Exact Gaussian Processes on a Million Data Points

Ke Alexander Wang[1*] **Geoff Pleiss**[1*] **Jacob R. Gardner**[2]
Stephen Tyree[3] **Kilian Q. Weinberger**[1] **Andrew Gordon Wilson**[1]
[1]Cornell University, [2]Uber AI Labs, [3]NVIDIA

Abstract

Gaussian processes (GPs) are flexible models with state-of-the-art performance on many impactful applications. However, computational constraints with standard inference procedures have limited exact GPs to problems with fewer than about ten thousand training points, necessitating approximations for larger datasets. In this paper, we develop a scalable approach for exact GPs that leverages multi-GPU parallelization and methods like linear conjugate gradients, accessing the kernel matrix only through matrix multiplication. By partitioning and distributing kernel matrix multiplies, we demonstrate that an exact GP can be trained on over a million points in 3 days using 8 GPUs and can compute predictive means and variances in under a second using 1 GPU at test time. Moreover, we perform the first-ever comparison of exact GPs against state-of-the-art scalable approximations on large-scale regression datasets with $10^4 - 10^6$ data points, showing dramatic performance improvements.

1 INTRODUCTION

Gaussian processes (GPs) have seen great success in many machine learning settings, such as black-box optimization (Snoek et al., 2012), reinforcement learning (Deisenroth and Rasmussen, 2011; Deisenroth et al., 2015), and time-series forecasting (Roberts et al., 2013). These models offer several advantages – principled uncertainty representations, model priors that require little expert intervention, and the ability to adapt to any dataset size (Rasmussen and Ghahramani, 2001; Rasmussen and Williams, 2006). GPs are not only ideal for problems with few observations; they also have great promise to exploit the available information in increasingly large datasets, especially when combined with expressive kernels (Wilson and Adams, 2013) or hierarchical structure (Wilson et al., 2012; Damianou and Lawrence, 2013; Wilson et al., 2016a; Salimbeni and Deisenroth, 2017).

In practice, however, exact GP inference can be intractable for large datasets, as it naïvely requires $\mathcal{O}(n^3)$ computations and $\mathcal{O}(n^2)$ storage for n training points (Rasmussen and Williams, 2006). Many approximate methods have been introduced to improve scalability, relying on mixture-of-experts models (Deisenroth and Ng, 2015), inducing points (Snelson and Ghahramani, 2006; Titsias, 2009; Wilson and Nickisch, 2015; Gardner et al., 2018b), random feature expansions (Rahimi and Recht, 2008; Le et al., 2013; Yang et al., 2015), or stochastic variational optimization (Hensman et al., 2013, 2015; Wilson et al., 2016b; Cheng and Boots, 2017; Salimbeni et al., 2018). However, choosing a suitable scalable approach involves many design choices, such as numbers of random features, types of features, and numbers and locations of inducing points. Performance is often sensitive to these choices and depends on the properties of a given dataset. Due to the historical intractablity of training exact GPs on large datasets, it is an open question how approximate methods compare to an exact approach when much more data is available.

In this paper, we develop a methodology to scale exact GP inference well beyond what has previously been achieved: we train an exact Gaussian process *on over a million data points without approximations*. Such a result would be intractable with standard implementations, which rely on the Cholesky decomposition. The scalability we demonstrate is made feasible through the recent Blackbox Matrix-Matrix multiplication (BBMM) inference procedure of Gardner et al. (2018a), which uses conjugate gradients and related methods to reduce GP inference to iterations of matrix multiplication. This drops the time complexity of GPs from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$.

[*] Equal contribution

However, in its original form BBMM with exact GPs could still only be applied to modestly sized datasets (i.e., $n < 10{,}000$), due to $\mathcal{O}(n^2)$ memory constraints. By partitioning and distributing kernel matrix multiplications, we demonstrate that the memory requirement for BBMM at train time can be reduced to $\mathcal{O}(n)$. Additionally, we introduce a number of practical heuristics to accelerate training and maximally utilize GPUs.

We perform the first ever comparison of exact GPs to popular approximate GP methods on datasets with $n \gg 10{,}000$. Exact GPs on these datasets offer notably better performance, often exceeding a two-fold gain in accuracy. With 8 GPUs, exact GPs can be trained in seconds for $n \approx 10^4$, hours for $n \approx 10^5$, and three days for $n \approx 10^6$. However, all trained models can *make predictions in less than 1 second on 1 GPU* due to a simple caching strategy. These results demonstrate that exact GPs can be scaled to much larger datasets than previously thought possible, with highly compelling performance and a straightforward training procedure that does not require expert knowledge of GPs.

2 BACKGROUND

Gaussian processes (GPs) are non-parametric machine leaning models that place a distribution over functions $f \sim \mathcal{GP}$. The function distribution is defined by a *prior mean function* $\mu : \mathbb{R}^d \to \mathbb{R}$, a *prior covariance function* or *kernel* $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, and observed (training) data $(X, \mathbf{y})$. The choice of μ and k encode prior information about the data. μ is typically chosen to be a constant function. Popular kernels include the RBF kernel and the Matérn kernels (Rasmussen and Williams, 2006).

Notation: Throughout this paper we will use the following notation: given training inputs $X \in \mathbb{R}^{n \times d}$, K_{XX} is the $n \times n$ kernel matrix containing covariance terms for all pairs of entries. The vector $\mathbf{k}_{X\mathbf{x}^*}$ is a vector formed by evaluating the kernel between a test point $\mathbf{x}^*$ and all training points. $\widehat{K}_{XX}$ is a kernel matrix with added Gaussian observational noise (i.e. $\widehat{K}_{XX} = K_{XX} + \sigma^2 I$).

Training: Most kernels include hyperparameters θ, such as the lengthscale, which must be fit to the training data. In regression, θ are typically learned by maximizing the GP's *log marginal likelihood* with gradient descent:

$$\mathcal{L} = \log p(\mathbf{y} \,|\, X, \theta) \propto -\mathbf{y}^\top \widehat{K}_{XX}\mathbf{y} - \log |\widehat{K}_{XX}|, \quad (1)$$

$$\frac{\partial \mathcal{L}}{\partial \theta} \propto \mathbf{y}^\top \widehat{K}_{XX} \frac{\partial \widehat{K}_{XX}}{\partial \theta} \widehat{K}_{XX}\mathbf{y} - \operatorname{tr}\left\{ \widehat{K}_{XX}^{-1} \frac{\partial \widehat{K}_{XX}}{\partial \theta} \right\}. \quad (2)$$

A typical GP has very few hyperparameters to optimize and therefore requires fewer iterations of training than most parametric models.

