KernelMatmul: Scaling Gaussian Processes to Large Time Series

Tilman Hoffbauer[1], **Holger Hoos**[1,2,3], **Jakob Bossek**[4]

[1]Chair for AI Methodology, RWTH Aachen University, Germany
[2]Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands
[3]University of British Columbia, Canada
[4]Chair for Machine Learning and Optimisation, Paderborn University, Germany
tilman.hoffbauer@rwth-aachen.de

Abstract

Time series forecasting requires reliable uncertainty estimates. Gaussian process regression provides a powerful framework for modelling this in a probabilistic fashion. However, its application to large time series is challenging, due to its cubic time complexity and quadratic memory requirement. In this work, we present KernelMatmul, a novel method that accelerates Gaussian process inference and thus facilitates scaling of Gaussian process regression to large, irregularly sampled and multi-output time series. Leveraging conjugate gradients in combination with sparsity approximation, KernelMatmul achieves time and memory complexity linear in the number of samples. We thoroughly benchmark our new method against multiple baselines to demonstrate its benefits and limitations, both in efficiency and accuracy.

Implementation —
https://github.com/Turakar/kernel-matmul
Experiments —
https://github.com/Turakar/kernel-matmul-benchmark

1 Introduction

Time series data is common in our modern society. Examples include environmental sensors, medical monitoring data, machine operation, and traffic measurements (Chandola and Vatsavai 2010; Dürichen et al. 2015; Chandola and Vatsavai 2011; Godahewa et al. 2021). These time series contain a wealth of information about the underlying processes. When performing predictions on these time series, we require not only an estimate of the expected value but also of the prediction confidence. Many predictions become uncertain the further away we move from our initial data due to unforeseen influences (e.g., weather changes).

Gaussian processes (GPs) provide us with a mathematically rigorous way to model these uncertainties (Rasmussen and Williams 2006). To this end, a probabilistic method is employed. GPs model the covariance between samples to obtain a prior distribution. The covariance matrix over all samples is called the kernel matrix. Given this prior, a predictive posterior distribution can be obtained from existing observations by Bayesian principles; this distribution is multivariate normal and provides both a mean and a confidence estimate. As such, the application of GPs to time series seems beneficial, but there is one important restriction: For inference of the posterior distribution, we need to perform costly linear algebra operations on the kernel matrix, which incur a computational complexity of $\Theta(N^3)$ and a memory requirement of $\Theta(N^2)$ for N samples when using the traditional approach to inference with the Cholesky decomposition (Rasmussen and Williams 2006). This quickly becomes infeasible for large N.

Thus, other inference schemes are required for analysing large time series using GPs. Often, this includes approximations: We can approximate the posterior of the GP, e.g., using variational inference (Hensman, Matthews, and Ghahramani 2015), or the prior kernel matrix, e.g., using structured kernel interpolation (Wilson and Nickisch 2015). In this work, we will focus on accelerating inference using conjugate gradients (CG) (Gardner et al. 2018), which replaces the Cholesky decomposition by an iterative solver and mainly requires one central operation: The multiplication of the kernel matrix by an arbitrary right-hand side (RHS). Thus we can avoid storing the kernel matrix in memory and instead compute it on-the-fly. Further, we add a sparsity assumption, assuming that the covariance of distant samples is zero.

Our core contribution is KernelMatmul, an implementation of the kernel matrix multiplication operator with key benefits:

- on-the-fly computation of the kernel matrix to reduce the $\Theta(N^2)$ memory requirement to $\Theta(N)$;
- an (optional) sparsity assumption suitable for large time series, reducing the computational complexity of kernel matrix multiplication from $\Theta(N^2)$ to $\Theta(N)$;
- GPU acceleration using CUDA (NVIDIA 2023b), which partially hides the latency of on-the-fly computation behind the global memory latency.

These benefits are implemented by a single CUDA kernel with accompanying Python bindings that performs the kernel matrix multiplication and is easily integrated into existing frameworks, such as GPyTorch (Gardner et al. 2018).

After the introduction of the background (Section 2) and a discussion of related work (Section 3), we introduce KernelMatmul (Section 4). Our work includes a benchmark of multiple approaches to kernel matrix multiplication performance (including KeOps (Charlier et al. 2021)). This bench-

mark demonstrates improved performance of KernelMatmul on many configurations (Section 5.1). An investigation of the residuals (Section 5.2) showcases the low error introduced by our sparsity approximation. Additionally, we provide a comparison (Section 5.3) to other approximation schemes, such as variational inference (Wu, Pleiss, and Cunningham 2022) and structured kernel interpolation (Wilson and Nickisch 2015). Finally, we end with a short conclusion (Section 6).

2 Gaussian Processes for Time Series

Instead of estimating a single function f to explain our data, we model a distribution over functions using a GP (Rasmussen and Williams 2006). A GP is defined by its mean and covariance function, which define a normal distribution for every finite set of samples. For a scalar input like in time series, we obtain the following definition for all finite sets of locations $\mathbf{x} = (x_1, \ldots, x_N)^\top \in \mathbb{R}^N$:

$$f \sim \mathcal{GP}(\mu(\cdot), K(\cdot, \cdot)) \tag{1}$$

$$\Leftrightarrow (f(x_1), \ldots, f(x_N))^\top \sim \mathcal{N}(\mu(\mathbf{x}), K(\mathbf{x}, \mathbf{x})), \tag{2}$$

where $\mu : \mathbb{R}^N \to \mathbb{R}^N$ defines the expected mean (often set to a trained constant) and $K : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^{N \times N}$ defines the covariance of all samples. Importantly, the kernel matrix $K(\mathbf{x}, \mathbf{x})$ has to be positive semi-definite for arbitrary inputs and is defined element-wise by the kernel function $(K(\mathbf{x}, \mathbf{x}))_{i,j} = k(x_i, x_j)$, $1 \leq i, j, \leq N$. Commonly, one uses the radial basis function (RBF) kernel. This kernel function is stationary and thus expressed by the distance $\tau = x - x'$. In the following, we use $k(\tau)$ and $k(x, x')$ interchangeably for stationary kernels.

