Low Precision Arithmetic for Fast Gaussian Processes

Abstract

Low precision arithmetic has had a transformative effect on the training of neural networks, reducing computation, memory and energy requirements. However, despite its promise, low precision arithmetic has received little attention for Gaussian process (GP) training, largely because GPs require sophisticated linear algebra routines that are unstable in low precision. We study the different failure modes that can occur when training GPs in half-precision. To circumvent these failure modes, we propose a multi-faceted approach involving conjugate gradients with re-orthogonalization, compact kernels, and preconditioners. Our approach significantly improves the numerical stability and practical performance of conjugate gradients in low precision over a wide range of settings, and reduces the runtime of 1.8 million data points to 10 hours on a single GPU, without requiring any sparse approximations.

1 INTRODUCTION

Low-precision training can profoundly improve training time, memory requirements, and energy consumption. Moreover, as hardware accelerators continue to develop, the scalability gains of low-precision algorithms will continue to appreciate into the future. However, training in low precision can increase the magnitude of round-off errors and decrease the supported range of the numerical representation, leading to lower quality gradients and ultimately sacrificing performance or, in the worst cases, convergence of the training procedure. Recent works have developed efficient and stable methods for training deep neural networks in low precision to great effect, often via the development of new floating point representations [Intel, 2018, Wang and Kanwar, 2019], mixed-precision representations [Micikevicius et al., 2017, Das et al., 2018, Yang et al., 2019], or alternative summation techniques [NVIDIA, 2017, Micikevicius et al., 2017, Zamirai et al., 2021].

Gaussian processes (GPs) are significantly more expensive than neural networks, and thus potentially stand more to gain from low precision computations. GPs typically require $\mathcal{O}(n^3)$ computations and $\mathcal{O}(n^2)$ memory, for n data points. Going from float to half precision therefore already halves the memory requirements and run-time for matrix vector multiplications. However, GPs typically require sophisticated algebraic operations such as Cholesky decompositions for solving linear systems, which suffer badly from round-off error and are very poorly suited to half precision. Developing these approaches for low precision computation is an open area of research [Higham and Pranesh, 2021].

Instead, we focus on iterative techniques for Gaussian processes [Gibbs and MacKay, 1997, Cutajar et al., 2016, Gardner et al., 2018] that use conjugate gradients to compute the solutions to linear systems while exploiting extremely efficient MapReduce based matrix vector multiplications using KeOps [Charlier et al., 2021, Feydy et al., 2020]. These approaches are particularly appealing in the context of low precision computations: (1) they suffer less from round-off errors than Cholesky decompositions [Gardner et al., 2018]; (2) they are easy to parallelize on GPUs, which are being designed specifically to accelerate low-precision computation; (3) memory consumption is a primary *computational* bottleneck, since more memory means fewer computationally expensive matrix partitions [Wang et al., 2019].

While there are many challenges to overcome, we demonstrate that training GPs in half-precision can be stable, efficient, and broadly practical. In particular, our contributions include:

- We show that half precision kernel matrices have qualitatively different eigenspectra than higher precision representations, and represent covariances at much shorter distances than in float or double precision.

Submitted to the 38th Conference on Uncertainty in Artificial Intelligence (UAI 2022). **To be used for reviewing only**.

- We investigate the effect of kernel choice, CG tolerance, pre-conditioner rank, and compact support on stability in half precision.
- Based on these insights, we propose a new numerically stable conjugate gradients (CG) algorithm that converges quickly in half precision, particularly due to the use of re-orthogonalization, mixed-precision and the logsumexp trick.
- We provide a powerful practical implementation of Gaussian processes in half precision that reduces training times on 1.8 million data point datasets to less than 10 hours on a single GPU, a major advance over recent CG milestones of 1 million data points in 3 days on 8 GPUs [Wang et al., 2019].

2 PRELIMINARIES

This section introduces the basic information on different floating point precision representations, on training Gaussian Process and how the optimization is translated into linear solves and, finally, on conjugate gradients as an accelerated linear solver.

2.1 NUMERICAL PRECISION

Modern computer architectures represent numbers in floating point precision, often with either 32-bit (float) or 64-bit (double) precision [Kahan, 1996]. *Float precision* uses 1 bit for the sign of the number (s), 8 bits for the exponent (e), and 23 bits for the mantissa (or significant, m). Then, the value of a number is represented as

$$\text{value}\,(s, e, m) = (-1)^s \times 2^{e-127} \times 1.m$$

where $s \in \{0, 1\}$, $e \in \{0, \cdots, 255\}$ and $m\{1, \cdots, 1 + \sum_{i=1}^{23} 2^{-i}\}$. Float precision can then represent values from $\pm\left(2 - 2^{-23}\right) \times 2^{127} \approx \pm 3.4028235 \times 10^{38}$ and measuring decimals less than one for values greater than $2^{-126} \approx 1.175 \times 10^{-38}$. The round-off error is of the order of $2^{-24} \approx 6 \times 10^{-8}$ (Chapter 2.5, Watkins [2010]).

In contrast, *half precision*, a representation with 16-bits, uses 1 bit for the sign, 5 bits for the exponent and 10 bits for the mantissa (or significant). Using again s, e and m to represent the previous bits, we obtain the following representation

$$\text{value}\,(s, e, m) = (-1)^s \times 2^{e-14} \times 1.m$$

where $s \in \{0, 1\}$, $e \in \{0, \cdots, 255\}$ and $m\{1, \cdots, 1 + \sum_{i=1}^{10} 2^{-i}\}$. Immediately we see that the range of values decreases to $\pm\left(2 - 2^{-10}\right) \times 2^{17} \approx \pm 2.620 \times 10^5$ and that the lowest value that can be represented is $2^{-14} \approx 6.1035 \times 10^{-5}$. The roundoff error is now significantly higher as well, on the order of $2^{-14} \approx \times 10^{-4}$. Much modern hardware attempts to reduce the roundoff error via fused multiply-and-accumulate (FMAC) compute units to accumulate operations at higher precisions [NVIDIA, 2017, Zamirai et al., 2021, Micikevicius et al., 2017]. We also note the existence of the bfloat16 standard [Intel, 2018], which uses a slightly different representation, preserving the range of float precision, while using fewer bits in the mantissa, thereby trading off the range for increased roundoff error.

2.2 GAUSSIAN PROCESSES

We consider regression tasks on observed data $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ for $\mathbf{x}_i \in \mathbb{R}^D$ and $y_i \in \mathbb{R}$. The Gaussian process (GP) model for the data is

$$\begin{aligned} f(\cdot) &\sim \mathcal{GP}\left(m(\cdot), k(\cdot,\cdot)\right), \\ y_i &\sim f(\mathbf{x}_i) + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}\left(0, \sigma^2\right) \end{aligned}$$

where $k_\theta(\cdot,\cdot)$ is the covariance kernel, $m(\cdot)$ is the mean function (set to zero without loss of generality) [Rasmussen and Williams, 2008]. We train the kernel hyperparameters, θ, via minimizing the the negative marginal likelihood:

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}) &= -\log p(\mathbf{y}|\mathbf{X};\boldsymbol{\theta}) \\ &= \frac{1}{2}\log\left|\tilde{\mathbf{K}}\right| + \frac{1}{2}\mathbf{y}^T\tilde{\mathbf{K}}^{-1}\mathbf{y} + \frac{N}{2}\log(2\pi) \end{aligned} \tag{1}$$

where $\tilde{\mathbf{K}} \in \mathbb{R}^{N\times N}$ is the Gram matrix of all data points with diagonal observational noise

$$\tilde{\mathbf{K}}_{i,j} = (\mathbf{K} + \sigma^2 I)_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j) + \sigma^2 \mathbb{I}_{i=j},$$

representing the evaluated covariance matrix as $\mathbf{K}$. To perform hyper-parameter estimation of $\boldsymbol{\theta}$, we need to compute the gradient of the loss function, which is given by