Predictions: For a test point $\mathbf{x}^*$, the GP predictive posterior distribution $p(f(\mathbf{x}^*) \mid X, \mathbf{y})$ with a Gaussian likelihood is Gaussian with moments:

$$\mathbb{E}\left[f(\mathbf{x}^*) \,|\, X, \mathbf{y}\right] = \mu(\mathbf{x}^*) + \mathbf{k}_{X\mathbf{x}^*}^\top \widehat{K}_{XX}^{-1}\mathbf{y} \quad (3)$$

$$\operatorname{Var}\left[f(\mathbf{x}^*) \,|\, X, \mathbf{y}\right] = k(\mathbf{x}^*, \mathbf{x}^*) - \mathbf{k}_{X\mathbf{x}^*}^\top \widehat{K}_{XX}^{-1} \mathbf{k}_{X\mathbf{x}^*} \quad (4)$$

Portions of these equations can be precomputed as part of training to reduce the test-time computation. In particular, (3) is reduced to an $\mathcal{O}(n)$ matrix-vector multiplication once $\widehat{K}_{XX}^{-1}\mathbf{y}$ is computed and cached. Similar caching techniques can reduce the asymptotic time complexity of (4) as well (Pleiss et al., 2018).

The Cholesky decomposition is used in many GP implementations to compute $\widehat{K}_{XX}^{-1}\mathbf{y}$, $\log |\widehat{K}_{XX}|$, and $\operatorname{tr}\left\{ \widehat{K}_{XX}^{-1} \left(\partial \widehat{K}_{XX} / \partial \theta \right) \right\}$ in (1) and (2). The positive definite kernel matrix $\widehat{K}_{XX}$ can be factorized into $LL^\top$, where L is lower triangular. Computing L requires $\mathcal{O}(n^3)$ time. After computing this factorization, matrix solves and log determinants take $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$ time respectively. The columns of $L = \begin{bmatrix} \mathbf{l}^{(1)} & \dots & \mathbf{l}^{(k)} \end{bmatrix}$ are computed recursively (Golub and Van Loan, 2012):

$$\mathbf{l}^{(i)} = \begin{bmatrix} \mathbf{0} \\ d_i \\ \frac{1}{d_i}\left(\mathbf{k}^{(i)}_{(i+1):n} - \sum_{j=i}^{i-1} l_i^{(j)} \mathbf{l}^{(j)\top}_{(i+1):n} \right) \end{bmatrix} \quad (5)$$

where $\mathbf{k}^{(i)}$ is the i^{th} column of the matrix $\widehat{K}_{XX}$, and $d_i = \sqrt{k_i^{(i)} - \sum_{j=1}^{i-1} l_i^{(j)}}$. Because of this recursive nature, the Cholesky algorithm is not easily parallelized and makes limited use of GPU acceleration.

Conjugate gradients (CG) is an alternative method for computing $\widehat{K}_{XX}^{-1}\mathbf{y}$. CG frames $\widehat{K}_{XX}^{-1}\mathbf{y}$ as the solution to an optimization problem: $\mathbf{v}^* = \arg\min_{\mathbf{v}} \frac{1}{2}\mathbf{v}^\top \widehat{K}_{XX}\mathbf{v} - \mathbf{v}^\top \mathbf{y}$, which is convex by the positive-definiteness of $\widehat{K}_{XX}$. The optimization is performed through iterations, each of which requires a matrix-vector multiplication with $\widehat{K}_{XX}$. For a specified tolerance ϵ of the relative residual norm $\|\widehat{K}_{XX}\mathbf{v}^* - \mathbf{y}\| / \|\mathbf{y}\|$, the solution can be found in t_ϵ iterations. The exact number of iterations depends on the conditioning and eigenvalue distribution of $\widehat{K}_{XX}$, but $t_\epsilon \ll n$ for reasonable values of ϵ. A *preconditioner* is commonly used to accelerate convergence (Golub and Van Loan, 2012). In this paper, we refer to preconditioned CG as PCG.

Blackbox Matrix-Matrix (BBMM) GP inference. While PCG has been proposed in some papers to compute $\widehat{K}_{XX}^{-1}\mathbf{y}$, additional methods were required to compute $\log |\widehat{K}_{XX}|$ and $\operatorname{tr}\left\{ \widehat{K}_{XX}^{-1} \left(\partial \widehat{K}_{XX} / \partial \theta \right) \right\}$, the other terms in (1) and (2). Gardner et al. (2018a) demonstrate that a slightly-modified PCG algorithm can estimate

these two terms. The modified PCG computes a constant number of additional solves $\widehat{K}_{XX}^{-1}\mathbf{z}_1, \ldots, \widehat{K}_{XX}^{-1}\mathbf{z}_s$ and stores coefficients that estimate the trace term and the log determinant term. (See Gardner et al. (2018a) for more details.) Additionally, Gardner et al. recommend using a batched version of PCG to compute the solves $\left[\widehat{K}_{XX}^{-1}\mathbf{y} \quad \widehat{K}_{XX}^{-1}\mathbf{z}_1 \quad \ldots \quad \widehat{K}_{XX}^{-1}\mathbf{z}_s\right]$ in parallel. Each iteration of this batched-solver only requires multiplying $\widehat{K}_{XX}$ by a small matrix of right-hand-side vectors.

Computing (1) and (2) with batched-PCG, which the authors refer to as *BlackBox Matrix-Matrix* (BBMM) inference, offers several advantages over standard Cholesky-based methods. In particular, BBMM reduces the asymptotic complexity from $\mathcal{O}(n^3)$ down to $\mathcal{O}(t_\epsilon n^2)$, where $t_\epsilon \ll n$ is the number of CG iterations required for convergence to within tolerance ϵ. Additionally, the algorithm mostly uses matrix multiplication, which is one of the most parallelizable BLAS routines. BBMM therefore utilizes GPU acceleration very effectively, making it faster than Cholesky methods on datasets of all sizes (Gardner et al., 2018a).

3 METHOD

To perform exact Gaussian process inference on large datasets, we must overcome the time and space requirements of solving linear systems. Most GP implementations use the Cholesky decomposition to solve linear systems required for inference (Rasmussen and Williams, 2006). Although such solvers are heavily optimized, they cannot be easily adapted to distributed hardware because of the Cholesky decomposition's recursive implementation. The $\mathcal{O}(n^3)$ time complexity of the decomposition makes it difficult to perform exact GP inference on datasets with $n > 10^4$ data points. In addition to this limitation, the Cholesky decomposition requires $\mathcal{O}(n^2)$ memory to store the lower-triangular factor L in addition to the kernel matrix itself. At $n = 500{,}000$, the decomposition requires a full terabyte of memory and a prohibitively large amount of computational resources.

We address the above challenges by taking a matrix-multiplication approach to GP inference. We follow the method in Gardner et al. (2018a) to reduce the time complexity to $\mathcal{O}(n^2)$ by using preconditioned conjugate gradients to solve linear systems. We overcome the memory limitations by partitioning the kernel matrix to perform all matrix-vector multiplications (MVMs) without ever forming the kernel matrix explicitly, reducing the memory requirement to $\mathcal{O}(n)$. In addition, we parallelize partitioned MVMs across multiple GPUs to further accelerate the computations, making training possible and timely even for datasets with $n > 10^6$.