$$k(\tau) = \exp\left(-\frac{\tau^2}{2 \cdot \lambda^2}\right), \tag{3}$$

where $\lambda > 0$ defines the lengthscale of the covariance expressed by the kernel function. Given a certan prediction horizon τ_*, we want to choose λ large enough for meaningful predictions at that horizon. If λ is too low, $k(\tau_*)$ would be close to zero, and the prediction would be dominated by the mean. Thus, we choose λ such that $k(\tau_*) \geq 0.9$, where 0.9 is an arbitrarily chosen threshold. This is fulfilled by

$$\lambda = \sqrt{-\frac{\tau_*^2}{2 \cdot \log(0.9)}} \tag{4}$$

for the RBF kernel (derivation provided in the supplementary material).

To model periodic patterns in time series, we decided to use the spectral kernel function (Wilson and Adams 2013):

$$k(\tau) = \exp\left(-\frac{\tau^2}{2 \cdot \lambda^2}\right) \cdot \cos\left(2 \cdot \pi \cdot \nu \cdot \tau\right) \tag{5}$$

where $\nu > 0$ defines the frequency of the modelled periodic signal. Periodic patterns are often found in time series, and they have proven beneficial in previous studies (de Wolff, Cuevas, and Tobar 2021; Altamirano and Tobar 2022). With an RBF kernel, no proper predictions can be made beyond a trivial interpolation to the mean. Note that this kernel converges to the RBF kernel for $\nu \to 0$. We use the same lengthscale as for the RBF kernel, as the periodic component is only a modulation of the RBF kernel.

Before training, we initialise the parameters from the peaks of a Lomb-Scargle periodogram (Press and Rybicki 1989) of the training data, similar to the work by de Wolff, Cuevas, and Tobar (2021). The Lomb-Scargle method fits a mixture of sine and cosine waves to the signal to find its periodic components. In contrast to the fast Fourier transform (FFT), it is applicable to time series with non-equidistant sampling. To model multi-output time series and multiple frequencies of the spectral kernel, the standard linear model of coregionalization (LMC) (Bonilla, Chai, and Williams 2007) is employed. As such, if there is only one output, as it is the case in our experiments, the LMC corresponds to replacing the kernel function by a weighted sum of spectral kernel functions with different parameters. We provide more details on this initialization in the supplementary material.

For inference, the following posterior is obtained for the expected values $\mathbf{f}^*$ at new locations $\mathbf{x}^*$ given the GP prior and the observed values $\mathbf{y}$ containing i.i.d. Gaussian noise with variance $\alpha > 0$ for the training locations $\mathbf{x}$:

$$\begin{aligned} \mathbf{f}^* \mid \mathbf{x}^*, \mathbf{x}, \mathbf{y} &\sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ \text{with } \boldsymbol{\mu} &= \mu(\mathbf{x}^*) + K(\mathbf{x}^*, \mathbf{x}) \cdot \mathbf{K}^{-1} \cdot \mathbf{y} \\ \text{and } \boldsymbol{\Sigma} &= K(\mathbf{x}^*, \mathbf{x}^*) - K(\mathbf{x}^*, \mathbf{x}) \cdot \mathbf{K}^{-1} \cdot K(\mathbf{x}, \mathbf{x}^*) \\ \text{where } \mathbf{K} &= K(\mathbf{x}, \mathbf{x}) + \alpha \cdot I \end{aligned} \tag{6}$$

For training, we optimise the likelihood of the training data

$$\begin{aligned} -\log p(\mathbf{y}|\mathbf{x}) = &\frac{1}{2} \cdot (\mathbf{y} - \boldsymbol{\mu})^\top \cdot \mathbf{K}^{-1} \cdot (\mathbf{y} - \boldsymbol{\mu}) + \\ &\frac{1}{2} \cdot \log(|\det \mathbf{K}|) + \frac{N}{2} \cdot \log(2 \cdot \pi) \end{aligned} \tag{7}$$

using gradient descent with Adam (Kingma and Ba 2017) until convergence on the parameters.

In the previous two equations, one can observe the computational issues associated with GP regression; in particular, the computation of the terms involving the inverse of the kernel matrix and its log determinant is challenging. Traditionally, one computes the Cholesky decomposition $\mathbf{K} = \mathbf{L} \cdot \mathbf{L}^\top$ defined by the lower triangular matrix $\mathbf{L}$. Once $\mathbf{L}$ is obtained, the mentioned computations can be performed efficiently. As mentioned, the computational complexity of obtaining $\mathbf{L}$ is in $\Theta(N^3)$, and we need to store $\mathbf{L}$, which requires $\Theta(N^2)$ memory.

3 Related Work

Usually, one uses the Cholesky decomposition of the kernel matrix for inference and training, as outlined by Rasmussen and Williams (2006). However, this incurs a computational complexity of $\Theta(N^3)$ and a memory requirement of $\Theta(N^2)$, as the decomposition has to be stored in memory. This renders this naïve scheme infeasible for large values of N.

Several approaches have been developed to account for this issue. The ones presented here can be categorised into three groups: (a) improving the inference for exact GPs, (b)

approximating the GP prior and (c) approximating the posterior (Liu et al. 2020).

