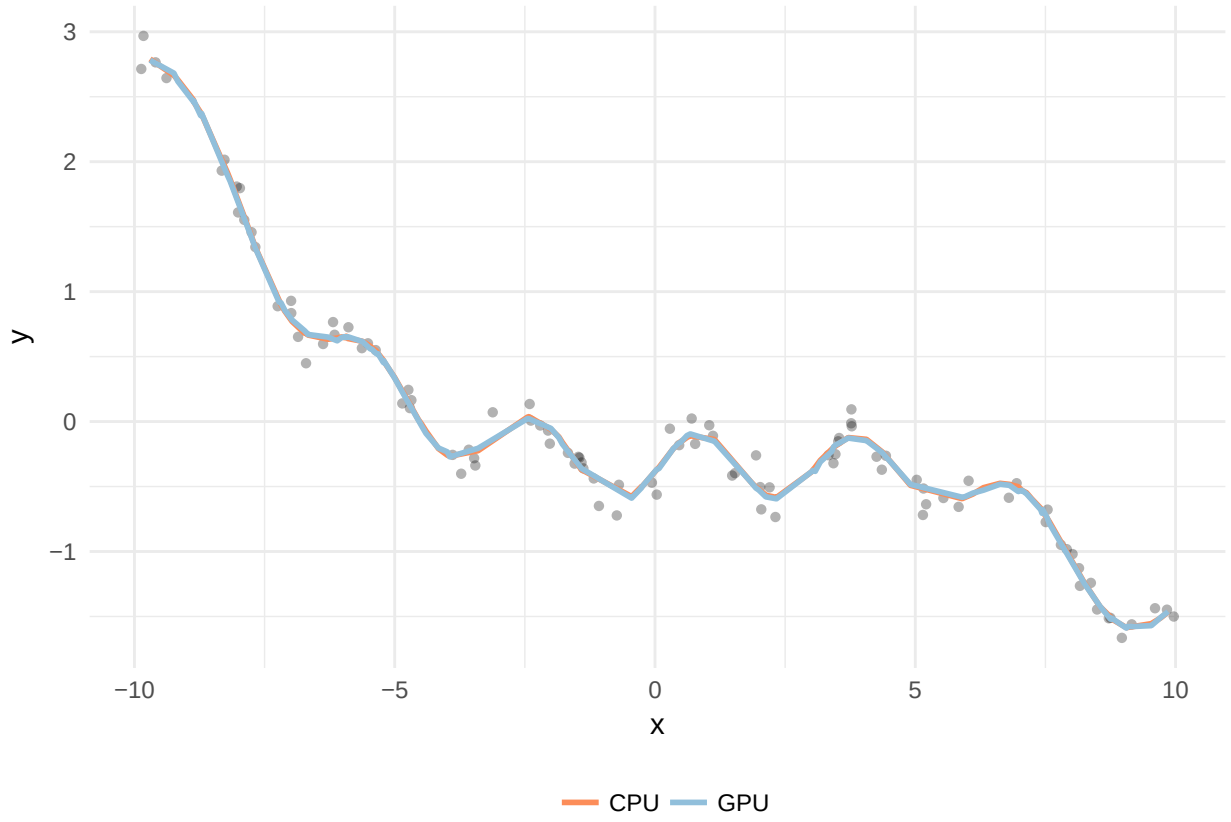

Figure 3: Comparison of CPU and GPU fits.