Algorithm 1: Preconditioned Conjugate Gradients (PCG)

Output: $\widehat{K}_{XX}^{-1}\mathbf{b}$

Input : $\texttt{mvm\_}\widehat{K}_{XX}\texttt{()}$ – matrix multiplication function $\widehat{K}_{XX}(\cdot)$
$\mathbf{b} \in \mathbb{R}^n$ – the vector to solve against
$P^{-1}\texttt{()}$ – a preconditioner function
ϵ – error tolerance

$\mathbf{u}_0 \leftarrow \mathbf{0}$ // Current solution
$\mathbf{r}_0 \leftarrow \texttt{mvm\_}\widehat{K}_{XX}\texttt{(}\mathbf{u}_0\texttt{)}$ - $\mathbf{b}$ // Current error
$\mathbf{z}_0 \leftarrow P^{-1}\texttt{(}\mathbf{r}_0\texttt{)}$ // Preconditioned error
$\mathbf{p}_0 \leftarrow \mathbf{z}_0$ // Next ``search'' direction

for $j \leftarrow 1$ **to** T **do**
  $\mathbf{v}_j \leftarrow \texttt{mvm\_}\widehat{K}_{XX}\texttt{(}\mathbf{p}_{j-1}\texttt{)}$
  $\alpha_j \leftarrow (\mathbf{r}_{j-1}^\top \mathbf{z}_{j-1})/(\mathbf{p}_{j-1}^\top \mathbf{v}_j)$
  $\mathbf{u}_j \leftarrow \mathbf{u}_{j-1} + \alpha_j \mathbf{p}_{j-1}$
  $\mathbf{r}_j \leftarrow \mathbf{r}_{j-1} - \alpha_j \mathbf{v}_j$
  if $\|\mathbf{r}_j\|_2 < \epsilon$ **then return** $\mathbf{u}_j$;
  $\mathbf{z}_j \leftarrow P^{-1}\texttt{(}\mathbf{r}_j\texttt{)}$
  $\beta_j \leftarrow (\mathbf{z}_j^\top \mathbf{z}_j)/(\mathbf{z}_{j-1}^\top \mathbf{z}_{j-1})$
  $\mathbf{p}_j \leftarrow \mathbf{z}_j - \beta_j \mathbf{p}_{j-1}$
end

return $\mathbf{u}_j$

$\mathcal{O}(n)$ memory MVM-based inference. Algorithm 1 is the standard implementation of the preconditioned conjugate gradients (PCG) algorithm. The primary input to PCG is $\texttt{mvm\_}\widehat{K}_{XX}$, a routine for performing MVMs using the kernel matrix $\widehat{K}_{XX}$.

Besides the storage cost associated with $\texttt{mvm\_}\widehat{K}_{XX}$, each iteration of PCG updates four vectors: $\mathbf{u}$ (the current solution), $\mathbf{r}$ (the current error), $\mathbf{p}$ (the "search" direction for the next solution), and $\mathbf{z}$ (a preconditioned error term). Storing these vectors requires exactly $4n$ space. In practice, we use a modified version of PCG (BBMM) which stores additional terms for estimating the log determinant, as proposed by Gardner et al. (2018a). These terms require only a small amount of additional $\mathcal{O}(n)$ storage. The quadratic space cost associated with PCG-based GP inference only comes from computing $\texttt{mvm\_}\widehat{K}_{XX}$.

Typically in the full GP setting, $\texttt{mvm\_}\widehat{K}_{XX}$ is implemented by first computing the full $n \times n$ kernel matrix $\widehat{K}_{XX}$, then computing the matrix-vector product with the full matrix. However, this would have the same $\mathcal{O}(n^2)$ memory requirement as Cholesky-based GP inference. Although forming $\widehat{K}_{XX}$ requires $\mathcal{O}(n^2)$ memory, the result of the MVM $\widehat{K}_{XX}\mathbf{v}$ requires only $\mathcal{O}(n)$ memory. Therefore, we reduce the memory requirement to $\mathcal{O}(n)$ by computing $\widehat{K}_{XX}\mathbf{v}$ in separate constant-sized pieces.

Partitioned kernel MVMs. To compute $\widehat{K}_{XX}\mathbf{v}$ in pieces, we partition the kernel matrix $\widehat{K}_{XX}$ such that *we only store a constant number of rows at any given time*. With the $4n$ memory requirement of storing the PCG vectors, our approach requires only $\mathcal{O}(n)$ memory.

We first partition the data matrix with n points in d dimensions, $X \in \mathbb{R}^{n \times d}$, into p partitions each of which contains roughly n/p data points:

$$X = \begin{bmatrix} X^{(1)}; & \cdots & ; X^{(p)} \end{bmatrix}$$

where we use ";" to denote row-wise concatenation. For each $X^{(l)}$, we can compute $\widehat{K}_{X^{(l)}X}$, which is a roughly $(n/p) \times n$ kernel matrix between the partition $X^{(l)}$ and the full data X. By partitioning the kernel matrix this way, we rewrite it as a concatenation of the p partitions:

$$\widehat{K}_{XX} = \begin{bmatrix} \widehat{K}_{X^{(1)}X}; & \cdots & ; \widehat{K}_{X^{(p)}X} \end{bmatrix}$$

Computing each partition requires access to the full training set X, which we assume fits in memory. However, each partition $\widehat{K}_{X^{(l)}X}$ contains only $1/p$ of the entries of the full kernel matrix. Rewriting the matrix-vector product $\widehat{K}_{XX}\mathbf{v}$ in terms of these partitions,

$$\widehat{K}_{XX}\mathbf{v} = \begin{bmatrix} \widehat{K}_{X^{(1)}X}\mathbf{v}; & \cdots & ; \widehat{K}_{X^{(p)}X}\mathbf{v} \end{bmatrix},$$

we see that this matrix-vector product can be computed in smaller components by separately computing each $\widehat{K}_{X^{(l)}X}\mathbf{v}$. We discard each kernel partition $\widehat{K}_{X^{(l)}X}$ once its vector product has been computed. When training the GP, we keep track of how we partitioned X and $\mathbf{v}$ in order to compute gradients, a procedure often referred to as checkpointing (Griewank and Walther, 2000). This leads to the implementation of $\texttt{mvm\_}\widehat{K}_{XX}$ for computing $\widehat{K}_{XX}\mathbf{v}$ described in Algorithm 2.

Algorithm 2: $\texttt{mvm\_}\widehat{K}_{XX}$ by partitioning

Output: $\widehat{K}_{XX}\mathbf{v}$
Input : $k(\cdot,\cdot)$ – covariance function defining $\widehat{K}_{XX}$
$X \in \mathbb{R}^{n \times d}$ – training data
$\mathbf{v} \in \mathbb{R}^n$ – vector to multiply
p – number of partitions

$\mathbf{z} \leftarrow \mathbf{0}$
for $l \leftarrow 1$ **to** p **do**
  Form $X^{(l)}$ from X
  Compute $\widehat{K}_{X^{(l)}X}$ using k, $X^{(l)}$, and X
  $\mathbf{z}^{(l)} \leftarrow K_{X^{(l)}X}\mathbf{v}$
end
return $\begin{bmatrix} \mathbf{z}^{(1)}; & \ldots & ; \mathbf{z}^{(p)} \end{bmatrix}$