For (a), Gardner et al. (2018) proposed to use CG for inference. For this, one reformulates $\mathbf{K}^{-1} \cdot (\mathbf{y} - \boldsymbol{\mu})$ to a linear equation system $\mathbf{K} \cdot \boldsymbol{\alpha} = \mathbf{y} - \boldsymbol{\mu}$ and solves it iteratively by minimizing the squared error $||\mathbf{K} \cdot \boldsymbol{\alpha} - (\mathbf{y} - \boldsymbol{\mu})||^2$. Interestingly, one can even estimate $\log|\det \mathbf{K}|$ by adding additional columns to the RHS of the linear equation system. For our purposes, we use the CG inference implemented in GPyTorch (Gardner et al. 2018). To use CG for inference, we must compute the kernel matrix multiplication $h(A) = K(\mathbf{x}, \mathbf{x}') \cdot A$ for an arbitrary matrix $A \in \mathbb{R}^{N \times D}$ in every iteration. Here, D is the number of RHS columns, which is $D = 11 \cdot L$ for L latent models in the LMC with default GPyTorch settings, i.e., the original $\mathbf{y} - \boldsymbol{\mu}$ plus 10 additional columns for the estimation of the log determinant. CG provides us with two important benefits: First, it removes the need for the costly Cholesky decomposition by replacing it with a series of kernel matrix multiplications. Secondly, this allows us to focus on the kernel matrix multiplication operator $h(A)$. One can employ multiple GPUs for higher memory capacity and increased througput when computing $h(A)$ (Wang et al. 2019). However, it is not necessary to store $K(\mathbf{x}, \mathbf{x}')$ to compute $h(A)$. For example, KeOps (Charlier et al. 2021) can be used to implement $h(A)$ using an on-the-fly computation of $K(\mathbf{x}, \mathbf{x}')$, reducing the memory requirement to $\Theta(N)$. Of course, this comes at the cost of re-computing the entire kernel matrix on every CG iteration, demanding an efficient implementation.

For (b), Wilson and Nickisch (2015) proposed to approximate the true kernel matrix by interpolation on a regular grid of N_{ind} points. Thus, all interpolation points must be equidistant. With this setup, the kernel matrix has Toeplitz structure, i.e., its main and off-diagonals are constant. This allows for the computation of the matrix product using the FFT (Gohberg and Olshevsky 1994). Thus, the complexity is reduced to $\Theta(N + N_{\text{ind}} \cdot \log(N_{\text{ind}}))$ and the memory requirement to $\Theta(N + N_{\text{ind}})$. We refer to this strategy as structured kernel interpolation (SKI). Alternatively, one can let the kernel infer a sparsity pattern during training (Noack et al. 2023).

For (c), one can try to approximate the posterior of the GP. This is often achieved with variational inference (Hensman, Matthews, and Ghahramani 2015), which has recently been extended by Wu, Pleiss, and Cunningham (2022) to account for locality in the data. In this variational nearest neighbour Gaussian process (VNNGP) framework, one only considers a small neighbourhood of N_{nbs} points for each point. By exploiting this locality, the complexity can be reduced to $\Theta(N \cdot N_{\text{nbs}}^3)$ while the memory requirement is reduced to $\Theta(N_{\text{nbs}}^2)$ by splitting the training dataset into multiple fixed-size batches. Note that CG-based inference can be combined with variational inference (Pleiss et al. 2020).

4 KernelMatmul: Large Sparse Kernel Matrix Multiplication

KernelMatmul builds on CG (Gardner et al. 2018) and focuses on the optimisation of the kernel matrix multiplication operation $h(A) = K(\mathbf{x}, \mathbf{x}') \cdot A$. To this end, we employ two important tricks: First, we observe that distant values are approximately uncorrelated, and we thus enforce sparsity in the kernel matrix, which leads to a time complexity of $\Theta(N)$. Second, the entries of $K(\mathbf{x}, \mathbf{x}')$ are computed on-the-fly, reducing the memory requirement to $\Theta(N)$.

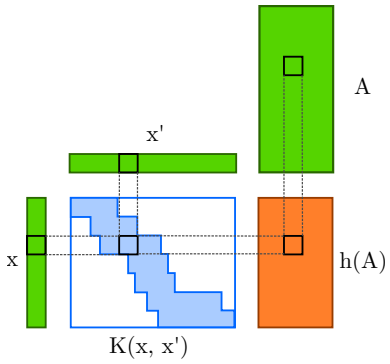

Figure 1: The KernelMatmul multiplication scheme. Shaded areas are dense. The black lines mark the elements participating in a single block of matrix multiplication. $K(\mathbf{x}, \mathbf{x}')$ (blue) is computed on-the-fly and follows the special sparsity structure. The input is marked green, while the output is shown in orange.

To implement the sparsity approximation, we assume $k(\tau) = 0$ if $\tau > c$, where $c > 0$ defines the cutoff distance. As the cutoff c might be hard to specify in general, we use the following observation:

Theorem 1. *For an RBF kernel and a relative error of $\epsilon > 0$, the requirement*

$$\int_{-c}^{c} |k(\tau)| \, \mathrm{d}\tau = (1 - \epsilon) \cdot \int_{-\infty}^{\infty} |k(\tau)| \, \mathrm{d}\tau \tag{8}$$

leads to the solution

$$c = \sqrt{2} \cdot \lambda \cdot \mathrm{erf}^{-1}(1 - \epsilon), \tag{9}$$

where erf^{-1} denotes the inverse error function.

This can be proven by first observing that $|k(\tau)| = k(\tau)$ for an RBF kernel, then rewriting the integrand to the PDF of a normal distribution and applying the known relationship between the CDF of the normal distribution and the error function. A detailed proof is presented in the supplementary material. For a cutoff c defined by Theorem 1, a fraction of ϵ of the total mass of the kernel function is lost. The cutoff scales proportionally with the lengthscale λ and the inverse error function of $1 - \epsilon$.