$$\begin{aligned} \nabla_{\boldsymbol{\theta}}\mathcal{L} &= \frac{1}{2}\operatorname{Tr}\left(\tilde{\mathbf{K}}^{-1}\nabla_{\boldsymbol{\theta}}\tilde{\mathbf{K}}\right) - \frac{1}{2}\mathbf{y}^T\tilde{\mathbf{K}}^{-1}\left(\nabla_{\boldsymbol{\theta}}\tilde{\mathbf{K}}\right)\tilde{\mathbf{K}}^{-1}\mathbf{y} \\ &\approx \frac{1}{M}\sum_{i=1}^{M}\mathbf{z}_i^\top\tilde{\mathbf{K}}^{-1}\nabla_{\boldsymbol{\theta}}\tilde{\mathbf{K}}\mathbf{z}_i) - \frac{1}{2}\mathbf{y}^T\tilde{\mathbf{K}}^{-1}\left(\nabla_{\boldsymbol{\theta}}\tilde{\mathbf{K}}\right)\tilde{\mathbf{K}}^{-1}\mathbf{y} \end{aligned} \tag{2}$$

with the first line coming from matrix derivatives [Rasmussen and Williams, 2008] and the second line coming from the application of Hutchinson's trace estimator to the trace term [Gibbs and MacKay, 1997, Gardner et al., 2018]. The limiting time complexity is that of computing $\mathbf{v} = \tilde{\mathbf{K}}^{-1}\mathbf{y}$, which costs $\mathcal{O}(N^3)$ operations and space when using the Cholesky decomposition [Rasmussen and Williams, 2008]. Instead, Gibbs and MacKay [1997], Gardner et al. [2018] proposed to use conjugate gradients (CG) to compute these linear systems, reducing the time complexity down to $\mathcal{O}(JN^2)$, for J steps of conjugate gradients.

2.3 CONJUGATE GRADIENTS

We use conjugate gradients (CG) to solve the linear system $\mathbf{A}x = \mathbf{b}$ for x, so that $x = \mathbf{A}^{-1}b$ [Hestenes and Stiefel, 1952, Nocedal and Wright, 2006, Demmel, 1997]. To estimate x, CG iteratively minimizes the CG uses a three-term recurrence where each new term requires only a matrix-vector multiplication with $\mathbf{A}$. Formally, each CG iteration computes a new term of the following summation: $\mathbf{A}^{-1}\mathbf{b} = \sum_{i=1}^{N} \gamma_i \mathbf{d}_i$, where, for simplicity, we assume that the algorithm is initialized at $\mathbf{x}_0 = \mathbf{0}$, γ_i are the coefficients and $\mathbf{d}_i$ are the orthogonal conjugate search directions Golub and Loan [2018]. In infinite precision representation, N iterations of CG produce all the N summation terms and recover the exact solution. The overall CG algorithm is shown in 2 in the Appendix.

3 RELATED WORK

Iterative Gaussian Processes: Due to the cubic runtime complexity of training Gaussian processes (GPs) using Cholesky, there has been extensive research on using alternative algorithms for reducing that overhead. In this work, we focus solely on comparing with iterative GPs, ignoring the vast literature of alternative approaches. Gibbs and MacKay [1997] first studied iterative techniques for GPs by using conjugate gradients to solve the linear systems and used trace estimation, which reduces the overall complexity of GP regression to $\mathcal{O}(kN^2)$ for k steps of CG, while Cutajar et al. [2016] re-discovered the technique and applied preconditioning to speed up the linear systems focusing solely on double precision. Gardner et al. [2018] proposed the batch conjugate gradients algorithm that we build off of and used Lanczos to estimate log determinants, focusing their efforts in float precision. Wang et al. [2019] extended this work and enabled exact GPs to be trained on 1 million data points by using 8 GPUs over 3 days via data partitioning. Outside explicit GP regression, Meanti et al. [2020] also used KeOps over several GPUs for Nyström-based kernel regression, achieving results on datasets of up to 1 billion data points.

Lower Precision Arithmetic: Interest in training neural network models in lower precisions than traditional double or floating point precision has been around as early as 1990 [Hammerstrom, 1990]. In the modern era, Gupta et al. [2015] and Chen et al. [2014] have pionneered the usage of lower precision arithmetic with the goal of reducing memory costs to enable the training of large deep neural network models. Gupta et al. [2015] proposed the usage of stochastic rounding to reduce errors from the loss of precision when moving to half (e.g. 16-bit arithmetic) precision down from floating point, wile Chen et al. [2014] used special representations to enable accurate training of DNNs. These works have led to a flurry of research into *mixed precision* training of deep neural networks [Micikevicius et al., 2017, Das et al., 2018, Yang et al., 2019]. In mixed precision training, which inspires our approach, the activations and gradients of each DNN layer are propagated in half but the master weight copy is stored in floating point precision [Micikevicius et al., 2017]. The success of these lower precision approaches has led to significant efforts in reducing the memory overhead even further via quantization of neural network layers [Jacob et al., 2018, Das et al., 2018], the development of specialized chip architectures that speed up lower precision arithmetic such as modern GPUs [NVIDIA, 2017] and TPU cores [Wang and Kanwar, 2019], and new standards that enhance mixed precision training such as bfloat16 [Wang and Kanwar, 2019, Intel, 2018].

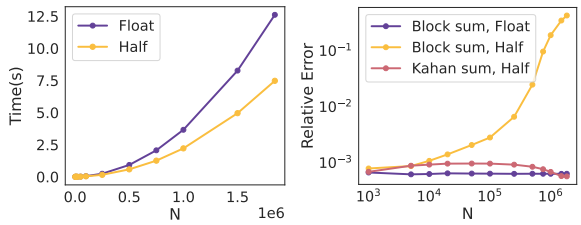
The promise of lower precision has crept outside of the deep learning community. Cutajar et al. [2016] found that pre-conditioning was necessary to prevent conjugate gradients (CG) to estimate solves towards numerical precision, a finding echoed by Gardner et al. [2018] and more recently Wenger et al. [2021]. Gardner et al. [2018], Wang et al. [2019], Wenger et al. [2021] argue for the usage of pivoted Cholesky preconditioners [Harbrecht et al., 2012]. In the numerical linear algebra community, Gratton et al. [2021] argued instead for the usage of full re-orthogonalization when using CG as a general purpose optimization algorithm. Haidar et al. [2017, 2018] alternatively proposed an iterative refinement and the usage of GMRES (a method related directly to CG) for solving dense linear systems on GPUs in lower precision arithmetic, while Abdelfattah et al. [2020] considered mixed precision solves using iterative refinement. Higham [2021], Higham and Pranesh [2021] argue for the usage of blocked arithmetic to reduce roundoff errors, which we demonstrate has considerable advantages in Figure 1b. Indeed, lower precision arithmetic has made its way directly into kernel approximations. However, these approaches have primarily either been structured kernel basis function based like Fastfood [Le et al., 2013, Yang et al., 2015] or quantization based using random fourier features [Zhang et al., 2019, Li and Li].

4 METHODS

Our methodology combines two sets of approaches. First, it expands upon the ideas of Gardner et al. [2018], Wang et al. [2019] for translating the problem of training GPs into one that only requires on matrix-vector operations. Second, it increases numerical stability via scale translation, mixed-precision aggregation and reorthogonalizaton.

4.1 HALF PRECISION MATRIX VECTOR MULTIPLIES

Following Wang et al. [2019], Meanti et al. [2020], we exploit KeOps Charlier et al. [2021] to produce our *matrix-*



(a) Time. (b) Accuracy.

Figure 1: **(a)** Timings for a single matrix vector multiplication on electric. Half is consistently about twice as fast as single precision. **(b)**Accuracies compared to the float precision for various schemes. Block summation performs each block within half precision, before doing accumulations in half or float and then casting the result back to half, while Kahan summation performs all operations in half. Block summation with float and Kahan summation are nearly as accurate as an entirely float precision MVM.

free approach. For a more detailed explanation of KeOps, please see Charlier et al. [2021], Feydy et al. [2020]. More specifically, to use conjugate gradients effectively, we need to evaluate $\mathbf{K}\mathbf{v}$, which can be structured as

$$\tilde{\mathbf{K}}\mathbf{v}_i = a^2 \sum_{j=1}^{N} k\left(x_i, x_j\right) v_j + \sigma^2 v_i. \tag{3}$$

KeOps performs the inner loop of this multiplication via partitioning the resulting rows and columns of the matrix into blocks $B_1, \cdots B_k$ before reducing the resulting matrices, computing the row-wise operations in parallel. To exploit parallelism on accelerated hardware, we decompose the data into blocks $B_1, \ldots, B_K$ of size $|B_k| = M << N$ (M is the CUDA block size of 192) such that $\bigcup_{k=1}^{K} B_k = \{1, \ldots, N\}$. To compute $\mathbf{K}\mathbf{v}$ we decompose operation into K separate products $\mathbf{K}^{(k)}\mathbf{v}^{(k)}$ where the $\mathbf{K}^{(k)}$ is an $M \times M$ matrix composed of $\mathbf{K}^{(k)}_{i,j} = a^2 k\left(x_i, x_j\right) + \sigma^2 \delta_{i,j}$ for all $i, j \in B_k$ and where $\mathbf{v}^{(k)}_i = \mathbf{v}_i$ for $i \in B_k$. Each separate product $\mathbf{K}^{(k)}\mathbf{v}^{(k)}$ is then computed as a regular matrix vector product (MVM). We clarify that a *matrix-free* approach does involve the creation of block matrices but not of the full matrix $\mathbf{K}$ but only ever requires storage of $M \times M$ matrices in memory at once. For accurate matrix products, we use block summation in a higher precision as is commonly used on specialized GPU hardware [Zamirai et al., 2021, NVIDIA, 2017].