Algorithm 2 requires access to the training data X and vector $\mathbf{v}$ already in memory and only allocates new memory to temporarily store the output vector $\mathbf{z}$ and a $(n/p) \times n$ kernel matrix partition $\widehat{K}_{X^{(l)}X}$. This algorithm allows us to reduce memory usage in exchange for sequential but easily parallizable computations. If $p = 1$ then we have the naïve $\mathcal{O}(n^2)$ memory MVM procedure. As $p \to n$, PCG will only require $O(n)$ memory. In practice, we set a constant number of rows per partition according to the amount of memory available rather the number of partitions p. By keeping a partition in memory only until its component of the MVM has been computed, we can use Algorithm 2 to train GPs with an $\mathcal{O}(n)$ memory requirement. *We can therefore perform exact GP inference as long as the dataset can fit in memory*. In the results section, we train exact GP models on a dataset with over a million data points.[1]

3.1 PRACTICAL CONSIDERATIONS

Distributed MVMs in Parallel. MVM-based inference can easily take advantage of multiple GPUs or distributed computational resources because each MVM $\widehat{K}_{X^{(l)}X}\mathbf{v}$ can be performed on a different device. Thus we can compute multiple such MVMs in parallel to attain wall-clock speedups proportional to the number of devices available on large data sets where computation time exceeds the distributed computing overhead. Although $O(n)$ memory is achievable by setting $p = \mathcal{O}(n)$, in practice one may prefer $\mathcal{O}\left(n^2/p\right)$ memory to more effectively accelerate MVMs on parallel hardware with the necessary memory resources.

Additionally, we note that distributed parallel MVMs with Algorithm 2 require $O(n)$ less communication between devices than distributed Cholesky decomposition. The input data matrix X does not change over the course of training, so it only has to be distributed to each device once. Moreover, each call to Algorithm 2 only has to supply each device with a new right-hand-side vector $\mathbf{v}$ and any updates to the kernel hyperparameters. Finally, if w devices are used, the output from each device will be a vector of length n/pw. Thus after the initial transfer of X to all devices, only $\mathcal{O}(n)$ data are copied to or from the devices. In contrast, distributing the Cholesky decomposition across multiple devices would require $\mathcal{O}(n^2)$ communication (Gustavson et al., 2006).

Although distributed computations have been utilized for approximate GP inference through mixture-of-experts GP models (Deisenroth and Ng, 2015), to the best of our knowledge, our method is the first to use distributed computation for *exact* Gaussian process inference.

[1] Cutajar et al. (2016) note the possibility of kernel partitioning with MVM-based inference. However, our approach, which builds on the BBMM framework (Gardner et al., 2018a), is the first to use partitioning to scale exact GPs to $n > 10^6$.

Predictions. At inference time, we must compute the predictive mean and variance given in (3) and (4). Although the predictive mean contains a linear solve $\widehat{K}_{XX}^{-1}\mathbf{y}$, this solve depends only on the training data. The result of this solve can be stored in a linear vector $\mathbf{a}$ and used for subsequent predictions. Therefore, computing the predictive mean requires computing $\widehat{K}_{\mathbf{x}^*X}\mathbf{a}$. Because this equation involves no linear solves, it can be computed efficiently on a single GPU.

Similarly, a training data dependent cache can be computed for the predictive variance using the method of Pleiss et al. (2018) with a satisfactorily tight convergence tolerance. On exact GPs, this approach affords $\mathcal{O}(n)$ predictive variance computations,[2] removing the need to perform a linear solve at test time. In practice, we observe that both the predictive mean and variance can be computed in less than a second on a single GPU, even if the full model required days to train on multiple GPUs. Because predictions are fast after these precomputations, we can afford to use more stringent criteria for CG convergence for these one-time precomputations.

Preconditioning. To accelerate the convergence of CG, Gardner et al. (2018a) introduced a preconditioner for $\widehat{K}_{XX}$ derived from its *partial pivoted Cholesky* decomposition. Preconditioning works by modifying CG to solve the related linear system $P^{-1}K_{XX}\mathbf{v} = P^{-1}\mathbf{y}$ instead of solving the original system $K_{XX}\mathbf{v} = \mathbf{y}$. These linear systems have the same solution $\mathbf{v}^*$. However, the number of CG iterations required depends on the eigenvalue distribution of $P^{-1}K_{XX}$ rather than that of K_{XX}.

As discussed in Gardner et al. (2018a), computing a rank k preconditioner requires only k kernel matrix rows: an already $\mathcal{O}(n)$ space dependence. While each iteration of CG requires computing each kernel matrix partition from scratch, the preconditioner is computed once before any iterations of CG are performed. Therefore, it can be efficient to increase the size of the preconditioner to an extent if it reduces the number of CG iterations. In particular, as the number of checkpoint partitions increases, it may be more efficient to compute a larger preconditioner, thereby reducing the iterations of CG. While in Gardner et al. (2018a) the preconditioner size is typically limited to under 20 by default, in our use case we found that preconditioners of up to size $k = 100$ provide a noticeable improvement to wall-clock speed for large datasets.

PCG Convergence Criteria. At test time, an accurate solve $\widehat{K}_{XX}^{-1}\mathbf{y}$ is critical for good predictive performance. We find that a CG tolerance of $\epsilon \leq 0.01$ is necessary for good predictions. At training time, however, we find that it is possible to use a less strict convergence criterion. Learning the hyperparameters with a stochastic optimizer like Adam (Kingma and Ba, 2015) is robust against noise in the gradients, so we use a much looser convergence criterion of $\epsilon = 1$. Each computation of (1) and (2) requires no more than 100 CG iterations to converge with this criterion, which results in faster training.

[2] We do not achieve constant-time predictions as described in (Pleiss et al., 2018). Reducing the $\mathcal{O}(n)$ prediction time to $\mathcal{O}(1)$ requires using structure kernel interpolation (Wilson and Nickisch, 2015) to approximate the kernel matrix.

4 RELATED WORK

MVM-based GP inference. Conjugate gradients and related MVM-based algorithms (Ubaru et al., 2017; Dong et al., 2017) have been used in certain settings throughout the GP literature. However, these methods have typically been used when the kernel matrix is structured and affords fast matrix-vector multiplies. Cunningham et al. (2008) note that CG reduces asymptotic complexity when the data lie on a regularly-spaced grid because the kernel matrix is structured and affords $\mathcal{O}(n \log n)$ MVMs. This idea was extended to multi-dimensional grids by Saatçi (2012). Wilson and Nickisch (2015) introduce a general-purpose GP approximation specifically designed for CG. They combine a structured inducing point matrix with sparse interpolation for approximate kernel matrices with nearly-linear MVMs.

More recently, there has been a push to use MVM-based methods on exact GPs. Cutajar et al. (2016) use conjugate gradients to train exact GPs on datasets with up to 50,000 points. The authors investigate using off-the-shelf preconditioners and develop new ones based on inducing-point kernel approximations. However, the size of the preconditioners tested by the authors scale as $\mathcal{O}(n^{5/3})$, which limits the scalability of the model. In contrast, our preconditioner is $\mathcal{O}(n)$. Gardner et al. (2018a) also note that a modified batched version of PCG is faster on the GPU than GP inference with Cholesky. As mentioned in Section 3, we extend their method to reduce the memory requirement of exact GPs and take advantage of distributed computational resources.