As the magnitude of the spectral kernel is bounded by the RBF kernel, and an analytical solution could not be retrieved, we use the same equation for the spectral kernel. Thus, the targeted mass (defined by ϵ) might differ from the achieved mass (defined by c) for the spectral kernel. We conducted a numerical study to estimate the resulting error, shown in Figure 2. Each plot shows the relative error expressed by

$$1 - \frac{\text{target mass}}{\text{achieved mass}} = 1 - \frac{(1 - \epsilon) \cdot \int_{-\infty}^{\infty} |k(\tau)| \, \mathrm{d}\tau}{\int_{-c}^{c} |k(\tau)| \, \mathrm{d}\tau}. \tag{10}$$

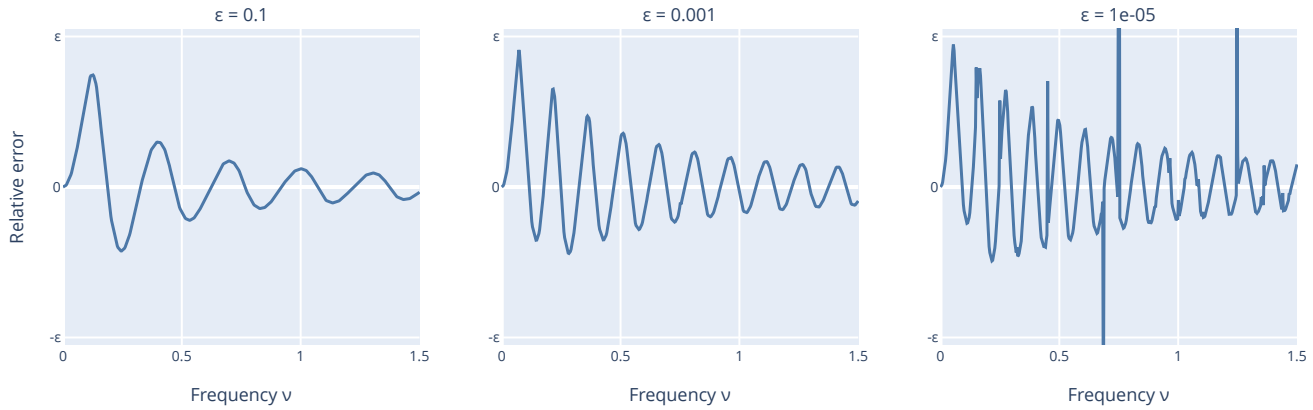

Figure 2: Relative error of the estimate of the approximated kernel function mass, i.e. $1-(1-\epsilon)\cdot\int_{-\infty}^{\infty}|k(\tau)|\,\mathrm{d}\tau/\int_{-c}^{c}|k(\tau)|\,\mathrm{d}\tau$ of a spectral kernel function with the approximation by Theorem 1. All values are computed using integral quadrature. The spikes are caused by numerical integration failure due to round-off errors. Without loss of generality, we set $\lambda = 1$, as this merely corresponds to a scaling of τ and ν.

Thus, this measure is positive if the achieved mass is higher than the target mass, i.e., if our approximation of the kernel function is better than expected. On the more interesting negative side, where we underapproximate the kernel function, we observe a maximum error of $\epsilon/2$ which quickly vanishes with higher frequencies. We note that these considerations also apply for the case of the LMC, as the integral in Theorem 1 can be decomposed into a sum of integrals in this case.

Combined with a block-wise computation and ordered $\mathbf{x}, \mathbf{x}'$, this cutoff leads to the special sparsity structure illustrated in Figure 1. We note that the layout of this sparsity pattern can be computed efficiently in $\Theta(N)$ time by iterating. To do this, we use the following technique: During iteration over the rows of the kernel matrix, we keep track of the start and end points of the previous row. Then, on each iteration, we know that the next start point cannot be before the previous one, and that the next end point cannot be before the previous one, because $\mathbf{x}$ is sorted.

To perform the kernel matrix multiplication under sparsity efficiently, we implemented a custom CUDA kernel. To parallelise the computation of the kernel matrix product, we split the problem into blocks, as shown in Figure 1. As stated previously, we compute the corresponding block of $K(\mathbf{x}, \mathbf{x}')$ on-the-fly to reduce GPU memory usage. Due to the comparatively low throughput of memory operations, we can hide the added computations partially by the use of instruction-level parallelism (ILP) (NVIDIA 2023a, Chapter 4). The blocks are distributed to the processors of the GPU by the NVIDIA driver. Each block iterates along the corresponding rows of the kernel matrix and aggregates the values using multiple threads. For increased parallelisation, the iteration along the rows can be split into multiple subblocks, and the number of threads per block is configurable. As the RHS entries and $\mathbf{x}'$ values corresponding to each block are used multiple times in every step, we cache these values in the shared memory.

The correctness of our KernelMatmul implementation was assured by means of an extensive suite of unit tests. Except for the block size, all other parameters of the kernel, including the number of threads, are auto-tuned on each first call. The block size cannot be autotuned easily because it defines the sparsity pattern which is computed beforehand.

5 Results

We performed a total of three experiments to assess the benefits and limitations of KernelMatmul. First, we did a thorough benchmark for kernel matrix multiplication. Second, we quantified the error introduced by the sparsity approximation in terms of the residual. Finally, we compared KernelMatmul to other GP scaling methods. All experiments were run on an NVIDIA H100 GPU with CUDA 12.1 on Linux. Detailed information on the software versions can be found in the code repository.

