University of Ljubljana
Faculty of Computer and Information Science

bstatcomp

# GPU Optimized Math Routines in the Stan Math Library

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

# MAIN GOAL

Faster model inference for Stan users …

… in a seamless and cost-effective way.

# TALK OUTLINE

- Motivation: GP regression.
- Parallelization & GPUs.
- Stan + OpenCL.
- Paralellizing the Cholesky decomposition.
- Challenges & Roadmap.

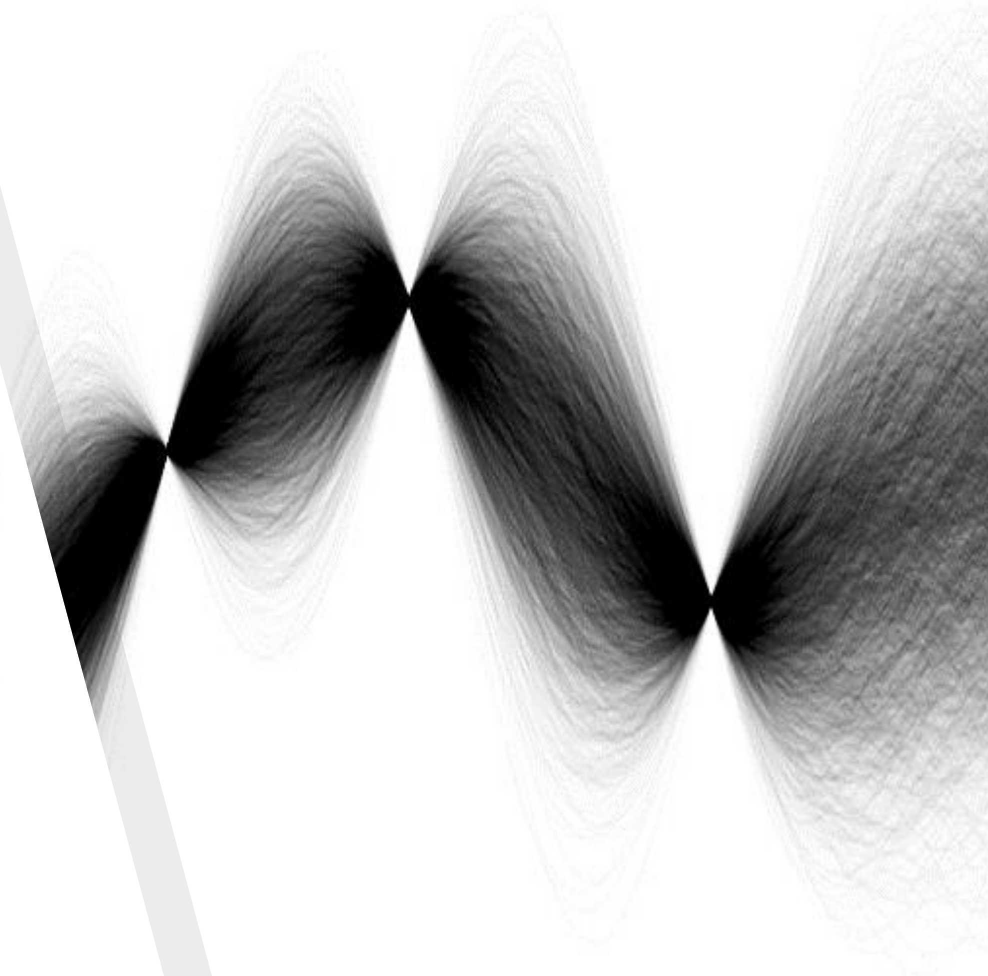
# GP REGRESSION

Gaussian Processes are very useful but ...

... computation scales unfavorably - $O(n^3)$.

What can we do?

- Approximate inference.
- Run the computation on a better CPU.
- Run the computation in parallel.



# PARALLELIZATION 101

Break the problem up into smaller pieces and compute them in parallel.

The pieces must be:

- Small enough to keep all individual processing units busy most of the time.
- Large enough to avoid too much overhead with breaking them up and putting them back together.

Maximum speedup limited by the parts we can‘t parallelize (thanks, Amdahl!).

