Training one DeePMD Model in Minutes: a Step towards Online Learning

Siyu Hu[1,2], Tong Zhao[1,2], Qiuchen Sha[1,2], Enji Li[1,2], Xiangyu Meng[3], Liping Liu[2,4], Lin-Wang Wang[2,4], Guangming Tan[1,2], Weile Jia[1,2]

1. State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
2. University of Chinese Academy of Sciences
3. College of Computer Science and Technology, Qingdao Institute of Software, China University of Petroleum
4. Institute of Semiconductor, Chinese Academy of Sciences

{husiyu20b,zhaotong,shaqiuchen22s,lienji23s,tgm,jiaweile}@ict.ac.cn,x.meng@s.upc.edu.cn,{liuliping,lwwang}@semi.ac.cn

Abstract

Neural Network Molecular Dynamics (NNMD) has become a major approach in material simulations, which can speed-up the molecular dynamics (MD) simulation for thousands of times, while maintaining *ab initio* accuracy, thus has a potential to fundamentally change the paradigm of material simulations. However, there are two time-consuming bottlenecks of the NNMD developments. One is the data access of *ab initio* calculation results. The other, which is the focus of the current work, is reducing the training time of NNMD model. The training of NNMD model is different from most other neural network training because the atomic force (which is related to the gradient of the network) is an important physical property to be fit. Tests show the traditional stochastic gradient methods, like the Adam algorithms, cannot efficiently deploy the multisample minibatch algorithm. As a result, a typical training (taking the Deep Potential Molecular Dynamics (DeePMD) as an example) can take many hours. In this work, we designed a heuristic minibatch quasi-Newtonian optimizer based on Extended Kalman Filter method. An early reduction of gradient and error is adopted to reduce memory footprint and communication. The memory footprint, communication and settings of hyper-parameters of this new method are analyzed in detail. Computational innovations such as customized kernels of the symmetry-preserving descriptor are applied to exploit the computing power of the heterogeneous architecture. Experiments are performed on 8 different datasets representing different real case situations, and numerical results show that our new method has an average speedup of 32.2 compared to the Reorganized Layer-wised Extended Kalman Filter with 1 GPU, reducing the absolute training time of one DeePMD model from hours to several minutes, making it one step toward online training.

CCS Concepts: • **Computing methodologies** → *Parallel algorithms.*

Keywords: Parallel training, Molecular dynamics, First principle, *ab initio*, GPU

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03
https://doi.org/10.1145/3627535.3638505

1 Introduction

Molecular Dynamics (MD) with *ab initio* accuracy is the method of choice in theoretical studies of many microscopic phenomena such as material defects [2], phase transition [38] and nanotechnology [41] and numerous other issues. Recent developed Neural Network Molecular Dynamics (NNMD) has advanced both numerical accuracy and computational efficiency, and it has resulted in a number of methods and packages such as SNAP [44], SIMPLE-NN [29], HDNNP [4–6], BIM-NN [48], CabanaMD-NNP [10], SPONGE [24], DeepMD-kit [46], Schnet [42], ACE [12, 33], NequIP [3], DimeNet++ [16, 17] and SpookyNet [45], etc. Although high-performance computing has greatly sped up the inference efficiency of NNMD models, e.g., DeePMD, a state-of-the-art NNMD package, can simulate billions of atoms with nanoseconds per day on top supercomputers [19, 26], the model training process remains a major issue in practical NNMD development and usage. More specifically, there are two challenges in NNMD training: 1). The absolute time for training one NN model with thousands of *ab initio* labeling data can be several hours; 2). Since the labeling data cannot cover all chemical space at *priori*, the training procedure is invoked repetitively to obtain a well-trained NN model. As shown in Figure. 1(a), for the same system, samples with different configurations will be incorporated for repetitive training. Furthermore, among different systems, each system needs to be trained individually due to the inherent differences, as shown in Figure. 1(b)(c). In one particular NNMD development, this retraining can take 20 to 100 times, as depicted in Figure. 1(d). Thus the practical use of the NNMD suffers significantly from the extended training time. This is common for many NNMD packages, like Schnet [42], NequIP [3], DeePMD-kit[46] etc. In this work, we use DeePMD as our example, as