5.1 Performance Comparison

At the core of KernelMatmul is its acceleration of kernel matrix multiplication. To assess its effectiveness, we performed an extensive benchmark on many matrix multiplication tasks, for which we used the RBF kernel with varying kernel matrix sizes N, number of RHS columns D and cutoff values c. We benchmark on 50 randomly spaced inputs $\mathbf{x}, \mathbf{x}'$ and compare on the dense case ($c \to \infty$), too. The RHS is fully random as it does not affect performance. With this design, the performance is independent of the lengthscale (as opposed to using ϵ values via Theorem 1), which we set to $\lambda = 1$. As baseline methods, we chose naïve dense matrix multiplication using the industry-standard cuBLAS implementation by NVIDIA, blocked sparse matrix multiplication using cuSPARSE, also developed by NVIDIA, and the previously mentioned on-the-fly approach KeOps (Charlier et al. 2021). For all sparse methods, we optimised the block size for maximum performance using some example configurations beforehand. The results are shown in Figure 3. We perform all experiments without the fast math

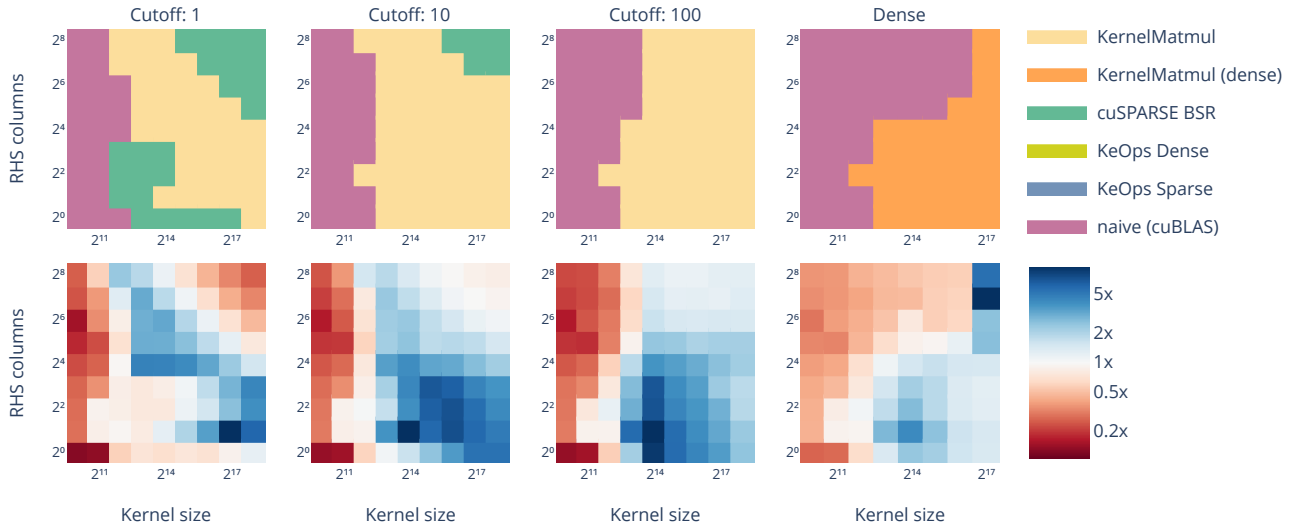

Figure 3: Benchmark of multiple baseline kernel matrix multiplication schemes and KernelMatmul. Columns correspond to increasing cutoff values, where the last value corresponds to a dense matrix without a cutoff. In the top row, each cell indicates the fastest method for this configuration. In the bottom row, each cell indicates the relative speedup of KernelMatmul (dense or sparse) over the best baseline method. For reference, the powers of two shown are $2^0 = 1$, $2^2 = 4$, $2^4 = 16$, $2^6 = 64$, $2^8 = 256$, $2^{11} = 2048$, $2^{14} = 16\,384$ and $2^{17} = 131\,072$. Performance is measured as the median wallclock time of 50 runs on an NVIDIA H100 GPU. All differences are significant according to a Wilcoxon signed-rank test with a false-discovery-rate correction ($\alpha = 0.05$).

compiler setting. With fast math, the accuracy of some functions would be unspecified beyond a certain range (NVIDIA 2023a, Chapter 13.2). The reciprocal of the lengthscale is computed ahead of time for KernelMatmul and KeOps to avoid expensive divisions during the matrix multiplication. For each configuration and algorithm, we collect 50 samples of the wallclock time. All measured differences are significant by a Wilcoxon signed-rank test (Wilcoxon 1992) with $\alpha = 0.05$ and false-discovery-rate correction.

Notably, KernelMatmul outperforms KeOps on all tested configurations. We thus conclude that KernelMatmul is better suited for large kernel matrix multiplication for time series (i.e., single-dimensional input) than KeOps in general. During development, we identified two main reasons for this: For one, KernelMatmul makes use of the shared memory, and second, KernelMatmul decreases the computational load by combining computed kernel matrix entries with multiple RHS entries if applicable. We suspect that KeOps might be limited by its generic approach.

While NVIDIA's cuBLAS implementation, which is inherently dense, is quickly outperformed by KernelMatmul on low cutoff values (i.e., high sparsity), we even outperform cuBLAS matrix multiplication on large dense matrices with few RHS columns. While this might seem counterintuitive at first glance, as cuBLAS does not perform a recomputation of the kernel matrix on every call, this type of matrix multiplication is bound by the speed of the memory transfers (memory-bound). As such, the on-the-fly computation of kernel matrix values trades off memory load time with unused compute time in these cases, leading to faster matrix multiplication. This effect also occurs for KeOps but is less pronounced there, due to the inefficiencies of KeOps. At the largest tested kernel matrix size of $N = 2^{17}$, the naïve approach runs out of memory. Thus, KernelMatmul performs best on these configurations, even for higher numbers of RHS columns.

The sparse complement of cuBLAS, namely cuSPARSE, is a competitor for KernelMatmul on very sparse matrices, i.e., for low cutoff values. In these cases, the highly optimized implementations of cuSPARSE offer a distinct advantage over KernelMatmul. However, even for very sparse matrices, KernelMatmul still outperforms cuSPARSE on certain configurations, e.g. on a moderate number of RHS columns and large kernel sizes. With a medium to high cutoff, KernelMatmul outperforms cuSPARSE on most configurations.

The speedups achieved by KernelMatmul over the baselines can be quite high. While only achieving medium speedups for cases with a high number of RHS columns, KernelMatmul is up to 4 times faster than the baseline methods on configurations with a low to medium number of RHS columns. The highest speedup is achieved in the dense case for very large kernel matrices and a high number of RHS columns. As cuBLAS runs out of memory in those cases, we observe a 9-fold improvement over the next best method, KeOps dense. As an example, this situation occurs in long time series with multiple outputs if an LMC is used.