To further exploit parallelism and to reduce memory overhead, we compute all matrix vector products in half precision, rather than floating point precision [Gardner et al., 2018] or double precision. Moving to half precision immediately produces about a 2x speedup compared to half precision, as shown in Figure 1a, as evaluated on dataset sizes successively larger on the 1.8 million data point house electric dataset from the UCI repository [Dua and Graff, 2017].

Algorithm 1 Enhanced Stability CG (blue font denotes differences from standard CG)

Input: MVM function $\mathbf{K}(\cdot)$, initial solution guess $\mathbf{x}_0$, linear system right hand side $\mathbf{b}$, tolerance ϵ, preconditioner function $\mathbf{P}(\cdot)$
Initialize: $k \leftarrow 0$, $\mathbf{r}_0 \leftarrow \mathbf{K}(x_0) - \mathbf{b}$, $\mathbf{d}_0 \leftarrow -\mathbf{r}_0$, $\mathbf{z}_0 = \mathbf{P}(\mathbf{r}_0)$ and $\log \gamma_0 \leftarrow \mathrm{L\Sigma E}\left(\mathbf{r}_0^T \mathbf{z}_0\right)$.
while $\|\mathbf{r}_k\|_2 < \epsilon$ **do**
  $\alpha_k = \exp\left(\log \gamma_k - \mathrm{L\Sigma E}\left(\mathbf{d}_k^T \mathbf{K}(\mathbf{d}_k)\right)\right)$
  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$
  $\mathbf{r}_{k+1} = \mathbf{r}_k + \alpha_k \mathbf{K}(\mathbf{d}_k)$
  for $j = 0$ to k **do**
    $\mathbf{r}_{k+1} = \mathbf{r}_{k+1} - \left(\mathbf{u}_j^T \mathbf{r}_{k+1}\right) \mathbf{u}_j$
  end for
  $\mathbf{z}_{k+1} = \mathbf{P}(\mathbf{r}_{k+1})$
  $\log \gamma_{k+1} = \mathrm{L\Sigma E}\left(\mathbf{r}_{k+1}^T \mathbf{z}_{k+1}\right)$
  $\beta_{k+1} = \exp\left(\log \gamma_{k+1} - \log \gamma_k\right)$
  $\mathbf{d}_{k+1} = -\mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k$
end while

Furthermore, as shown in Figure 1b, these matrix vector multiplications are highly accurate with relative errors averaging less than 0.1% when we perform block summation in floating point precision. By comparison, block summations in half precision is significantly less accurate, while Kahan summation [Kahan, 1965] in half is nearly as accurate as block summation in float. All three approaches have similar speeds as shown in the Appendix. Thus, by moving to half precision we have immediately achieved a nearly 2x speedup in time complexity (and a similar reduction in memory) at negligible accuracy costs.

However, there is a slight issue with the limitations of the range, which can easily cause numerical overflow for large matrix vector multiplications. To see why this is problematic, we note that the largest value representable in half precision is 2^{13}, while the output of a kernel matrix vector multiply scales quasi-linearly with the size of the kernel matrix, N,

$$\begin{aligned}\tilde{\mathbf{K}}(\mathbf{v})_i &= a^2 \sum_{j=1}^{N} k\left(x_i, x_j\right) v_j + \sigma^2 v_i. \\ &\leq N\left(a^2 + \sigma^2\right) \|v\|_\infty .\end{aligned}$$

To get around these numerical overflow issues, we instead downscale every matrix multiply by $N^{-1/2}$, computing instead $\tilde{\mathbf{K}}(\mathbf{v}/N^{1/2})$, which produces an upper bound that scales as $N^{1/2}$ instead. We could try to divide by N instead, but for the largest datasets cannot even store N in half precision!

4.2 HALF PRECISION CONJUGATE GRADIENTS

Rescaling Conjugate Gradients The step sizes of the CG algorithm have a natural interpretation: α_k ensures that

we are minimizing the objective function along the path $\mathbf{x}_k + \alpha_k \mathbf{p}_k$ and β_k guarantees conjugacy between the search directions. The accelerated convergence of CG depends on the correct computation of α_k and β_k, however in finite arithmetic we cannot compute these quantities exactly. Worse, the round-off error gets amplified when using computations in half precision as we compute the step size α_k and β_k terms

$$\alpha_k = \frac{\mathbf{z}_k^T \mathbf{r}_k}{\mathbf{d}_k^T \mathbf{K} \mathbf{d}_k} \quad \text{and} \quad \beta_{k+1} = \frac{\mathbf{z}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{z}_k^T \mathbf{r}_k}.$$

We need to prevent roundoff and overflow error with these terms so we store them solely in the log scale (as they are positive by definition). To do so, we exploit the well known logsumexp trick, which states that for a given vector $\mathbf{w}$ and $\mathbf{z}$ such that $\mathbf{w}^T \mathbf{z} > 0$, to compute $\log \mathbf{w}^T \mathbf{z}$ we use the following transformation $\mathrm{L\Sigma E}(\cdot)$ such that

$$\mathrm{L\Sigma E}\left(\mathbf{w}^T \mathbf{z}\right) = y_{\max} + \log\left(\sum_{i=1}^{N} s_i \exp\left(y_i - y_{\max}\right)\right)$$

where $y_i = \log|w_i| + \log|z_i|$ and $s_i = \operatorname{sign}(w_i z_i)$. For example, we compute and store $\log \alpha_k = \mathrm{L\Sigma E}(\mathbf{r}_k^\top \mathbf{z}_k) - \mathrm{L\Sigma E}(\mathbf{d}_k^\top \mathbf{K} \mathbf{d}_k)$.

Reorthogonalization In infinite precision, each direction vector $\mathbf{d}_j$ for CG is $\mathbf{K}$ orthogonal, that is $\mathbf{d}_i^T K \mathbf{d}_j = 0$ whenever $i \neq j$, producing residual vectors, $\mathbf{r}_k$, that are orthogonal to each other [Nocedal and Wright, 2006]. In practice, orthogonality is lost due to round-off error leading to slower convergence or even divergences in the residual vectors [Gratton et al., 2021, Cutajar et al., 2016]. To prevent these buildups of roundoff errors, we apply explicit Gram-Schmidt reorthogonalization inside the CG algorithm [Gratton et al., 2021]. Specifically, we re-orthogonalize the residual vector for time step k, updating the new residual $\mathbf{r}_k = \mathbf{K}\mathbf{x}_k - \mathbf{b}$ as

$$\mathbf{r}_{k+1} \leftarrow \mathbf{r}_{k+1} - (\mathbf{u}_i^T \mathbf{r}_{k+1}) \mathbf{u}_i$$

for $i = 1, \ldots, k$ and where $\mathbf{u}_i$ are the previous iteration's orthonormal vectors created from the residual $\mathbf{r}_i$. Reorthogonalization comes at an increase memory cost as we must store all previous residuals, adding a memory complexity of $\mathcal{O}(JN)$ for J steps of CG. However, we find that with re-orthogonalization (and reasonably high tolerances) we are able to converge in very few steps, often $J < 50$.