Approximate GP methods. There are several approximations to GP inference that require $\leq O(n^2)$ memory and scale to large datasets. Perhaps the most common class of approaches are *inducing point methods* (Quiñonero-Candela and Rasmussen, 2005; Snelson and Ghahramani, 2006), which introduce a set of $m \ll n$ data points Z to form a low-rank kernel approximation:

$$K_{XX} \approx K_{XZ}K_{ZZ}^{-1}K_{ZX}.$$

Training and predictions with this approximation take $\mathcal{O}(nm^2)$ time and $\mathcal{O}(nm)$ space. Here we highlight some notable variants of the basic approach, though it is by no means an exhaustive list – see (Liu et al.,

2018) for a more thorough review. *Sparse Gaussian process regression* (SGPR) (Titsias, 2009; Hensman et al., 2013) selects the inducing points Z through a regularized objective. *Structured kernel interpolation* (SKI) (Wilson and Nickisch, 2015) and its variants (Gardner et al., 2018b) place the inducing points on a regularly-spaced grid, in combination with sparse interpolation, for $\mathcal{O}(n+g(m))$ computations and memory, where $g(m) \approx m$. *Stochastic variational Gaussian processes* (SVGP) (Hensman et al., 2015) introduces a set of variational parameters that can be optimized using minibatch training. Recent work has investigated how to scale up the number of inducing points using tensor decompositions (Evans and Nair, 2018; Izmailov et al., 2018). It is typical for practitioners to use one or more of these methods on problems with more than 10^4 data points. In the next section, we show that exact GPs outperform SGPR and SVGP in terms of accuracy on datasets with $10^4 - 10^6$ training points.

5 RESULTS

We compare the performance of exact Gaussian processes against widely-used scalable GP approximation methods on a range of large-scale regression datasets from the UCI dataset repository (Asuncion and Newman, 2007). Our experiments demonstrate that exact GPs: (1) outperform popular approximate GPs methods SGPR and SVGP on all benchmarking datasets in our study; (2) compute thousands of test-point predictions in less than a second, even when $n > 10^6$; (3) utilize all available data when making predictions, even when $n > 10^5$; and (4) achieve linear training speedups on large datasets by adding additional GPU devices.

Baselines. We compare against two scalable GP approximations: Sparse Gaussian Process Regression (SGPR) (Titsias, 2009; Hensman et al., 2013; Matthews, 2016), and Stochastic Variational Gaussian Processes (SVGP) (Hensman et al., 2013, 2015). We choose these methods due to their popularity and general applicability, enabling a comparison over a wide range of datasets. SGPR is an inducing point method where the inducing points are learned through a variational objective. This method is typically used on medium-sized regression datasets. SVGP can also be viewed as an inducing point method, but where the predictive distribution is defined by a set of variational parameters. Unlike SGPR, the variational parameters and hyperparameters are learned through minibatch optimization. Therefore, SVGP does not have to store the entire dataset and is typically used when SGPR doesn't fit into memory. Both methods scale as $\mathcal{O}(nm^2+m^3)$, where m is the number of inducing points and n is the number of data points used per training iteration. In practice, these methods use $m \ll n$ to scale to large datasets. We use $m = 512$ for SGPR and $m = 1{,}024$ for SVGP, which are common values used for these methods (Matthews et al., 2017). We later experiment with varying the number of inducing points.

Experiment details. We extend the GPyTorch library (Gardner et al., 2018a) to perform all experiments. Each dataset is randomly split into $2/3$ training and $1/3$ testing. The data is whitened to be mean 0 and standard deviation 1 as measured by the training set. We use a constant prior mean and a Matérn 3/2 kernel with a shared lengthscale across all dimensions for all models.

We learn model hyperparameters and variational parameters by optimizing the log marginal likelihood using the the same Adam optimizer by Kingma and Ba (2015) to keep experiments consistent. For exact GPs and SGPR, we perform 100 iterations of Adam with a learning rate of 0.1. For SVGP, we perform 100 epochs of Adam with a minibatch size of 1,024 and a learning rate of 0.01, which we found to perform better than using the same learning rate as exact GPs and SGPR. We use a rank-100 partial pivoted-Cholesky preconditioner and run PCG with a tolerance of $\epsilon = 1$ during training. We perform all training on a single machine with 8 NVIDIA Tesla V100-SXM2-32GB-LS GPUs. Code to reproduce the experiments is available at `https://gpytorch.ai`.

Accuracy. Table 1 displays the accuracy of exact GPs and approximate methods on several large-scale datasets. We find that exact GPs consistently achieve lower error than approximate methods on every dataset.

The first section contains medium-sized datasets with $n \leq 50{,}000$. Most notably, on Kin40K, an exact GP achieves half the error of SVGP and a third the error of SGPR, while on CTSlice, an exact GP achieves a quarter of the error of SGPR.

The second section of Table 1 contains datasets with $n \geq 100{,}000$, including one dataset with over 1M points. Here we also find that exact GPs outperform the baseline approximate GPs, achiving a quarter of the error of SVGP on 3DRoad and half the error on HouseElectric. To the best of our knowledge, this is the first set of results comparing exact GPs to approximate methods on datasets of this size.

Interestingly, the size or the dimensionality of the dataset does not seem to influence the relative performance of the approximate methods. For example, though Protein and Kin40K are similar in size and have almost the same dimensionality, the approximate methods perform worse on Kin40K (relative to the performance of exact GPs). Approximate methods have 18% higher error on Protein, yet they have 100% higher error on Kin40K. Moreover,

Table 1: Exact GPs vs. approximate methods on medium and large regression datasets using a constant prior mean and a Matérn 3/2 kernel with a shared lengthscale across all dimensions. The table shows the root-mean-square error (RMSE) computed on the test set, as well as training time. n and d are the size and dimensionality of the training dataset, respectively, and p is the number of kernel partitions. All models were trained using V100 GPUs. Best results are in bold (lower is better).

			RMSE (random = 1)			**Training Time**				
Dataset	n	d	**Exact GP** (BBMM)	**SGPR** ($m=512$)	**SVGP** ($m=1{,}024$)	**Exact GP** (BBMM)	**SGPR** ($m=512$)	**SVGP** ($m=1{,}024$)	#GPU	p
PoleTele	9,600	26	**0.154**	0.219	0.218	**22.1 s**	40.6 s	68.1 s	1	1
Elevators	10,623	18	**0.374**	0.436	0.386	**17.1 s**	41.2 s	112 s	1	1
Bike	11,122	17	**0.216**	0.345	0.261	**18.8 s**	41.0 s	109 s	1	1
Kin40K	25,600	8	**0.093**	0.257	0.177	83.3 s	**56.1 s**	297 s	1	1
Protein	29,267	9	**0.545**	0.659	0.640	120 s	**65.5 s**	300 s	1	1
KeggDirected	31,248	20	**0.078**	0.089	0.083	107 s	**67.0 s**	345 s	1	1
CTslice	34,240	385	**0.050**	0.199	1.011	148 s	**77.5 s**	137 s	1	1
KEGGU	40,708	27	**0.120**	0.133	0.123	**50.8 s**	84.9 s	7.61 min	8	1
3DRoad	278,319	3	**0.106**	0.654	0.475	7.06 hr	**8.53 min**	22.1 min	8	16
Song	329,820	90	**0.761**	0.803	0.999	6.63 hr	**9.38 min**	18.5 min	8	16
Buzz	373,280	77	**0.265**	0.387	0.270	11.5 hr	**11.5 min**	1.19 hr	8	19
HouseElectric	1,311,539	9	**0.049**	—	0.086	3.29 days	—	**4.22 hr**	8	218