5.2 Sparsity Approximation Error

In this experiment, we quantified the error introduced by the sparsity approximation. We used $N = 10\,000$ randomly placed samples of $\sin(2 \cdot \pi \cdot \nu \cdot x)$ for $x \in [0, 100]$ with Gaus-

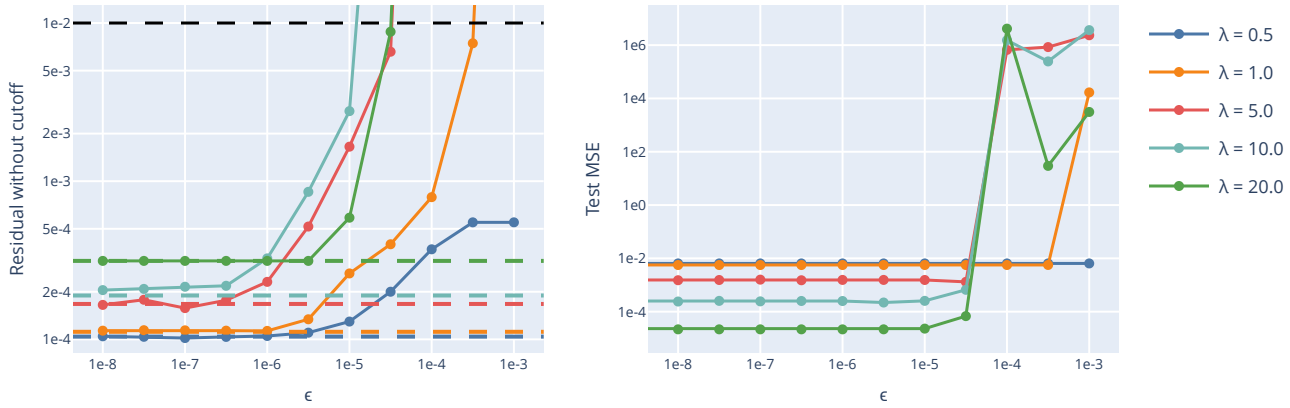

Figure 4: The different coloured lines correspond to varying lengthscales λ of the kernel function. Left: Residual of the solution found with cutoff ϵ under the dense kernel matrix. For each value of λ, the dashed line indicates the residual obtained without sparsity. The black dashed line indicates the CG tolerance of GPyTorch (Gardner et al. 2018) during inference. Right: MSE of predictions on a separate test dataset with GPyTorch default CG tolerance. Note that the variance of $\mathbf{y}$ as defined is 0.5. In both plots, the variance of the measurements over 50 random versions of the data $\mathbf{x}$ and $\mathbf{y}$ is too small for plotting.

sian noise drawn from $\mathcal{N}(0, 0.01)$ as an example dataset. The large value of N and the range for x ensured that small ϵ values, i.e., large cutoff values c, do not immediately lead to a dense kernel matrix. For this dataset, we constructed a perfectly matching spectral kernel (i.e., $\nu = 1$) with $\alpha = 0.01$. Commonly, e.g., in GPyTorch (Gardner et al. 2018), the error of a solution X to $K(\mathbf{x}, \mathbf{x}) \cdot X = \mathbf{y}$ is assessed by its residual $||\mathbf{y} - K(\mathbf{x}, \mathbf{x}) \cdot X||$. Thus, we first computed a solution using the approximated kernel matrix with a sparsity of ϵ and a low CG tolerance, and then calculated the residual under the dense kernel matrix. To additionally account for the influence this residual has on the prediction, we generated another 1000 samples after the training data for testing and report the mean squared error (MSE) in a separate plot. The predictions for this second comparison were generated with GPyTorch default settings, i.e., a CG tolerance of 0.01. The results from both experiments are shown in Figure 4.

After a certain ϵ value, the residual of the solution under sparsity quickly increases. For both sparse and dense solutions, the higher the lengthscale λ, the larger the residual becomes in general. This is likely caused by the increasingly worse conditioning of the kernel matrix under high lengthscales, a phenomenon well understood for RBF kernels (Rasmussen and Williams 2006). Thus, even though the effective cutoff c grows proportionally with λ, the residual is still higher than for a lower lengthscale. Once the residual of the solution exceeds the GPyTorch tolerance, the test MSE quickly deteriorates.

In general, the residuals are comparatively low. For an $\epsilon \leq 10^{-5}$, the sparse solutions are indistinguishable from the dense solution with the default tolerance of GPyTorch, both in terms of their residual and the test MSE. During training, GPyTorch even chooses a tolerance of 1.0. Thus, for sufficiently low ϵ, KernelMatmul can make use of the implied sparsity with a very low approximation error and almost no influence on predictions.

5.3 Accuracy Comparison

This experiment compares KernelMatmul-based CG inference ($\epsilon = 10^{-5}$) to other scaling methods for GPs. It was executed on three datasets from the Monash Forecasting Repository: London Smart Meters, Solar and Traffic (Godahewa et al. 2021). We chose this repository because of its design as a benchmark dataset collection. The datasets were selected based on size (many different series) and length (at least 10000 training samples). Because London Smart Meters contains more than 5000 series, we limited our comparison to the first 1000 series. All datasets contain equidistant samples. We use the splits into training, validation and test data provided by the the Monash Forecasting Repository. We additionally limit the training context to the last 10 000 samples for all series, in order to equalise compute requirements.

We compared KernelMatmul to VNNGP (Wu, Pleiss, and Cunningham 2022) and SKI (Wilson and Nickisch 2015), as these are representative methods of the ideas in Section 3. We chose VNNGP over the more standard SVGP because of its neighbourhood approximation, which seems suitable to time series. SKI used as many grid points as there are samples in the training set, while VNNGP used 64 neighbours per sample. The latter value is limited by compute requirements.