# WHY GPUs?

Properties of GPUs:

- Everyone has a GPU.
- Massive parallelism – thousands of „cores“ (best performance/cost ratio).
- Optimized for vector and matrix problems.
- Faster data transfers compared to clusters.
- Energy-efficient.

# Stan
# + OpenCL

# OpenCL

- Parallel framework CPUs, GPUs, DSPs, FPGAs,...
- Special functions (kernels) are executed by N threads on target devices.
- APIs for C, C++, Python, Julia...
- Open standard maintained by the Khronos Group.

**Typical example:**

- Copy input data to the OpenCL device,
- parallel execution of special functions on the OpenCL device and
- copy the results back to the host.

# What‘s an OpenCL Context?

It‘s like a scheduler:

- Manages the devices, queues, platforms, memory alloc, etc.
- Devices: GPUs and CPUs.
- Platforms: Implementations of OpenCL (Khronos’s OpenCL vs. Intel’s OpenCL).
- The context has a ‘program’ object that manages kernels for devices and platforms.

# OpenCL **Context**

- We only want one context to exist so it's stored as a singleton.
- Access the context through an adapter API.

IE: In the adapter class `opencl_context`

```
// Return the stan program's context
inline cl::Context& context() {
  return opencl_context_base::getInstance().context_;
}

...

// developers access
auto ctx = opencl_context.context()
```

# Making **Kernels**

- We want to make it simple for users to add and use kernels.
- Reworked design with Sean Talts and Rok.

**Making a kernel:**

A kernel that only needs to set the global work size.

Name of kernel

```
const global_range_kernel<cl::Buffer, cl::Buffer, int, int> copy("copy", copy_kernel_code);
```

Argument Types kernel accepts

String literal holding the kernel code

# Accessing Matrices
## on the Device

Developers move Stan matrices over to `matrix_cl` matrices:

```
matrix_d d1
matrix_cl d1_cl(d1)
```

Users operate on these like Stan matrices:

```
matrix_cl d3_cl = d1_cl + d2_cl
matrix_cl d4_cl = cholesky_decompose(d1_cl)
```

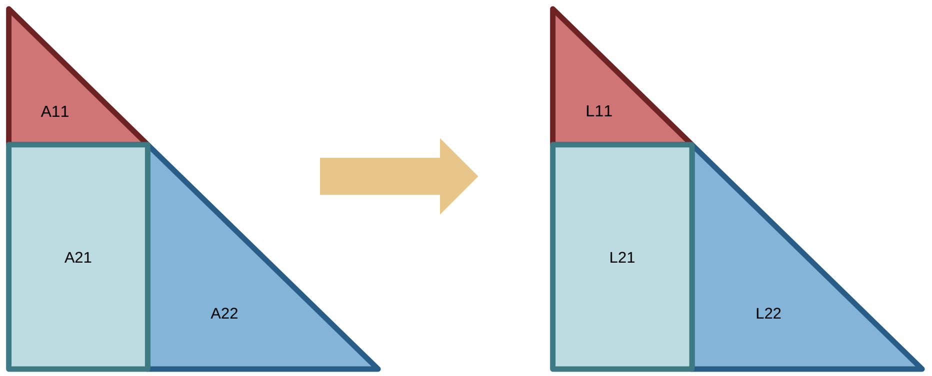
OpenCL

First GPU-optimization available to Stan users:

# Cholesky Decomposition

# Cholesky Decomposition

- Our first bottleneck target in Stan.
- Almost no naive parallelim in the basic algorithm.
- No real speedup on the GPU.
- Blocked-cholesky is computationally more complex, but GPUs are made for fast matrix multiplications.




$$L_{21} = A_{21}(L_{11}^T)^{-1}$$
$$L_{22} = A_{22} - L_{21}(L_{21})^T$$

# Derivative of the Cholesky Decomposition

GPU implementation of Murray (2016):

- Eigen version already in Stan Math,
- largest bottlenecks are matrix multiplication and lower triangular inverse.