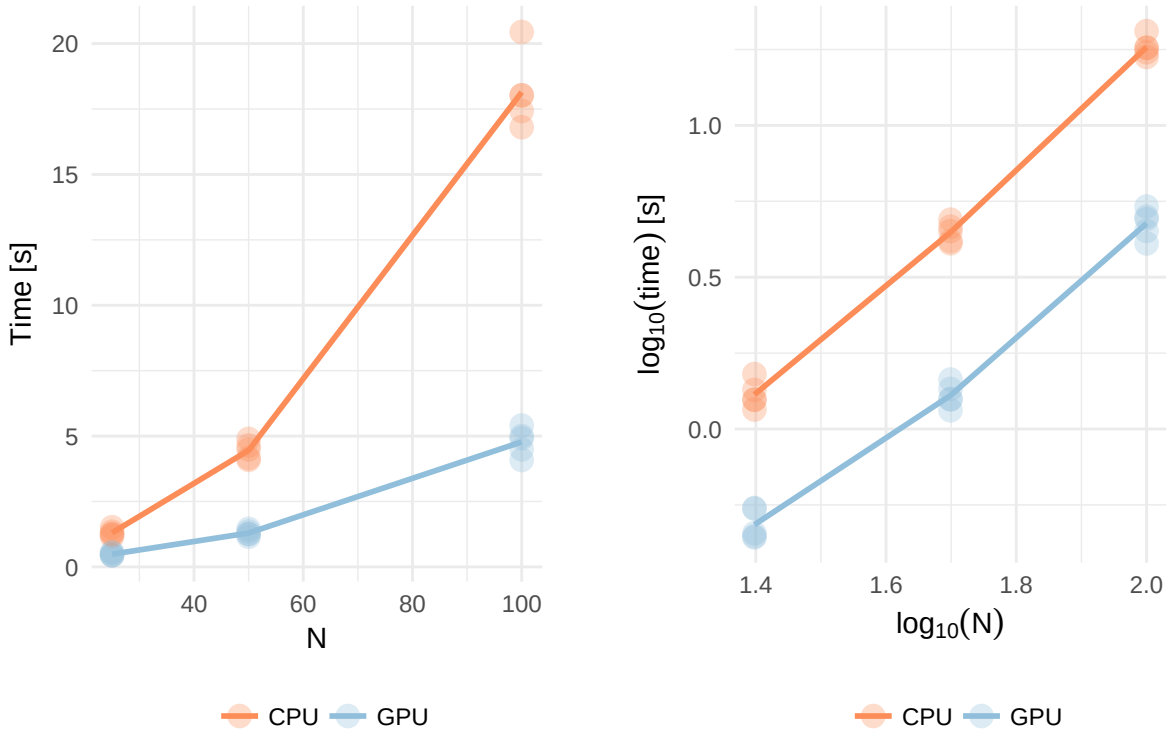

Figure 4: Visualizations of speedup when using our GPU approach compared to default CPU implementation.

The model is executed and timing results are stored for multiple input sizes over the GPU and CPU implementation. The measurements include sampling and warmup iterations, but not model compilation time[2]. 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.

To estimate the timing of the GPU vs. CPU routines we build a small linear regression on the simulations of the form $time \sim D_{GPU} \times N_{sims} + N_{sims} + \epsilon$ and predict the GPU and CPU times for $N = \{150, 200, 250\}$.

Table 1: The predictions for CPU vs. GPU times for larger N

N	CPU	GPU
150	23.65	6.34
200	31.54	8.45
250	39.42	10.57

Futher tests are necessary, but basic intuition and the back of a napkin regression implies that this method will continue

[2]The experiments were executed on a desktop computer with a Intel Xeon CPU running at $2.3GHz$ and a NVIDIA GTX1070 GPU

Example 2: Spatial Gaussian process

Advances in Global Positioning System (GPS) and Geographical Information Systems (GIS) enable us to accurately determine positions of points where we collect scientific data. In turn, this spurned growth of statistical modelling of large spatial datasets in many different scientific fields. This example represents one such toy dataset.

The toy dataset and the Stan model are implementented the same way as in Max Jospeh's study (Joseph 2016). The dataset mimics data collection at n different spatial locations. Each spatial location has its coordinates drawn from $\mathcal{U}(0,1)$. The data collected at a particular point depends on a mean zero Gaussian process with an isotropic stationary exponential covariance function:

$$[C(d)]_{i,j} = \eta^2 \cdot \exp(-d_{ij} \cdot \phi))$$

$$w(s) \sim GP(0, C(d))$$

$$y(s) \sim \text{Poisson}(\exp(X^T(s) \cdot \beta + w(s) + \mathcal{N}(0, \sigma))),$$

where d_{ij} is the distance between locations s_i and s_j, η^2 is the variance parameter of the Gaussian process, ϕ determines how quickly the correlation in w decays as distance increases, $y(s)$ is data collected at the geospatial location s, X is an $n \times p$ design matrix, β is a length p parameter vector, and σ is a "nugget" parameter. We used the following parameter values in our simulations: $\eta = 0.8, \sigma = 0.3, \phi = 7, \beta = 2$. Stan model is attached in the appendix section.

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

TODO: Text.

Results are shown in Figure 6.

