# GPU Optimized Math Routines in the Stan Math Library

*Rok Cesnovar, Davor Sluga, Jure Demsar, Steve Bronder, Erik Strumbelj*

*Jul 9, 2018*

## Introduction

The Stan library's Hamilton Monte Carlo sampler (NUTS) typically explores the target distribution more efficiently than alternative MCMC methods and requires fewer iterations. However, it requires the computation of the gradient and is computationally more expensive per iteration. The makes it an excellent candidate for GPU optimization.

Our motivating example were Gaussian Process (GP) models, where the computation is dominated by the inversion of the covariance matrix of the size of the data, which is typically done through Cholesky decomposition. We implemented GPU optimizations for the Cholesky decomposition and its derivative in the Stan Math library (Carpenter et al. 2015). This is the first known open source GPU implementation of the Cholesky decomposition in an MCMC setting. Furthermore, the GPU kernels use OpenCL and are not restricted to a particular GPU vendor. While results show that GPU optimizations are not optimal for small $n \times m$ matrices, large matrices can see speedups of 7.8x while retaining precision.

## GPU implementation

One of the most significant linear algebra bottlenecks in GP (and many other statistical models) is matrix inversion. In particular, the inversion of a positive semi-definite covariance matrix, which is typically done through Cholesky decomposition. This requires the computation of the decomposition, its derivative, and the derivative of solving the linear system $Ax = B$. To reduce these bottlenecks, we implemented GPU optimizations of the following Stan Math library methods:

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.

The execution times of methods (1-4) are negligible and thus our GPU implementations of these methods are simple and naive. For instance, in the multiplication of a $m \times n$ matrix with a scalar we create $m \times n$ threads, where each thread is assigned a single multiplication. These implementations are necessary to perform methods (6-8) on the GPU.

Stan's GPU matrix multiplication routines are based on the the routines in cuBLAS (NVIDIA 2017) and clBLAST. 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. Specific

cases allow for specific optimization. For example, $A \times A^T$. Because the result is symmetric, the routine will reduce the number of multiplications by one half.

The GPU implementations and optimizations of the lower triangular matrix inverse and the Cholesky decomposition are improvements on our previous work (Češnovar and Štrumbelj 2017). Details of these implementations are available in the following sections. The first derivative of the Cholesky decomposition is implemented using methods (1-7).

The OpenCL (Stone, Gohara, and Shi 2010) context which manages the devices, platforms, memory, and kernels sits in `opencl_context_base::getInstance()` and is implemented in the Math library as a singleton. Developers can access the context through a friend adapter class called `opencl_context` which provides a simple wrapper API for accessing the base context.

## Inverting a lower triangular matrix

The most 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) that replaces most of the sequential code with matrix multiplications which are more suited for many-core systems.

The input matrix is split into blocks[1] 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 $C3 = -C2 \times A3 \times C1$.

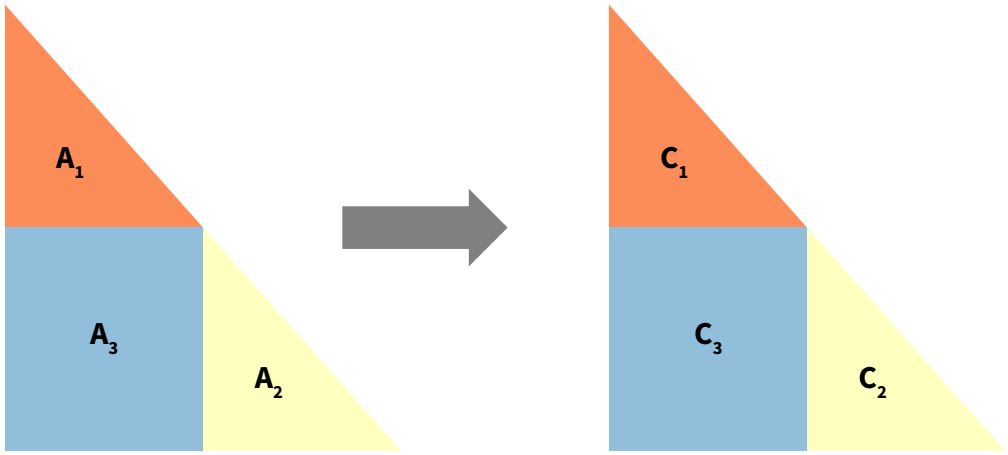


Figure 1: Blocked version of the lower triangualar matrix inverse.

## Cholesky decompostion

The GPU implementation of the Cholesky Decomposition comes from the blocked algorithm proposed in (Louter-Nool 1992). Similar to the application of the lower triangular matrix inverse, the input matrix is split into blocks, as shown in Figure 2. A basic algorithm is first used to calculate the Cholesky Decomposition of $A_{11}$ and then the calculation of the inverse of $L_{11}^T$. Calculations for $L_{21}$ and $L_{22}$ proceeds as follows:

[1]The optimal number of blocks depends on the input matrix size and the GPU used. Thread blocks and warps will be in groupings of powers of two, so the optimal block size is recommended to be a power of two such as 32x32

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

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

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.

