An Asynchronous Multi-GPU Algorithm for Training Kernel Machines

Vijay Giri [1] **Parthe Pandit** [2]

Abstract

Kernel machines are a popular class of estimators commonly used in machine learning and statistics. EigenPro is an iterative and scalable solver for training kernel machines on large datasets exceeding 1 million samples. This algorithm is highly parallelizable and can take advantage of multiple GPUs. However using multiple GPUs poses the challenge of synchronization delays which can plateau speed-ups leading to poor resource utilization. In this paper we propose an improvement, AsyncEigenPro, a parallel and completely lock-free asynchronous algorithm, that is resilient to synchronization delays between multiple GPUs. This algorithm is inspired by Hogwild!, a popular asynchronous alternative to SGD, but exploits the special structure of the kernel regresison problem. Our algorithm enables efficient multi-GPU training for kernel methods. We also rigorously analyze the convergence properties of the algorithm which brings out the effect of delayed gradients. Importantly, through kernel regression, we show that the asynchronous SGD in the overparametereized regime has a faster convergence rate than in the classical regime with a better dependence on the delay. Our large scale numerical experiments on upto 10 million training samples, shows a near-linear speedup in training time with respect to the number of GPUs.

1. Introduction

Deep neural networks (DNNs) are the state-of-the-art models in many machine learning applications today. They can be trained on large scale datasets using graphics processing units (GPUs). However they are challenging to train and require a lot of domain knowledge and heuristics to achieve good performance. Furthermore, tuning hyperparameters, both for modelling and optimization, remains expensive. The prevailing folklore is that it is the overparameterization of DNNs that enables their strong performance. See (19; 20; 9) for example.

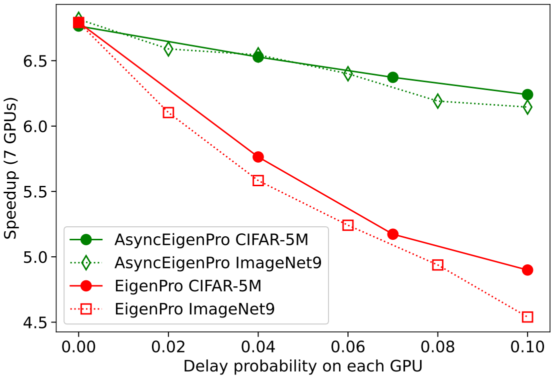

Figure 1. **Degradation due to synchronization delays:** In this experiment we run EigenPro and AsyncEigenPro with 7 GPUs to measure the decrease in speed-ups due to synchronization delays. We add an independent controlled delay at any time instant on every GPU. As the plot shows, AsyncEigenPro is more resilient to synchronization delays. More details in Section 5.

Recent work has shown an equivalence of DNNs in certain regimes to kernel machines, a well-studied class of models with stable training procedures (13; 8). Kernel machines can also surpass the performance of deep networks in certain settings with small and medium scale datasets (4; 11). Furthermore, kernel machines can also be modified to learn features that can enhance the performance of these models (21). This has renewed the interest of the research community to investigate whether kernel machines can be a principled alternative for DNNs.

However, a major challenge for deploying kernel models to practical applications is that the training complexity scales super-linearly with the number of samples. An unresolved question is whether kernel machines trained on large datasets are able to compete with DNNs. To resolve this question, we first need to develop tools that enable training kernel machines over large datasets, while retaining their overparameterization.

Recently, the EigenPro algorithm was proposed as a scal-

[1] Department of Computer Science and Engineering, and [2] Halicioglu Data Science Institute, UC San Diego, USA. Correspondence to: Parthe Pandit <parthepandit@ucsd.edu>.

able iterative solver for training kernel machines (15). This algorithm has a $\Omega(n)$ memory footprint, and a $\Omega(n)$ per iteration complexity, where n is the number of training samples. Furthermore, this algorithm is highly parallel since it only involves pairwise kernel-evaluations and matrix-vector multiplications. Importantly, the algorithm can, in principle, take advantage of multiple GPUs, in a Data-Parallel manner, with a potential to scale linearly with the number of GPUs. This is an attractive property since the V-RAM on a single GPU is limited and remains an expensive resource.

However, when using multiple GPUs to deploy EigenPro for large-data problems, a major challenge is the delay costs due to synchronization between the GPUs. Figure 1, shows the degradation in speed-up due to synchronization delays. Importantly these costs can grow with the number of GPUs and preclude efficient resource utilization at scale.

In the rest of the paper, by EigenPro, we mean EigenPro2 – the improved version presented in (16).

1.1. Main contributions

In this paper we provide a:

1. **New asynchronous algorithm:** for training kernel machines using multiple GPUs with a near-linear speed-up. We provide a PyTorch implementation of our algorithm. Our algorithm has been tested on compute nodes with upto 8 GPUs, but has the potential to scale beyond to multi-node multi-GPU setups.

2. **Convergence analysis:** We provide a rigorous analysis for the convergence of our algorithm. Asynchronous alternatives to SGD such as Hogwild! and its variants have been well studied in the classical setting. Typically both SGD and asynchronous SGD in the classical regime need an annealed learning rate. However, in overparameterized settings, SGD possesses an important property called *variance reduction for free* (VRF), which allows exponential convergence of SGD with a fixed learning rate. We show that asynchronous SGD in the overparameterized case has better convergence rate compared to the classical regime. Our analysis shows that the effect of delay is sublinear for this problem unlike the quadratic effect in the classical regime. A summary of comparative results is provided in Table 1.

3. **Large-scale experiment:** We train kernel machines in the overparameterized setting with upto 10 million training samples. To the best of our knowledge, kernel machines have not been trained at this scale before, without using an approximate model.

Regime	SGD	Async-SGD
Classical	(6)	(17)
learning rate	annealed	fixed, $O(1/\mathsf{delay}^2)$
error	$O(1/\mathsf{T})$	$\widetilde{O}(1/\mathsf{T})$
Overparam.	(14)	Theorem 1
learning rate	fixed, $O(1)$	fixed, $O(1/\sqrt{\mathsf{delay}})$
error	$O(\exp(-\mathsf{T}))$	$\widetilde{O}(1/\mathsf{T}^2)$

Table 1. Summary of convergence analyses of SGD, and asynchronous SGD for linear regression in the classical regime and in the overparameterized regime. While the classical regime requires an annealed learning rate, in the overparameterized regime a fixed learning rate SGD can converge due to the *variance reduction for free (VRF)* phenomenon from (14). The results also suggest that overparameterization helps in asynchronous case too but not to the same effect as synchronous. Here T is the number of iterations .

1.2. Prior work

EigenPro was derived in (15) as a preconditioned stochastic gradient descent, with a space complexity and computational complexity of $O(mn)$ for n training samples and a batch size of m. The preconditioner is designed to ensure maximal GPU memory utilization. The first version of EigenPro required a setup cost of $O(nq^2)$ where q is the level of the preconditioner, and an additional per iteration cost of $O(mn)$ for preconditioning. The second version in (16) employed a Nyström approximation (27) to find a preconditioner that reduced the setup cost to $O(sq^2)$ and per iteration cost to $O(sm)$, for $q \ll s \ll n$. This made the preconditioning step scalable at each iteration. With this the per iteration complexity is $O(m(n+s))$. We provide a detailed description of the mechanics of the EigenPro algorithm in Section 2.1

Solving the above system of linear equations using matrix inversion scales as $O(n^3)$ making it prohibitively expensive for large-scale applications. Thankfully, the problem possesses a lot of structure which enables iterative algorithms to yield solutions in a scalable manner.

Other iterative kernel solvers: A few well-known iterative solvers are PEGASOS (primal Kernel-SVM), Kernel gradient descent (or Richardson iteration (23)), NYTRO (early stopping + Nyström subsampling), and FALKON (NYTRO+preconditioning). Kernel matrices are typically poorly conditioned, which necessitates preconditioning to speed-up iterative training. While there are more versatile solvers such as GPYTORCH and GPFLOW which can minimize a variety of loss functions, they typically do not scale to datasets with 100,000+ samples.

Our work is most closely related to EigenPro2, which is yet another iterative solver which applies preconditioning. A more detailed review of EigenPro2 is provided in Section 2. Perhaps the most significant difference between EigenPro

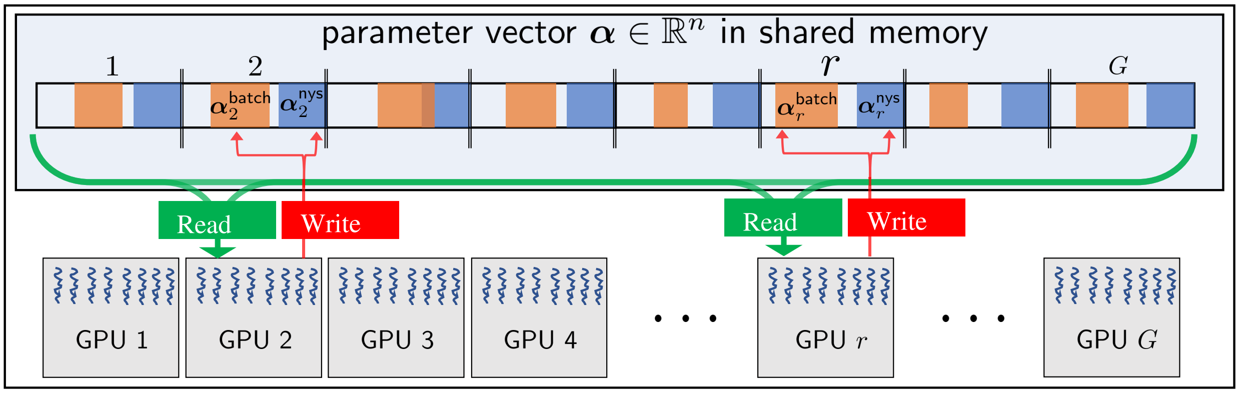
parameter vector $\boldsymbol{\alpha} \in \mathbb{R}^n$ in shared memory
1
2
r
G
$\boldsymbol{\alpha}_2^{\text{batch}}$
$\boldsymbol{\alpha}_2^{\text{nys}}$
$\boldsymbol{\alpha}_r^{\text{batch}}$
$\boldsymbol{\alpha}_r^{\text{nys}}$
Read
Write
Read
Write
GPU 1
GPU 2
GPU 3
GPU 4
GPU r
GPU G

Figure 2. Schematic of AsyncEigenPro (Algorithm 1). In each iteration, every GPU reads the entire parameter vector asynchronously from shared memory (line 8 from Algorithm 1), and writes asynchronously to a partition of the parameter vector (line 10 from Algorithm 1). The coordinates of the subvector $\boldsymbol{\alpha}_r^{\text{nys}}$ correspond to the indices of the data samples in the Nyström approximating subset X_r^{nys}, whereas the coordinates of the subvector α_r^{bat} coresond to the indices of the data samples in the minibatch X_r^{bat} for the iteration. In every iteration the location of the batch update changes, but the Nyström update location remains the same.

and FALKON is that the EigenPro predictor has n degrees of freedom and is well defined even for kernel ridgeless case, whereas FALKON have an approximate model with $\ll n$ degrees of freedom and necessarily require a non-zero ridge parameter. Hence FALKON solvers are unstable to solve the kernel interpolation problem.