For each dataset, we compiled a list of 11 randomly selected subsets of 20 series each. Then, we performed individual hyperparameter optimizations (HPOs) for each subset and method. Every HPO is performed with SMAC3 (Lindauer et al. 2022) in a multi-instance setting for 4 hours wall time. The best configuration of each HPO is then evaluated on all series in the test dataset by retraining on the training and validation data of that series. In the end, we report the mean absolute scaled error (MASE) on the test dataset in Figure 5. The MASE was proposed by Hyndman and Koehler (2006) for a fair comparison of different forecasting methods. We provide more details on the HPO in the supplementary material.

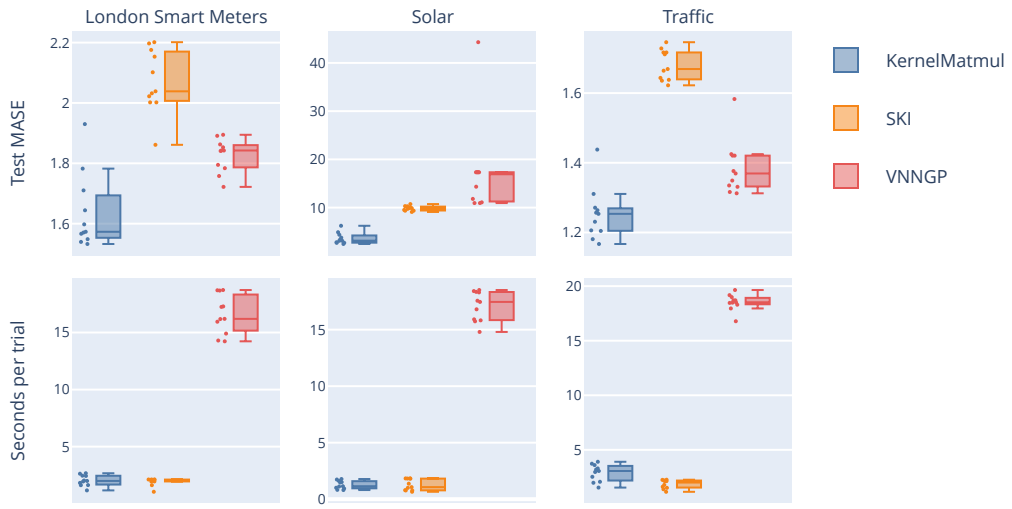

Figure 5: Top: Test MASE on different datasets from the Monash forecasting repository (Godahewa et al. 2021) for KernelMatmul, SKI and VNNGP. For each dataset and method, we perform individual HPOs on eleven different subsets of the validation data. In the end, the models are re-trained with the best configuration on the validation data and evaluated on the test data. Bottom: Average seconds per trial for each of the optimization runs from Figure 5.

On all three datasets, we observed an improvement of KernelMatmul over the other methods. This improvement is significant according to a paired Wilcoxon signed-rank test (Wilcoxon 1992) ($\alpha = 0.05$). We notice that there is no consistent difference in accuracy between SKI and VNNGP. The average time per trial is also shown in Figure 5. While SKI is slightly faster than KernelMatmul on the tested kernel size, we note that SKI has a worse scaling.

6 Conclusion

In this work, we present an implementation of kernel matrix multiplication, dubbed KernelMatmul, that is optimised for very fast GP regression on large time series. We thoroughly benchmarked its performance on a wide variety of configurations. KernelMatmul additionally introduces a cutoff-based sparsity approximation of the kernel matrix. Our mathematical analysis allowed us to choose a cutoff with a known error ϵ. KernelMatmul scales well to low ϵ values with negligible impact on accuracy. In comparison to other approximation methods, we observed improved forecasting accuracy of KernelMatmul. We note that KernelMatmul is especially well-suited for irregular time series. By putting no constraints on the sample distances $x_{i+1} - x_i$, KernelMatmul can natively adapt to arbitrary spacing of the samples. This is in contrast to other approaches like SKI.

We believe that it should be possible to extend KernelMatmul by kernel functions with compact support like those presented by Barber (2020). This would allow for exact inference and maybe a tighter cutoff. Due to its support for irregularly sampled time series, KernelMatmul applies to specialised downsampling methods that go beyond averaging; for example, one might consider sampling more recent and less distant samples for training the GP. More flexible kernel functions might benefit prediction for the large context sizes achievable with KernelMatmul. Over the larger time window that KernelMatmul provides, more complex dynamics must be modelled by the kernel function. For deep Gaussian processes (Jankowiak, Pleiss, and Gardner 2020), it is currently unclear how the sparsity approximation can be deployed. However, kernel functions based on the Fourier series, the sinc-function (Tobar 2019) or the multi-objective spectral mixture kernel (MOSM (Altamirano and Tobar 2022)) might be interesting. Finally, the current preconditioning approach of GPyTorch (Wenger et al. 2022) does not take into account the sparsity introduced by KernelMatmul.

Overall, we are convinced that KernelMatmul can enable new applications of GPs. By using it, more faithful predictions can be obtained, which also include an uncertainty estimate, a feature much needed for many applications.

Acknowledgements

Experiments were performed with computing resources granted by RWTH Aachen University under projects `thes1400` and `supp0006` and on the Kathleen cluster funded by the Alexander von Humboldt Professorship held by HH.

References

Altamirano, M.; and Tobar, F. 2022. Nonstationary multi-output Gaussian processes via harmonizable spectral mixtures. In *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, 3204–3218. PMLR. ISSN: 2640-3498.

Barber, J. 2020. Sparse Gaussian Processes via Parametric Families of Compactly-supported Kernels. ArXiv:2006.03673 [cs, stat].

Bonilla, E. V.; Chai, K.; and Williams, C. 2007. Multi-task Gaussian Process Prediction. In *Advances in Neural In-*

formation Processing Systems, volume 20, 153–160. Curran Associates, Inc.