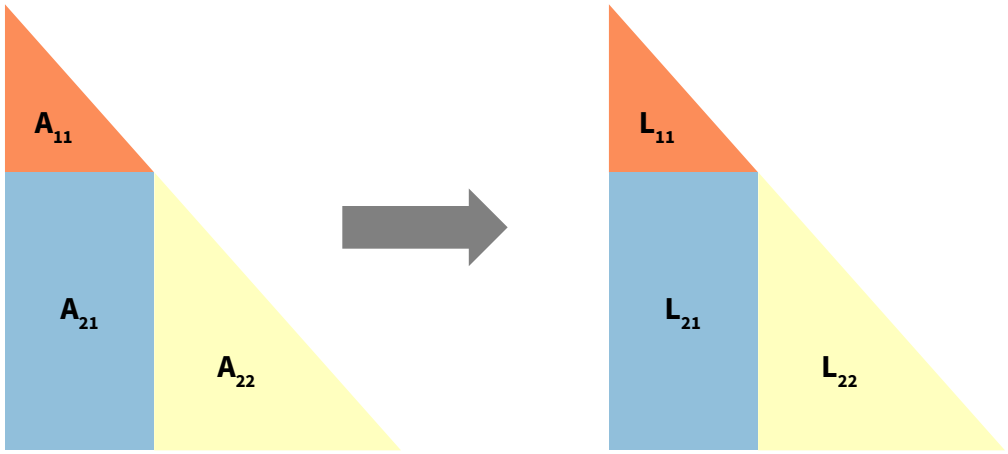


Figure 2: Blocked version of the Cholesky decomposition.

The implementation of the derivative of the Cholesky decomposition comes from the blocking method 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 the algorithm performs various 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()` and `multi_normal_cholesky_gpu()` in the stan language. In the latter, only the derivative of solving $Ax = b$ is run on the GPU. In the future, all GPU methods will be implemented in the same way so that users can make their code access the GPU routines by calling `<func_name>_gpu()`.

## Example: GP regression

Models that use large covariance matrices benefit from the Cholesky GPU routines. The example below uses 1D GP regression with hyperpriors from the case study (Betancourt 2017) (see the 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 $\beta$ and $\alpha$ were set so that $E[f] = 0$ and $Var[f] = 1$. Figure 3 shows that there is no practical difference between GPU and CPU fits (however, the solutions are not identical).

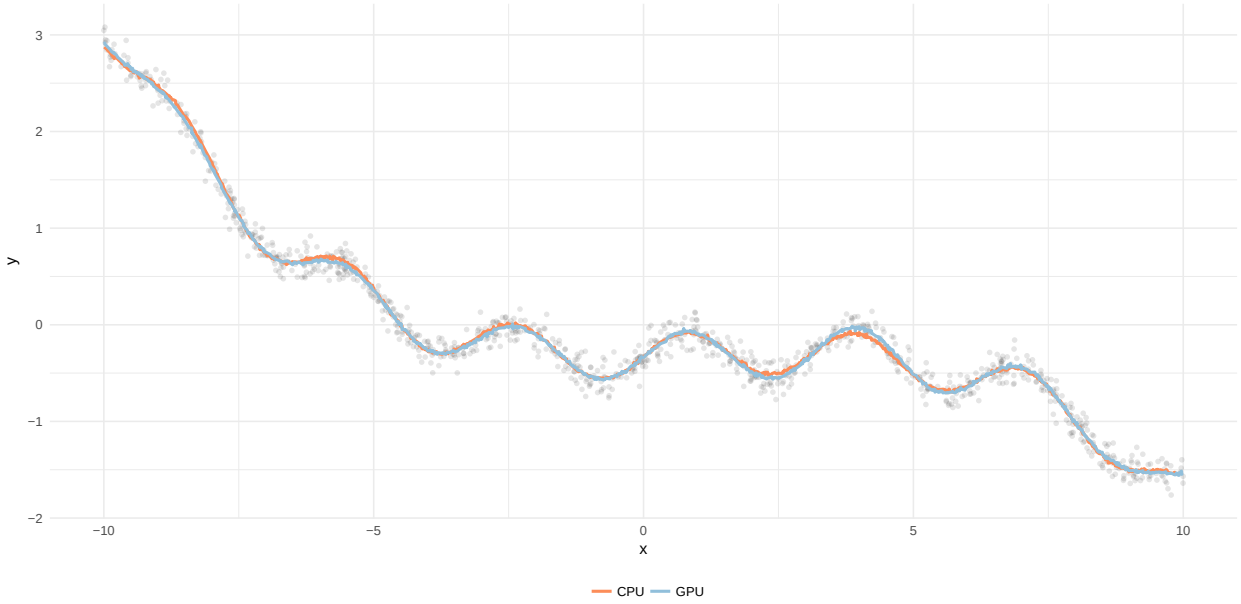


Figure 3: Comparison of CPU and GPU fits.