Preconditioning We additionally use preconditioning, specifically using a pivoted cholesky preconditioner [Gardner et al., 2018, Harbrecht et al., 2012]. With preconditioning, we hope to find an operator such that $\mathbf{P}^{-1}\mathbf{K} \approx \mathbf{I}$ with $\mathbf{P}^{-1}(\mathbf{v})$ cheap to compute and solve the system $\mathbf{P}^{-1}\mathbf{K}v = \mathbf{P}^{-1}w$ instead of $\mathbf{K}v = w$.

Following Gardner et al. [2018], we use the pivoted Cholesky decomposition, which requires solely access to the rows of the matrix (e.g. $\mathbf{K}_{i.} = \mathbf{K}e_i$ where e_i is a zero vector with ith entry equal to one) and an approximate diagonal (constant in our case). Running k steps of the pivoted Cholesky decomposition on $\mathbf{K}$ produces an approximation $\tilde{K} \approx \mathbf{L}_k \mathbf{L}_k^\top + \sigma^2 I$ and we can then use the Woodbury matrix identity to construct $\mathbf{P}^{-1}$.

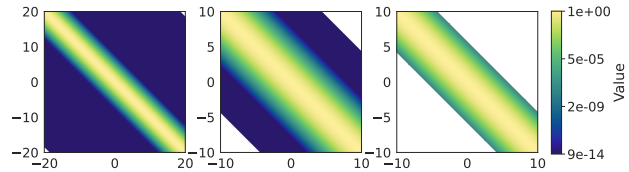


Figure 2: Elementwise logarithms of RBF kernel matrices in double (left), float (middle), and half (right) precisions. Values that are white indicate the kernel matrix is exactly zero at those points, producing a kernel with compact support. A compactly supported kernel produces more efficient kernel matrix MVMs as we can safely remove data points that have distances from a given test point larger than numerical precision.

$$\mathbf{P}^{-1}\mathbf{w} = \sigma^{-2}\mathbf{w} - \sigma^{-4}\mathbf{L}_k \left(\mathbf{I} + \sigma^{-2}\mathbf{L}_k^T \mathbf{L}_k\right)^{-1} \mathbf{L}_k^T \mathbf{w}. \tag{4}$$

The inner system is of size $k << N$ and so can be solved using direct methods (e.g. Cholesky or QR) in float precision as there is not native LAPACK support for these methods in half precision.

To summarize, our revised CG procedure is shown in algorithm 1 with the differences from the BBMM CG algorithm of Gardner et al. [2018] shown in bolded blue font.

4.3 LOSS FUNCTION

Computation of Eq. (1) requires either an expensive eigendecomposition or Lanczos iteration in the forwards pass to compute the log determinant term, which can be quite unstable [Gardner et al., 2018]. Instead, we use a pseudo-loss that has the same gradient as Eq. (2) via solving $\tilde{K}$ against both the response $\mathbf{y}$ and the probe vectors $\mathbf{z}_i$ simultaneously. That is, we find solutions to the linear system using Alg. 1:

$$\mathbf{K}\left[\mathbf{u}_0, \mathbf{u}_1, \cdots, \mathbf{u}_M\right] = \left[\mathbf{y}, \mathbf{z}_1, \cdots, \mathbf{z}_M\right]. \tag{5}$$

for $\mathbf{u}_i$. We then detach these solutions and compute simply:

$$\tilde{\mathcal{L}}(\boldsymbol{\theta}) = \frac{1}{2M}\sum_{j=1}^{M} \mathbf{u}_j^T \left(\mathbf{K}_{\boldsymbol{\theta}} \mathbf{z}_j\right) - \frac{1}{2}\mathbf{u}_0^T \left(\mathbf{K}_{\boldsymbol{\theta}} \mathbf{u}_0\right), \tag{6}$$

which has gradient

$$\nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}} = \frac{1}{2M}\sum_{j=1}^{M} \mathbf{u}_j^T \left(\nabla_{\boldsymbol{\theta}} \mathbf{K} \mathbf{z}_j\right) - \frac{1}{2}\mathbf{u}_0^T \left(\nabla_{\boldsymbol{\theta}} \mathbf{K} \mathbf{u}_0\right), \tag{7}$$

which is the same as Eq. 2. Eq. 7 then only requires gradients with respect to matrix vector products, which are again computed with matrix vector products (and is done natively in KeOps). (7) does not use variance reduction strategies such as Wenger et al. [2021] but this can be straightforwardly included in the future.

5 EXPERIMENTS

Our experiments are split into two parts. In Section 5.1, we empirically study the effect of half precision on kernel matrices, finding that reducing the precision increases sparsity. In Section 5.2, we study the effect of reducing the precision on our stable CG solver, while demonstrating signficant speedups on large-scale UCI benchmarks with greater than $100,000$ data points.

5.1 HALF PRECISION KERNEL MATRICES

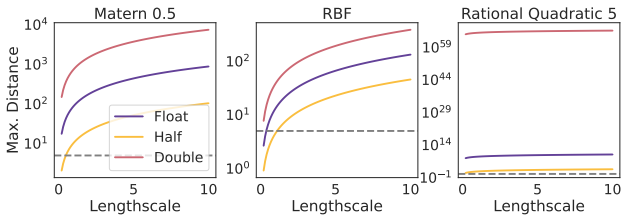
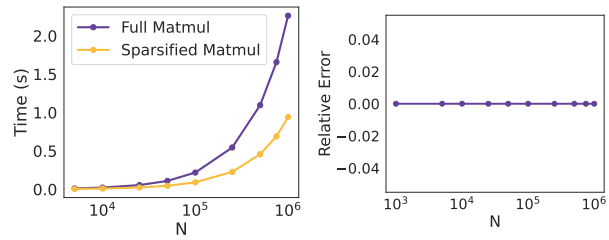


Figure 3: Maximum representable distance across precisions for Matérn-$1/2$, RBF, and rational quadratic (5) kernels. Shown as a gray line for reference is 5 units of distance. RBF kernels have significantly shorter representable distances, even for reasonably large lengthscales, while rational quadratic kernels have by far the highest especially in double and float.

Our first set of experiments attempts to understand and quantify the differences between standard kernel matrices various precisions. We find that lower precision kernel matrices are sparser leading to potential speedups, but have more slowly converging eigenvalues potentially impacting generalization.

First, in Figure 2, we display the elementwise logarithms of RBF kernel matrices of size 500 as we lower the precision from double (left) to float (middle) to half precision. The elementwise logarithm allows us to see that there are significantly more values of the kernel matrix represented as *exactly zero* and thus displayed as white for the half precision kernels than for the higher precisions. For the half and floating point kernels, we consider data evenly spaced in $[-10, 10]$ while for double we consider data in $[-20, 20]$.

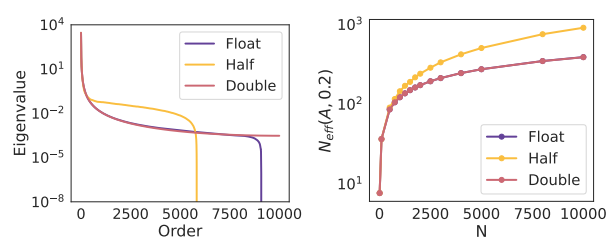
We can exactly quantify the range of support of common stationary kernel matrices for a given lengthscale. For example, for a Matérn-$1/2$ kernel with lengthscale, l, the kernel



(a) Time of MVMs. (b) Error of sparsified MVM.

Figure 4: **(a)** Time for both the full and truncated MVMs as the dataset size grows. The truncated MVM grows significantly less fast than the full MVM because it only operates on one quarter of the full matrix. **(b)** Error of the truncated MVM is zero as expected. To produce the sparsified MVM, we truncated any data points that had kernel entries that were un-representable in half precision.

function will be exactly zero when the $k(x, x') = k(d) = \exp\{-d/l\} < \epsilon$ (satisfied when $d > -\log \epsilon/l$) where ϵ is the smallest representable value in that given precision.[1] Thus, if we have $d = 0.75$ (equivalent to $x = -0.5$ and $x' = 0.25$), then we cannot represent a relationship in half precision between the two points for lengthscales greater than about $9.24 \approx -\log \epsilon/0.75$. We show the maximum distance representable in double, half, and float precisions for several kernels with various lengthscales in Figure 3, displaying 5 units (a proxy for the maximum distance of input data standardized to have mean 0 and variance 1) as a gray dashed line for reference. RBF kernels have the smallest support, especially for short lengthscales, followed by Matérn kernels; rational quadratic kernels have high support.