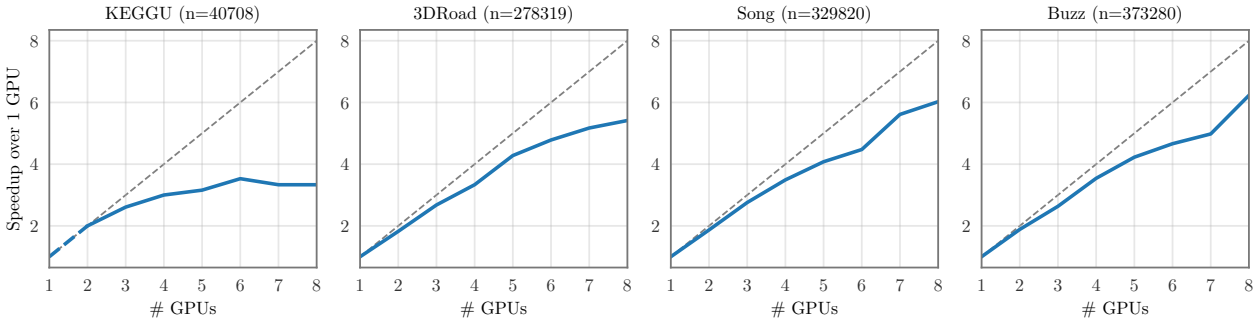

Figure 1: Training speedup from using additional GPUs at training time. Since our exact GPs use matrix-multiplication based inference, they achieve a near linear speedup with more computing resources on large datasets.

we also see that the choice of approximate method affects performance. On certain datasets like Bike, SVGP outperforms SGPR. On other datasets, such as CTSlice and Song, SVGP fails to learn anything, attaining test RMSEs equal to the random baseline. This suggests that the performance of SVGP and SGPR cannot be interpreted as reliable indicators of exact GP performance and that the chosen approximation can greatly affect predictive performance.

Training time. Table 1 also displays the amount of time to train exact and approximate GPs. On datasets with $n < 35{,}000$, an exact GP fits on a single 32GB GPU without any partitioning and can be trained in seconds. For $n \geq 100{,}000$, the kernel matrices need to be partitioned and must use the matrix-vector multiplication strategy outlined in Algorithm 2 which significantly increases the training time for exact GPs. However, if the necessary computational resources are available, these experiments show that it may be preferable to train an exact GP to make more accurate predictions in exchange for longer training times.

Acceleration with multiple GPUs. Because we use matrix-multiplication-based approaches to train exact GPs, the computations can be easily parallelized and distributed. Moreover, matrix multiplication is one of the most commonly distributed routines, so parallelized GP implementations can be built using readily-available routines in libraries like PyTorch. Figure 1 plots the speedup as more GPUs are used for training on the KEGGU, 3DRoad, Song, and Buzz datasets. Each of these datasets achieve a nearly linear speedup when adding up to 4 GPUs. The speedup is more pronounced for the two large datasets (3DRoad and Song). The training time can be further improved by using more GPUs to reduce the

number of kernel partitions.

Acceleration with L-BFGS. The results in Table 1 use Adam to train exact GPs despite the optimization problem not being inherently stochastic in the exact setting. We use Adam since it generally provides the best results for the alternative methods, and SVGP is intended for stochastic optimization. To compare to a more sophisticated full-batch gradient descent method, we also train exact GPs with L-BFGS (Liu and Nocedal, 1989). On each dataset, we run L-BFGS until 10 steps of linesearch are unable to make further progress. We find that performance on a few datasets is improved by additionally finetuning with 10 iterations of Adam. Table 2 shows the results of this comparison.

Table 2: Training exact GPs using L-BFGS compared to Adam. All results were run using the same hardware and other experimental details as in Table 1.

	Test RMSE		Training Time	
Dataset	**Exact GP** (L-BFGS)	**Exact GP** (Adam)	**Exact GP** (L-BFGS)	**Exact GP** (Adam)
PoleTele	**0.150**	0.154	**8.8 s**	22.1 s
Elevators	**0.374**	**0.374**	**6.5 s**	17.1 s
Bike	0.222	**0.216**	**9.7 s**	18.8 s
Kin40K	0.094	**0.093**	**27.7 s**	83.3 s
Protein	**0.534**	0.545	**17.3 s**	120 s
KeggDirected	**0.078**	**0.078**	**50.6 s**	107 s
CTslice	**0.045**	0.050	**42.1 s**	148 s
KEGGU	0.121	**0.120**	**38.7 s**	50.8 s
3DRoad	**0.106**	**0.106**	**7.03 hr**	7.06 hr
Song	0.767	**0.761**	**37.7 min**	6.63 hr
Buzz	0.268	**0.265**	**10.6 hr**	11.5 hr

We find that on smaller and medium sized datasets L-BFGS converges significantly faster than Adam. In the large dataset regime, Adam becomes more competitive with L-BFGS. We believe this finding is partly due to the costly nature of line search in the setting where kernel partitioning is used, as each function evaluation is more expensive than a gradient evaluation. Additionally, the running time of L-BFGS varies considerably on the larger datasets. We hypothesize that this variability is a result of L-BFGS line searches venturing into hyperparameter regimes that result in poorly conditioned kernel matrices, requiring many more iterations of CG to reach the specified tolerance ϵ.

Prediction time. Although exact GPs take longer to train, we find that *their speed is comparable to approximate methods at test time.* After training various GPs, we follow the common practice of precomputing the solves required for GP predictions. For predictive mean computations, we pre-compute the solve $\widehat{K}_{XX}^{-1}\mathbf{y}$ in (3). For predictive variances, we compute a low-rank approximation of $\widehat{K}_{XX}^{-1}$ in (4) using the method described by Pleiss et al. (2018). Table 3 measures the time for the precomputations for exact GPs. On large datasets, these precomputations can take a few hours. However, subsequent predictions are reduced to $\mathcal{O}(n)$ vector multiplications.

Table 3 displays the time to compute 1,000 predictive means and variances at test time after precomputation. All predictions are made on one NVIDIA RTX 2080 Ti GPU. We see exact GPs take *less than a second for all predictions* across all dataset sizes used, nearly matching the prediction speed of SGPR and consistently beating the prediction speed of SVGP.

Table 3: Prediction time for exact GPs and approximate methods, as measured on 1 NVIDIA RTX 2080 Ti GPU. Times measure computation of 1,000 predictive means and variances. An asterisk (*) indicates the one-time pre-computed cache was calculated using 8 V100 GPUs. Best results are in bold (lower is better).