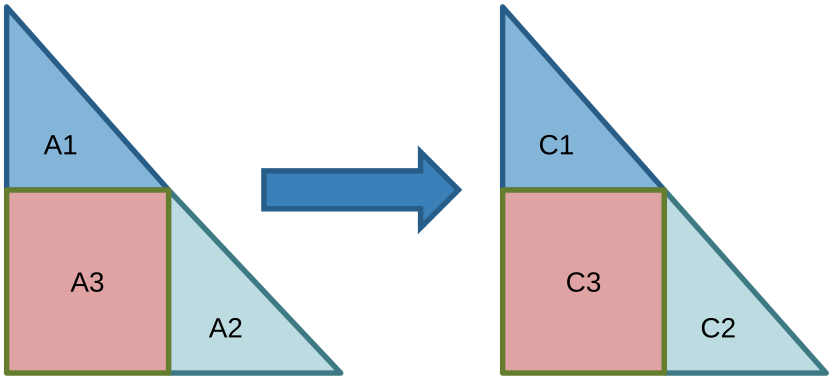
As a consequence - GPU implementations of

- matrix multiplication,
- lower triangular inverse,
- +, -, transpose, partition.

function chol_blocked_rev($L$, $\bar{A}$)
    # *If at input* $\mathrm{tril}(\bar{A}) = \bar{L}$, *at output* $\mathrm{tril}(\bar{A}) = \mathrm{tril}(\bar{\Sigma})$, *where* $\Sigma = LL^{\top}$.
    for $k = N$ to no less than 1 in steps of $-N_b$:
        $j \leftarrow \max(1,\ k - N_b + 1)$
        $R,\ D,\ B,\ C$ = level3partition($L$, $j$, $k$)
        $\bar{R},\ \bar{D},\ \bar{B},\ \bar{C}$ = level3partition($\bar{A}$, $j$, $k$)
        $\bar{C} \leftarrow \bar{C}D^{-1}$
        $\bar{B} \leftarrow \bar{B} - \bar{C}R$
        $\bar{D} \leftarrow \bar{D} - \mathrm{tril}(\bar{C}^{\top}C)$
        $\bar{D} \leftarrow$ chol_unblocked_rev$(D, \bar{D})$
        $\bar{R} \leftarrow \bar{R} - \bar{C}^{\top}B - (\bar{D} + \bar{D}^{\top})R$
    return $\bar{A}$

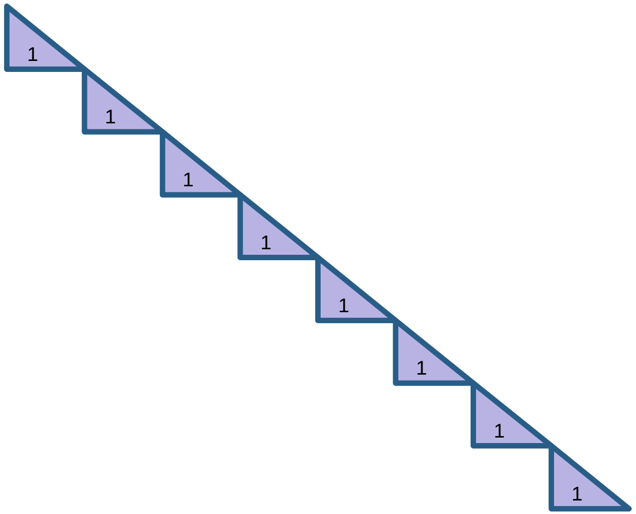
# Lower triangular inverse

GPU Basic forward substiution algorithm not suitable for GPUs.

