GPU-based Parallel Computation Support for Stan

Rok Češnovar  
University of Ljubljana

Steve Bronder  
Capital One

Davor Sluga  
University of Ljubljana

Jure Demšar  
University of Ljubljana

Tadej Ciglarič  
University of Ljubljana

Sean Talts  
Columbia University

Erik Štrumbelj  
University of Ljubljana

Abstract

This paper details an extensible OpenCL framework that allows Stan to utilize heterogeneous compute devices. It includes GPU-optimized routines for the Cholesky decomposition, its derivative, other matrix algebra primitives and some commonly used likelihoods, with more additions planned for the near future. Stan users can now benefit from speedups offered by GPUs with little effort and without changes to their existing Stan code. We demonstrate the practical utility of our work with two examples – logistic regression and Gaussian process regression.

Keywords: statistical inference, Bayesian statistics, automatic differentiation, parallel computation, GPU, Stan.

1. Introduction

Stan is an open-source probabilistic programming language for Bayesian modelling and inference (Carpenter, Gelman, Hoffman, Lee, Goodrich, Betancourt, Brubaker, Guo, Li, and Riddell 2017). It has become the system of choice for statisticians and scientists as well as the reference project for Bayesian inference. Hundreds of research papers using Stan are published every year, ranging from cognitive anthropology and the structure of gravitational clusters to clinical trial design, and sports. Dozens of tools utilize Stan such as rstanarm (Gabry and Goodrich 2016), brms (Bürkner et al. 2017), and Facebook’s forecasting tool Prophet (Taylor and Letham 2018).

There exist many other languages and software tools similar to Stan. Some focus more on statistical inference, while others focus more on machine learning and deep learning. To name just
a few of the most popular: Edward/Edward2 (TensorFlow) (Tran, Hoffman, Saurous, Brevdo, Murphy, and Blei 2017) and PyMC3 (Salvatier, Wiecki, and Fonnesbeck 2016) (Theano) (Bergstra, Breuleux, Bastien, Lamblin, Pascanu, Desjardins, Turian, Warde-Farley, and Bengio 2010), Pyro (Bingham, Chen, Jankowiak, Obermeyer, Pradhan, Karaletsos, Singh, Szerlip, Horsfall, and Goodman 2019) (PyTorch) (Adam, Paszke, Gross, Chintala, Chanan, Yang, DeVito, Lin, Desmaison, Antiga, and Lerer 2017), and MxNet (Zheng, Chen, Li, Li, Lin, Wang, Wang, Xiao, Xu, Zhang, and Zhang 2015).

Stan has three distinct components: a probabilistic programming language, the Stan Math library that supports auto-differentiation, and algorithms for inference. The main advantages of Stan are a rich math library and state-of-the-art inference with a variant of Hamiltonian Monte Carlo – the NUTS (No-U-turn) sampler (Hoffman and Gelman 2014) – which makes Stan suitable for robust fully-Bayesian inference. Moreover, the Stan probabilistic programming language is easier to understand than systems embedded in other languages (Baudart, Hirzel, and Mandel 2018).

Stan initially focused on precision over performance and has only recently started building out low-level parallelism. While Stan supports threading and MPI to execute disjoint sets in the auto-differentiation expression tree, it did not have support for specialized hardware such as GPUs. An ideal case for GPU based optimization are models based on Gaussian Processes (GP). The computation in GP-based models is, even for moderate input sizes, dominated by computing the inverse of the covariance matrix. The $O(n^3)$ time of this operation also dominates the quadratic costs associated with transferring matrices to and from a GPU. In turn, computing the Cholesky decomposition of the positive definite matrix dominates the computation time of the matrix inverse. Because these costs can be broken up and executed in parallel they make the Cholesky decomposition an ideal target for GPU-based computation.

This paper describes a framework for GPU support in Stan and GPU implementations of the Cholesky decomposition, its derivative, other matrix algebra primitives, and GLM likelihoods with derivatives in the Stan Math library. Unlike most similar libraries, our framework relies on OpenCL 1.2 (Stone, Gohara, and Shi 2010), so it supports a variety of devices. This includes GPUs of different vendors, multi-core CPUs, and other accelerators.

The integration with Stan is seamless and user friendly - setting a flag moves the computation of supported routines to the GPU, with no need to change Stan code. The API provides experts with a simple way of implementing their GPU kernels, using existing GPU kernels as building blocks. We demonstrate the practical utility of our work – ease of use and speedups – with two examples, logistic regression and GP regression.

2. Integrating OpenCL with Stan

Stan’s reverse mode automatic differentiation uses Eigen’s (Guennebaud, Jacob et al. 2010) Matrix type to store data as a matrix of type double or Stan’s var type, where the var holds the value and adjoint used in automatic differentiation. Stan uses expression templating to build up an expression tree used in automatic differentiation and stores all the data needed in the expression tree via its local allocator. When a node’s children in the expression graph are disjoint, Stan can utilize C++11 threading or MPI to compute the log probability evaluation in parallel. When an operation within a node is expensive, Stan can use the OpenCL backend to parallelize the operation on the node.
2.1. OpenCL base

Stan’s OpenCL backend uses a single context to receive data and routines from individual nodes in the expression tree. Ensuring there is only one context and queue per device for the life of the program makes context management simpler. The implementation of the OpenCL context which manages the device queue follows the Meyers singleton pattern and sits in the class opencl_context_base.

Instead of calling opencl_context_base::getInstance().method(), developers can access the context through a friend adapter class opencl_context which provides an API for accessing the base context. If the OpenCL implementation supports asynchronous operations, then the context asynchronously executes kernels. Asynchronous operations are particularly useful in conjunction with threading as the individual threads will be able to enqueue operations, which will execute while threads do other calculations using CPU.