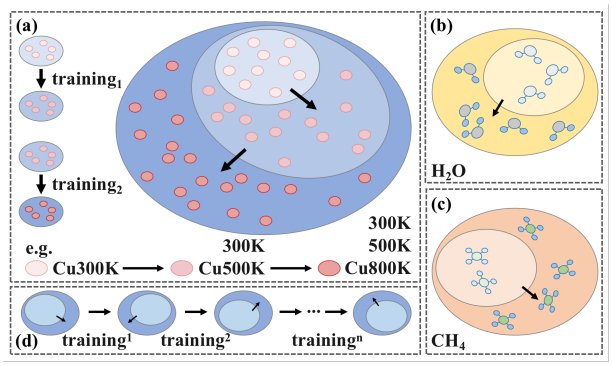

Figure 1. The repeated training in NNMD. (a) An example of the copper system under different temperatures. (b) and (c) is the H_2O and CH_4 retraining process. (d) The retraining loop.

it is a state-of-the-art NNMD method. It combines a physical symmetry-preserving descriptor and a deep neural network model. Also a broad spectrum of physical phenomena such as the phase diagram of water [52], nucleation of liquid silicon [7], diffusion in Lithium battery [34] and absorption of N_2O_5 [15], etc, has been studied using DeePMD.

An effective way to increase the computational efficiency of NNMD would be to use a larger minibatch method. Here, the batch size means the number of atomic configuration "images" to be used together for gradient calculations with the same model parameters. However, our test shows, that it is infeasible for DeePMD to apply a larger minibatch under the Adam training method currently deployed in the DeePMD package. When the larger minibatch is adopted in Adam based method, more tactical parameter tuning may be needed to avoid instability and convergence issues. To our knowledge, a universal parameter tuning strategy has not been found due to different situations of physical systems. Without fine-tuning, training by a large batch can even deteriorate the convergence rate. Table 1 shows the convergence of Adam-based DeePMD under different training batch sizes. Note that the training of Adam-based DeePMD with batch size 32 and 64 is readjusted by multiplying the learning rate with their square root of the minibatch respectively. For a simple copper bulk system, when increasing the batch size from 1 to 32, using the default training parameters, the corresponding number of epochs increases from 17 to 327 and the wall clock time extends from 9 hours to 20 hours. We remark that the default setting (scaling the learning rate by multiplying with the square root of minibatch size) converges faster than other heuristics such as adjusting the learning rate by multiplying the minibatch size. This indicates, currently, there is no practical way to increase batch size in the Adam algorithm used in many NNMD training. In summary,

Table 1. The Adam-based DeePMD convergence under different training batch sizes. The first two columns are the physical systems and converged Energy RMSE (training by DeePMD by using default setting: Adam optimizer with batch size set to be 1). The column "batch size" records the converged epochs under different batch sizes when reaching the given error (the second column). The column "epoch growth" is a factor of consumed number of epochs of batch size 32 to 1 and batch size 64 to 32. The "-" denotes that when the batch size increases, the error cannot decrease to the default Energy RMSE.

System	Energy RMSE(eV)	batch size			epoch growth	
		1	32	64	32/1	64/32
Cu	0.0427±0.0004	17	327	703	19.2x	2.1x
Mg	0.01565±0.00005	99	1528	3042	15.4x	2.0x
Al	0.7105±0.0002	52	680	1493	13.1x	2.2x
Si	0.0808±0.0057	31	375	755	12.1x	2.0x
CuO	0.0638±0.003	105	–	–	–	–
NaCl	0.0113±0.0002	31	777	1525	25.1x	2.0x
HfO2	0.5219±0.0007	45	576	1148	12.8x	2.0x
H2O	0.0582±0.0001	45	617	1174	13.7x	1.9x

a conflict arises between the requirement for repeated training and the essentiality of manually readjusting parameters for every training procedure.