Recently, EigenPro3 was proposed in (1), for learning so-called general kernel models. The rate determining step in EigenPro3 is a projection subproblem that applies EigenPro2. Hence faster alternatives to EigenPro2, such as ours, benefit EigenPro3.

Distributed kernel regression: Distributed approaches kernel regression has also been studied using the random features approximation in (12). There have been several other approaches to solving the kernel regression problem given in equation (4) in a distributed manner, (see (18)). However these algorithms rely on a matrix inverse or equivalent operations which makes them (i) unable to scale to large models, and (ii) unstable for the kernel interpolation problem (no ridge regularization). Distributed kernel regression has been considered in (28) and several follow-ups. However akin to EigenPro our training algorithm does not approximate the kernel model.

Asynchronous SGD analysis with delays: Hogwild! (22) is a well-known algorithm for applying SGD with asynchronous updates from multiple processors into a single parameter in shared memory. This algorithm is particularly useful when SGD iterates are sparse. It's analysis is further simplified using the perturbed iterate framework (17). We apply the theoretical frameworks from these works to analyze our algorithm's convergence properties. Generic analysis for SGD under delayed updates and slightly different environment has also been considered in (2).

2. Problem Formulation and Background

Kernel machines are models of the form

$$\widehat{f}(x) = \sum_{i=1}^{n} \alpha_i^* K(x_i, x) \tag{1}$$

where α_i^* are learnable parameters, x_i are training data, and $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a positive definite kernel function. See (3; 26) for an in-depth review of kernel machines and its applications to machine learning.

Given a set of labelled training data $(X, Y) = \{(x_i, y_i)\}_{i=1}^n \in \mathbb{R}^d \times \mathbb{R}^k$, we wish to find an interpolating predictor from an RKHS $\mathcal{H}$ corresponding K,

$$\widehat{f} = \operatorname*{argmin}_{f} \|f\|_{\mathcal{H}} \qquad \text{subject to } f(X) = Y, \tag{2}$$

where the constraint means $f(x_i) = y_i$ for all i. One can show that this estimator is equivalent to the following kernel ridge-less regression with ridge penalty parameter $\lambda = 0^+$,

$$\widehat{f}_\lambda = \operatorname*{argmin}_{f \in \mathcal{H}} \frac{1}{2n} \sum_{i=1}^{n} \|f(x_i) - y_i\|^2 + \frac{\lambda}{2} \|f\|_{\mathcal{H}}^2 \tag{3}$$

Due to the representer theorem of (10) and (25) we know that the optimal solution has the form in equation (1).

Furthermore, $\boldsymbol{\alpha}^* = (\alpha_i^*) \in \mathbb{R}^{n \times k}$, with $\alpha_i^* \in \mathbb{R}^k$, is the solution to the linear system of equations

$$(\mathrm{K}(X, X) + \lambda I_n)\boldsymbol{\alpha}^* = Y \tag{4}$$

where $Y = (y_i) \in \mathbb{R}^{n \times k}$. Thus learning a kernel machine is simply solving the above $n \times n$ linear system of equations.

In this paper we give an algorithm to solve for $\boldsymbol{\alpha}^*$ with a highly parallelized mechanism using multiple GPUs operating asynchronously.

Notation: For any function f, and a set $X = \{x_i\}$, we mean by $f(X)$ the vector of stacked evaluations of f, i.e., $[f(X)]_i = f(x_i)$. Similarly, for kernel functions K, and sets X, Z, by $\mathrm{K}(X, Z)$ we mean the matrix of pairwise kernel evaluations $[\mathrm{K}(X, Z)]_{ij} = K(x_i, z_j)$.

A concept used often in what follows is that of the top-q eigensystem of a positive definite matrix. Formally,

Definition 1 (Top-q eigensystem)**.** Let $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_s$, be the eigenvalues of a symmetric positive definite matrix $\mathbf{K} \in \mathbb{R}^{s \times s}$, i.e., for unit-norm $\boldsymbol{e}_i$, we have $\mathbf{K}\boldsymbol{e}_i = \lambda_i \boldsymbol{e}_i$. We call the tuple $(\Lambda, \mathbf{E}, \lambda_{q+1})$ the top-q eigensystem, where

$$\Lambda := \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_q) \in \mathbb{R}^{q \times q}, \text{ and}$$
$$\mathbf{E} := [\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_q] \in \mathbb{R}^{s \times q}.$$

The computational complexity of finding the top-q eigensystem is $O(sq^2)$.

We now describe the mechanics of the EigenPro algorithm. Our algorithm is an improvement to EigenPro.

2.1. Mechanics of EigenPro

EigenPro is a $O(n)$ space algorithm for training Kernel machines. The first version in (15) used preconditioned stochastic gradient descent to solve problem (3) efficiently, with a mini-batch size of m. The preconditioner is derived by flattening the eigenvalues corresponding to the first q eigenspaces of $\mathbf{K}$.

A further improvement to EigenPro was proposed in (16) using a Nyström extension for constructing the preconditioner. This improvement drastically reduced the per iteration complexity of preconditioning from $O(nmq)$ to $O(smq)$, where $s \ll n$ is the size of the Nyström approximating subset $X_{\mathsf{nys}} \subset X$.

In each iteration of EigenPro, a minibatch $X^{\mathsf{bat}}, Y^{\mathsf{bat}}$ is sampled from X, Y, and a stochastic gradient is obtained for this minibatch,

$$\boldsymbol{g}^{\mathsf{bat}} := \frac{1}{m}\left(\mathrm{K}(X^{\mathsf{bat}}, X)\boldsymbol{\alpha} - Y^{\mathsf{bat}}\right). \tag{5}$$

Using this stochastic gradient, a preconditioned update is performed to the current estimate of the parameters $\boldsymbol{\alpha}$. This results in two simultaneous updates to $\boldsymbol{\alpha}$ in each iteration,

$$\boldsymbol{\alpha}^{\mathsf{bat}} \leftarrow \boldsymbol{\alpha}^{\mathsf{bat}} - \eta \boldsymbol{g}^{\mathsf{bat}} \tag{6a}$$
$$\boldsymbol{\alpha}^{\mathsf{nys}} \leftarrow \boldsymbol{\alpha}^{\mathsf{nys}} + \eta \mathbf{M}\mathrm{K}(X^{\mathsf{nys}}, X^{\mathsf{bat}})\boldsymbol{g}^{\mathsf{bat}} \tag{6b}$$

where $\boldsymbol{\alpha}^{\mathsf{bat}}$ is the subvector of $\boldsymbol{\alpha}$ with indices corresponding to the samples in X^{bat}. Similarly $\boldsymbol{\alpha}^{\mathsf{nys}}$ is the subvector with indices corresponding to samples in X^{nys}. In the equation above, the matrix $\mathbf{M}$ is a low rank matrix, given by

$$\mathbf{M} := \mathbf{E}\Lambda^{-1}\left(\boldsymbol{I} - \lambda_{q+1}\Lambda^{-1}\right)\mathbf{E}^{\top}, \tag{7}$$

where $(\Lambda, \mathbf{E}, \lambda_{q+1})$ forms the top-q eigensystem for the $s \times s$ matrix $\mathrm{K}(X^{\mathsf{nys}}, X^{\mathsf{nys}})$ (see definition 1 above).

Algorithm 1 AsyncEigenPro

```
Require: Data $(X, Y)$ and $G$ GPUs, kernel function $K(\cdot,\cdot)$, shared parameter vector $\boldsymbol{\alpha} \in \mathbb{R}^n$
Require: Batch size $m$, learning rate $\eta$, Nyström size $s$, preconditioner level $q$.

  Preprocessing:
  Sample subsets $X_1^{\mathsf{nys}}, X_2^{\mathsf{nys}}, \ldots, X_G^{\mathsf{nys}}$ without replacement from $X$, each of size $s$.
  for $r \leftarrow 1, \ldots, G$ do in parallel on each GPU
      $\mathbf{M}_r \leftarrow$ preconditioner from $X_r^{\mathsf{nys}}$ using eq. (10)
  end for

  Iteration:
  for $r \leftarrow 1, \ldots, G$ do in parallel on each GPU
      for $t \leftarrow 1, 2, \ldots$ do
          $(X_r^{\mathsf{bat}}, Y_r^{\mathsf{bat}}) \leftarrow$ mini-batch of size $m$
          Read $\boldsymbol{\alpha}$ from shared memory as $\widehat{\boldsymbol{\alpha}}$
          calculate gradient $\boldsymbol{g}_r^{\mathsf{bat}}$ using equation (11)
          Write $(\boldsymbol{\alpha}_r^{\mathsf{bat}}, \boldsymbol{\alpha}_r^{\mathsf{nys}})$ updates equation (12)
      end for
  end for
```

Note that X^{nys} is chosen once at preprocessing and $\mathbf{M} \in \mathbb{R}^{s \times s}$ is calculated once but stored as the tuple $(\Lambda, \mathbf{E}, \lambda_{q+1})$ for fast low-rank matrix multiplication. Thus the steps involved at setup are

(P1.) Kernel evaluations $\mathrm{K}(X^{\mathsf{nys}}, X^{\mathsf{nys}})$

(P2.) Top-q eigensystem computation $\mathbf{M} \equiv (\Lambda, \mathbf{E}, \lambda_{q+1})$

whereas the steps involved in each iteration are:

(I1.) Kernel evaluations $\mathrm{K}(X^{\mathsf{bat}}, X)$

(I2.) Gradient computations (equation (5))

(I3.) Preconditioning (reusing $\mathrm{K}(X^{\mathsf{nys}}, X^{\mathsf{bat}})$ from (I1.))

	FLOPS	Memory
Preprocessing (P1-2.)	$s^2c + sq^2$	s^2
Iteration (I1-3.)	$mnc_k + mnk$	nm

Table 2. Cost of EigenPro. c is the cost of 1 kernel evaluation, k is the target dimension. See notation Table 5.

3. Design of AsyncEigenPro

Before designing AsyncEigenPro, we first describe how we can distribute the computation of EigenPro over multiple GPUs.

EigenPro is highly parallelizable and can utilize multiple GPUs by distributing the kernel and gradient computations. These results can then be gathered and merged into an update for parameter vector.