(a) Eigenvalue spectrum. (b) Effective dimensionality.

Figure 5: **(a)** An observed eigenvalue spectrum for a Matérn-$1/2$ kernel across precisions; the half evaluated kernel has a much more slowly decaying spectrum than either float or double. **(b)** Effective dimensionality of the matrix is considerably higher for the half precision kernel than for either float or half as the matrix size increases. ED is higher because the intermediate eigenvalues are higher in half.

These representations of data points that are exactly zero produces a kernel that has compact support in practice, despite theoretically having infinite support. Indeed, we demon-

[1] Equations for other kernels are given in the Appendix.

strate the potential for this type of structure exploitation when computing GP predictive means in Figure 4; first, we compute the predictive mean cache $\mathbf{v} = \tilde{\mathbf{K}}^{-1}\mathbf{y}$ before either computing a full predictive mean $\mu_{\text{full}} = K(x^*, \mathbf{X})\mathbf{v}$ or a truncated predictive mean by dropping all data points that will be represented as zero in half precision, e.g. $\mu_{\text{trunc.}} = K(x^*, \mathbf{X}_{\text{close}})\mathbf{v}_{\text{close}}$, which drops approximately $3/4$ of the data in this example. As expected, the error is zero, indicating the predictive mean does not change at all!

We next investigate the eigenvalue spectrum across precisions, considering a Matérn-$1/2$ kernel and data drawn from $[-3, 3]$ displying the empirical spectrum in Figure 5a. The eigenvalues of the half precision kernel tend to decay slower before dropping off completely. However, the maximum estimated eigenvalue is the same across precisions (see Appendix) indicating the the condition number of the linear system is approximately the same across precisions as the smallest eigenvalue is σ^2. While many bounds on the convergence of CG depend on the condition number, there are tighter alternative bounds for the convergence of CG that depend on the internal eigenvalues, see Thm 5.5 of Nocedal and Wright [2006], that describe the behavior of CG to converge from the "outside in" alongside the spectrum (that is implicitly estimating the largest and smallest eigenvalues first) [Demmel, 1997]. That is, the more slowly decaying eigenvalue spectrum of the half precision kernel will tend to require more iterations of CG to solve.

Furthermore, the differing spectrum of the kernel matrix across precisions and in particular the slower decay can also play a role in the generalization capabilities of the method itself. Here, there is a clear metric, the effective dimensionality of the kernel matrix, defined as $N_{\text{eff}}(A, \sigma^2) := \sum_{i=1}^{N} \frac{\lambda_i}{\lambda_i + \sigma^2}$, which plays a significant role in generalization bounds for kernel learning [Zhang, 2005, Caponnetto and Vito, 2007]. We choose $\sigma^2 = 0.2$ (although the general trend is robust to σ^2) and find that as N increases, the effective dimensionality increases much faster for the half precision kernel as shown in Figure 5b, indicating half precision may have worse generalization performance.

5.1.1 What about Direct Methods?

A natural question to ask is to see if a direct method such as a Cholesky factorization can solve the linear system. However, support for Cholesky factorizations in half precision is not included in GPU based linear algebra libraries (LAPACK) and approaches for half precision Cholesky factorizations use iterative techniques on the backend [Higham and Pranesh, 2021]. We attempted to mock the effectiveness of direct methods in solving linear systems: by using Eq. 4 to solve linear systems on the UCI *protein* dataset in half precision, varying the size of the preconditioner directly, showing the results in Figure 7. The solve produces residual norms vastly greater than 1 with preconditioners greater than $30,000$ failing.

5.2 BENCHMARK RESULTS (UCI)

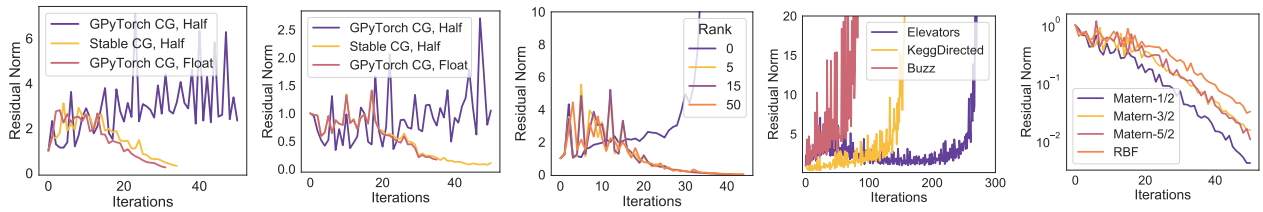
We compare directly to floating point precision results using KeOps and GPyTorch on benchmarks of up to 1.8 million data points from the UCI repository [Dua and Graff, 2017] using RBF ARD kernels and a single NVIDIA V100 GPU with results for other kernels shown in the Appendix. We find that a mixed precision strategy of using solely half precision matrix vector multiplications produces speedups of at least two times over an equivalent floating point precision model.

Residual Norms First, we consider the relative solve error on three different datasets of varying size: elevators, kegg directed, and buzz for hyperparmeters collected at the end of the optimization runs. We find that our solver typically converges to a residual tolerance less than 0.5 in less than 50 optimization steps, as re-orthogonalization and preconditioning make the system quite easy to solve, as shown in Figures 6a for *elevators* and 6b for *keggDirected*, finding that running CG in half precision diverges very quickly. However, our stable CG implementation reaches a similar tolerance as float precision, but requires a few more MVMs. Furthermore, we show the effect of preconditioning on Buzz for no preconditioner, a rank 50 preconditioner and a rank 50 preconditioner, finding that preconditioning is necessary to reach low residual errors in Figure 6c. Similarly to Maddox et al. [2021], we find that the size of the preconditioner has minimal effect on solve tolerance, but that it is needed for convergence.

However, despite preconditioning, the effect of low precision does produce round-off errors that grow with the iteration number, preventing us from running our method with many CG iterations or with low tolerances (at train time at least). We show the effect across three datasets in Figure 6d, suggesting that further work needs to be done to encourage convergence. Finally, we show the results across Matérn kernels with varying ν and RBF kernels without ARD on *pol* in Figure 6e, finding that Matérn-$1/2$ kernels tend to converge fastest.

Optimization Trajectories Next, in Figure 8, we display the evolution of the three hyperparameters (noise, outputscale, and lengthscale) on *protein* with the same plot shown for *3droad* in Figure A.5. We see that the solver produces accurate enough solves so that gradients are unaffected and the hyper-parameters train as stably as they do with the GPyTorch solves in floating point precision.

Benchmark Results Finally, in Table 2 we display the root mean square errors (RMSEs), negative log likelihoods (NLLs), and fitting times for both float and half precision over 5 seeds, displaying the standard deviation. We see



(a) Elevators (b) *KeggD* (c) *Buzz* (d) Tolerances (e) *Pol*

Figure 6: **(a,b)** Residual norms on *elevators* and *keggD*; standard half precision CG fails while our approach converges in a similar number of iterations to float CG. **(c)** Residual norms using our CG solver on *buzz*; using preconditioning prevents divergences due to roundoff errors. **(d)** Running CG for too many iterations produces roundoff errors that build up over time producing divergences. **(e)** Matérn-$1/2, 3/2$ kernels tend to converge quicker than RBF kernels on *pol*.

Table 1: RMSEs, NLL, and training time across 3 seeds on a suite of UCI tasks. Here, we use RBF ARD kernels with 50 CG iterations and 50 optimization steps. We included KeOps compilation times which plays a significant timing role for the smaller datasets and observe that on Song, half precision requires about $1/3$ of the time with a significant improvement on House Electric as well. $*$Float did not converge on Buzz so we use the timings for 8 GPUs from Wang et al. [2019].