The Adam algorithm is a first-order gradient algorithm in optimization. Theoretically, second-order gradient methods in optimization can have much better efficiency compared with first-order methods, and also importantly, it exhibits greater simplicity and robustness in hyperparameter tuning compared with first-order methods [36, 37] to some degree. The second-order optimizers can converge faster than first-order stochastic gradient descent (SGD) methods by leveraging curvature information to accelerate model training. In NNMD, a representative quasi-Newtonian method is Extended Kalman Filter(EKF) based optimizer. One example is the Reorganized Layer-wised Extended Kalman Filter(RLEKF) based DeePMD [23]. RLEKF can converge in much fewer epochs (11.7% epochs) compared with Adam optimizer. While the training wall clock time is still 80% of single-sample minibatch Adam-based DeePMD due to the additional calculation required in Kalman Filter theory and the instance-by-instance updating in RLEKF [23]. The second-order method has inherent advantages in NNMD model training. We believe an efficient multi-minibatch Extended Kalman Filter optimizer can further reduce the training wall clock time.

Optimizers, in essence, update each individual entry in the parameter vector. In the first-order optimizer, each entry has its own step size which is adaptively updated using gradients. In second-order methods, Hessian matrix is involved. To enhance practicability, a lot of approximation algorithms [9] have emerged to reduce the computation and communication cost. In this work, we propose Fast Extended

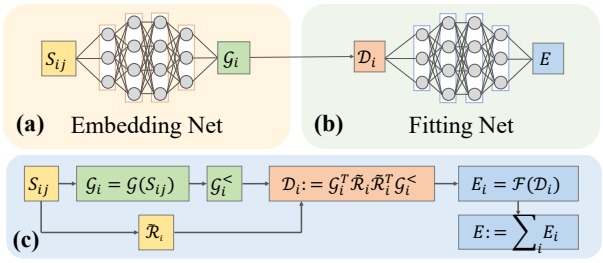

Figure 2. The network of DeePMD. (a) The 3-layer fully-connected embedding network. (b) The fitting net aims at fitting Energy potential from the Descriptor Di. The translation from Gi to Di is by the symmetry operation in (c). (c) The entire workflow of the DeepMD network.

Kalman Filter (FEKF) algorithm, a Quasi-Newtonian Method designed for fast convergence. FEKF is an approximation algorithm based on Kalman Filter Theory. The early reduction strategy is adopted to benefit from memory footprint and communication overhead. In the aforementioned copper training example, by using the FEKF optimizer, the 32-sample minibatch version can reduce the wall time from 26132s to 576s without sacrificing the final model accuracy. The batch size can scale up to 4096 and be distributed among 16 GPUs, reducing the absolute training time to 281s with little sacrificing model accuracy. Our contributions are:

- We propose a parallel framework for multi-sample minibatch Extended Kalman Filter training. Based on the framework we investigate two algorithms with different orders of computing and aggregation.
- We formulate several key factors to tune convergence and data movement in the parallel training algorithm. They provide support for fast convergence with reduced epoch numbers and communication costs.
- We implement GPU-friendly programming rules such as extracting and refining the derivative of the symmetry-preserving operator based on mathematical derivation.
- Testing is performed on 8 physical systems representing different situations. With no sacrifice of accuracy, the FEKF with training batch size 32 gained an average 11.61x algorithm speedup compared with RLEKF, and a further 3.25x averaged speedup is achieved via systematic optimization.

2 Background

2.1 DeePMD

The essential stages of DP training are illustrated in Figure. 2. To incorporate physical symmetries like translational, rotational, and permutational invariances into the atomic descriptor $\mathcal{D}_i$ can be viewed as the feature of atom i, an embedding net consisting of three fully connected layers is employed for each atom i. Subsequently, a fitting net with three layers is used to train $\mathcal{D}_i$.