		1 GPU	4 GPUs		8 GPUs	
	Accuracy	Time	Time	Speedup	Time	Speedup
(async) CIFAR-5M	89.15%	27.58	7.09	**3.89**×	3.70	**7.45**×
(async) ImageNet9	74.21%	10.73	2.78	**3.85**×	1.37	**7.83**×
(sync) CIFAR-5M	89.16%	27.58	7.04	**3.92**×	3.715	**7.42**×
(sync) ImageNet9	74.30%	10.73	2.74	**3.91**×	1.39	**7.71**×
(sync) ImageNet	66.71%	8.241	2.76	**2.98**×	1.30	**6.33**×
(sync) HIGGS	70.35%	88.16	22.20	**3.97**×	11.51	**7.65**×
(sync) TAXI	0.382(mse)	74.87	18.80	**3.98**×	9.42	**7.95**×

Table 3. Speedup of 1-epoch multi-GPU training. Time in minutes. Here, we try to get maximum speedup from both EigenPro and AsyncEigenPro. The shared parameter is in GPU. Hence, the communication delay is minimal or negligible except for ImageNet whose parameter vector is large (due to 1000 classes) which results in significant transfer costs.

We assume that (X, Y), $(\Lambda, \mathbf{E}, \lambda_{q+1})$ are stored on each of the G GPUs.

In the distributed computation, at each iteration, we first split the minibatch $X^{\mathsf{bat}}, Y^{\mathsf{bat}}$ into G virtual disjoint sub-batches,

$$(X_1^{\mathsf{bat}}, Y_1^{\mathsf{bat}}), (X_2^{\mathsf{bat}}, Y_2^{\mathsf{bat}}), \ldots, (X_G^{\mathsf{bat}}, Y_G^{\mathsf{bat}}).$$

The gradients for each sub-batch are computed as,

$$\boldsymbol{g}_r^{\mathsf{bat}} := \tfrac{1}{m}(\mathrm{K}(X_r^{\mathsf{bat}}, X) - Y_r^{\mathsf{bat}}), \quad r = 1, \ldots, G.$$

Finally, a reduction is performed by stacking and summing,

$$\boldsymbol{g}^{\mathsf{bat}} = \begin{bmatrix} \boldsymbol{g}_1^{\mathsf{bat}\top} & \boldsymbol{g}_2^{\mathsf{bat}\top} & \ldots & \boldsymbol{g}_G^{\mathsf{bat}\top} \end{bmatrix}^\top, \tag{8}$$

$$\mathrm{MK}(X^{\mathsf{nys}}, X^{\mathsf{bat}})\boldsymbol{g}^{\mathsf{bat}} = \sum_{r=1}^{G} \mathrm{MK}(X^{\mathsf{nys}}, X_r^{\mathsf{bat}})\boldsymbol{g}_r^{\mathsf{bat}}. \tag{9}$$

And then the updates equation (6) are performed.

Asynchronization: The sub-batching can be trivially asynchronous. Similarly, instead of stacking the $\boldsymbol{g}_r^{\mathsf{bat}}$'s as in equation (8), the coordinates corresponding to $\boldsymbol{\alpha}_r^{\mathsf{bat}}$ can be updated directly asynchronously.

However performing the asynchronous update to $\boldsymbol{\alpha}^{\mathsf{nys}}$ using (6b) without the sum reduction equation (9) is problematic.

The need for multiple preconditioners: Observe that at each iteration, the same set of coordinates corresponding to X^{nys} are updated. Consequently, each of the G GPUs attempts to write to the same shared memory. This can lead to unstable behavior unless proper locking is utilized which can be costly. See Figure 3 for an illustration.

Instead if the preconditioning step in equation (6b) involved updating separate parts of the shared parameter vector, the problem of locking can be avoided.

This is indeed the main design innovation of AsyncEigenPro. We enable this behavior by using multiple preconditioners for the EigenPro algorithm, where each GPU contains a unique preconditioner $\mathbf{M}_r$ corresponding to X_r^{nys}. We also enforce that X_r^{nys} are disjoint subsets of X, which imply the preconditioning write operations are non-overlapping. Let $(\Lambda_r, \mathbf{E}_r, \lambda_{q+1}^{(r)})$ be the top-q eigensystem of $\mathrm{K}(X_r^{\mathsf{nys}}, X_r^{\mathsf{nys}})$ calculated on the r^{th} GPU. Then define the preconditioner,

$$\mathbf{M}_r := \mathbf{E}_r \Lambda_r^{-1} \left(\boldsymbol{I} - \lambda_{q+1}^{(r)} \Lambda_r^{-1} \right) \mathbf{E}_r^\top \tag{10}$$

Note that having multiple preconditioners does not incur any additional setup time. Since we have multiple GPUs and computation of preconditioners are independent of each other, we can completely parallelize these computations.

The problem of inconsistent reads: The parameter vector $\boldsymbol{\alpha}$ is in a shared memory which every GPU can access. Each GPU would need the whole parameter vector to calculate gradients. Since each GPU is updating independent of other GPUs, the value of the the parameter during read may be different from the value of parameter at the time of update because other GPUs may have updated the parameter. So, the gradient updates may not correspond to the current parameter, also known as the problem of *stale gradients*.

Similarly, while a GPU is reading the shared parameter vector other GPUs may have partially updated the parameter which leads inconsistencies while reading the parameter. We call this event *inconsistent reads*. We follow notions from (17) to encapsulate both these events as perturbations to the parameters and denote the read parameter by any GPU as $\widehat{\boldsymbol{\alpha}}$.

Mechanics of AsyncEigenPro: Each GPU runs the EigenPro training algorithm independently of each other in an asynchronous and lock-free manner. We first randomly partition the data into mutually exclusive and collectively exhaustive sets $\{X_r\}_{r=1}^G$, one for each GPU. Nyström and mini-batch samples for each GPU are sampled from its corresponding set X_r. Then, during training, each GPU asynchronously fetches the most updated $\boldsymbol{\alpha}$ from shared memory which we call $\widehat{\boldsymbol{\alpha}}$, and computes the mini-batch

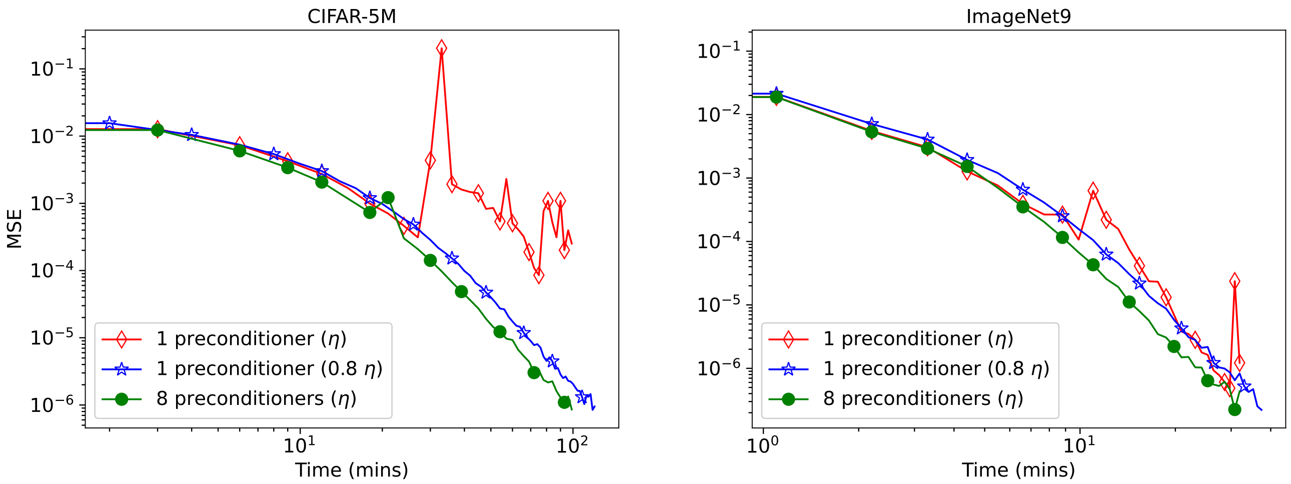

Figure 3. **The need for multiple preconditioners** in AsyncEigenPro: A single preconditioner would mean every GPU updating the same coordinates in equation (12b) in each iteration. In a lock-free environment, this easily leads to race conditions. Mutliple preconditioners with disjoint Nyström coordinates overcomes this challenge. Most runs with a single preconditioner diverged, and in the runs in which it converged (plotted here), we can see, AsyncEigenProis very unstable. For a single preconditioner to converge we need to decrease the learning rate even further. This effect can only get worse as we increase the number of GPUs

gradient

$$\boldsymbol{g}_r^{\mathsf{bat}} := \frac{1}{m}\left(K(X_r^{\mathsf{bat}}, X)\widehat{\boldsymbol{\alpha}} - Y_r^{\mathsf{bat}}\right) \tag{11}$$

and the preconditioning $\mathbf{M}_r\mathrm{K}^{(\mathsf{nys}_r,\mathsf{bat}_r)}\boldsymbol{g}_r^{\mathsf{bat}}$. Finally, the parameter in shared memory is updated asynchronously as,

$$\boldsymbol{\alpha}_r^{\mathsf{bat}} \leftarrow \boldsymbol{\alpha}_r^{\mathsf{bat}} - \eta\boldsymbol{g}_r^{\mathsf{bat}} \tag{12a}$$

$$\boldsymbol{\alpha}_r^{\mathsf{nys}} \leftarrow \boldsymbol{\alpha}_r^{\mathsf{nys}} + \eta\mathbf{M}_r K(X_r^{\mathsf{nys}}, X_r^{\mathsf{bat}})\boldsymbol{g}_r^{\mathsf{bat}} \tag{12b}$$

We emphasize that there is no locking or synchronization at any point. We allow for inconsistent reads, whereas writes are non-overlapping by design. Hence, unlike prior works, we do not need to assume *atomic writes* even for single dimension.

Note that $\mathbf{M}_r$ is calculated using top-q eigensystem of $K(X_r^{\mathsf{nys}}, X_r^{\mathsf{nys}})$. For pseudo code see Algorithm 1.

4. Convergence Analysis

To account for delayed gradients we define the following.

Definition 2 (Maximum iteration latency τ)**.** We assign a time t to an iteration at the start of the write operation to the shared parameter vector. Between the time of read and time of update of iteration t, other GPUs may have updated the shared parameter. We assume there exists a constant integer τ such that the maximum number of updates between time-of-read and time-of-write is no more than τ.

Assumption 1 (Strong convexity). The smallest eigenvalue of the kernel matrix $\mathrm{K}(X,X)/n$ satisfies $\lambda_n > 0$.