		RMSE		**NLL**		**Time**	
Dataset	(N, d)	**Single**	**Half**	**Single**	**Half**	**Single**	**Half**
PoleTele	(13.5K, 26)	0.117 ± 0.004	0.121 ± 0.003	-0.349 ± 0.004	-0.316 ± 0.004	57.6 ± 0.6	88.3 ± 3.2
Elevators	(14.9K, 18)	0.364 ± 0.004	0.382 ± 0.001	0.515 ± 0.0195	0.663 ± 0.024	58.4 ± 3.1	90.1 ± 2.9
Bike	(15.6K, 17)	0.074 ± 0.003	0.083 ± 0.009	-0.3714 ± 0.0066	-0.413 ± 0.008	62.3 ± 0.7	85.6 ± 0.41
Kin40K	(36K, 8)	0.099 ± 0.001	0.100 ± 0.003	0.2352 ± 0.005	0.241 ± 0.005	65.5 ± 5.3	92.9 ± 0.28
Protein	(41.1K, 9)	0.055 ± 0.006	0.635 ± 0.002	0.9802 ± 0.0115	1.412 ± 0.001	70.9 ± 5.7	115.7 ± 0.10
3droad	(391.4K, 3)	0.194 ± 0.01	0.215 ± 0.004	1.249 ± 0.0129	1.201 ± 0.005	1260 ± 35	$1{,}003.2 \pm 0.97$
Song	(463.8K, 90)	0.761 ± 0.004	0.779 ± 0.022	1.146 ± 0.0043	1.765 ± 0.819	24357 ± 2613	$9{,}930.1 \pm 129.7$
Buzz	(524.9K, 77)	–	0.454 ± 0.184	–	0.898 ± 0.714	20750^*	$23{,}030.6 \pm 1{,}675.3$
HouseElectric	(1844.3K, 9)	0.052 ± 0.002	0.052 ± 0.003	-0.72 ± 0.002	-0.439 ± 0.084	42751 ± 2180	$35{,}966.1 \pm 1{,}346.3$

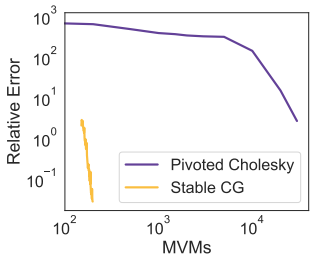


Figure 7: Failure of a pivoted cholesky approach to solve a linear system in half precision. CG, including preconditioner MVMs, solves the system more efficiently and accurately.

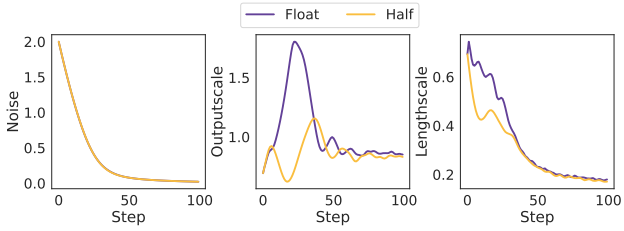


Figure 8: Optimization and parameter trajectories on *protein* for both a standard float precision model and our half precision implementation. The noise term (left) and lengthscale (right) are very close for both precisions, while the outputscale (center) shows some minor differences.

that it takes up to data points of $100,000$ for there to be a speedup from using half precision, a fact we attribute to increased compilation times in KeOps in half. However, for song and houseelectric, the two largest datasets we see significant speedups (of up to 3x) for the same number of optimization steps. Interestingly, we were unable to get buzz to converge in float but were in half, suggesting that lower precision might help somewhat with regularization.

6 CONCLUSION

We have demonstrated that reduced precision has considerable benefits for CG based Gaussian processes due to the reduced memory and computational gains, especially for larger datasets exceeding $100,000$ points. While a reduced range and roundoff error present significant challenges in low precision arithmetic, we show that pleasingly simple strategies such as re-orthogonalization and the logsumexp trick can largely alleviate these issues. In the future, we hope to push to increasingly lower precisions [Yang et al., 2021] further reducing the memory overheads by exploiting the properties of stochastic roundoff error [Higham and Mary, 2019], and extending our mixed precision approach to other iterative methods used for iterative GPs such as Lanczos [Gardner et al., 2018] and MINRES [Pleiss et al., 2020].

References

A. Abdelfattah, S. Tomov, and J. Dongarra. Investigating the benefit of fp16-enabled mixed-precision solvers for symmetric positive definite matrices using gpus. In *ICCS*, 2020.

A. Caponnetto and E. D. Vito. Optimal rates for the regularized least-squares algorithm. *FOCM*, 2007.

B. Charlier, J. Feydy, J. Glaunès, F.-D. Collin, and G. Durif. Kernel operations on the gpu, with autodiff, without memory overflows. *JMLR*, 2021.

J. Chen, N. Cao, K. H. Low, R. Ouyang, C. K.-Y. Tan, and P. Jaillet. Parallel gaussian process regression with low-rank covariance matrix approximations. *arXiv:1305.5826*, 2013.

Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. Dadiannao: A machine-learning supercomputer. In *IEEE/ACM International Symposium on Microarchitecture*, 2014.

K. Cutajar, M. Osborne, J. P. Cunningham, and M. Filippone. Preconditioning Kernel Matrices. *ICML*, 2016.

D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, and E. Georganas. Mixed Precision Training of Convolutional Neural Networks using Integer Operations. *arXiv:1802.00930*, 2018.

J. W. Demmel. *Applied numerical linear algebra*. SIAM, 1997.

D. Dua and C. Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

J. Feydy, A. Glaunès, B. Charlier, and M. Bronstein. Fast geometric learning with symbolic matrices. *NeurIPS*, 2020.

J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. *NeurIPS*, 2018.

M. G. Genton. Classes of kernels for machine learning: a statistics perspective. *JMLR*, 2001.

M. Gibbs and D. J. MacKay. Efficient implementation of gaussian processes. Technical report, 1997.

T. Gneiting. Compactly supported correlation functions. *Journal of Multivariate Analysis*, 2002.

G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, 2018. Fourth Edition.

S. Gratton, S. E., and P. L. Titley-Peloquin D., Toint. Minimizing convex quadratics with variable precision conjugate gradients. *Numer Linear Algebra Appl.*, 2021.

S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. *ICML*, 2015.

A. Haidar, P. Wu, S. Tomov, and J. Dongarra. Investigating half precision arithmetic to accelerate dense linear system solvers. In *8th workshop on latest advances in scalable algorithms for large-scale systems*, 2017.

A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18*, pages 603–613, 2018.

D. Hammerstrom. A vlsi architecture for high-performance, low-cost, on-chip learning. In *IJCNN*, 1990.

H. Harbrecht, M. Peters, and R. Schneider. On the Low-Rank Approximation by the Pivoted Cholesky Decomposition. *Applied Numerical Mathematics*, 2012.

J. Hensman, N. Fusi, and N. D. Lawrence. Gaussian Processes for Big Data. *UAI*, 2013.

M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 1952.

N. J. Higham. Numerical stability of algorithms at extreme scale and low precisions. *Manchester Institute for Mathematical Sciences*, 2021.

N. J. Higham and T. Mary. A new approach to probabilistic rounding error analysis. *SIAM journal on scientific computing*, 41(5):A2815–A2835, 2019.

N. J. Higham and S. Pranesh. Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. *SIAM Jour. on Scientific Computing*, 2021.

Intel. bfloat16 - Hardware Numerics Definition. Technical report, 2018. URL `https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf`.

B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CVPR*, 2018.

W. Kahan. Further Remarks on Reducing Truncation Errors, Commun. *Assoc. Comput. Mach, 8:40*, 1965.

W. Kahan. IEEE standard 754 for Binary Floating-Point Arithmetic. *Lecture Notes on the Status of IEEE*, 1996.

Q. Le, T. Sarlós, A. Smola, et al. Fastfood-approximating kernel expansions in loglinear time. In *ICML*, 2013.

X. Li and P. Li. Quantization algorithms for random fourier features. In *ICML*. PMLR.

W. J. Maddox, S. Kapoor, and A. G. Wilson. When are iterative gaussian processes reliably accurate? *arXiv preprint arXiv:2112.15246*, 2021.

G. Meanti, L. Carratino, L. Rosasco, and A. Rudi. Kernel methods through the roof: handling billions of points efficiently. *NeurIPS*, 2020.

P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, and G. Venkatesh. Mixed Precision Training. *arXiv:1710.03740*, 2017.

D.-T. Nguyen, M. Filippone, and P. Michiardi. Exact gaussian process regression with distributed computations. In *ACM/SIGAPP Symposium on Applied Computing*, 2019.

J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2006.

NVIDIA. Nvidia tesla v100 gpu architecture. Technical report, Technical Report WP-08608. http://www. nvidia. com/object/volta-architecture . . . , 2017.