2.2. Matrix class

The base matrix class matrix_cl holds the device memory buffer, meta-information on the matrix, and methods for reading and writing event stacks for asynchronous computation. When a kernel receives a matrix_cl, the kernel’s event is attached to the appropriate read or write event stack. Reading and writing to OpenCL buffers uses the generic enqueue(Write)/(Read)Buffer methods. Because Stan Math heavily relies on Eigen matrices, constructors and methods are available for easily passing data back and forth.

Developers can pass in Eigen matrices directly to the matrix_cl constructor or use the to_matrix_cl() or from_matrix_cl() methods.

```cpp
Eigen::MatrixXd m(2, 2);
m << 1, 2, 3, 4;

matrix_cl A(m);
matrix_cl B(2, 2);
B = to_matrix_cl(m);
Eigen::MatrixXd C = from_matrix_cl(B);
```

Similar constructors for the matrix_cl class are available for standard vectors std::vector<T> and arrays of doubles.

We can reduce the data transfers of triangular matrices by only transferring the non-zero parts of the matrix in a packed form. The kernel unpack deals with unpacking the packed form shown on the right-hand side on Figure 1 to a flat matrix shown on the left-hand side. For lower (upper) triangular matrices, the upper (lower) triangular fill with zeros. The kernel pack packs the flat matrix to packed form for the transfer back to the host’s global memory.

```cpp
matrix_cl L = packed_copy<TriangularViewCL::Lower>(L_val_cpu, M_);
```

When operating on GPUs, transferring data from host to device and making copies can be the most expensive operations. To reduce the burden of data transfers, Eigen matrices with primitive types use the EIGEN_MATRIXBASE_PLUGIN to attach an OpenCL Buffer object to the Eigen matrix. Because Eigen matrices with double types are static throughout the computation (inference), the data in the cache does not change. The first time a matrix is transferred over to the GPU, if the type is primitive, then the matrixâ€™s data is copied
GPU-based Parallel Computation Support for Stan

Figure 1: Packing and unpacking a triangular matrix.

to the **Eigen** matrix’s OpenCL Buffer. The next time a request is made to have the **Eigen** matrices data transferred, the **Stan** program uses the cached matrix on the GPU to perform a device to device copy instead of a host to device transfer. The cache is unsafe for **var** type matrices because the **var** changes through the life of the **Eigen** program. Caching large matrices can cause the GPU to run out of memory. Caching large matrices can cause the GPU to run out of memory and so is turned off by default. Users can turn on the cache method by passing the macro flag **STAN_OPENCL_CACHE**.

2.3. Kernel construction

The **OpenCL** specification demands that strings are used to represent **OpenCL** kernels. However, having a large number of files comprised of strings is unwieldy and difficult to maintain. **Stan** wraps its kernels inside of a **STRINGIFY** macro, which gives developers access to the standard set of developer tools such as code highlighting, linting, Doxygen (Van Heesch 2008), and auto-formatting. This style makes the kernel code easier to maintain compared to having files full of strings. An example of how a developer brings a new kernel into **Stan**:

```c
// Items in between \ cond and \ endcond are ignored by doxygen .
// \ cond
const char * exampleKernelCode = STRINGIFY (  
// \ endcond  
/**
* Example of adding new kernel in Stan
* *
* @param [ out ] A An example output matrix .
* @param [ in ] B An example input matrix .
* @param val Some other input value
*/
__kernel void example ( double *A, double *B, int * val ) {
    // kernel code ...
}  
// \ cond
);
// \ endcond
/**
* See the docs for \ link kernels / example .hpp example () \ endlink
*/
const kernel_cl< out_buffer , in_buffer , int > example (  
"_example␣", exampleKernelCode , ("␣THREAD_BLOCK_SIZE␣", 32));
```

In the above, a developer uses **STRINGIFY** to create a **const char** that holds the kernel code. That string passes into the **kernel_cl** struct templated by the kernel argument types and with arguments giving the name of the kernel, the kernel code, and optional kernel macros they would like to have defined in the kernel.
Internally, we keep track of OpenCL events via queues on each matrix_cl object that we use to conservatively prevent race conditions and provide ordering where necessary. out_buffer and in_buffer are empty structs that we pass as template arguments to configure the kernel during construction to indicate the directionality of each input buffer. At runtime, the kernel will check the correct event queues on its arguments for events it needs to wait for and then attach the event representing the kernel’s completion to each buffer’s queues correctly. That way we ensure that an operation that, for example, writes to a buffer, is completed before we allow the OpenCL runtime to read from that buffer.

The parameter pack of types in the template for kernel_cl are unpacked and passed as the argument types for the operator() and down to the template arguments for OpenCL’s make_kernel functor. Below is a simplified version of the code used to construct and call the kernel.

```cpp
template <typename... Args>
struct kernel_cl {
    const kernel_functor<to_const_buffer_t<Args>&...> make_functor;

    kernel_cl(const char* name, const std::vector<const char*>& sources,
              const std::map<const char*, int>& options = {})
        : make_functor(name, sources, options) {}

    auto operator() (cl::NDRange global_thread_size, cl::NDRange thread_block_size,
                     to_const_matrix_cl_t<Args>&... args) const {
        auto f = make_functor();
        const std::vector<cl::Event> kernel_events
            = vec_concat(select_events<Args>(args)...);
        cl::EnqueueArgs eargs(opencl_context.queue(), kernel_events,
                               global_thread_size, thread_block_size);
        cl::Event kern_event = f(eargs, get_kernel_args(args)...);
        assign_events<Args...>(kern_event, args...);
        return kern_event;
    }
};
```

Note that the meta-programming traits to_const_buffer_t<> and to_const_matrix_cl_t<> override the in_buffer and out_buffer template types in order to propagate a cl::Buffer or matrix_cl to subsequent templates and signatures.

In the above code, the kernel_cl’s constructor passes the name, kernel code, and kernel options to the kernel functor in the initialization list which compiles the kernel. Kernel arguments declared with in_buffer or out_buffer should be of type matrix_cl. When a kernel is called, the events that are in each matrix_cl’s read or write stacks are collected depending on whether it was designated as a in or out buffer. The kernel will then wait to execute until the previous events complete. The kernel’s event is assigned to each matrix_cl’s appropriate read and write event stack via assign_events() depending on whether it was defined as an in_buffer or an out_buffer.