Theorem 1 (Convergence of $\boldsymbol{\alpha}$)**.** *Consider the iteration given in equation* (12) *initialized at $\boldsymbol{\alpha}_0$, with maximum iteration latency τ. If the learning rate $\eta \leq \frac{m}{\beta+\lambda_{q+1}(m-1)}$, $\beta = \max_i k(\boldsymbol{x}_i, \boldsymbol{x}_i)$, $(\lambda_1, \lambda_2, ..., \lambda_n)$ are the eigenvalues of $K(X,X)/n$, then after T iterations, we have*

$$\mathbb{E}\left\|\boldsymbol{\alpha}_T - \boldsymbol{\alpha}^*\right\|^2 \leq \left(1 - \tfrac{\eta\lambda_n}{2}\right)^T \left\|\boldsymbol{\alpha}_0 - \boldsymbol{\alpha}^*\right\|^2 + \frac{2\eta^2\tau\mathcal{L}_{\mathsf{init}}}{m}\left(1 - \left(1 - \tfrac{\eta\lambda_n}{2}\right)^T\right) \tag{13}$$

where $\mathcal{L}_{\mathsf{init}}$ is the square loss at initialization.

See Appendix A.1 for proof. For $\tau = 0$, we are in the synchronous setting, in which case the result matches the parameter bound in (**?**)Thm. 1]ma2018power.

Remark 1 (Residual variance due to delay τ). **Lack of automatic variance reduction.** The first term of the bound is an exponential convergence term since we are dealing with a linear dynamical system. The second term is a variance term which is introduced because of the delay τ. Notice that despite of non-overlapping updates and overparameterization we do not see a complete VRF.

Dependence on delay: Note that, there is only a linear dependence on the delay τ as against the quadratic dependence in Hogwild!. Comparing the analysis with Hogwild!, our analysis uses two key aspects of the kernel regression problem at hand – (i) interpolation and (ii) multiple preconditioners. Since we can interpolate, we can apply the analysis from (14). Furthermore, we have non-overlapping write operation, which simplifies analysis significantly.

To get to a specified error ϵ we can make each of the two terms $\epsilon/2$ which results in following

Corollary 2. *Error rate of ϵ is reached i.e., $\left\|\boldsymbol{\alpha}_T - \boldsymbol{\alpha}^*\right\|_2^2 \leq$*

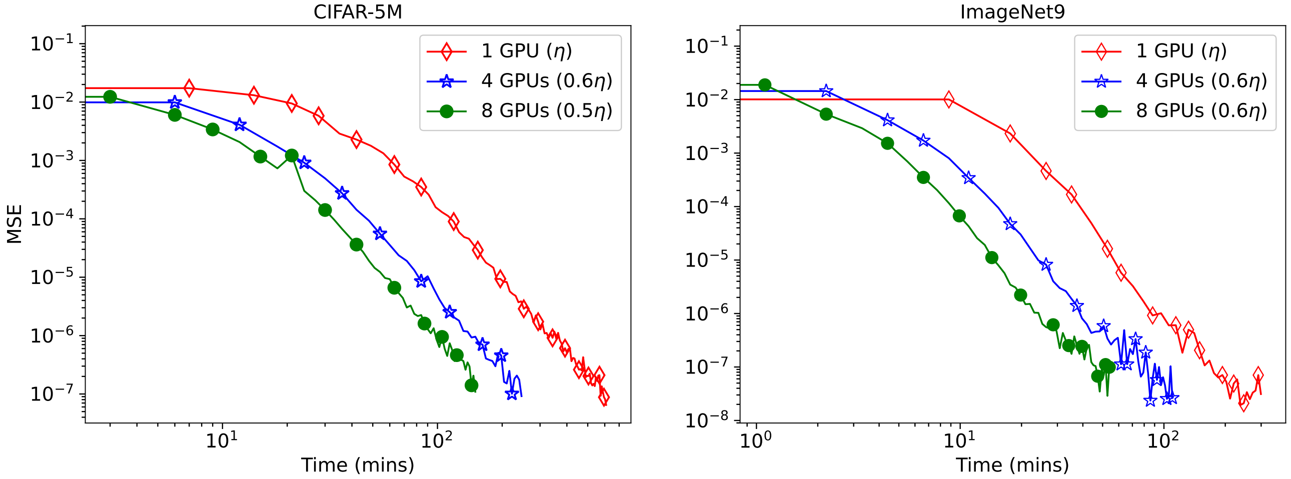

Figure 4. For AsyncEigenPro, we plot the training loss as a function of wall clock time. Due to delayed gradients, learning rate had to be decreased. Also, decrease in learning rate with respect to the number of GPUs is very mild. For both CIFAR-5M and ImageNet9 we did not have to decrease learning rate more than a factor of half as compared to that of 1 GPU.

ϵ for $\eta \leq \sqrt{\frac{m\epsilon}{\tau\mathcal{L}_{\text{init}}}}$ after T iterations given by

$$T \geq \mathcal{O}\left(\sqrt{\frac{\tau\mathcal{L}_{\text{init}}}{m\epsilon}}\frac{\log\left(\frac{2\|\boldsymbol{\alpha}_0-\boldsymbol{\alpha}^*\|_2^2}{\epsilon}\right)}{\lambda_n}\right)$$

Note that upto logarithmic factor our analysis provides $1/T^2$ rate of convergence for constant step size. This is in contrast to synchronous and asynchronous SGD analysis (17) which guarantee $1/T$ convergence rate upto log factors. Comparing these analysis, we see that the better rates are mainly due to the interpolation properties.

Remark 2 (Speedup challenges in the interpolation regime). If number of processors is G then $\tau = \mathcal{O}(G)$. We see from Corollary 2 that as we increase the number of processors, the number of iterations increase by $\sqrt{G}$. So, the theoretical guarantee is for $\sqrt{G}$ speedup. In interpolation regime, synchronous SGD has strong convergence properties due to VRF. Hence, introducing a variance due to delayed gradients results in less than linear speedup guarantee. However, in experiments we see that much better speedup is obtained in many cases.

5. Numerical experiments

We run experiments on various datasets with both multi-GPU EigenPro and AsyncEigenPro. We also run experiments with delay models on each GPU. Then, we explore a scenario where we can get maximum speedup. Experiments were conducted on Linux machine, AMD Milan CPUs where we use upto 80GB memory, 8 NVIDIA A100 GPUs contained in a single node. For a given dataset we use a fixed preconditioner level (see Appendix B), consequently the mini-batch size (see (14)), for all the runs such that we use maximum memory capacity of the GPUs. For certain asynchronous runs we tune the learning rate.

Datasets: CIFAR-5M is a 5 million subset of CIFAR10-like images from (20). We use a 3 million subset of this data. Next, we generate a subset of ImageNet dataset using 9 classes belonging to living animals (7). Then, we perform random augmentations to get a dataset of size 1.85 million images which we call ImageNet9. All image datasets are featurized using MobileNetV2 model (24) and then used as input for kernel regression. HIGGS data is from (5). TAXI dataset is originally from this[1] repository. We borrow a subset of this data from (18) and further sample to get a data size of 10 million.

Table 4. Datasets for experiments. TAXI is a regression problem.

Dataset	# samples	# features	# classes
CIFAR-5M	3 million	1280	10
ImageNet9	1.85 million	1280	9
ImageNet	1.28 million	1280	1000
HIGGS	10.5 million	28	2
TAXI	10 million	9	1

Effect of random delays: Figure 1 shows the comparison between synchronous and asynchronous training when some processors are delayed.

We assume a delay model where at each iteration each GPU has the same probability of a delay event. So a GPU can randomly delay gradient computation by a given probability. The effect of delay is distributed across iterations in the case of AsyncEigenProwhereas the delay gets added to each iteration in EigenPro. Thus we see that as the probability

[1] https://github.com/toddwschneider/nyc-taxi-data

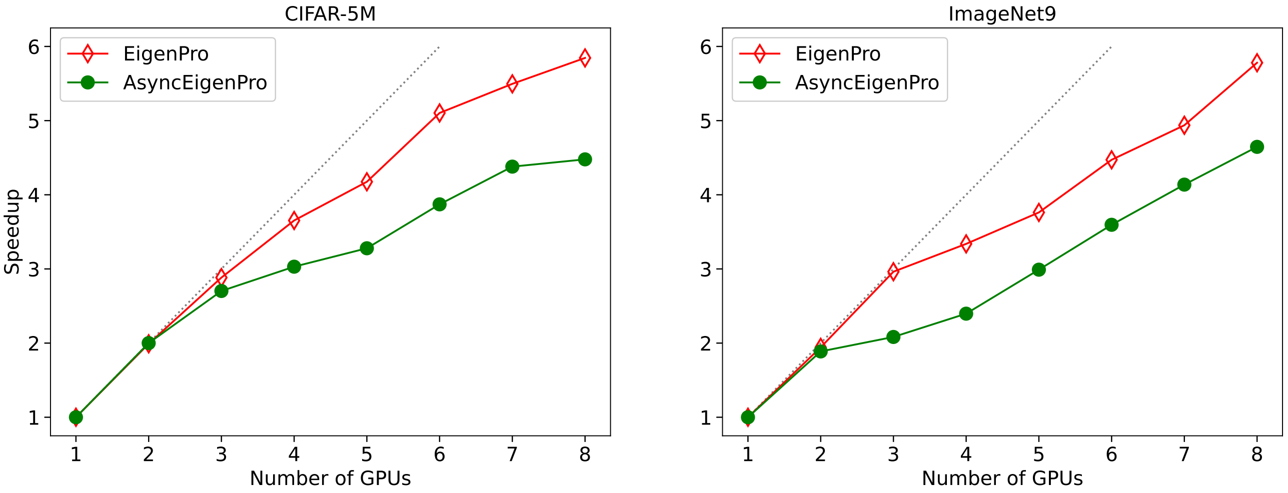

Figure 5. We show speedups of both EigenPro and AsyncEigenPro for up to 8 GPUs. All the runs are driven to same loss value. Both EigenPro and AsyncEigenPro exhibit a significant speedup with increasing number of GPUs. Here, the EigenPro fares better due to the idealized conditions with minimum communication delays. However, as we show in Figure 1, in the presence of delays, the speedup of AsyncEigenPro degrades more gracefully than EigenPro.

of delay increases, the gap between synchronous and asynchronous increases. We see in Figure 1 that for probability as low as 0.02 we see a significant decrease in speedup for EigenPro. Also, the delay model considered here is fairly benign. Other delay models like having a consistently slow GPU or high communication delay can make synchronous algorithm prohibitive.

Speedups: We see in Figure 5 the speedup obtained by both EigenProand AsyncEigenPro. The shared parameters in these experiments reside in CPU. The modern NVIDIA A100 GPUs have a very efficient CPU-GPU transfer mechanism which results in minimum communication delay. We see that synchronous case fares well in these scenarios. We show the progression of training procedure for AsyncEigenPro in Section 4. Due to multiple preconditioners, even with lock-free algorithm we get stable training along with significant speedup.