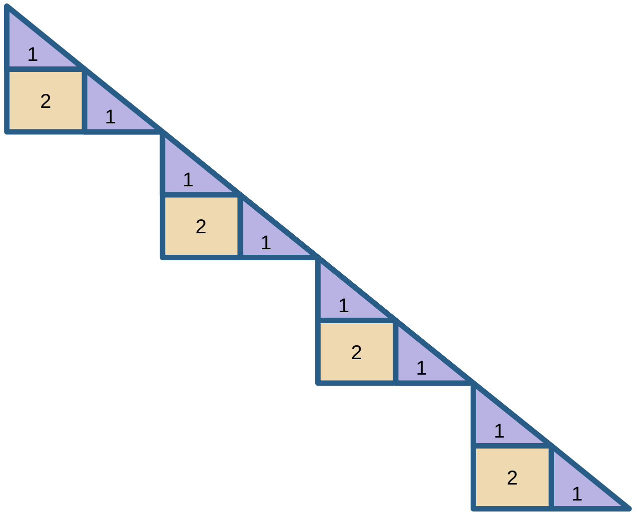



$$C_1 = A_1^{-1}$$
$$C_2 = A_2^{-1}$$
$$C_3 = -C_2 A_3 C_1$$

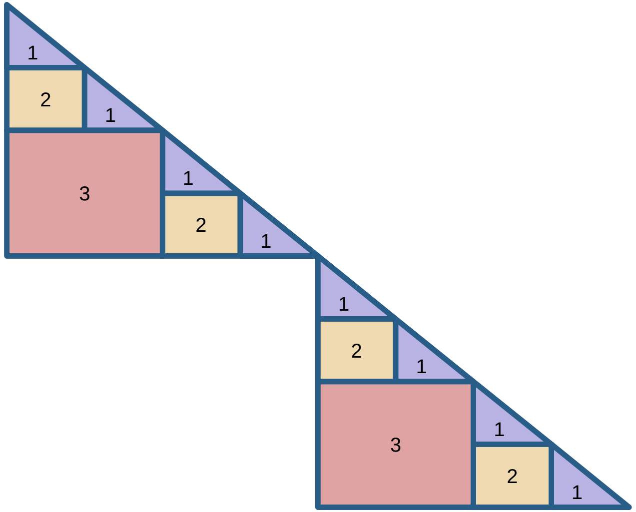
# Lower triangular inverse



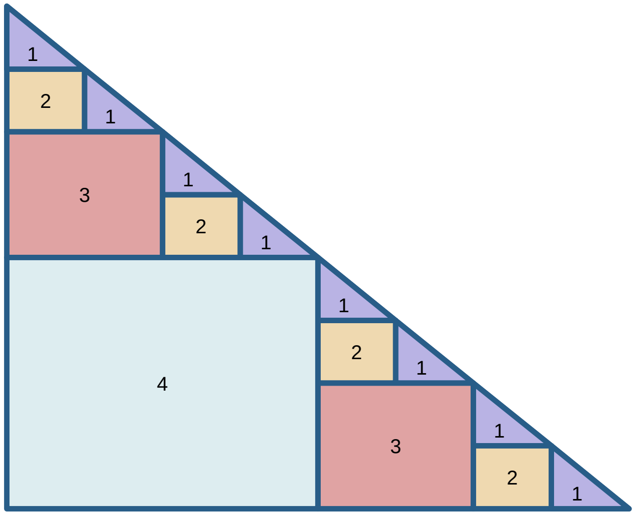