When the kernel_cl struct is constructed, it compiles the kernel and developers call the kernel with

```cpp
matrix_cl foo = //...```

Depending on the in/out_buffer passed when constructing the kernel, events will be added to the appropriate matrix_cl read and/or write event stack. For instance, in the above, goo in the output and will have the kernel’s event attached to its write_stack. While foo will have the kernel’s event attached to it’s read_stack. Later kernel calls that write to foo will know to wait for all the event’s in foo’s read_stack and write_stack while kernels that use goo as input will know to wait for the event’s in goo’s write_stack.

The kernel functions for addition, subtraction, and multiplication are wrapped in their standard operators so users of the Stan Math library can call the operations such as:

```cpp
matrix_cl A = B * C + D - E;
```

### 3. GPU-optimized routines in the Stan Math library

In this section we describe the three GPU-optimized matrix algebra primitives that are currently implemented in Stan Math. They are currently not accessible directly in the Stan language but are used inside existing primitive and reverse functions. Several supporting routines that were used in the implementation of these primitives are described in Section 3.5.

#### 3.1. Matrix multiplication

Efficient implementations of the general matrix multiplication in OpenCL can be found in various best practices guides (Nvidia 2009, 2010) and research papers (Matsumoto, Nakasato, and Sedukhin 2014; Nugteren 2017, 2018). In Stan Math we implemented general matrix multiplication with optimizations for triangular input matrices, vector and row vector inputs, and multiplication in the form \( C = A A^T \). Multiplications of a matrix or its diagonal with a scalar are also supported and explained in Section 3.5.

We implemented a kernel for general matrix multiplication (GEMM) that is based on the aforementioned best practices guides and is exposed through the `operator*(matrix_cl& A, matrix_cl& B)` function. Matrix multiplication \( C = AB \), where \( A \) is \( n \times k \) and \( B \) is \( k \times m \) is executed with a 2D grid of \( \frac{n \times m}{WPT} \) threads, where WPT (Work Per Thread) is an implementation parameter. Thread \((i,j)\) computes the values \( C_{i,j}, ..., C_{i,j+WPT-1} \) of the resulting matrix, therefore performing up to WPT dot products of rows in \( A \) and columns in \( B \). The default value is \( WPT = 8 \) as it gives good overall performance on GPUs of different architectures but it can be tuned for the target GPU. The dot products are done in smaller chunks (tiles) in order to facilitate the use of the GPUs small and fast shared memory. The use of shared memory with tiling is another common approach to optimization in OpenCL and CUDA GEMM implementations. When \( A \) or \( B \) are triangular, threads automatically adapt the start and end of the dot products to avoid unnecessary reads for elements known to be zero. Triangularity is passed with the last two parameters of the kernel or as the template parameters of the `multiply` function in the Stan Math API. Example calls are below:

```cpp
matrix_cl C = multiply(A, B);
```
matrix_cl C = multiply<triangular_view_cl::LOWER>(A, B);
matrix_cl C = multiply<triangular_view_cl::ENTIRE, triangular_view_cl::UPPER>(A, B);
matrix_cl C = multiply<triangular_view_cl::UPPER, triangular_view_cl::LOWER>(A, B);

Special case: large $k$

When $k$ is multiple orders of magnitude larger than $n$ and $m$ and $n \times m$ is small, a small number of threads will compute long dot products thus reducing the occupancy of the GPU. In such cases we instead create $n \times m \times s$ threads, splitting the dot products into $s$ parts, each thread calculating a part of the scalar product. In order to avoid race conditions, we create $s$ copies of the resulting matrix, with the thread’s ID in the 3rd dimension determining the copy to write the result to. A separate support kernel then adds these copies together using $n \times m$ threads with thread $(i,j)$ assigned to add the $s$ copies of $C_{i,j}$. Since $n \times m$ is small, these extra copies do not have a large memory footprint. The value $s$ is determined based on the size of the input matrices. This optimization offers up to a 45% reduction in execution time compared to the GEMM kernel.

Special case: $C = AA^T$

Because $C$ is symmetric the threads where $j < i$ compute the values $C_{i,j}, ..., C_{(i,j+WPT-1}$ and map these values over the diagonal to $C_{j,i}, ..., C_{j+WPT-1,i}$. Threads where $j > i$ return immediately. This kernel is accessible through `multiply_transpose(matrix_cl& A)`.

Special case: vector inputs

We also implemented kernels that optimize multiplication of matrices with vectors, row vectors and scalars. The latter is only used as an internal function to implement other primitive function or gradients and is described in Section 3.5.

The general matrix multiplication kernel is not suitable for cases where the inputs are vector or row vectors as it would create additional threads without assigned work. In the case of matrix vector multiplication we create 1D work groups instead of 2D as we do in the GEMM kernel in order to avoid spawning overhead threads. When multiplying a $1 \times n$ row vector with a $n \times m$ matrix we create $m$ work groups of arbitrary size (default is 64). Each work group is assigned one of the $m$ scalar products.

Primitive function

The primitive function for general matrix multiplication accepts two Eigen matrices with elements of type double. Setting STAN_OPENCL flag does not guarantee that all functions that have GPU support will use the GPU. The function will use the existing CPU sequential implementations if the problem is small enough that no speedup is expected. For the primitive GEMM, the GPU routines are used if $n \times m > 250000$ and $k > 100$. These values are set so that the matrix multiplication that meets the criteria is guaranteed to be faster on any given GPU. These values could generally be set lower on mid and high-end GPUs.
The input Eigen matrices are used to create matrix_clobjects, which initiates the data transfer to the GPU if the matrices are not already cached (see Section 2.2).

Reverse mode function

We focused on optimizing the reverse mode for the general matrix multiplication (GEMM). Reverse modes for multiplication that involve scalars are not optimized using OpenCL. There are three implementations for GEMM in reverse mode: matrix multiplication of two matrices of stan::math::var objects, matrix multiplication of a matrix of doubles with a matrix of var objects and vice versa. The description of the general case is given here, with the other two cases only omitting certain copies.

In the forward pass we extract the values from the matrices and multiply them using the same kernel that is used in the primitive function. If the input matrix is a matrix of doubles, the extraction step can be ignored. In the chain function, that is used to evaluate the gradient, we have three input matrices of doubles (adjAB, B, A) and we need to calculate the following expressions:

\[ \text{adjA} = \text{adjAB} \ast \text{B.transpose();} \]
\[ \text{adjB} = \text{A.transpose()} \ast \text{adjAB;} \]

The transpose kernel is described in Section 3.5 while the GEMM kernel explained above is used for the multiplication. The thresholds on when to move the computation to the OpenCL device are the same as for the primitive form of matrix multiplication.

3.2. Solving triangular systems

Primitive function

We implemented a general solver for \(Ax = b\), where \(A\) is triangular and a special case where \(b\) is an identity matrix. In the latter, we transfer the input matrix to the GPU, if not already present in the GPUs global memory, and calculate the lower triangular inverse of \(A\) on the GPU. If \(A\) is upper triangular the input and output of the lower triangular inverse are transposed. The result is then copied back to the global memory of the host. The general solver only adds a multiplication of the inverse \(A^{-1}\) with the right-hand side \(b\).

The OpenCL implementation of the lower triangular inverse is based on our previous work (Češnovar and Štrumbelj 2017). The forward substitution algorithm that is commonly used in sequential implementations is not suitable even for multi-core parallel implementations as it requires constant communication between threads. The communication-free algorithm for multi-core CPUs proposed in (Mahfoudhi, Mahjoub, and Nasri 2012) is based on the divide-and-conquer approach. The idea is that we can split the matrix into submatrices as shown in Figure 2. The input matrix is split in three submatrices: \(A_1\), \(A_2\) and \(A_3\). We calculate the inverses of \(A_1\) and \(A_2\) in parallel to get \(C_1\) and \(C_2\). The remaining submatrix is then calculated using \(C_3 = -C_2A_3C_1\), with matrix multiplication also parallelized as shown in Section 3.1.

Our approach generalizes this for many-core architectures, calculating a batch of smaller inverses along the diagonal in the first step (blocks labeled S1 in Figure 3). For this step we use kernel batch_identity (see 3.5). It creates a large number of smaller identity matrices that are then used in diag_inv to calculate the inverses of the smaller submatrices along the diagonal.
Figure 2: Splitting into submatrices when computing the lower triangular inverse.

For a $n \times n$ matrix, this kernel is executed with $n$ threads split into $b$ work groups, where $b$ is the number of submatrices along the diagonal. Each work group is thus assigned one of the inverses along the diagonal. The rest of the submatrices are calculated by applying equation $C_3 = -C_2 A_3 C_1$. For Figure 3gu this is done in four steps, first calculating submatrices labeled $S_2$, then reapplying the same equation to calculate submatrices $S_3$, $S_4$ and $S_5$. Each step is done in two phases, first calculating $T = C_2 A_3$ with the \texttt{inv_lower_tri_multiply} and then $C_3 = -TC_1$ with the \texttt{neg_rect_lower_tri_multiply} kernel. Both kernels are based on the general matrix multiply kernel but modified to handle a batch of multiplications of small submatrices. The GPU support is only used when $n > 500$.

Figure 3: Splitting into submatrices when computing the lower triangular inverse.

Reverse mode function

In order to add GPU support to the reverse mode implementation of triangular system solvers, we used the lower triangular inverse kernel, the \texttt{GEMM} kernels, and the trivial transpose kernel. There are again three implementations for solving triangular systems in the form $Ax = b$, one with $A$ and $b$ being matrices of \texttt{var}, and two cases where either of them is a matrix of doubles. In the function evaluation phase, we use the same principles as in the primitive function with the added steps of extracting a matrix of doubles from the inputs, if needed. In the \texttt{chain()} function that is used to evaluate the gradient, we have three input matrices: $adjC$, $A$ and $C$. To evaluate the gradient we calculate the following:
A \times \text{adj}B = \text{adj}C; \\
\text{adj}A = - \text{adj}B \times C^\text{transpose};

Solving the system is done the same way as for the primitive function while the rest is done using a \texttt{GEMM} kernel, explained in Section 3.1, a transposing kernels and a kernel to multiply a matrix with a scalar, both described in Section 3.5.

### 3.3. Cholesky decomposition

**Primitive function**

There has been a recent push for implementing fast GPU implementations of dense Cholesky decompositions with the developments in the area of the so-called batched linear algebra, where the goal is to solve a large number of smaller problems in parallel (Abdelfattah, Haidar, Tomov, and Dongarra 2016; Dongarra, Duff, Gates, Haidar, Hammarling, Higham, Hogg, Lara, Relton, Tomov, and Zounon 2016). The developments in this area have also brought advances for the native mode Cholesky decomposition of a single large matrix. The state-of-the-art implementations presented in (Dong, Haidar, Tomov, and Dongarra 2014; Abdelfattah, Haidar, Tomov, and Dongarra 2017; nVidia 2012) use CUDA and are limited to NVIDIA GPUs. Our approach is based on the blocked version of the algorithm proposed in Louter-Nool (1992). It is based on the idea that if we partition the input matrix as shown in Figure 4, we can calculate the resulting submatrices as follows:

\[
L_{11} = \text{chol}(A_{11}) \\
L_{21} = A_{21}(L_{11}^T)^{-1} \\
L_{22} = A_{22} - L_{21}(L_{21}^T)
\]

The function \texttt{chol} computes the Cholesky decomposition of a smaller $b \times b$ block $A_{11}$ using the classic sequential algorithm. It is implemented with a simple kernel executed with $b$ threads. Parallel execution in the sequential algorithm is limited to parallelizing the inner loop and requires a lot of synchronization so all threads are organized in a single work group. These computations utilize the \texttt{lower_triangular_inverse} from Section 3.2 and the matrix multiplication and \texttt{multiply_transpose} from Section 3.1. We traverse through the input matrix using:
cholesky_decompose(A) {
    if (A.rows() == 0)
        return A;
    if (A.rows() <= min_L11_size) {
        return cholesky_decompose_kernel(A)
    }
    block = A.rows() / partition
    A_11 = A(0:block, 0:block)
    L_11 = cholesky_decompose(A_11);
    A(0:block, 0:block) = L_11
    block_subset = A.rows() - block
    A_21 = A(block:A.rows(), 0:block)
    L_21 = A_21 * transpose(lower_triangular_inverse(L_11))
    A(block:A.rows(), 0:block) = L_21
    A_22 = A(block:A.rows(), block:A.rows())
    L_22 = A_22 - multiply_transpose(L_21)
    L_rem_11 = cholesky_decompose(L_22);
    A(block:N, block:N) = L_rem_11
    return A;
}

Note that partition and min_L11_size are implementation parameters.

Reverse mode function

We re-implemented the Stan Math blocked Cholesky decomposition gradient in OpenCL. The blocked version of the gradient is based on Murray (2016). The input to the gradient is the values and adjoints of the reverse mode input matrix. Both matrices are lower triangular so their values are copied to the GPU in a packed form and then unpacked with the unpack kernel (see Section 3.5). The resulting values of the adjoints are packed with the pack kernel and transferred back to the host’s global memory. The gradient calculation pseudo-code is given below:

```plaintext
for (k = N; k > 0; k -= block_size_) {
    j = max(0, k - block_size_);
    R = L(j:k, 0:j)
    D = L(j:k, j:k)
    B = L(k:N, 0:j)
    C = L(k:N, j:k)
    R_adj = L_adj(j:k, 0:j)
    D_adj = L_adj(j:k, j:k)
    B_adj = L_adj(k:N, 0:j)
    C_adj = L_adj(k:N, j:k)
    C_adj = C_adj * lower_triangular_inverse(D)
    B_adj = B_adj - C_adj * R;
    D_adj = D_adj - transpose(C_adj) * C;
```
GPU-based Parallel Computation Support for Stan

Figure 5: Cholesky decomposition speedups for different GPUs and varying input matrix size $n \times n$. We measured the performance of the Cholesky decomposition on matrices of doubles, matrices of var, and the gradient of the Cholesky decomposition.

```cpp
D_adj = transpose(D) * D_adj
copy_lower_tri_to_upper_tri(D_adj)
D = transpose(lower_triangular_inverse(D));
D_adj = D * transpose(D * D_adj)
copy_lower_tri_to_upper_tri(D_adj)
R_adj = R_adj - transpose(C_adj) * B - D_adj * R;
D_adj = D_adj.diagonal() * 0.5
set_zeros_in_upper_tri(D_adj)
```

Note that `block_size` is set heuristically with default values that can be tuned for target GPUs.

**Speedup measurements**

We measured computation times of the Cholesky decomposition on an Intel Core i7-5600U CPU at 3.60GHz and three different GPUs: NVIDIA Titan XP, NVIDIA Tesla V100 and AMD R9 Fury. We ran each experiment for different input matrix sizes in increments of 1000 up to the size that caused a `std::bad_alloc` or out of memory on the GPU. We generated Toeplitz matrices with $A_{i,j; i \neq j} = n - |i - j|$ and $A_{i,i} = n^2$ to ensure positive definiteness. The measurement used at each input size was the median of 9 repetitions. Results are shown in Figure 5.

### 3.4. Generalized linear models

Generalized linear models (GLMs) are the most commonly used family of statistical models. A GLM is characterized by $E[Y] = g^{-1}(X\beta)$, where $Y$ is the target variable, $X\beta$ is a linear combination of input variables and coefficients, and $g$ is the link function.

Computing the likelihood for a GLM consists of computing the linear combination, transforming it with $g$ and finally computing the likelihood according to our choice of distribution for $Y$. We can improve performance by computing all steps simultaneously (including
analytically derived gradient), instead of performing each step individually and relying on autodifferentiation for the gradient. Four such GLM likelihood primitives are implemented in the Stan Math library: normal-identity (normal distribution with identity link, linear regression), Bernoulli-logit (logistic regression), Poisson-log, and negative binomial-log. Stan users can access them by calling the appropriate log-pdf/pmf function. These likelihoods can also be used as components in more complex models, such as regularized regression.

Note that the GLM likelihoods are heavily templated. With the exception of integers, every argument can have type var or double. Many arguments can either be scalars or vectors. For example, standard deviation in the normal-identity GLM can be the usual scalar parameter or, if we want to implement heteroskedasticity, a vector.

The GPU implementation of the GLM primitives is based on their CPU implementation that is already in Stan and not part of this work. First, the data are transferred to the GPU. The argument types also determine which derivatives we need to compute. This information is transferred to the GPU kernel by kernel arguments and it allows kernels to skip unnecessary computation.

Each GLM is implemented in a single kernel. The kernel requires execution of one thread per input variable. The number of threads is rounded up and they are organized into work groups of size 64. Each thread calculates one scalar product of the matrix-vector product and additional GLM-specific values and derivatives. Computation of the (log) likelihood ends with the summation over intermediate values, computed by each thread. Threads in a work group execute parallel reduction. First thread of each work group writes partial sum to the global memory. Partial sums are then transferred to the main memory and summation is completed by the CPU.

Derivatives with respect to coefficients and input variables require calculation of another matrix-vector product. Since another product cannot be efficiently computed in the same kernel, a GEMV kernel is run if these derivatives are needed.

GLM likelihood computation is executed on the GPU only when the dimension of the problem is large enough. More precisely, when \( c_1 \times n \times k + c_2 \times n > 1 \), where \( n \) and \( k \) are the number of observations and input variables, respectively, and \( c_1 \) and \( c_2 \) are constants that we determined empirically for each GLM separately.

### 3.5. Supporting GPU routines

Here we describe the remaining OpenCL kernels in Stan Math. These routines are not bottlenecks of statistical computation, so we opted for simplicity. All kernels use the default work group size determined by OpenCL.

These OpenCL kernels often do not outperform their CPU or Eigen equivalents, especially when we consider data transfers to or from the OpenCL GPU device. They should only be used as parts of an algorithm or when there is no data copy overhead. Note that all OpenCL kernels in Stan Math can be used as building blocks for new GPU routines.

All the kernels below assume input matrices of size \( m \times n \).

**Add and subtract**

These kernels add or subtract two matrices of the same size and store the result to a third matrix buffer. They execute on a grid of \( m \times n \) threads, where thread \((i, j)\) adds or subtracts
elements $A_{i,j}$ and $B_{i,j}$ and stores them to $C_{i,j}$.

The kernel code for adding two matrices:

```c
__kernel void add(__global double *C, __global double *A,
                  __global double *B, unsigned int rows,
                  unsigned int cols) {
    int i = get_global_id(0);
    int j = get_global_id(1);
    if (i < rows && j < cols) {
        C(i, j) = A(i, j) + B(i, j);
    }
}
```

**Multiplication with scalar**

Kernels `scalar_mul` and `scalar_mul_diagonal` can be used to multiply a matrix with a scalar. The former multiplies the entire input matrix with the given scalar while the latter multiplies only the diagonal elements. Similarly to the add and subtract kernels, the kernel `scalar_mul` is executed with $m \times n$ threads with each thread multiplying one matrix element. The kernel `scalar_mul_diagonal` is executed with $\min(m, n)$ threads with each thread multiplying one diagonal element.

**Matrix transpose**

Two kernels can be used for transposing a matrix and both are executed with $m \times n$ threads. In kernel `transpose` each thread copies an element from $A_{i,j}$ to $B_{j,i}$ and `triangular_transpose` copies the lower triangular of a matrix to the upper triangular or vice-versa. In the former case, the threads with indices under and on the diagonal perform $A_{j,i} = A_{i,j}$ and threads above the diagonal do nothing. In the latter case the roles are reversed.

**Matrix initialization**

Three kernels can be used to initialize a matrix: `zeros`, `identity` and `batch_identity`. For kernel `zeros` we specify the output matrix, its size, and if we want to set zeros only on the lower or upper triangular of the matrix. In both cases, we spawn $m \times n$ threads, with each thread assigned a single matrix element. Similarly, for kernel `identity` each thread is assigned a single matrix element. The `batch_identity` kernel is used to make a batch of smaller identity matrices in a single continuous buffer. This kernel is executed with $k \times n \times n$ threads, where $k$ is the number of $n \times n$ identity matrices to create. Each thread is assigned a single element in one of the batch matrices, with the thread ID $(0, \ldots, k-1)$ on the first dimension determining the matrix and the IDs on the remaining two dimensions determining the element of the matrix.

**Submatrix copy**

OpenCL provides functions that enable copying of entire matrix buffers. In order to add the functionality of copying arbitrary submatrices we implemented the `sub_block` kernel. This kernel copies a rectangular or triangular submatrix of size $k \times l$ from a source matrix to a given submatrix of the destination matrix of size $m \times n$. Each of the $k \times l$ threads that execute this kernel is assigned one element of the submatrix. The thread then determines
whether to copy the element based on the triangular view argument and its indices on the
two dimensions. Similarly to the `zeros` kernel, `sub_block` has a triangular view argument
that determines whether to copy the entire input matrix or only the lower/upper triangular
parts.

**Input checking**

We implemented kernels that allow the user to check whether the matrices stored on the
OpenCL devices are symmetric, contain NaN values or contain zeros on the diagonal. The
inputs are a pointer to the matrix buffer, its size and a pointer to a flag variable. If the
conditions of the check are met, the matrix sets the flag. No threads reset the flag so there
are no race conditions. `check_diagonal_zeros` is executed with \(\min(m, n)\) threads, each
thread checking one diagonal element for zeros. `check_nan` is executed with \(m \times n\) threads,
each thread checking one matrix element for NaN. For the kernel `check_symmetric` the flag
should be set before executing the kernel with the threads resetting it upon discovery of non-
symmetry. The kernel is again executed using \(m \times n\) threads, each thread checking whether
\(A_{i,j} - A_{j,i}\) is within tolerance.

4. Illustrative examples

In this section we show two illustrative examples that demonstrate the use of the new GPU
routines and what speedups we can expect.

- The experimental framework, data generation and measurements are written in R and
  are available as supplementary material. The framework calls `cmdstan` to compile the
  model and then calls the executable to run the sampling.

- The Stan code is identical for the CPU and GPU models. The only difference is that
  `cmdstan` is used with a different set of compilation flags.

- The measurements are end-to-end, measuring the speedups that a user would experience
during typical use. That is, we measure the time it takes from calling the model to
  sample to when the samples are output by the model. This includes reading the data
  and outputting the samples, but not model compilation time. Model compilation only
  has to be done once and we find it reasonable to assume that compilation time is
  negligible over the life span of the model. We ran each experiment for 200 warmup and
  200 sampling iterations.

- We ran the experiments on an Intel Core i7-5600U CPU at 3.60GHz and two different
  GPUs: NVIDIA Titan XP and AMD Sapphire Radeon VII.

- We made three measurements for each hardware-model-input size configuration. We
  calculate speedups by dividing the average of the three measurements for the device
  with the average of the three measurements for the reference device.

4.1. Logistic regression

We generated toy datasets as follows:
Each element of the $n \times k$ matrix $X$ is an independent draw from the standard normal. Dimensions $n$ and $k$ will vary.

The latent linear term $\mu_i$ depends only on the first two input variables and is drawn from $N(3X_{i,1} - 2X_{i,2} + 5, 1)$.

Target variable $y_i$ is set to 1 with probability $\frac{1}{1 + e^{-\mu_i}}$.

The model is standard logistic regression with Bernoulli likelihood and logit link function:

$$ y_i | \alpha, \beta \sim_{iid} \text{Bernoulli} \left( \frac{1}{1 + e^{-(X_i \beta + \alpha)}} \right). $$

We used the default flat priors provided by Stan:

```stan
data {
  int<lower=1> k;
  int<lower=0> n;
  matrix[n, k] X;
  int y[n];
}

parameters {
  vector[k] beta;
  real alpha;
}

model {
  target += bernoulli_logit_glm_lpmf(y | X, alpha, beta);
}
```

To measure the benefits of the GLM primitive without GPU support we also included the model that uses only the `bernoulli_logit` primitive. The only difference is in the model block:

```stan
model {
  target += bernoulli_logit_lpmf(y | X * beta + alpha);
}
```

The results are summarized in Figure 6.

### 4.2. GP regression

We generated toy datasets as follows:

- Each of the $n$ elements of $x$ is an independent draw from $U(-10, 10)$. Dimension $n$ will vary.
- Target variable $y_i$ is drawn from $N(f(x), \frac{1}{10})$, where $f(x) = \beta(x + x^2 - x^3 + 100 \sin 2x - 1)$. Parameters $\beta$ and $\alpha$ were set so that $E[f] = 0$ and $\text{Var}[f] = 1$.

The model is a 1D GP regression with hyperpriors (Betancourt 2017). For the sake of brevity, we do not include the model code in the text.

The results are summarized in Figure 7.
Figure 6: Times and speedups for logistic regression. Note the that $n$ and $k$ were limited by CPU computation time. The plots indicate that GPU performance has not yet peaked for these GPUs – better speedups can be expected for larger $n$ and/or $k$. 
GPU-based Parallel Computation Support for Stan

Figure 7: Times and speedups for GP regression. Again, \( n \) was limited by CPU computation time. The plots indicate that GPU performance has not yet peaked for these GPUs – better speedups can be expected for larger \( n \). Note that speedups for 4096 and 6144 are based on extrapolated CPU times. We used linear regression in log-log space, where CPU measurements form an almost perfect line, as we would expect from the cubic time complexity of GP regression.

5. Discussion and conclusion

GPU support will provide Stan users with practically meaningful speedups in moderate to large problems, even on mid-range retail gaming GPUs. Furthermore, the speedup estimates from our illustrative examples are very conservative for two reasons. First, additional speedup can be achieved by using compute-only GPUs. And second, a substantial part of the time in the logistic regression example is due to slow I/O in cmdstan – the data (samples) are passed (from) the model as text files.

The benefits come with almost zero user effort and without the need to change existing Stan code. As part of future work, we will be adding other matrix algebra and statistical likelihood primitives, such as other matrix decompositions, multinomial regression, ordinal regression and covariance functions. Expert users can already take advantage of the API and add their own GPU kernels.

The API for Stan’s OpenCL backend is still evolving. The next improvements include easier access to the specialized triangular multiplication routines, better caching of immutable data on the device between MCMC (Markov chain Monte Carlo) iterations, and easier access and construction of matrix\_cl types when passing an Eigen matrix or vector of vars to the OpenCL device. As we write more functions that Stan can use to access OpenCL we will develop a load balancer to handle multiple devices.

Tuning GPU-based computation is also an interesting challenge. It consists of two parts: (a) how to set optimal parameters for the GPU routines and (b) when to move computation to the GPU (when does the speedup justify the overhead). The iterative MCMC (or optimization) setting lends itself to the possibility of efficient online tuning, because computation and input
dimensions are constant over all iterations.

### Computational details

The results in this paper were obtained using the post 2.19.1 develop branch of Stan and cmdstan and an experimental branch of Stan Math. All the components can be obtained via:

```bash
git clone --shallow-submodules --recursive --depth 1 -b jss_special_issue --single-branch https://github.com/bstatcomp/cmdstan.git
```

All the functionality is planned for release in Stan Math 2.20.

Detailed instructions on how to install cmdstan and activate GPU support are available here:

[https://github.com/bstatcomp/gpu-stan-paper-materials](https://github.com/bstatcomp/gpu-stan-paper-materials)

This public repository also contains all the replication and visualization scripts, measurement data, and a simple example that can be used to check if cmdstan was successfully compiled with GPU support. Note that a standalone replication script for all the measurement results in this paper is not feasible, because the measurements were made on different hardware and cmdstan compiled with and without GPU support.

### Acknowledgments

This research was supported by the Slovenian Research Agency (ARRS, project grant L1-7542 and research core funding P5-0410). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan XP GPU used for this research. We gratefully acknowledge the support of Amazon with the Amazon Research Award.

### References

Abdelfattah A, Haidar A, Tomov S, Dongarra J (2016). “Performance Tuning and Optimization Techniques of Fixed and Variable Size Batched Cholesky Factorization on GPUs.” *Procedia Computer Science*, **80**, 119 – 130. ISSN 1877-0509. [doi:https://doi.org/10.1016/j.procs.2016.05.303](https://doi.org/10.1016/j.procs.2016.05.303). International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA, URL [http://www.sciencedirect.com/science/article/pii/S1877050916306548](http://www.sciencedirect.com/science/article/pii/S1877050916306548).

Abdelfattah A, Haidar A, Tomov S, Dongarra J (2017). “Fast Cholesky Factorization on GPUs for Batch and Native Modes in MAGMA.” *Journal of Computational Science*, **20**, 85 – 93. ISSN 1877-7503. [doi:https://doi.org/10.1016/j.jocs.2016.12.009](https://doi.org/10.1016/j.jocs.2016.12.009). URL [http://www.sciencedirect.com/science/article/pii/S1877750316305154](http://www.sciencedirect.com/science/article/pii/S1877750316305154).

Adam, Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer (2017). “Automatic Differentiation in PyTorch.” In *NIPS Autodiff Workshop*.

Baudart G, Hirzel M, Mandel L (2018). “Deep Probabilistic Programming Languages: A Qualitative Study.” *arXiv preprint arXiv:1804.06458*. 
GPU-based Parallel Computation Support for Stan

Bergstra J, Breuleux O, Bastien F, Lamblin P, Pascanu R, Desjardins G, Turian J, Warde-Farley D, Bengio Y (2010). “Theano: a CPU and GPU Math Expression Compiler.” In Proceedings of the Python for scientific computing conference (SciPy), volume 4. Austin, TX.

Betancourt M (2017). “Robust Gaussian Processes in Stan, Part 3.” Retrieved from https://betanalpha.github.io/assets/case_studies/gp_part3/part3.html.

Bingham E, Chen JP, Jankowiak M, Obermeyer F, Pradhan N, Karaletsos T, Singh R, Szerlip P, Horsfall P, Goodman ND (2019). “Pyro: Deep Universal Probabilistic Programming.” The Journal of Machine Learning Research, 20(1), 973–978.

Bürkner PC, et al. (2017). “brms: An R Package for Bayesian Multilevel Models Using Stan.” Journal of Statistical Software, 80(1), 1–28.

Carpenter B, Gelman A, Hoffman MD, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017). “Stan: A Probabilistic Programming Language.” Journal of Statistical Software, 76(101).

Češnovar R, Štrumbelj E (2017). “Bayesian Lasso and Multinomial Logistic Regression on GPU.” PLoS ONE, 12(6), e0180343.

Dong T, Haidar A, Tomov S, Dongarra J (2014). “A Fast Batched Cholesky Factorization on a GPU.” In 2014 43rd International Conference on Parallel Processing, pp. 432–440. ISSN 0190-3918. doi:10.1109/ICPP.2014.52.

Dongarra JJ, Duff I, Gates M, Haidar A, Hammarling S, Higham NJ, Hogg J, Lara P, Relton SD, Tomov S, Zounon M (2016). “A Proposed API for Batched Basic Linear Algebra Subprograms.”

Gabry J, Goodrich B (2016). “rstanarm: Bayesian Applied Regression Modeling via Stan.” R package version, 2(1).

Guennebaud G, Jacob B, et al. (2010). “Eigen v3.” http://eigen.tuxfamily.org.

Hoffman MD, Gelman A (2014). “The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo.” Journal of Machine Learning Research, 15(1), 1593–1623.

Louter-Nool M (1992). “Block-Cholesky for Parallel Processing.” Appl. Numer. Math., 10(1), 37–57. ISSN 0168-9274. doi:10.1016/0168-9274(92)90054-H. URL http://dx.doi.org/10.1016/0168-9274(92)90054-H.

Mahfoudhi R, Mahjoub Z, Nasri W (2012). “Parallel Communication-Free Algorithm for Triangular Matrix Inversion on Heterogeneous Platform.” In 2012 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 553–560.

Matsumoto K, Nakasato N, Sedukhin SG (2014). “Implementing Level-3 BLAS Routines in OpenCL on Different Processing Units.”

Murray I (2016). “Differentiation of the Cholesky Decomposition.” arXiv e-prints, arXiv:1602.07527. 1602.07527.
Nugteren C (2017). “CLBlast: A Tuned OpenCL BLAS Library.” CoRR, abs/1705.05249. 
1705.05249, URL http://arxiv.org/abs/1705.05249.

Nugteren C (2018). “CLBlast: A Tuned OpenCL BLAS Library.” In Proceedings of the International Workshop on OpenCL, IWOCL ‘18, pp. 5:1–5:10. ACM, New York, NY, USA. ISBN 978-1-4503-6439-3. 
doi:10.1145/3204919.3204924. URL http://doi.acm.org/10.1145/3204919.3204924.

Nvidia (2009). “NVIDIA OpenCL Best Practices Guide .”

Nvidia (2010). “NVIDIA CUDA C Programming Guide.”

nVidia (2012). CUBLAS Library User Guide. nVidia, v5.0 edition. URL http://docs.nvidia.com/cublas/index.html.

Salvatier J, Wiecki TV, Fonnesbeck C (2016). “Probabilistic Programming in Python using PyMC3.” PeerJ Computer Science, 2, e55.

Stone JE, Gohara D, Shi G (2010). “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems.” IEEE Des. Test, 12(3), 66–73. ISSN 0740-7475. 
doi: 10.1109/MCSE.2010.69. URL http://dx.doi.org/10.1109/MCSE.2010.69.

Taylor SJ, Letham B (2018). “Forecasting at Scale.” The American Statistician, 72(1), 37–45.

Tran D, Hoffman MD, Saurous RA, Brevdo E, Murphy K, Blei DM (2017). “Deep Probabilistic Programming.” arXiv preprint arXiv:1701.03757.

Van Heesch D (2008). “Doxygen: Source Code Documentation Generator Tool.” URL: http://www.doxygen.org.

Zheng, Chen T, Li M, Li Y, Lin M, Wang N, Wang M, Xiao T, Xu B, Zhang C, Zhang (2015). “Mxnet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems.” arXiv preprint arXiv:1512.01274.
Affiliation:
Rok Češnovar, Davor Sluga, Jure Demšar, Tadej Ciglarič, Erik Štrumbelj
University of Ljubljana
Faculty of Computer and Information Science
Večna pot 113
1000 Ljubljana, Slovenia
E-mail: erik.strumbelj@fri.uni-lj.si

Steve Bronder
Financial Institutions Group
Capital One
11 West 19th Street
New York City, NY
E-mail: sbronder@stevebronder.com

Sean Talts
ISERP
Columbia University
New York City, NY
E-mail: sean.talts@gmail.com