Approximation-Optimization tradeoff In many use cases, higher computational efficiency is desired at a slight cost of model performance. For such approximate solutions, a larger η can suffice. We note in Table 3 that it is possible to get good accuracy without complete convergence in training error. For cases like TAXI, where the size of the parameter vector is not large, we see full linear speedup.

6. Discussion and Conclusion

In this paper we provided a new algorithm called AsyncEigenPro, a multi-GPU lock-free parallel training algorithm for large-scale kernel regression.

This algorithm is resilient against synchronization delays as shown in Figure 1, and is suited for multi-GPU training with a large number of GPUs, since synchronization delays grow with the number of GPUs.

We provided the first convergence analysis for overparametrized SGD in an asynchronous setting. This analysis shows that the convergence rate of asynchronous SGD in overparametrized regime is better than that of the classical setting.

Our numerical experiments provided kernel regression on 10 Million training samples. To the best of our knowledge this is the largerst kernel machine trained without reducing the model size.

Currently, we performed all experiments on a single node with at most 8 GPUs, which is the maximum number of GPUs available per node in our computer cluster. A multinode training can also be setup, with some additional technicalities. The algorithm algorithm being robust to communication delays shows potential to scale even further.

References

[1] Amirhesam Abedsoltan, Mikhail Belkin, and Parthe Pandit. Toward large kernel models. *arXiv preprint arXiv:2302.02605*, 2023.

[2] Yossi Arjevani, Ohad Shamir, and Nathan Srebro. A tight convergence analysis for stochastic gradient descent with delayed updates. In *Algorithmic Learning Theory*, pages 111–132. PMLR, 2020.

[3] Nachman Aronszajn. Theory of reproducing kernels. *Transactions of the American mathematical society*, 68(3):337–404, 1950.

[4] Sanjeev Arora, Simon S Du, Zhiyuan Li, Ruslan Salakhutdinov, Ruosong Wang, and Dingli Yu. Harnessing the power of infinitely wide deep nets on small-data tasks. *arXiv preprint arXiv:1910.01663*, 2019.

[5] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 07 2014.

[6] Léon Bottou. Online algorithms and stochastic approximations. *Online learning and neural networks*, 1998.

[7] Logan Engstrom, Andrew Ilyas, Hadi Salman, Shibani Santurkar, and Dimitris Tsipras. Robustness (python library), 2019.

[8] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31, 2018.

[9] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[10] George S Kimeldorf and Grace Wahba. A correspondence between bayesian estimation on stochastic processes and smoothing by splines. *The Annals of Mathematical Statistics*, 41(2):495–502, 1970.

[11] Jaehoon Lee, Samuel Schoenholz, Jeffrey Pennington, Ben Adlam, Lechao Xiao, Roman Novak, and Jascha Sohl-Dickstein. Finite versus infinite neural networks: an empirical study. *Advances in Neural Information Processing Systems*, 33:15156–15172, 2020.

[12] Jian Li, Yong Liu, and Weiping Wang. Towards sharp analysis for distributed learning with random features. *arXiv preprint arXiv:1906.03155*, 2019.

[13] Chaoyue Liu, Libin Zhu, and Misha Belkin. On the linearity of large non-linear models: when and why the tangent kernel is constant. *Advances in Neural Information Processing Systems*, 33:15954–15964, 2020.

[14] Siyuan Ma, Raef Bassily, and Mikhail Belkin. The power of interpolation: Understanding the effectiveness of sgd in modern over-parametrized learning. In *International Conference on Machine Learning*, pages 3325–3334. PMLR, 2018.

[15] Siyuan Ma and Mikhail Belkin. Diving into the shallows: a computational perspective on large-scale shallow learning. *Advances in neural information processing systems*, 30, 2017.

[16] Siyuan Ma and Mikhail Belkin. Kernel machines that adapt to gpus for effective large batch training. *Proceedings of Machine Learning and Systems*, 1:360–373, 2019.

[17] Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. Perturbed iterate analysis for asynchronous stochastic optimization. *SIAM Journal on Optimization*, 27(4):2202–2229, 2017.

[18] Giacomo Meanti, Luigi Carratino, Lorenzo Rosasco, and Alessandro Rudi. Kernel methods through the roof: handling billions of points efficiently. *Advances in Neural Information Processing Systems*, 33:14410–14422, 2020.

[19] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124003, 2021.

[20] Preetum Nakkiran, Behnam Neyshabur, and Hanie Sedghi. The deep bootstrap framework: Good online learners are good offline generalizers. In *International Conference on Learning Representations*, 2021.

[21] Adityanarayanan Radhakrishnan, Daniel Beaglehole, Parthe Pandit, and Mikhail Belkin. Feature learning in neural networks and kernel machines that recursively learn features. *arXiv preprint arXiv:2212.13881*, 2022.

[22] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, volume 24, 2011.

[23] L. F. Richardson. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the

stresses in a masonry dam. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 210:307–357, 1911.

[24] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[25] Bernhard Schölkopf, Ralf Herbrich, and Alex J Smola. A generalized representer theorem. In *International conference on computational learning theory*, pages 416–426. Springer, 2001.

[26] Bernhard Schölkopf, Alexander J Smola, Francis Bach, et al. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[27] Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. *Advances in neural information processing systems*, 13, 2000.

[28] Yuchen Zhang, John Duchi, and Martin Wainwright. Divide and conquer kernel ridge regression: A distributed algorithm with minimax optimal rates. *Journal of Machine Learning Research*, 16(102):3299–3340, 2015.

APPENDICES

Symbol	Purpose
n	Number of samples
m	Batch-size
G	Number of GPUs
c	cost of 1 pairwise kernel evaluation
d	input dimension
k	output dimension
s	Nyström approximation subsample size
q	Preconditioner level

Table 5. Symbolic notation. They satisfy $m < n$, and $q < s < n$.

A. Proofs

In many of the steps we will be using the following result

$$\mu_n \|\boldsymbol{v}\|_2^2 \leq \|\mathbf{M}\boldsymbol{v}\|_2^2 \leq \mu_1 \|\boldsymbol{v}\|_2^2 \tag{14}$$

where μ_1 is the largest eigenvalue and μ_n is the smallest eigenvalue of the full rank matrix $\mathbf{M}^\top\mathbf{M} \in \mathbb{R}^{n\times n}$. $\boldsymbol{v}$ is any vector.

Definition 3 (Selector matrix $\boldsymbol{Q}_m$)**.** For any index set $\mathcal{B}_m$ with size m we define a diagonal matrix which acts as a selector.

$$[\boldsymbol{Q}_m]_{ij} = \begin{cases} 1 & i = j, i \in \mathcal{B}_m \\ 0 & \text{otherwise} \end{cases}$$

Definition 4 (Redefining $\mathbf{M}_r$)**.** Let's define a $n \times n$ version of $\mathbf{M}_r$ defined in equation (10) where we assign 0 to all the entries which do not correspond to Nyström indices.

$$\boldsymbol{A} = \begin{bmatrix} \mathbf{M}_r & \mathbf{0} \\ \mathbf{0} & \mathbf{0}_{n-s} \end{bmatrix} \in \mathbb{R}^{n\times n},\ \mathbf{M}_r \in \mathbb{R}^{s\times s} \tag{15}$$

From here on in the appendix we will refer to $\boldsymbol{A}$ as $\mathbf{M}_r$ for convenience in notation.

Definition 5 (Preconditioners $\boldsymbol{P}_r$ and $\boldsymbol{P}$)**.**

$$\boldsymbol{P}_r := \boldsymbol{I} - \mathbf{M}_r\mathbf{K} \tag{16}$$

$$\boldsymbol{P} := \boldsymbol{I} - \mathbf{M}\mathbf{K} \tag{17}$$

where $\mathbf{M}$ is the matrix obtained when $s = n$ in equation (10) i.e., without Nyström approximation

Lemma 3 (Update equation for $\boldsymbol{\alpha}$)**.** *Updates in equation* (12a) *and* (12b) *can be merged into a single update equation as follows*

$$\boldsymbol{\alpha}_{t+1} = \boldsymbol{\alpha}_t - \frac{\eta}{m}\boldsymbol{P}_r\boldsymbol{Q}_m\left(\mathbf{K}\boldsymbol{\alpha}_t - \boldsymbol{y}\right)$$

where $\boldsymbol{Q}_m$ is the selector matrix (definition 3) for the set of mini-batch indices

Proof. For analysis, we want to merge (12a), (12b) into a single update equation. We will use the selector matrix $\boldsymbol{Q}_m$ and $\boldsymbol{Q}_s$ for selecting set of mini-batch indices $\mathcal{B}_m$ and set of Nyström samples' indices $\mathcal{B}_s$ respectively. This gives a full update equation for $\boldsymbol{\alpha}$ with zeros in the non-batch and non-Nyström indices.

Now, merging (12a) and (12b) gives

$$\begin{aligned} \boldsymbol{\alpha}_{t+1} &= \boldsymbol{\alpha}_t - \frac{\eta}{m}\boldsymbol{Q}_m\left(\mathbf{K}\widehat{\boldsymbol{\alpha}}_t - \boldsymbol{y}\right) + \frac{\eta}{m}\mathbf{M}_r(\boldsymbol{Q}_m\mathbf{K}\boldsymbol{Q}_s)^\top\boldsymbol{Q}_m\left(\mathbf{K}\widehat{\boldsymbol{\alpha}}_t - \boldsymbol{y}\right) \\ &= \boldsymbol{\alpha}_t - \frac{\eta}{m}\left(\boldsymbol{I} - \mathbf{M}_r\mathbf{K}\right)\boldsymbol{Q}_m\left(\mathbf{K}\widehat{\boldsymbol{\alpha}}_t - \boldsymbol{y}\right) \end{aligned}$$

where, we have used $\mathbf{M}\boldsymbol{Q}_s = \mathbf{M},\ \boldsymbol{Q}_m^2 = \boldsymbol{Q}_m$. Define

$$\boldsymbol{P}_r := \boldsymbol{I} - \mathbf{M}_r\mathrm{K}$$

to get

$$\boldsymbol{\alpha}_{t+1} = \boldsymbol{\alpha}_t - \frac{\eta}{m} \boldsymbol{P}_r \boldsymbol{Q}_m \left(\mathbf{K} \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{y} \right)$$

Note that only $\mathcal{B}_m \cup \mathcal{B}_s$ indices are updated in each iteration just like in (12a), (12b). Now we have a single update equation for each iteration of EigenPro. □

Definition 6 (Selector $\boldsymbol{Q}_i^t$ for inconsistencies)**.** The updates from other processors might be partially complete when iteration t starts the update. To account for such scenarios define $\boldsymbol{Q}_i^t$. For iteration t and iteration i such that $t - \tau \le i < t$, there exists a diagonal matrix $\boldsymbol{Q}_i^t \in \mathbb{R}^{n \times n}$ with diagonal entries in $\{0, 1\}$ such that

$$\widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}_t = \frac{\eta}{m} \sum_{i=t-\tau}^{t-1} \boldsymbol{Q}_i^t \left\{ \boldsymbol{P} \boldsymbol{Q}_m \left(\mathbf{K} \widehat{\boldsymbol{\alpha}}_i - \boldsymbol{y} \right) \right\} \tag{18}$$

i.e., between time of read and time of update, the difference in parameter is only because of delayed gradients and partial writes.