Conclusion

This project shows the GPU routines in stan can give both practical and powerful speedups. Parallelizing the cholesky and it's derivative can give 13 fold speedups or more for programs which depend on large covariance matrices. As this project continues, future areas of reasearch include (a) removing unnecessary data transfers to and from the GPU, which is currently our largest bottleneck, (b) allowing `rstan` (Stan Development Team 2018) to access the GPU methods, and (c) adding GPU-parallel implementations for other computational building blocks such as other matrix methods, density calculations, and generating random variates.

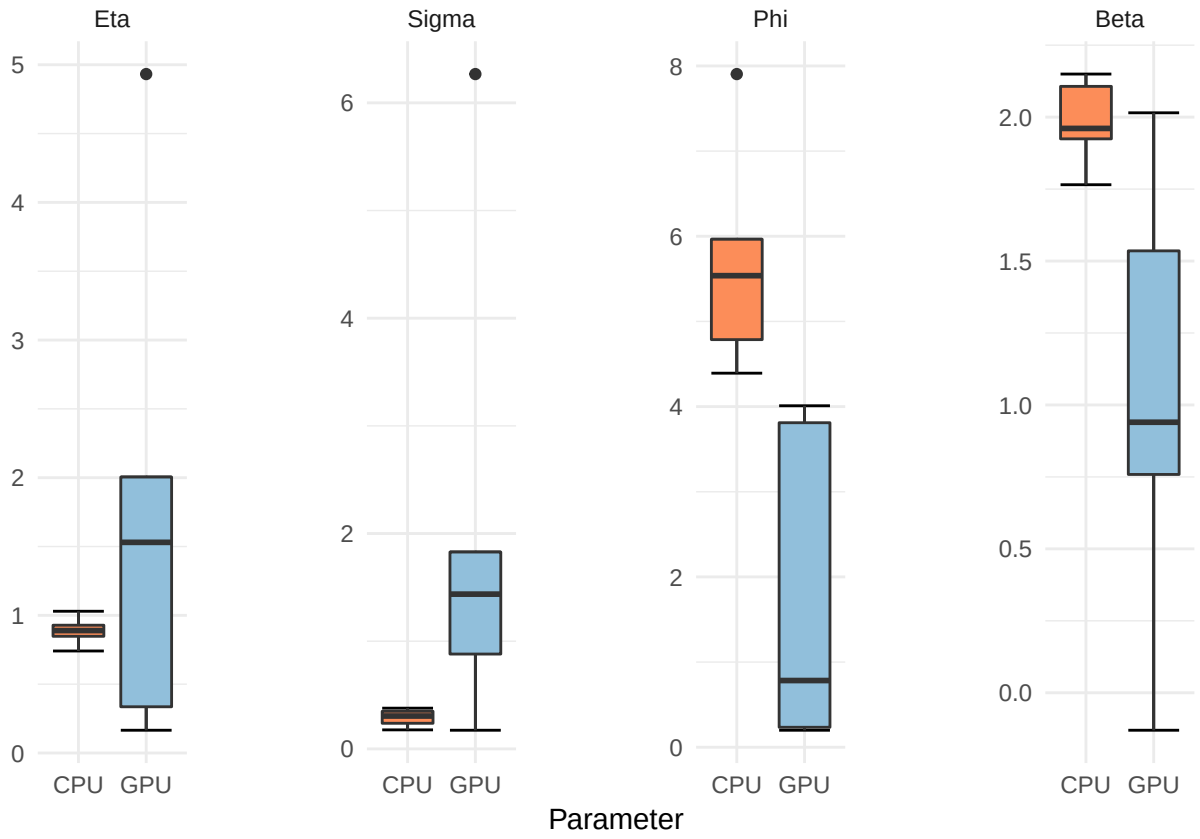

Figure 5: Comparison of calculated parameters between CPU and GPU.