	Precomputation	Prediction		
Dataset	**Exact GP** (BBMM)	**Exact GP** (BBMM)	**SGPR** ($m=512$)	**SVGP** ($m=1{,}024$)
PoleTele	5.14 s	**6 ms**	**6 ms**	273 ms
Elevators	0.95 s	**7 ms**	**7 ms**	212 ms
Bike	0.38 s	**7 ms**	9 ms	182 ms
Kin40K	12.3 s	**11 ms**	12 ms	220 ms
Protein	7.53 s	14 ms	**9 ms**	146 ms
KeggDirected	8.06 s	**15 ms**	16 ms	143 ms
CTslice	7.57 s	**22 ms**	14 ms	133 ms
KEGGU	18.9 s	**18 ms**	13 ms	211 ms
3DRoad	118 m*	119 ms	**68 ms**	130 ms
Song	22.2 m*	123 ms	**99 ms**	134 ms
Buzz	42.6 m*	131 ms	**114 ms**	142 ms
HouseElectric	7.20 hr*	958 ms	—	**166 ms**

5.1 Ablation Studies

To the best of our knowledge, this is the first time that exact GPs have been trained on datasets with $n \gg 10^4$. With our method, we can better understand how exact GPs and approximate GPs scale to larger datasets. Here, we demonstrate how the amount of data affects exact GP performance, and how the number of inducing points affects the performance of approximate GPs.

Do GPs need the entire dataset? As a non-parametric model, Gaussian processes naturally adapt to the amount of training data available (Wilson et al., 2014). Figure 2 shows an increase in accuracy as we increase the amount of training data on the KEGGU, 3DRoad, and Song datasets. For each dataset, we subsample a fraction of the training data and plot the resulting root-mean-square error on the test-set as a function of subsampled training set size. We use the same 1/3 holdout of the full dataset to test in each case. As expected, the test RMSE

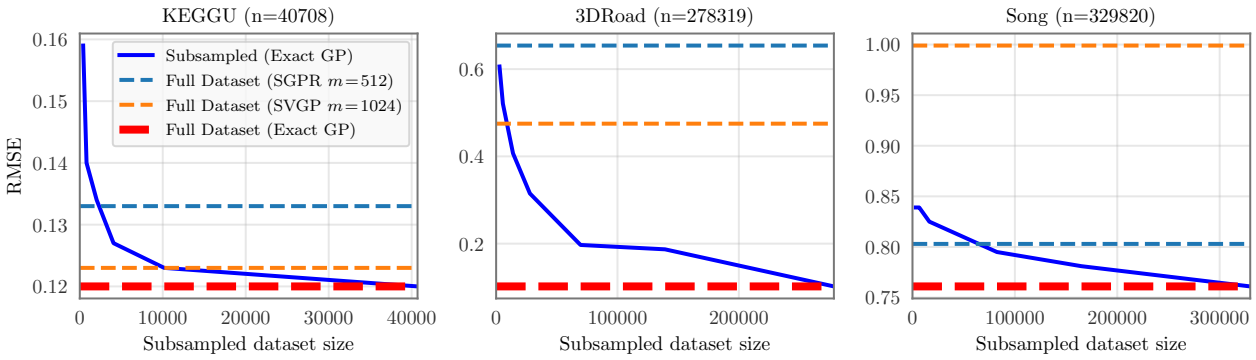

Figure 2: Test root-mean-square error (RMSE) as a function of subsampled dataset size (lower is better). Subsampled exact GPs outperform approximate GPs even with a quarter of the training set. Exact GP error continues to decrease as data is added until the full dataset is used.

decreases monotonically as we increase the subsample size. Figure 2 also shows the performance of exact GP, SGPR, and SVGP trained on the entire training set. Strikingly, in all three cases, *an exact GP with less than a quarter of the training data outperformed approximate GPs trained on the entire training set*. Furthermore, test error continues to decrease with each addition of training data.

Would more inducing points help? In Table 1, exact GPs uniformly take substantially longer to train on the largest datasets than the approximate methods. This naturally raises the question "can approximate models with more inducing points recover the performance of exact methods?" We plot test RMSE on two datasets, Bike and Protein, as a function of the number of inducing points in Figure 3. The test RMSE of both inducing point methods saturates on both datasets well above the test RMSE of an exact GP trained. Furthermore, we note that using m inducing points introduces a $m \times m$ matrix and a $\mathcal{O}(nm^2 + m^3)$ time complexity (Hensman et al., 2013, 2015) which makes it difficult to train SGPR with $m \gg 1024$ inducing points on one GPU. It is possible to combine kernel partitioning in Algorithm 2 with inducing-point methods to utilize even larger values of m. However, as Figure 3 and Table 1 show, it may be preferable to use the extra computational resources to train an exact GP on more data rather than to train an approximate GP with more inducing points.

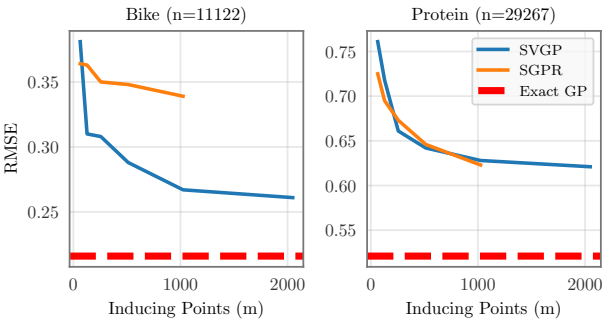

Figure 3: Error of SVGP and SGPR methods as a function of the number of inducing points (m). Both methods scale cubically with m. We were unable to run SGPR with more than 1,024 inducing points on a single GPU. Exact GPs have lower error than both methods.

6 DISCUSSION

For Gaussian processes, "a large dataset is one that contains over a few thousand data points" (Hensman et al., 2013). Bigger datasets have traditionally necessitated scalable approximations. In this paper, we have extended the applicability of exact GPs far beyond what has been thought possible — to datasets with *over a million training examples* through MVM-based GP inference. Our approach uses easily parallelizable routines that fully exploit modern parallel hardware and distributed computing. In our comparisons, we find that exact GPs are more widely applicable than previously thought; they perform dramatically well on very large datasets while requiring fewer design choices than approximate methods.

Is CG still exact? In the GP literature, *exact* GP inference typically refers to using the Cholesky decomposition with exact kernels (Rasmussen and Williams, 2006). A natural question to ask is whether we can consider our approach "exact" in light of the fact that CG perform solves only up to a pre-specified error tolerance. However, unlike the approximate methods presented in this paper, the difference between a CG-based model and a theoretical model with "perfect" solves can be precisely controlled by this error tolerance. We therefore consider CG exact in a sense that is commonly used in the context of mathematical optimization — namely that it computes solutions up to arbitrary precision. In

fact, CG-based methods can often be more precise than Cholesky based approaches in floating-point arithmetic due to fewer round-off errors (Gardner et al., 2018a).

When to approximate? There are many approximate methods for scalable Gaussian processes, with varying statistical properties, advantages, and application regimes. We chose to compare exact GPs to approximation methods SVGP and SGPR for their popularity and available GPU implementations. There may be some regimes where other approximate methods or combinations of methods outperform these two approximations. Our objective is not to perform an exhaustive study of approximate methods and their relative strengths.

Indeed, there are cases where an approximate GP method might still be preferable. Examples may include training on large datasets with limited computational resources. In certain regimes, such as low dimensional spaces, there are approximations that are designed to achieve high degrees of accuracy in less time than exact GPs. Additionally, GP inference with non-Gaussian likelihoods (such as for classification) requires an approximate inference strategy. Some approximate inference methods, such as Laplace and MCMC (Rasmussen and Williams, 2006; Murray et al., 2010), may be amenable to the parallelisation approaches discussed here for approximate inference with exact kernels.