We ran the model for different input sizes $n$ with and without GPU support. In both cases NUTS was used to sample from the posterior and all the settings were the same. Therefore, the only difference between the CPU and GPU experiments was that the latter peformed some Math routines on the GPU. We used a desktop computer with an Intel Core i7-4790 CPU running at $3.6GHz$ and a Nvidia GTX1070 GPU.

Timing results are shown in Figure 4 with measured times include sampling and warmup iterations, but not model compilation time. Due to unnecessary data transfers, the GPU implementation is not faster than the CPU version for smaller input sizes ($n < 750$). For larger $n$, the data transfer becomes negligible, and we can observe a speedup of 7.8 for $n = 5120$. Speedup measurements for larger $n$ were infeasible due to large CPU computation times.

## Conclusion

Our GPU optimized methods in Stan result in practically meaningful speedups. Parallelizing the Cholesky, its derivative and the derivative of solving $Ax = B$ provides 7.8-fold speedups or more for programs which depend on large covariance matrices. As this project continues, we plan to (a) removing unnecessary data transfers to and from the GPU, which is currently our most significant bottleneck, (b) allow `rstan` (Stan Development Team 2018) access to the GPU methods, and (c) add GPU-optimized implementations for other computational building blocks, such as other matrix methods, density computation, and random variate generation.

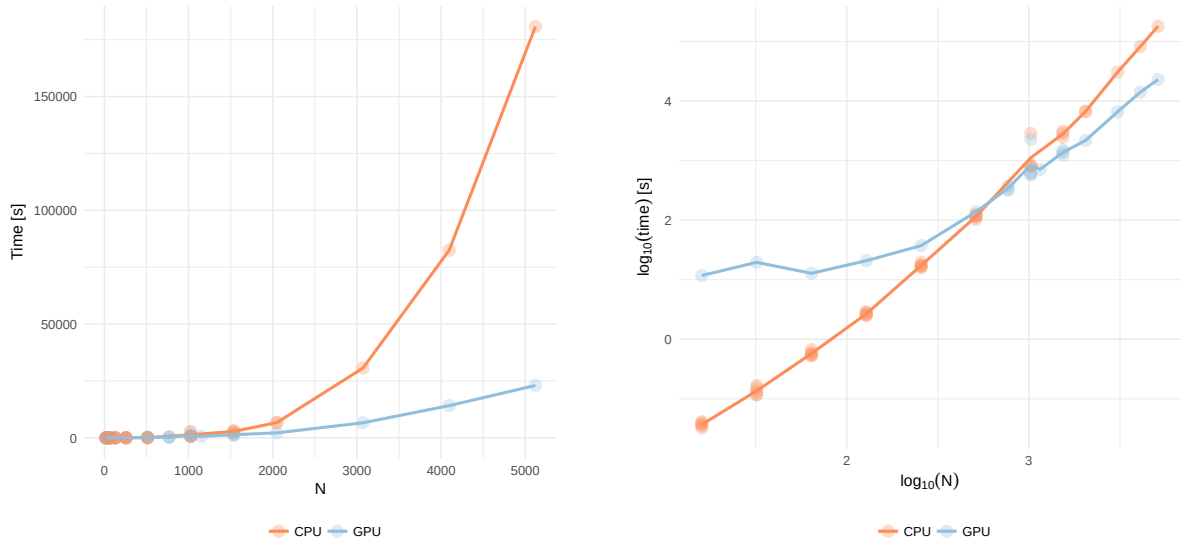


Figure 4: Visualizations of speedup when using the GPU approach compared to the default CPU implementation. For simulations ran with the default CPU implementation executed for only simulations up to $n = 2048$, times reported for larger n are estimated from measurements for smaller n.

# Appendix

## Reproducing the simulations

Our simulations take several days to complete, so the results in this R Markdown file are not computed each time the manuscript is compiled. In order to achieve a reasonable compilation time of the R Markdown file, we decided to use precomputed results.

To recompute the results, you have to use the GP.R script from the _Simulations folder. Newly calculated results will be saved into the _Simulations/_Output/GP folder. To replace precomputed results with the new ones you have to delete all files from the _Results folder and replace them with files from _Simulations/_Output/GP folder.

Unforuntately, using the GPU routines in Stan Math Library is not straightforward. To use these routines you must first install the appropriate version of the library and then recompile CmdStan. See for detailed instructions. Pay special attention to the section Integration with CmdStan. Once you succesfully compile GpuStan with GPU support you only have to change the working and CmdStan directories at the top of the GP.R script and you are ready to go! Currently, simulations can only be run on Windows.

## 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); // cholesky_decompose_gpu in GPU model

  // 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);
}
```

## Original Computing Environment

```
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.”

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.

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.

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.