# Lower triangular inverse



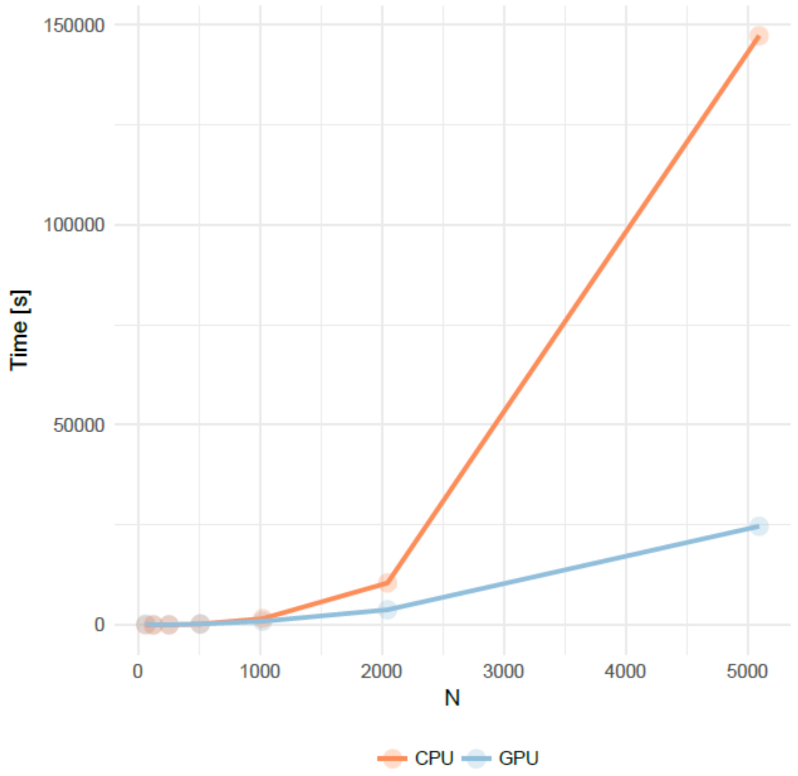
# Lower triangular inverse

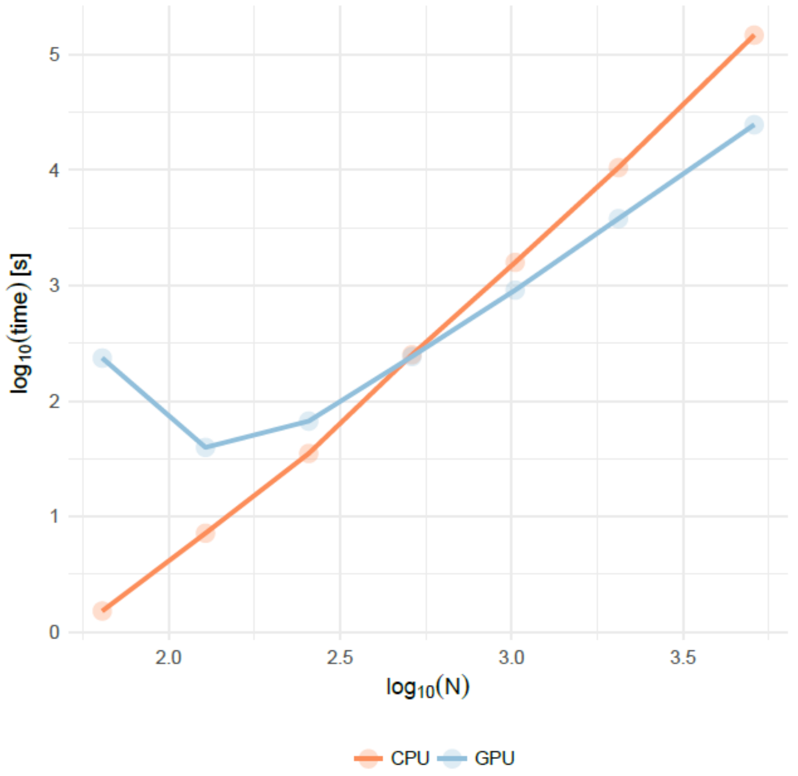


# Lower triangular inverse