Chandola, V.; and Vatsavai, R. 2010. Scalable time series change detection for biomass monitoring using gaussian process.

Chandola, V.; and Vatsavai, R. R. 2011. A Gaussian Process Based Online Change Detection Algorithm for Monitoring Periodic Time Series. In *Proceedings of the 2011 SIAM International Conference on Data Mining (SDM)*, Proceedings, 95–106. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-992-5.

Charlier, B.; Feydy, J.; Glaunès, J. A.; Collin, F.-D.; and Durif, G. 2021. Kernel Operations on the GPU, with Autodiff, without Memory Overflows. *Journal of Machine Learning Research*, 22(74): 1–6.

de Wolff, T.; Cuevas, A.; and Tobar, F. 2021. MOGPTK: The multi-output Gaussian process toolkit. *Neurocomputing*, 424: 49–53.

Dürichen, R.; Pimentel, M. A. F.; Clifton, L.; Schweikard, A.; and Clifton, D. A. 2015. Multitask Gaussian Processes for Multivariate Physiological Time-Series Analysis. *IEEE Transactions on Biomedical Engineering*, 62(1): 314–322. Conference Name: IEEE Transactions on Biomedical Engineering.

Gardner, J.; Pleiss, G.; Weinberger, K. Q.; Bindel, D.; and Wilson, A. G. 2018. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In *Advances in Neural Information Processing Systems*, volume 31, 7576–7586. Curran Associates, Inc.

Godahewa, R. W.; Bergmeir, C.; Webb, G.; Hyndman, R.; and Montero-Manso, P. 2021. Monash Time Series Forecasting Archive. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 1.

Gohberg, I.; and Olshevsky, V. 1994. Fast Algorithms with Preprocessing for Matrix-Vector Multiplication Problems. *Journal of Complexity*, 10(4): 411–427.

Hensman, J.; Matthews, A.; and Ghahramani, Z. 2015. Scalable Variational Gaussian Process Classification. In *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics*, 351–360. PMLR. ISSN: 1938-7228.

Hyndman, R. J.; and Koehler, A. B. 2006. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4): 679–688.

Jankowiak, M.; Pleiss, G.; and Gardner, J. R. 2020. Deep Sigma Point Processes. ArXiv:2002.09112 [cs, stat].

Kingma, D. P.; and Ba, J. 2017. Adam: A Method for Stochastic Optimization. ArXiv:1412.6980 [cs].

Lindauer, M.; Eggensperger, K.; Feurer, M.; Biedenkapp, A.; Deng, D.; Benjamins, C.; Ruhkopf, T.; Sass, R.; and Hutter, F. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research*, 23(54): 1–9.

Liu, H.; Ong, Y.-S.; Shen, X.; and Cai, J. 2020. When Gaussian Process Meets Big Data: A Review of Scalable GPs. *IEEE Transactions on Neural Networks and Learning Systems*, 31(11): 4405–4423. Conference Name: IEEE Transactions on Neural Networks and Learning Systems.

Noack, M. M.; Krishnan, H.; Risser, M. D.; and Reyes, K. G. 2023. Exact Gaussian processes for massive datasets via non-stationary sparsity-discovering kernels. *Scientific Reports*, 13(1): 3155. Publisher: Nature Publishing Group.

NVIDIA. 2023a. CUDA C++ Programming Guide v12.1.1.

NVIDIA. 2023b. CUDA v12.1.1.

Pleiss, G.; Jankowiak, M.; Eriksson, D.; Damle, A.; and Gardner, J. 2020. Fast Matrix Square Roots with Applications to Gaussian Processes and Bayesian Optimization. In *Advances in Neural Information Processing Systems*, volume 33, 22268–22281. Curran Associates, Inc.

Press, W. H.; and Rybicki, G. B. 1989. Fast algorithm for spectral analysis of unevenly sampled data. *Astrophysical Journal, Part 1*, 338: 277–280.

Rasmussen, C. E.; and Williams, C. K. I. 2006. *Gaussian Processes for Machine Learning*. The MIT Press. ISBN 0-262-18253-X.

Tobar, F. 2019. Band-Limited Gaussian Processes: The Sinc Kernel. In *Advances in Neural Information Processing Systems*, volume 32, 12749–12759. Curran Associates, Inc.

Wang, K.; Pleiss, G.; Gardner, J.; Tyree, S.; Weinberger, K. Q.; and Wilson, A. G. 2019. Exact Gaussian processes on a million data points. *Advances in neural information processing systems*, 32.

Wenger, J.; Pleiss, G.; Hennig, P.; Cunningham, J.; and Gardner, J. 2022. Preconditioning for Scalable Gaussian Process Hyperparameter Optimization. In *Proceedings of the 39th International Conference on Machine Learning*, 23751–23780. PMLR. ISSN: 2640-3498.

Wilcoxon, F. 1992. Individual Comparisons by Ranking Methods. In Kotz, S.; and Johnson, N. L., eds., *Breakthroughs in Statistics: Methodology and Distribution*, Springer Series in Statistics, 196–202. New York, NY: Springer. ISBN 978-1-4612-4380-9.

Wilson, A.; and Adams, R. 2013. Gaussian Process Kernels for Pattern Discovery and Extrapolation. In *Proceedings of the 30th International Conference on Machine Learning*, 1067–1075. PMLR. ISSN: 1938-7228.

Wilson, A.; and Nickisch, H. 2015. Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP). In *Proceedings of the 32nd International Conference on Machine Learning*, 1775–1784. PMLR. ISSN: 1938-7228.

Wu, L.; Pleiss, G.; and Cunningham, J. P. 2022. Variational nearest neighbor Gaussian process. In *Proceedings of the 39th International Conference on Machine Learning*, 24114–24130. PMLR. ISSN: 2640-3498.