Nonetheless, with efficient utilization of modern hardware, exact Gaussian processes are now an appealing option on substantially larger datasets than previously thought possible. Exact GPs are powerful yet simple – achieving remarkable accuracy without requiring much expert intervention. We expect exact GPs to become ever more scalable and accessible with continued advances in hardware design.

References

Asuncion, A. and Newman, D. (2007). Uci machine learning repository. https://archive.ics.uci.edu/ml/. Last accessed: 2018-02-05.

Cheng, C.-A. and Boots, B. (2017). Variational inference for gaussian process models with linear complexity. In *NeurIPS*, pages 5184–5194.

Cunningham, J. P., Shenoy, K. V., and Sahani, M. (2008). Fast gaussian process methods for point process intensity estimation. In *ICML*, pages 192–199. ACM.

Cutajar, K., Osborne, M., Cunningham, J., and Filippone, M. (2016). Preconditioning kernel matrices. In *ICML*.

Damianou, A. and Lawrence, N. (2013). Deep gaussian processes. In *AISTATS*, pages 207–215.

Deisenroth, M. and Rasmussen, C. E. (2011). Pilco: A model-based and data-efficient approach to policy search. In *ICML*, pages 465–472.

Deisenroth, M. P., Fox, D., and Rasmussen, C. E. (2015). Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(2):408–423.

Deisenroth, M. P. and Ng, J. W. (2015). Distributed gaussian processes. In *ICML*, pages 1481–1490.

Dong, K., Eriksson, D., Nickisch, H., Bindel, D., and Wilson, A. G. (2017). Scalable log determinants for gaussian process kernel learning. In *NeurIPS*.

Evans, T. and Nair, P. (2018). Scalable gaussian processes with grid-structured eigenfunctions (gp-grief). In *ICML*, pages 1416–1425.

Gardner, J., Pleiss, G., Weinberger, K. Q., Bindel, D., and Wilson, A. G. (2018a). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *NeurIPS*, pages 7587–7597.

Gardner, J. R., Pleiss, G., Wu, R., Weinberger, K. Q., and Wilson, A. G. (2018b). Product kernel interpolation for scalable gaussian processes. In *AISTATS*.

Golub, G. H. and Van Loan, C. F. (2012). *Matrix computations*, volume 3. JHU Press.

Griewank, A. and Walther, A. (2000). Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45.

Gustavson, F. G., Karlsson, L., and Kågström, B. (2006). Three algorithms for cholesky factorization on distributed memory using packed storage. In *International Workshop on Applied Parallel Computing*, pages 550–559. Springer.

Hensman, J., Fusi, N., and Lawrence, N. D. (2013). Gaussian processes for big data. In *UAI*.

Hensman, J., Matthews, A., and Ghahramani, Z. (2015). Scalable variational gaussian process classification. In *AISTATS*.

Izmailov, P., Novikov, A., and Kropotov, D. (2018). Scalable gaussian processes with billions of inducing inputs via tensor train decomposition. In *ICML*, pages 726–735.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR*.

Le, Q., Sarlós, T., and Smola, A. (2013). Fastfood: approximating kernel expansions in loglinear time. In *ICML*.

Liu, D. C. and Nocedal, J. (1989). On the limited memory bfgs method for large scale optimization. *Math. Program.*, 45(1-3):503–528.

Liu, H., Ong, Y.-S., Shen, X., and Cai, J. (2018). When gaussian process meets big data: A review of scalable gps. *arXiv preprint arXiv:1807.01065*.

Matthews, A. G. d. G. (2016). *Scalable Gaussian process inference using variational methods*. PhD thesis, University of Cambridge.

Matthews, A. G. d. G., van der Wilk, M., Nickson, T., Fujii, K., Boukouvalas, A., León-Villagrá, P., Ghahramani, Z., and Hensman, J. (2017). Gpflow: A gaussian process library using tensorflow. *Journal of Machine Learning Research*, 18(40):1–6.

Murray, I., Prescott Adams, R., and MacKay, D. J. (2010). Elliptical slice sampling.

Pleiss, G., Gardner, J. R., Weinberger, K. Q., and Wilson, A. G. (2018). Constant-time predictive distributions for gaussian processes. In *ICML*.

Quiñonero-Candela, J. and Rasmussen, C. E. (2005). A unifying view of sparse approximate gaussian process regression. *Journal of Machine Learning Research*, 6(Dec):1939–1959.

Rahimi, A. and Recht, B. (2008). Random features for large-scale kernel machines. In *NeurIPS*, pages 1177–1184.

Rasmussen, C. E. and Ghahramani, Z. (2001). Occam's razor. In *NeurIPS*, pages 294–300.

Rasmussen, C. E. and Williams, C. K. (2006). *Gaussian processes for machine learning*, volume 1. MIT press Cambridge.

Roberts, S., Osborne, M., Ebden, M., Reece, S., Gibson, N., and Aigrain, S. (2013). Gaussian processes for time-series modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1984).

Saatçi, Y. (2012). *Scalable inference for structured Gaussian process models*. PhD thesis, University of Cambridge.

Salimbeni, H., Cheng, C.-A., Boots, B., and Deisenroth, M. (2018). Orthogonally decoupled variational gaussian processes. In *NeurIPS*, pages 8725–8734.

Salimbeni, H. and Deisenroth, M. (2017). Doubly stochastic variational inference for deep gaussian processes. In *NeurIPS*, pages 4588–4599.

Snelson, E. and Ghahramani, Z. (2006). Sparse Gaussian processes using pseudo-inputs. In *NeurIPS*, pages 1257–1264.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Titsias, M. K. (2009). Variational learning of inducing variables in sparse gaussian processes. In *AISTATS*, pages 567–574.

Ubaru, S., Chen, J., and Saad, Y. (2017). Fast estimation of tr (f (a)) via stochastic lanczos quadrature. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1075–1099.

Wilson, A. and Adams, R. (2013). Gaussian process kernels for pattern discovery and extrapolation. In *ICML*, pages 1067–1075.

Wilson, A. G., Gilboa, E., Nehorai, A., and Cunningham, J. P. (2014). Fast kernel learning for multidimensional pattern extrapolation. In *Advances in Neural Information Processing Systems*.

Wilson, A. G., Hu, Z., Salakhutdinov, R., and Xing, E. P. (2016a). Deep kernel learning. In *AISTATS*, pages 370–378.

Wilson, A. G., Hu, Z., Salakhutdinov, R. R., and Xing, E. P. (2016b). Stochastic variational deep kernel learning. In *NeurIPS*, pages 2586–2594.

Wilson, A. G., Knowles, D. A., and Ghahramani, Z. (2012). Gaussian process regression networks. In *International Conference on Machine Learning*.

Wilson, A. G. and Nickisch, H. (2015). Kernel interpolation for scalable structured gaussian processes (kiss-gp). In *ICML*, pages 1775–1784.

Yang, Z., Wilson, A., Smola, A., and Song, L. (2015). A la carte–learning fast kernels. In *AISTATS*, pages 1098–1106.