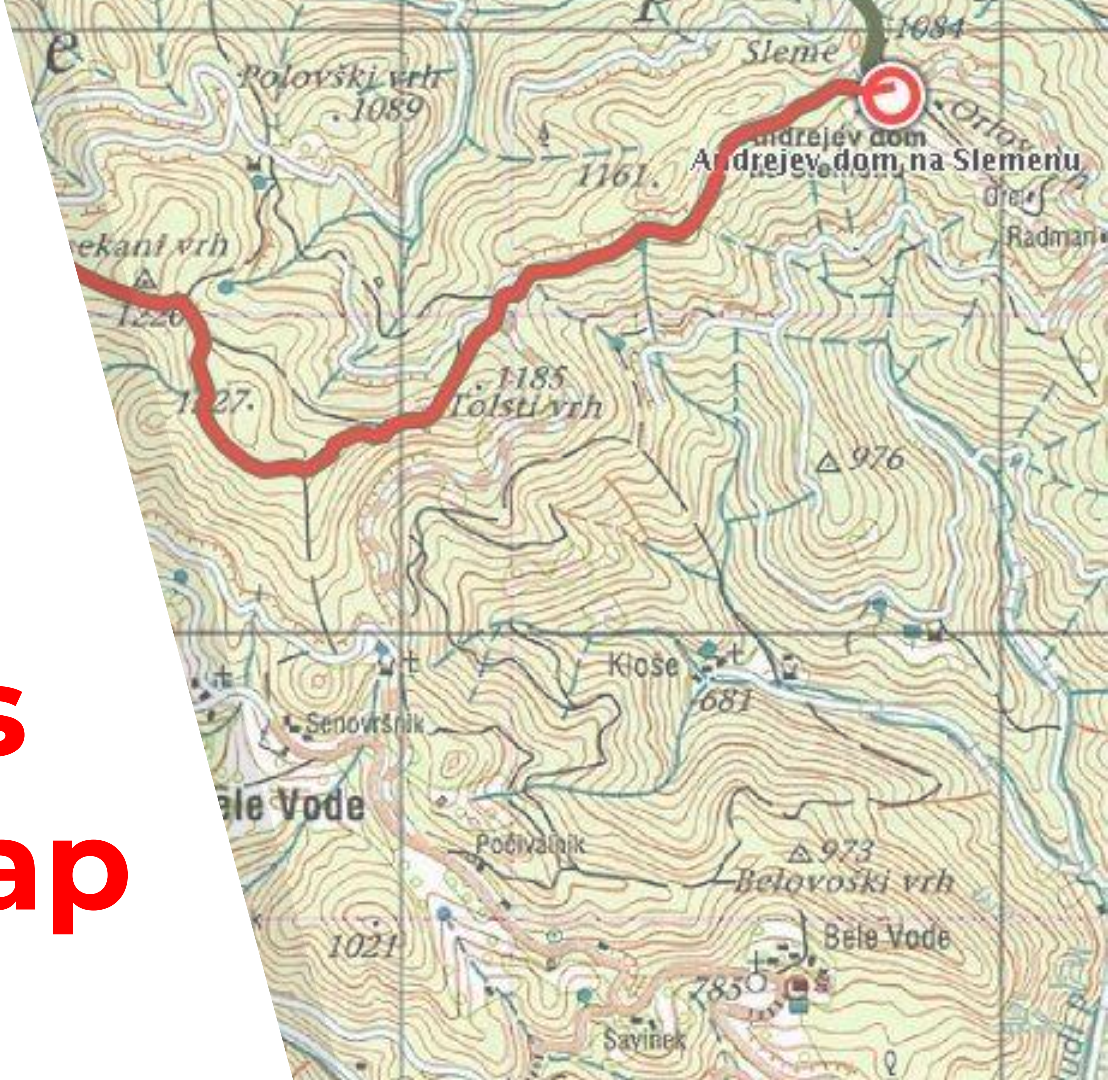


# RESULTS (simple 1D GP regression on N points)



# Challenges & Roadmap



# Issue: Data tranfers

Currently, speedups are limited by data transfers:

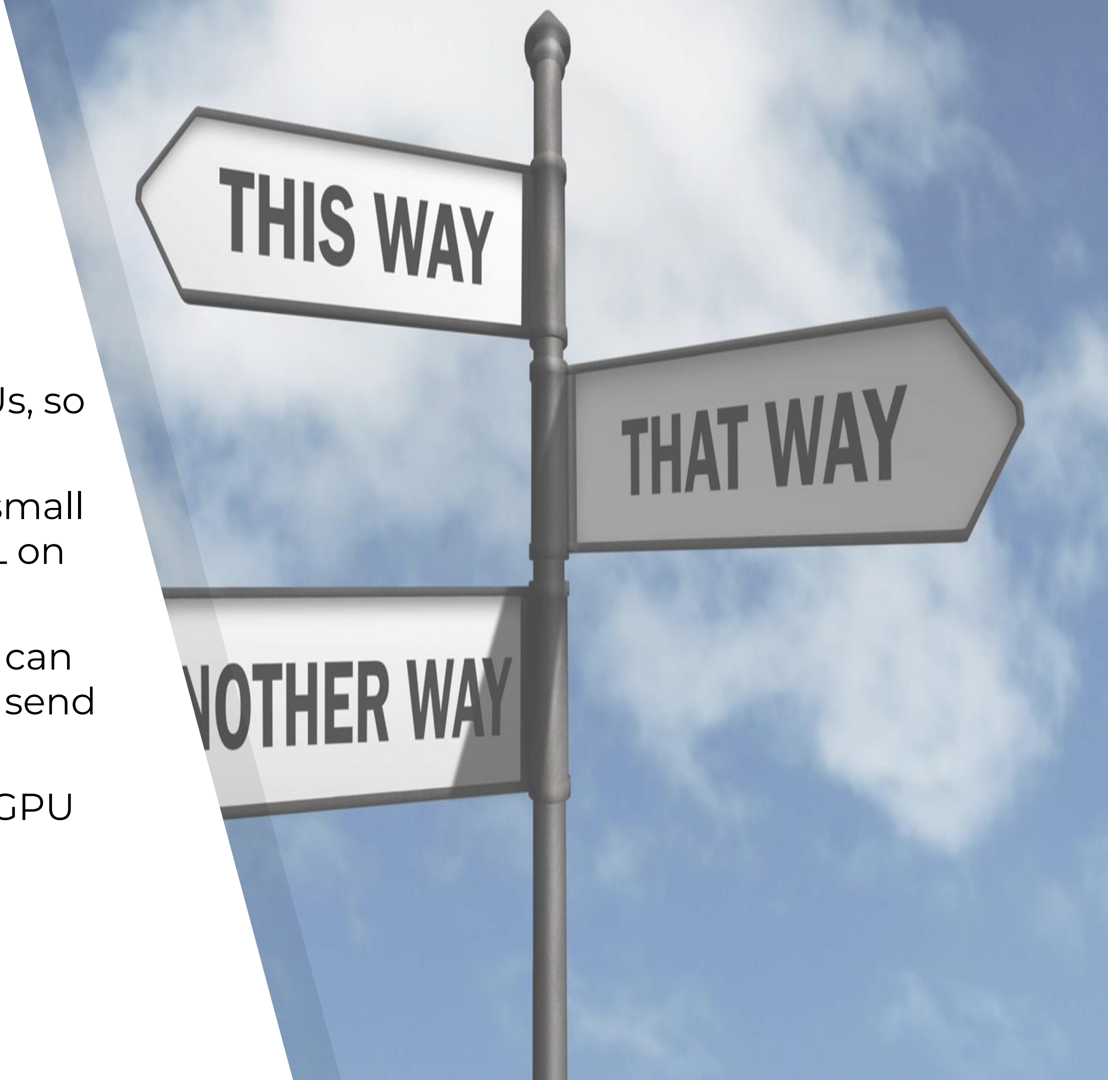
- Copying data to the GPU costs us some time.
- Most functions scale linearly and can‘t justify this cost for every call.
- Even is such functions represent 10% of the computation, it drastically limits the maximum speedup.

**Example:**

```
cov_exp_quad(x1, magnitude_1, length_scale_1)
  + cov_exp_quad(x1, magnitude_2, length_scale_2)
  + gp_periodic_cov(x1, magnitude_3, length_scale_3_1, 7)
 .* cov_exp_quad(x1, 1.0, length_scale_3_2)
  + gp_periodic_cov(x1, magnitude_4, length_scale_4_1, 365.25)
 .* cov_exp_quad(x1, 1.0, length_scale_4_2)
  + gp_dot_prod_cov(I_s, magnitude_5_1)
  + gp_dot_prod_cov(I_ss, magnitude_5_2)
  + gp_dot_prod_cov(I_ws, magnitude_5_3)
  + diag_matrix(rep_vector(jitter, N2));`
```

# Multiple Devices and OOM Algorithms

- OpenCL can run on CPUs and GPUs, so why not use both?
- We know some problems are too small to send to the GPU, so use OpenCL on the CPU.
- Needs a 'smart' load balancer that can look at a 'job' and decide where to send it.
- Current algorithms are limited by GPU DRAM.



# From **single routines** to (almost) **entire models**

**Idea:** move the bulk of the log-posterior computation (including „hand-made“ gradients) to the GPU.

Promising results on some models (~100x speedup).

Currently in the works:

- GLM (linear, logistic, Poisson, NB regression...). For example:

  ```
  bernoulli_logit_glm_lpdf(y | X, beta, alpha);
  ```

- Roadblock: Data transfers!

# Conclusion

- GPU support in Stan: Coming very soon!
- First, inverting covariance matrices…
- … other building-blocks will follow.
- Reasonable expectation: 10-200x speedup
- Prototype branch:

  https://github.com/bstatcomp/math/tree/stancon2018

If you have any questions, comments, ideas… let us know! We‘re also accepting requests. ☺