A.1. Proof of Theorem 1

Proof. First we will merge the two step update in equation (12) into a single update equation in Lemma 3. Then we derive a bound on $\|\boldsymbol{\alpha}_T - \boldsymbol{\alpha}^*\|^2$ for after T iteration. The upper bound involves two terms. The first term is an exponential convergence term since we are dealing with strong convexity and the second term is due variance introduced by the delayed gradients.

Notice that the parameter updates in AsyncEigenPro are additive. However, the updates are asynchronous, whereby the parameter value on which $\boldsymbol{g}_m$ in equation (11) is computed could be different than the parameter value at which it updates. Furthermore, there could be inconsistent reads which will result in noisy updates. We define $\widehat{\boldsymbol{\alpha}}$ to denote both delays (stale updates) and inconsistent reads. Using $\widehat{\boldsymbol{\alpha}}$ in the update rule of Lemma 3 will give

$$\boldsymbol{\alpha}_{t+1} = \boldsymbol{\alpha}_t - \frac{\eta}{m} \boldsymbol{P}_r \boldsymbol{Q}_m \left(\mathbf{K} \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{y} \right)$$

We define a function $g_m : \mathbb{R}^n \to \mathbb{R}^n$

$$g_m(\boldsymbol{\alpha}) := \frac{1}{m} \boldsymbol{P}_r \boldsymbol{Q}_m \left(\mathbf{K} \boldsymbol{\alpha} - \boldsymbol{y} \right) = \frac{1}{m} \boldsymbol{P}_r \boldsymbol{Q}_m \mathbf{K} \left(\boldsymbol{\alpha} - \boldsymbol{\alpha}^* \right) \tag{19}$$

$$\boldsymbol{\alpha}_{t+1} = \boldsymbol{\alpha}_t - \eta g_m(\widehat{\boldsymbol{\alpha}_t}) \tag{20}$$

Lemma 6 gives the convergence of the above iterations and the theorem statement.

□

Definition 7. Define a function $h : \mathbb{R}^n \to \mathbb{R}$ as below

$$h(\boldsymbol{\alpha}) := \frac{1}{n} \left(\frac{1}{2} \boldsymbol{\alpha}^\top \boldsymbol{P} \mathbf{K} \boldsymbol{\alpha} - \boldsymbol{\alpha}^\top \boldsymbol{P} \boldsymbol{y} \right) \tag{21}$$

$$:= \frac{1}{n} \left(\frac{1}{2} \boldsymbol{\alpha}^\top \mathbf{K}_p \boldsymbol{\alpha} - \boldsymbol{\alpha}^\top \mathbf{K}_p \boldsymbol{\alpha}^* \right) \tag{22}$$

$$\text{where} \quad \mathbf{K}_p := \boldsymbol{P} \mathbf{K} \tag{23}$$

here, we have used $\boldsymbol{y} = \mathbf{K} \boldsymbol{\alpha}^*$

Lemma 4 (Eigensystems of matrices)**.** *Consider the loss function $\mathcal{L}$*

$$\mathcal{L}(\boldsymbol{\alpha}) = \frac{1}{n} \|\mathbf{K} \alpha - \boldsymbol{y}\|_2^2 \tag{24}$$

where $\mathbf{K}$ is the kernel matrix. Let $\lambda_1 \ge \lambda_2 \ge \ldots \lambda_n > 0$ be the eigenvalues of $\mathbf{K}/n$ and the corresponding eigenvectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n$. Not that these different from definition 1 by a scaling factor of n.

(1) Eigenvalues of $\boldsymbol{P}$ are

$$1 \ge 1 \ge \ldots \ge 1 \ge \frac{\lambda_{q+1}}{\lambda_q} \ge \ldots \ge \frac{\lambda_{q+1}}{\lambda_1}$$

and corresponding eigenvectors are $\boldsymbol{e}_n, \boldsymbol{e}_{n-1}, \ldots, \boldsymbol{e}_{q+1}, \boldsymbol{e}_q, \ldots, \boldsymbol{e}_1$

(2) Largest eigenvalue of $\boldsymbol{P}_r^\top \boldsymbol{P}_r$ *is* 1.

(3) Eigenvalues of $\mathbf{K}_p := \boldsymbol{P}\mathbf{K}$ *are*

$$n\lambda_{q+1} = n\lambda_{q+1} = \ldots = n\lambda_{q+1} \geq n\lambda_{q+2} \geq \ldots \geq n\lambda_n$$

and corresponding eigenvectors are $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_{q+1}, \boldsymbol{e}_{q+2}, \ldots, \boldsymbol{e}_n$

Proof.

(1) $\mathbf{M}$ can be expressed in terms of eigenvectors as follows

$$\mathbf{M} = \sum_{i=1}^{q} \left(1 - \frac{\lambda_{q+1}}{\lambda_i}\right) \frac{1}{n\lambda_i} \boldsymbol{e}_i \boldsymbol{e}_i^\top \tag{25}$$

Next, we verify the eigenvectors

$$\begin{aligned} \boldsymbol{P}\boldsymbol{e}_i &= \boldsymbol{e}_i - \mathbf{M}\mathrm{K}\boldsymbol{e}_i = \boldsymbol{e}_i - \mathbf{M}n\lambda_i \boldsymbol{e}_i \\ &\overset{(a)}{=} \boldsymbol{e}_i - \left(1 - \frac{\lambda_{q+1}}{\lambda_i}\right)\boldsymbol{e}_i = \frac{\lambda_{q+1}}{\lambda_i}\boldsymbol{e}_i \qquad \forall i = 1, 2, \ldots, q \end{aligned}$$

where (a) follows from (25)

$$\begin{aligned} \boldsymbol{P}\boldsymbol{e}_j &= \boldsymbol{e}_j - \mathbf{M}\mathrm{K}\boldsymbol{e}_j = \boldsymbol{e}_j - n\lambda_j \mathbf{M}\boldsymbol{e}_j \\ &\overset{(a)}{=} \boldsymbol{e}_j \qquad \forall j = q+1, \ldots, n \end{aligned}$$

where (a) follows from (25)

(2) By Nyström approximation eigenvalues of $\boldsymbol{P}_r^\top \boldsymbol{P}_r$ is approximately same as $\boldsymbol{P}^\top \boldsymbol{P}$. Largest eigenvalue of $\boldsymbol{P}^\top \boldsymbol{P}$ is 1.

(3) Verifying eigenvectors

$$\begin{aligned} \mathrm{K}_p \boldsymbol{e}_i &= \boldsymbol{P}\mathrm{K}\boldsymbol{e}_i = n\lambda_i \boldsymbol{P}\boldsymbol{e}_i \\ &\overset{(a)}{=} n\lambda_{q+1}\boldsymbol{e}_i \qquad i = 1, 2, \ldots, q \end{aligned}$$

where (a) from eigensystem of $\boldsymbol{P}$

$$\mathrm{K}_p \boldsymbol{e}_j = \boldsymbol{P}\mathrm{K}\boldsymbol{e}_j = n\lambda_j \boldsymbol{P}\boldsymbol{e}_j = n\lambda_j \boldsymbol{e}_j \qquad \forall j = q+1, \ldots, n$$

□

Lemma 5. *Properties of* $g_m(\boldsymbol{\alpha})$ *defined in* (19)

(1) Unbiased estimator

$$\mathbb{E}_m\left[g_m(\boldsymbol{\alpha})\right] = \nabla h(\boldsymbol{\alpha}) \tag{26}$$

(2) Decomposing $g_m(\boldsymbol{\alpha})$

$$\mathbb{E}_m \left\|g_m(\boldsymbol{\alpha})\right\|_2^2 = \frac{1}{m}\mathbb{E}_{i_1}\left\|g_{1,i_1}(\boldsymbol{\alpha})\right\|_2^2 + \frac{m-1}{m}\left\|\nabla h(\boldsymbol{\alpha})\right\|_2^2 \tag{27}$$

$$\text{where} \quad g_{1,i_k}(\boldsymbol{\alpha}) := \boldsymbol{P}_r \boldsymbol{Q}_{i_k}\left(\mathbf{K}\boldsymbol{\alpha} - \boldsymbol{y}\right) \tag{28}$$

(3) Bound on $g_m(\boldsymbol{\alpha})$

$$\mathbb{E}\left\|g_m(\boldsymbol{\alpha})\right\|_2^2 \leq \frac{\mathcal{L}_0}{m} \tag{29}$$

where $\mathcal{L}_0$ *is the value of square loss at initialization.*

Proof.

(1)

$$\begin{aligned}\mathbb{E}_m\left[g_m(\boldsymbol{\alpha})\right] &= \frac{1}{m}\boldsymbol{P}_r\mathbb{E}_m[\boldsymbol{Q}_m]\left(\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right)\\ &= \frac{1}{m}\boldsymbol{P}_r\left(\frac{m}{n}\boldsymbol{I}\right)\left(\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right)\\ &= \frac{1}{n}\boldsymbol{P}_r\left(\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right)\\ &\overset{(a)}{\approx} \frac{1}{n}\boldsymbol{P}\left(\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right) = \nabla h(\boldsymbol{\alpha})\end{aligned}$$

where (a) is from the Nyström approximation (27).

Note that only the preconditioner is specific to a GPU. Hence, if we are in a heterogeneous setting we can rotate the association of GPUs and random data (along with preconditioner) periodically. This gives an unbiased estimation of gradients. Also, this operation is cheap because the size of preconditioner and data indices are small.