G. Pleiss, M. Jankowiak, D. Eriksson, A. Damle, and J. R. Gardner. Fast Matrix Square Roots with Applications to Gaussian Processes and Bayesian Optimization. *NeurIPS*, 2020.

C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. 2008.

A. J. Smola and B. Schölkopf. Sparse Greedy Matrix Approximation for Machine Learning. *ICML*, 2000.

M. K. Titsias. Variational Learning of Inducing Variables in Sparse Gaussian Processes. *AISTATS*, 2009.

K. A. Wang, G. Pleiss, J. R. Gardner, S. Tyree, K. Q. Weinberger, and A. G. Wilson. Exact Gaussian Processes on a Million Data Points. *NeurIPS*, 2019.

S. Wang and P. Kanwar. BFloat16: The secret to high performance on Cloud TPUs, 2019. URL `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus/`.

D. S. Watkins. *Fundamentals of Matrix Computations*. Wiley, 2010. Third Edition.

J. Wenger, G. Pleiss, P. Henning, J. P. Cunningham, and J. R. Gardner. Reducing the Variance of Gaussian Process Hyperparameter Optimization with Preconditioning. *arXiv:2107.00243*, 2021.

C. Williams and M. Seeger. Using the Nyström Method to Speed Up Kernel Machines. *NeurIPS*, 2000.

C. Yang, Z. Wu, J. Chee, C. De Sa, and M. Udell. How Low Can We Go: Trading Memory for Error in Low-Precision Training. *arXiv:2106.09686*, 2021.

G. Yang, T. Zhang, P. Kirichenko, J. Bai, A. G. Wilson, and C. De Sa. SWALP: Stochastic Weight Averaging in Low Precision Training. *ICML*, 2019.

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

P. Zamirai, J. Zhang, C. R. Aberger, and C. De Sa. Revisiting BFloat16 Training. *arXiv:2010.06192*, 2021.

J. Zhang, A. May, T. Dao, and C. Re. Low-precision random fourier features for memory-constrained kernel approximation. In *AISTATS*, 2019.

T. Zhang. Learning bounds for kernel regression using effective data dimensionality. *Neural Computation*, 2005.

A EXTENDED RELATED WORK

Conjugate Gradients: A description of the conjugate gradients algorithm is given in Alg. 2 while using preconditioning [Nocedal and Wright, 2006, Golub and Loan, 2018]. Gardner et al. [2018] propose a variant of conjugate gradients that they call modified batched CG (mBCG) which we use in our work. The primary difference between mBCG and CG is that mBCG enables solving several linear systems at once by performing all computations in batch mode so that linear operators such as $\mathbf{K}(\mathbf{v})$ are actually matrix matrix multiplications rather than matrix vector products. Then, an individual set of learning rates α_k and β_k is used for each system. Our stable CG implementation (Alg. 1) is actually based off of mBCG, but for didactic purposes we display only the standard CG version.

Algorithm 2 CG

Input: MVM function $\mathbf{K}(\cdot)$, initial solution guess $\mathbf{x}_0$, linear system right hand side $\mathbf{b}$, tolerance ϵ, preconditioner function $\mathbf{P}(\cdot)$
Initialize: $k \leftarrow 0$, $\mathbf{r}_0 \leftarrow \mathbf{K}(x_0) - \mathbf{b}$, $\mathbf{d}_0 \leftarrow -\mathbf{r}_0$, $\mathbf{z}_0 = \mathbf{P}(\mathbf{r}_0)$ and $\gamma_0 = \mathbf{r}_0^T \mathbf{z}_0$.
while $\|\mathbf{r}_k\|_2 < \epsilon$ **do**
  $\alpha_k = \frac{\gamma_k}{\mathbf{d}_k^T \mathbf{K}(\mathbf{d}_k)}$
  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$
  $\mathbf{r}_{k+1} = \mathbf{r}_k + \alpha_k \mathbf{K}(\mathbf{d}_k)$
  $\mathbf{z}_{k+1} = \mathbf{P}(\mathbf{r}_{k+1})$
  $\gamma_{k+1} = \mathbf{r}_{k+1}^T \mathbf{z}_{k+1}$
  $\beta_{k+1} = \frac{\gamma_{k+1}}{\gamma_k}$
  $\mathbf{d}_{k+1} = -\mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k$
end while

Other Scalable Gaussian Processes We note that our matrix-free schemes can be used to scale up approximate kernel methods such as Nystrom style approximations [Smola and Schölkopf, 2000, Williams and Seeger, 2000]; indeed, Meanti et al. [2020] use Nystrom approximations and KeOps for their kernel ridge regression approach. However, Zhang et al. [2019] found limited speedups when quantizing (which is slightly distinct from half precision) Nystrom approximations. Similarly, Pleiss et al. [2020] used iterative methods (in their cast MINRES) to speed up variational Gaussian processes [Titsias, 2009, Hensman et al., 2013], and we hope to speed up their approach as well. Chen et al. [2013], Nguyen et al. [2019] proposed parallel direct Cholesky based GP schemes for more scalable GP regression; however, their approaches will probably perform poorly in lower precision, as we demonstrate is the case for pivoted Cholesky based solves in Section 5.1.1.

Finally, kernels with compact support have been previously studied from a kernel approximation point of view [Genton, 2001, Gneiting, 2002]. However, these works focused on developing new techniques to approximate an infinitely supported kernel with a kernel that has demonstrated compact support, rather than using floating point precision to develop an approximate kernel with compact support.

B MAXIMUM DISTANCE REPRESENTABLE

To compute these distances, we consider four separate stationary kernels [Rasmussen and Williams, 2008] with distance $d = |x - x'|$ and lengthscale l, focusing on determining what values they drop below a given ϵ. For Matérn-$1/2$ kernels, we need $\exp\{-d/l\} < \epsilon$ and solving gives $d > -\log \epsilon * l$. For other Matérn kernels (e.g. $3/2$ and $5/2$), there is no straightforward closed form solution, but empirical investigations showed that the maximum distance representable is somewhere between Matérn-$1/2$ and RBF.

For RBF Kernels, we have $\exp\{-1/2d^2/l^2\} < \epsilon$ and solving gets $d > (-2\log\epsilon)^{1/2} l$. For rational quadratic kernels, we have $(1 + \frac{d^2}{2\alpha l^2})^{-\alpha} < \epsilon$ and solving for d gets $d > (2\alpha(\epsilon^{1/\alpha} - 1))^{1/2} * l$. We found that for $\alpha = 2, 3$ the size of the support was much larger, and so showed only $\alpha = 5$. For periodic kernels, we have $-\frac{2}{\lambda}\sin^2\left(\frac{\pi}{p}|d|\right) < \log\epsilon$ and solving gets $|d| > \frac{p}{\pi}\arcsin\left(-\log\epsilon\frac{\lambda}{2}\right)$ which will only have solutions when $-\log\epsilon\frac{\lambda}{2} \leq 1$.

C OTHER EXPERIMENTS

In Figure A.1, we display the different times of block summation across precisions, as well as Kahan summation and floating point accumultion of float MVMs, finding that all half precision accumulation mechanisms behave similarly, with Kahan summation being slightly slower than the other two.

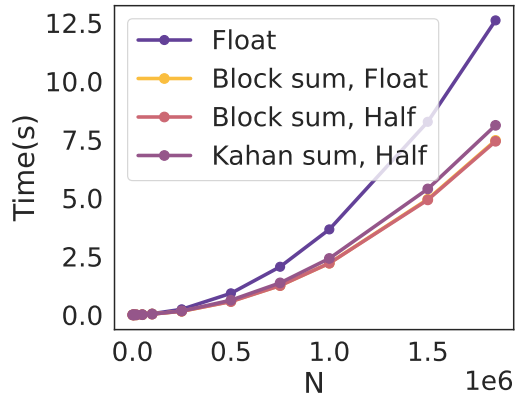


Figure A.1: Block summation in floating point adds negligible overhead compared with block summation in half precision, while being as accurate as Kahan summation.