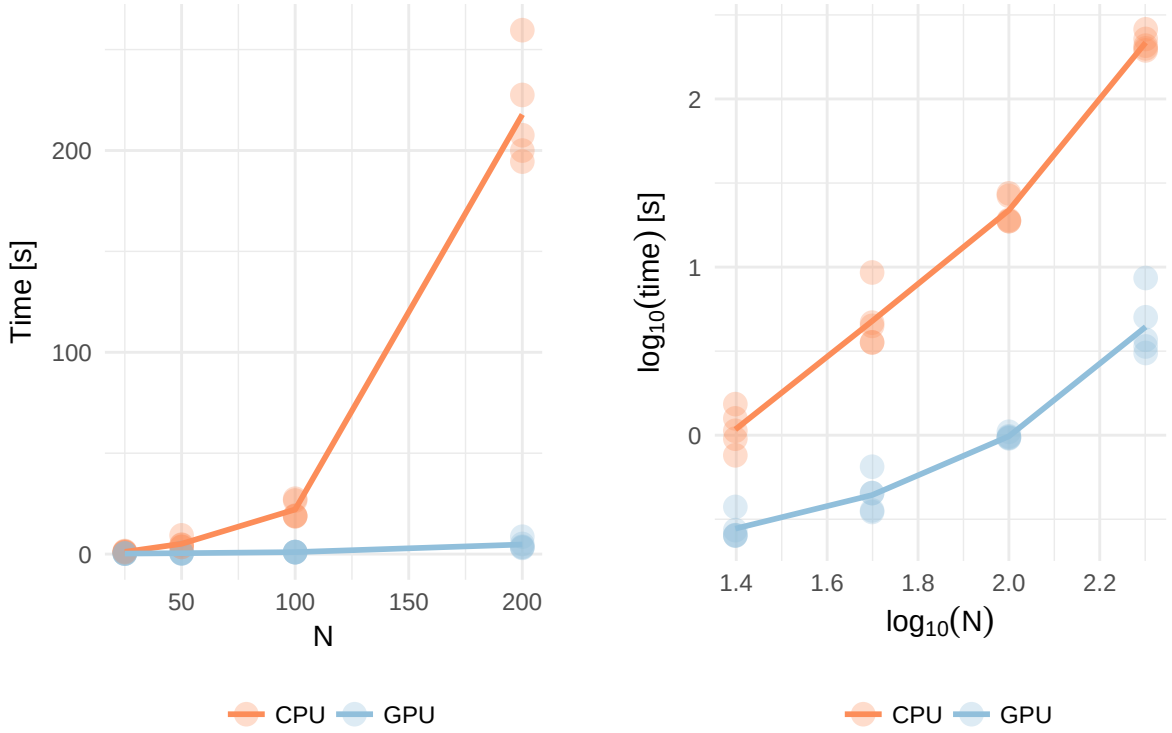

Figure 6: Visualizations of speedup when using our GPU approach compared to default CPU implementation.

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 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

```
sessionInfo()
```

```
## R version 3.3.2 (2016-10-31)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 16299)
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## 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
##  [7] RevoUtilsMath_10.0.0 RevoUtils_10.0.2     RevoMods_10.0.0
## [10] MicrosoftML_1.0.0    mrsdeploy_1.0        RevoScaleR_9.0.1
## [13] lattice_0.20-34      rpart_4.1-10
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.15           highr_0.6              pillar_1.1.0
##  [4] iterators_1.0.8        tools_3.3.2            digest_0.6.15
##  [7] jsonlite_1.5           evaluate_0.10.1        tibble_1.4.2
## [10] gtable_0.2.0           rlang_0.1.6            foreach_1.4.3
## [13] CompatibilityAPI_1.1.0 curl_3.1               yaml_2.1.16
## [16] gridExtra_2.3          stringr_1.2.0          knitr_1.19
## [19] stats4_3.3.2           rprojroot_1.3-2        grid_3.3.2
## [22] inline_0.3.14          R6_2.2.2               rmarkdown_1.9
## [25] magrittr_1.5           backports_1.1.2        scales_0.5.0
## [28] codetools_0.2-15       htmltools_0.3.6        colorspace_1.3-2
## [31] labeling_0.3           stringi_1.1.6          lazyeval_0.2.1
## [34] munsell_0.4.3
```

References

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

Carpenter, Bob, Matthew D. Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. 2015. “The Stan Math Library: Reverse-Mode Automatic Differentiation in C++.” *CoRR* abs/1509.07164. http://arxiv.org/abs/1509.07164.

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

Joseph, Max. 2016. “Gaussian Predictive Process Models in Stan.” https://github.com/mbjoseph/gpp-speed-test.

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.”

Stan Development Team. 2018. “RStan: The R Interface to Stan.” http://mc-stan.org/.

Stone, John E., David Gohara, and Guochun Shi. 2010. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems.” *IEEE Des. Test* 12 (3). Los Alamitos, CA, USA: IEEE Computer Society Press: 66–73. doi:10.1109/MCSE.2010.69.