(2) Let the set of i.i.d. indices for the mini-batch be $\boldsymbol{R}_m = \{i_k : k \leq m, k \in \mathbb{N}\}$. Then, g_m can be written as sum of terms where each term deals with an independent random variable.

$$g_m(\boldsymbol{\alpha}) = \frac{1}{m}\sum_{k=1}^{m}\boldsymbol{P}_r\boldsymbol{Q}_{i_k}\left(\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right)$$

$$g_m(\boldsymbol{\alpha}) = \frac{1}{m}\sum_{k=1}^{m}g_{1,i_k}(\boldsymbol{\alpha})$$

$$\begin{aligned}\mathbb{E}_m\left\|g_m(\boldsymbol{\alpha})\right\|_2^2 &= \mathbb{E}_m\left\langle \frac{1}{m}\sum_{k=1}^{m}g_{1,i_k}(\boldsymbol{\alpha}),\ \frac{1}{m}\sum_{k=1}^{m}g_{1,i_k}(\boldsymbol{\alpha})\right\rangle\\ &= \frac{1}{m^2}\sum_{k=1}^{m}\mathbb{E}_{i_k}\left\|g_{1,i_k}(\boldsymbol{\alpha})\right\|_2^2 + \frac{1}{m^2}\sum_{\substack{j,k\\ j\neq k}}\mathbb{E}_{i_k,i_j}\left\langle g_{1,i_k}(\boldsymbol{\alpha}),\ g_{1,i_j}(\boldsymbol{\alpha})\right\rangle\\ &= \frac{1}{m}\mathbb{E}_{i_1}\left\|g_{1,i_1}(\boldsymbol{\alpha})\right\|_2^2 + \frac{m-1}{m}\left\|\nabla h(\boldsymbol{\alpha})\right\|_2^2\end{aligned}$$

(3)

$$\begin{aligned}\mathbb{E}\left\|g_m(\boldsymbol{\alpha})\right\|_2^2 &= \frac{1}{m^2}\mathbb{E}\left\|\boldsymbol{P}_r\boldsymbol{Q}_m\left(\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right)\right\|_2^2\\ &\leq \frac{1}{m^2}\mathbb{E}\left\|\boldsymbol{Q}_m\left(\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right)\right\|_2^2\\ &= \frac{1}{m^2}\mathbb{E}\left\langle\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y},\mathbb{E}_m[\boldsymbol{Q}_m]\left(\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right)\right\rangle\\ &= \frac{1}{mn}\mathbb{E}\left\|\mathbf{K}\boldsymbol{\alpha}-\boldsymbol{y}\right\|_2^2\\ &\overset{(a)}{\leq} \frac{1}{mn}\left\|\mathbf{K}\boldsymbol{\alpha}_0-\boldsymbol{y}\right\|_2^2\\ &= \frac{\mathcal{L}_0}{m}\end{aligned}$$

In (a), we assume that the expected value of loss is less than initial loss. where $\mathcal{L}_0$ is the value of the square loss at initialization.

□

Lemma 6 (Convergence of $\boldsymbol{\alpha}$)**.** *If $\eta \lesssim \frac{1}{3}\left(\frac{m}{\nu+\lambda_{q+1}^{(s)}(m-1)}\right)$, then the iteration given in* (20) *satisfies the following after T iterations*

$$\mathbb{E}\left\|\boldsymbol{\alpha}_T-\boldsymbol{\alpha}^*\right\|_2^2 \leq \left(1-\frac{\eta\lambda_n}{2}\right)^T\left\|\boldsymbol{\alpha}_0-\boldsymbol{\alpha}^*\right\|_2^2 \tag{30}$$

Here, $\nu = max_i[\mathbf{K}]_{ii}$*,*

Proof. $h(\boldsymbol{\alpha})$ is λ_{q+1}-smooth and λ_n-strongly convex (See definition 7 and Lemma 4).

Strong convexity implies the following

$$h(\boldsymbol{\alpha}) - h(\boldsymbol{\alpha}^*) = h(\boldsymbol{\alpha}) \leq \langle \nabla h(\boldsymbol{\alpha}), \boldsymbol{\alpha} - \boldsymbol{\alpha}^* \rangle + \frac{\lambda_n}{2} \left\| \boldsymbol{\alpha} - \boldsymbol{\alpha}^* \right\|_2^2 \tag{31}$$

Analysing the norm $\left\| \boldsymbol{\alpha}_{t+1} - \boldsymbol{\alpha}^* \right\|_2^2$

$$\begin{aligned}
&\mathbb{E} \left\| \boldsymbol{\alpha}_{t+1} - \boldsymbol{\alpha}^* \right\|_2^2 \\
&= \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* - \eta g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \\
&= \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \mathbb{E} \left\langle \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^*,\ g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\rangle + \eta^2 \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \\
&= \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \mathbb{E} \left\langle \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}^*,\ g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\rangle - 2\eta \mathbb{E} \left\langle \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}_t,\ g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\rangle + \eta^2 \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \\
&\overset{(a)}{=} \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \mathbb{E} \left\langle \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}^*,\ g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\rangle + \eta^2 \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \\
&\overset{(b)}{=} \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \mathbb{E} \left\langle \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}^*,\ \nabla h(\widehat{\boldsymbol{\alpha}}_t) \right\rangle + \eta^2 \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \\
&\overset{(c)}{\leq} \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \left(\mathbb{E}\left[h(\widehat{\boldsymbol{\alpha}}_t) \right] + \frac{\lambda_n}{2} \mathbb{E} \left\| \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}^* \right\|_2^2 \right) + \eta^2 \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \\
&\overset{(d)}{\leq} \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \left(\mathbb{E}\left[h(\widehat{\boldsymbol{\alpha}}_t) \right] + \frac{\lambda_n}{4} \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 - \frac{\lambda_n}{2} \mathbb{E} \left\| \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}_t \right\|_2^2 \right) + \eta^2 \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \\
&= \left(1 - \frac{\eta \lambda_n}{2} \right) \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 + \eta \lambda_n \mathbb{E} \left\| \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}_t \right\|_2^2 - 2\eta \left(\mathbb{E}\left[h(\widehat{\boldsymbol{\alpha}}_t) \right] - \frac{\eta}{2} \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \right) \\
&\overset{(e)}{\leq} \left(1 - \frac{\eta \lambda_n}{2} \right) \mathbb{E} \left\| \boldsymbol{\alpha}_t - \boldsymbol{\alpha}^* \right\|_2^2 + \eta^3 \lambda_n \sum_{i=t-\tau}^{t-1} \mathbb{E} \left\| g_{m_i}(\widehat{\boldsymbol{\alpha}}_i) \right\|_2^2 - 2\eta \left(\mathbb{E}\left[h(\widehat{\boldsymbol{\alpha}}_t) \right] - \frac{\eta}{2} \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \right)
\end{aligned}$$

where (a) is from Lemma 7, (b) follows from (26), (c) is due to strong convexity of h, (d) uses triangle inequality of norms and (e) follows from Lemma 8.

Now, we telescope. Let $\gamma = 1 - \eta\lambda_n/2$

$$\begin{aligned}
&\mathbb{E} \left\| \boldsymbol{\alpha}_{T+1} - \boldsymbol{\alpha}^* \right\|_2^2 \\
&\leq \gamma^{T+1} \left\| \boldsymbol{\alpha}_0 - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \sum_{t=0}^{T} \gamma^{T-t} \mathbb{E}\left[h(\widehat{\boldsymbol{\alpha}}_t) \right] + \eta^2 \sum_{t=0}^{T} \gamma^{T-t} \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 + \eta^3 \lambda_n \sum_{t=0}^{T} \gamma^{T-t} \sum_{i>0, i=t-\tau}^{t-1} \mathbb{E} \left\| g_{m_i}(\widehat{\boldsymbol{\alpha}}_i) \right\|_2^2 \\
&\leq \gamma^{T+1} \left\| \boldsymbol{\alpha}_0 - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \sum_{t=0}^{T} \gamma^{T-t} \mathbb{E}\left[h(\widehat{\boldsymbol{\alpha}}_t) \right] + \sum_{t=0}^{T} \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \left(\eta^2 \gamma^{T-t} + \eta^3 \lambda_n \sum_{i=1}^{\tau} \gamma^{T-t-i} \right) \\
&= \gamma^{T+1} \left\| \boldsymbol{\alpha}_0 - \boldsymbol{\alpha}^* \right\|_2^2 - 2\eta \sum_{t=0}^{T} \gamma^{T-t} \left\{ \mathbb{E}\left[h(\widehat{\boldsymbol{\alpha}}_t) \right] - \frac{\eta}{2} \left(1 + 2 \frac{1 - \gamma^\tau}{\gamma} \right) \mathbb{E} \left\| g_{m_t}(\widehat{\boldsymbol{\alpha}}_t) \right\|_2^2 \right\} \\
&\overset{(a)}{\leq} \gamma^{T+1} \left\| \boldsymbol{\alpha}_0 - \boldsymbol{\alpha}^* \right\|_2^2
\end{aligned}$$

where (a) follows from Lemma 9, which also gives the constraint that

$$\eta \left(1 + 2 \frac{1 - \gamma^\tau}{\gamma} \right) \leq \frac{m}{\nu + \lambda_{q+1}^{(s)}(m-1)} := c$$

We calculate η in the worst case of $\tau \to \infty$

$$\eta \left(1 + \frac{2}{1 - \frac{\eta\lambda_n}{2}}\right) \leq c$$

$$\eta^2 \lambda_n - \eta(\lambda_n c + 6) + 2c \geq 0$$

One of the root cannot be used since $|\gamma| < 1$

$$\eta \leq \frac{1}{2\lambda_n}\left(\lambda_n c + 6 - \sqrt{(\lambda_n c + 2)^2 + 32}\right)$$

To eliminate λ_n, we can minimize w.r.t. λ_n. The minimum occurs at the maximum value of λ_n, which in our case is β/n. To get a simpler expression we consider large dataset i.e., large n. $n \to \infty$ gives

$$\eta \lesssim \frac{c}{3}$$

□

Lemma 7 (Non overlapping updates)**.** *All the changes to the parameter vector between the time a processor reads the parameter and updates it is in the orthogonal direction of the update to be done.*

$$\langle \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}_t,\ g_{m_t}(\widehat{\boldsymbol{\alpha}}_t)\rangle = 0 \tag{32}$$

Proof. Let r be the processor index that is updating at time t. Recall from Section 3 that each processor r updates indices of $\boldsymbol{\alpha}$ belonging to data $\mathcal{D}_r$ exclusively. Let the index set of this data $\mathcal{D}_r$ be $\mathcal{A}_r$. Each processor r has different preconditioner $\boldsymbol{P}_r$ whose Nystöm indices are chosen from $\mathcal{A}_r$ which results in $\boldsymbol{P}_r$ transforming only a subset $\mathcal{A}_r$ indices i.e.,

$$[\boldsymbol{P}_r]_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \qquad \forall i \notin \mathcal{A}_r \tag{33}$$

Also, note that only processor r can update indices in $\mathcal{A}_r$ and updates from one processor is sequential. So, between the time of read and time of update, the value in parameter vector corresponding to indices in $\mathcal{A}_r$ has not changed i.e.,

$$[\widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}_t]_i = 0 \qquad \forall i \in \mathcal{A}_r$$

Also, by the structure of $\boldsymbol{P}_r$ and definition of g_m, we know that only indices belonging to $\mathcal{A}_r$ are updated in each iteration.

$$[g_{m_t}(\widehat{\boldsymbol{\alpha}}_t)]_i = 0 \qquad \forall i \notin \mathcal{A}_r$$

which leads to

$$\langle \widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}_t,\ g_{m_t}(\widehat{\boldsymbol{\alpha}}_t)\rangle = 0$$

□

Lemma 8 (Bounding the perturbation)**.** *At all times, the norm of the change in parameter between the time of read and time of update is bounded as follows*

$$\mathbb{E}\,\|\widehat{\boldsymbol{\alpha}}_t - \boldsymbol{\alpha}_t\|_2^2 \leq \eta^2 \sum_{i=t-\tau}^{t-1} \mathbb{E}\,\|g_{m_i}(\widehat{\boldsymbol{\alpha}}_i)\|_2^2 \tag{34}$$

where τ is defined in definition 2 and $\mathcal{L}_0$ is the value of the square loss at initialization.