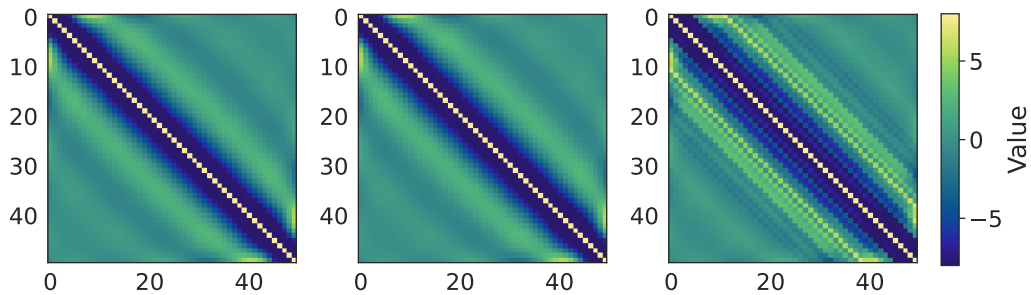
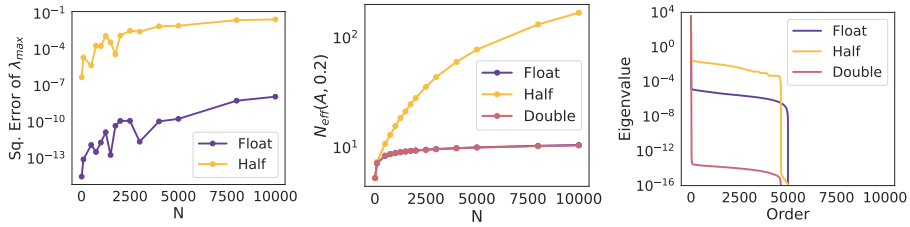


Figure A.2: Matrix inverses of an RBF kernel in double (left), float (middle), and half (right) precisions. The inverse is performed in double precision, while the evaluation itself is performed in half precision. The half precision inverse is qualitatively distinct from the other two indicating a slightly distinct spectrum.

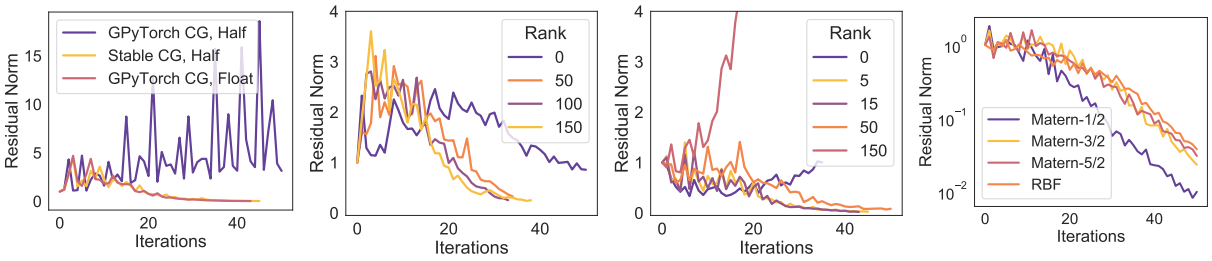
In Figure A.2, we display $(\mathbf{K} + 0.01)^{-1}$ for RBF kernels with lengthscale 1 and 50 data points in $[-3, 3]$ across double (left), float (middle), and half (right) precisions. We first evaluate the kernel to a lower precision and then pass into double precision before using a cholesky factorization to invert the kernel matrix, finding that the half precision kernel inverse has a distinct pattern (larger magnitude off-diagonal values) compared to the float and double inverse matrices.

In Figure A.3**(b)**, we show ED for RBF kernels with the associated spectrum in **(c)**.



(a) Difference of λ_{max}. (b) ED, RBF. (c) Eigenvalue spectrum, RBF

Figure A.3: **(a)** Difference of λ_{max} across precisions for Matérn-$1/2$ kernel. **(b)** ED for RBF kernels, the trend is similar to that of the Matérn-$1/2$ kernel because the eigenvalue spectrum **(c)** has a similar bunched up pattern in half.



(a) Buzz. (b) Elevators. (c) KeggD (d) pol

Figure A.4: **(a)** Residual norms across solvers on buzz. **(b)** Residual norm for stable CG on elevators. Here no preconditioning also converges. **(c)** Residual norm for stable CG on Kegg Directed. **(d)** Residual norm for no ARD on Pol.

In Figure A.4a, we display how CG in half diverges, but our stable CG converges as does CG in float. In Figure A.4b and A.4d, we display the effect of preconditioning on solves, finding again that larger preconditioners tend to converge very slightly faster.

In Figure A.5, we display the optimization trajectory on *3droad* finding that there are clearer divergences in terms of the outputscale; however, each parameter converges to similar values by the end of training.

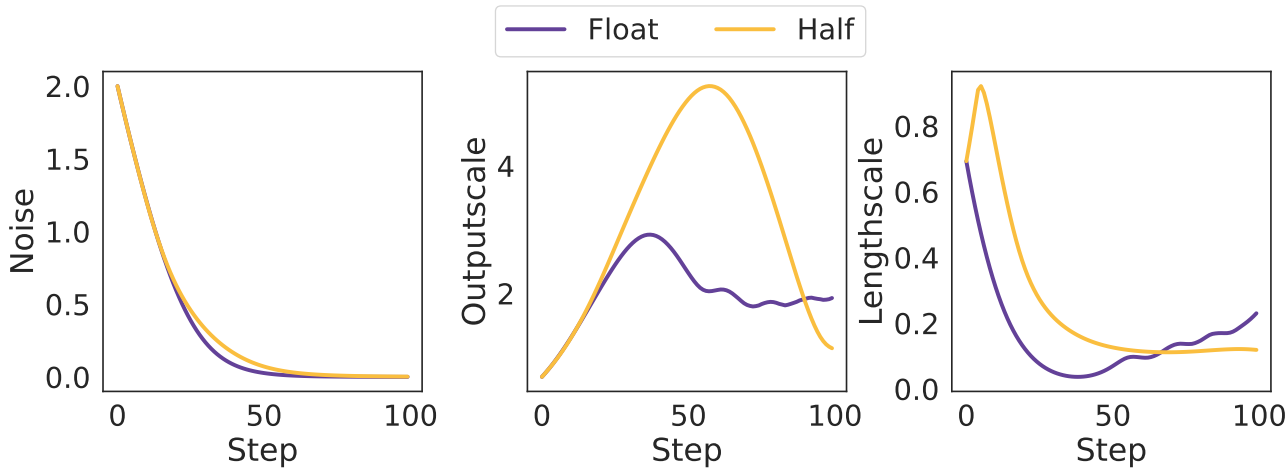


Figure A.5: Optimization trajectory on *3droad*.

Table 2: RMSEs, NLL, and training time across 3 seeds on a suite of UCI tasks. Here, we use Matérn-$5/2$ ARD kernels with 50 CG iterations and 50 optimization steps.

		RMSE		NLL		Time	
Dataset	(N, d)	**Single**	**Half**	**Single**	**Half**	**Single**	**Half**
PoleTele	(13.5K, 26)	0.098 ± 0.002	0.102	-0.454 ± 0.004	-0.447	49.1 ± 1.74	97
Protein	(41.1K, 9)	0.498 ± 0.007	0.509 ± 0.005	1.07 ± 0.01	1.04 ± 0.03	69.17 ± 1.7	110 ± 10
3droad	(391.4K, 3)	0.254 ± 0.006	0.231	0.812 ± 0.014	0.899	1666 ± 214	1501

D DESCRIPTION

All timing based experiments were performed using single NVIDIA V100 GPUs with 32GB of memory on a shared supercomputing cluster. Non-timing experiments also used NVIDIA RTX GPUs with either 24 or 48 GB of memory on either the same cluster or on a private internal server.

We used GPyTorch [Gardner et al., 2018] with the default parameter settings from Botorch's single task GP model[2] which

[2] `https://botorch.org/api/models.html#module-botorch.models.gp_regression`.

are constraining the noise to be greater than $1e-4$ and a Gamma$(1.1, 0.05)$ prior on the noise with initialization to 2, and Gamma$(2.0, 0.15)$ prior on the outpuscale and a Gamma$(3.0, 6.0)$ prior on the lengthscale(s). We fit using Adam for either 50 or 100 iterations unless otherwise documented and used a tolerance of 1.0 for the CG iterations unless otherwise stated.

For the datasets, we used the bayesian benchmarks package of `https://github.com/hughsalimbeni/bayesian_benchmarks/`, following their default training and testing splits.

At test time, we converted the models back to float precision; however, our experiments found that this actually had limited impact on the RMSE and NLLs.