Proof. Using to equation (18)

$$\begin{aligned}\mathbb{E}\left\|\widehat{\boldsymbol{\alpha}}_t-\boldsymbol{\alpha}_t\right\|_2^2&=\mathbb{E}\left\langle\sum_{i=t-\tau}^{t-1}\eta\boldsymbol{Q}_i^t g_{m_i}(\widehat{\boldsymbol{\alpha}}_i),\ \sum_{i=t-\tau}^{t-1}\eta\boldsymbol{Q}_i^t g_{m_i}(\widehat{\boldsymbol{\alpha}}_i)\right\rangle\\&=\eta^2\sum_{i=t-\tau}^{t-1}\mathbb{E}\left\|\boldsymbol{Q}_i^t g_{m_i}(\widehat{\boldsymbol{\alpha}}_i)\right\|_2^2+\eta^2\sum_{\substack{i,j\\i\neq j}}\mathbb{E}\left\langle\boldsymbol{Q}_i^t g_{m_i}(\widehat{\boldsymbol{\alpha}}_i),\boldsymbol{Q}_j^t g_{m_j}(\widehat{\boldsymbol{\alpha}}_j)\right\rangle\\&\overset{(a)}{=}\eta^2\sum_{i=t-\tau}^{t-1}\mathbb{E}\left\|\boldsymbol{Q}_i^t g_{m_i}(\widehat{\boldsymbol{\alpha}}_i)\right\|_2^2\\&\overset{(b)}{\leq}\eta^2\sum_{i=t-\tau}^{t-1}\mathbb{E}\left\|g_{m_i}(\widehat{\boldsymbol{\alpha}}_i)\right\|_2^2\end{aligned}$$

Recall from Section 3 that updates from each processor is non-overlapping. Here g_{m_i}, g_{m_j} are updates from different processors. (a) follow from this fact. (b) is from the definition of $\boldsymbol{Q}_i^t$ which is a diagonal matrix with 0 or 1 in the diagonal entries. □

Lemma 9 (Interpolation to variance reduction)**.** *For $h(\boldsymbol{\alpha})$ defined in definition 7, we show the following*

$$\mathbb{E}_m\left[h(\boldsymbol{\alpha})-\frac{\eta}{2}\left\|g_m(\boldsymbol{\alpha})\right\|_2^2\right]\geq 0 \tag{35}$$

$$\textit{if}\quad \eta\leq\frac{m}{\nu+\lambda_{q+1}^{(s)}(m-1)} \tag{36}$$

where $\nu = max_i[\mathbf{K}\boldsymbol{P}^{-1}]_{ii}$

Proof. Define a function $\overline{h}(\boldsymbol{\gamma}):\mathbb{R}^n\to\mathbb{R}$ on the transformed space $\boldsymbol{\alpha}\mapsto\boldsymbol{P}^{-1}\boldsymbol{\gamma}$ as

$$\overline{h}(\boldsymbol{\gamma}):=\frac{1}{n}\left(\frac{1}{2}\boldsymbol{\gamma}^\top\mathbf{K}\boldsymbol{P}^{-1}\boldsymbol{\gamma}-\boldsymbol{\gamma}^\top\mathbf{K}\boldsymbol{P}^{-1}\boldsymbol{\gamma}^*\right)$$

For an arbitrary index i_k, define a function $\overline{H}_{i_k}:\mathbb{R}^n\to\mathbb{R}$ as follows

$$\overline{H}_{i_k}(\boldsymbol{\gamma}):=\mathbf{1}^\top\boldsymbol{Q}_{i_k}[\overline{h}_1(\gamma_1),\overline{h}_2(\gamma_2),\ldots,\overline{h}_n(\gamma_n)]^\top \tag{37}$$

$$\overline{h}_i(\gamma_i):=\overline{h}_i(\gamma_i;\gamma_{\backslash i}):=\overline{h}(\boldsymbol{\gamma}) \tag{38}$$

$\boldsymbol{Q}_{i_k}$ is a selector matrix with one non-zero entry at $[\boldsymbol{Q}_{i_k}]_{i_k i_k}=1$ and $h_i(\gamma_i):\mathbb{R}\to\mathbb{R}$ treats the i^{th} dimension of $\boldsymbol{\gamma}$ as variable and rest of the dimensions of $\boldsymbol{\gamma}$ as constants. The value of $h_i(\gamma_i;\gamma_{\backslash i})$ is same as $h(\boldsymbol{\gamma})$

For $\boldsymbol{\gamma}=\boldsymbol{P}\boldsymbol{\alpha}$,

$$\mathbb{E}_{i_k}[\overline{H}_{i_k}(\boldsymbol{\gamma})]=\overline{h}(\boldsymbol{\gamma})=h(\boldsymbol{\alpha}) \tag{39}$$

$$\nabla\overline{H}_{i_k}(\boldsymbol{\gamma})=\frac{1}{n}\boldsymbol{Q}_{i_k}\mathbf{K}\boldsymbol{P}^{-1}(\boldsymbol{\gamma}-\boldsymbol{\gamma}^*) \tag{40}$$

$$\nabla^2\overline{H}_{i_k}(\boldsymbol{\gamma})=\frac{1}{n}\boldsymbol{Q}_{i_k}\mathbf{K}\boldsymbol{P}^{-1}\boldsymbol{Q}_{i_k} \tag{41}$$

Define $\nu:=max_i\ [\mathbf{K}\boldsymbol{P}^{-1}]_{ii}$. We see that $\forall_i\overline{h}_i(\gamma_i)$ is $\frac{\nu}{n}$-smooth and also $\overline{H}_{i_k}(\boldsymbol{\alpha})$ is $\frac{\nu}{n}$-smooth

$$\overline{H}_{i_k}(\boldsymbol{\gamma})-\frac{1}{2\nu}n\left\|\nabla\overline{H}_{i_k}(\boldsymbol{\gamma})\right\|_2^2\geq 0 \tag{42}$$

Relating g_{1,i_k} defined in lemma 5 to $\nabla\overline{H}_{i_k}$

$$\mathbb{E}_{i_k}\left\|g_{1,i_k}(\boldsymbol{\alpha})\right\|_2^2=\mathbb{E}_{i_k}\left\|\boldsymbol{P}_r\boldsymbol{Q}_{i_k}\mathbf{K}\left(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*\right)\right\|_2^2 \tag{43}$$

$$\overset{(a)}{\leq}\mathbb{E}_{i_k}\left\|\boldsymbol{Q}_{i_k}\mathbf{K}\left(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*\right)\right\|_2^2 \tag{44}$$

$$=\mathbb{E}_{i_k}\left\|\boldsymbol{Q}_{i_k}\mathbf{K}\boldsymbol{P}^{-1}(\boldsymbol{\gamma}-\boldsymbol{\gamma}^*)\right\|_2^2=n\mathbb{E}_{i_k}\left\|\nabla\overline{H}_{i_k}(\boldsymbol{\gamma})\right\|_2^2 \tag{45}$$

where (a) uses lemma 4

Using (27)

$$\mathbb{E}_m \left[h(\boldsymbol{\alpha}) - \frac{\eta}{2} \left\|g_{m_t}(\boldsymbol{\alpha})\right\|_2^2\right] \geq h(\boldsymbol{\alpha}) - \frac{\eta}{2m}\mathbb{E}_{i_1} \left\|g_{1,i_1}(\boldsymbol{\alpha})\right\|_2^2 - \frac{\eta(m-1)}{2m} \left\|\nabla h(\boldsymbol{\alpha})\right\|_2^2$$

Choosing appropriate $\eta(p)$ for $p \in [0,1]$ and using (39)

$$\begin{aligned}\mathbb{E}_m \left[h(\boldsymbol{\alpha}) - \frac{\eta}{2} \left\|g_m(\boldsymbol{\alpha})\right\|_2^2\right] &\geq \mathbb{E}_{i_1} \left[p\overline{H}_{i_1}(\boldsymbol{\gamma}) - \frac{\eta(p)}{2m} \left\|g_{1,i_1}(\boldsymbol{\alpha})\right\|_2^2\right] + (1-p)h(\boldsymbol{\alpha}) - \frac{\eta(p)(m-1)}{2m} \left\|\nabla h(\boldsymbol{\alpha})\right\|_2^2 \\ &\overset{(a)}{\geq} \mathbb{E}_{i_1} \left[p\overline{H}_{i_1}(\boldsymbol{\gamma}) - \frac{\eta(p)}{2m} n \left\|\nabla \overline{H}_{i_1}(\boldsymbol{\gamma})\right\|_2^2\right] + (1-p)h(\boldsymbol{\alpha}) - \frac{\eta(p)(m-1)}{2m} \left\|\nabla h(\boldsymbol{\alpha})\right\|_2^2\end{aligned}$$

where (a) is from (45)

Solving $\eta(p) \leq \min \left\{\frac{mp}{\nu}, \frac{m(1-p)}{\lambda_{q+1}^{(s)}(m-1)}\right\}$ for $p \in [0,1]$ we get $\eta \leq \frac{m}{\nu + \lambda_{q+1}^{(s)}(m-1)}$. Now we can use smoothness below

$$\mathbb{E}_m \left[h(\boldsymbol{\alpha}) - \frac{\eta}{2} \left\|g_m(\boldsymbol{\alpha})\right\|_2^2\right] \geq p\mathbb{E}_{i_1} \left[\overline{H}_{i_1}(\boldsymbol{\gamma}) - \frac{1}{2\nu} n \left\|\nabla \overline{H}_{i_1}(\boldsymbol{\gamma})\right\|_2^2\right] + (1-p) \left[h(\boldsymbol{\alpha}) - \frac{1}{2\lambda_{q+1}^{(s)}} \left\|\nabla h(\boldsymbol{\alpha})\right\|_2^2\right] \geq 0$$

□

B. Hyperparameters

Table 6. Hyperparameters

Dataset	q	s	kernel	bandwidth
CIFAR-5M	2000	10000	gaussian	5
ImageNet9	2000	10000	gaussian	5
ImageNet	1000	10000	gaussian	5
HIGGS	300	10000	gaussian	5
TAXI	100	10000	gaussian	5