1. The DeePMD neural network takes a snapshot of the molecular system as input, which includes the 3D Cartesian coordinates of each atom in the system. To represent the input, a neighbor list is constructed for each atom by including all other atoms that are within a certain cutoff distance r_c of that atom. The resulting neighbor list for each atom is then transformed into a smooth version, denoted as $\tilde{\mathcal{R}}i$, which is a matrix of size $N_m \times 4$, where N_m is the maximum length of all the neighbor lists. Each row of $\tilde{\mathcal{R}}i$ corresponds to a neighbor atom j of atom i and is a 4-dimensional vector given by $s(|\mathbf{r}_{ij}|)(1, \mathbf{r}_{ij}/|\mathbf{r}_{ij}|)$, where $s(x)$ is a smooth function that smoothly decays to zero between two thresholds, r_{cs} and r_c, and is $1/x$ for $x < r_{cs}$. The resulting matrix is then used as input to the DeePMD neural network for training and prediction.
2. The embedding net $\mathcal{G}i \in \mathbb{R}^{Nm \times M}$ is defined as $\mathcal{G}i = \mathcal{G}(s(|\mathbf{r}i\cdot|))$. Here, M is referred to as the symmetry order, and $\mathcal{G}$ is a composite function that involves three transformations: $\mathcal{E}0$, $\mathcal{E}1$, and $\mathcal{E}2$. The transformations are defined as follows:

$$\mathcal{E}_l(X) = X + \tanh\left(XW_l + \mathbf{1} \otimes w_l\right), l \in \{1, 2\},$$

$$\mathcal{E}_0(\mathbf{x}) = \tanh\left(\mathbf{x} \otimes W_0 + \mathbf{1} \otimes w_0\right),$$

where $\mathbf{1} \in \mathbb{R}^{Nm \times 1}, w_l \in \mathbb{R}^{1 \times M}, W_l \in \mathbb{R}^{N_m \times M}, l \in \{0, 1, 2\}$, and $\otimes$ denotes the outer product. tanh is an activation function.
3. The descriptor $\mathcal{D}_i$ is a matrix of size $M \times M^<$. It is defined as $\mathcal{D}_i := \mathcal{G}_i^{\mathrm{T}} \tilde{\mathcal{R}}_i \tilde{\mathcal{R}}_i^{\mathrm{T}} \mathcal{G}_i^<$, where $\mathcal{G}_i^<$ consists of the first several columns of $\mathcal{G}_i$.
4. The fitting net maps vectorized descriptor $\mathcal{D}_i$ with dimension $\mathbb{R}^{MM^<}$ into the potential energy E_i in the following way

$$E_i = \mathcal{F}(\mathcal{D}_i) = \mathcal{F}_3 \circ \mathcal{F}_2 \circ \mathcal{F}_1 \circ \mathcal{F}_0(\mathcal{D}_i),$$

$$\mathcal{F}_0(\mathbf{x}) = \tanh\left(\tilde{W}_0 \mathbf{x} + \tilde{w}_0\right),$$

$$\mathcal{F}_l(\mathbf{x}) = \mathbf{x} + \tanh\left(\tilde{W}_l \mathbf{x} + \tilde{w}_l\right), l \in \{1, 2\},$$

$$\mathcal{F}_3(\mathbf{x}) = \tilde{W}_3 \mathbf{x} + \tilde{w}_3,$$

where $\tilde{w}_0 \in \mathbb{R}^{d \times 1}$, $\tilde{W}_0 \in \mathbb{R}^{d \times MM^<}$, $\tilde{w}_l \in \mathbb{R}^{d \times 1}, \tilde{W}_l \in \mathbb{R}^{d \times d}, l \in \{1, 2\}$, $\tilde{w}_3 \in \mathbb{R}$, $\tilde{W}_3 \in \mathbb{R}^{1 \times d}$.
5. The outputs potential energy of systems of interest $E_{tot} := \sum_i E_i$, and force on atom i: $F_i := -\nabla_{\mathbf{r}_i} E_{tot}$.

The optimizer of the current DeePMD model is Adam. A widely used first-order method by the vast majority of applications in the AI for Science field for it is more stable, readable, and better supported by TensorFlow and PyTorch than second-order method. However, the biggest disadvantage of first-order methods compared to second-order methods is that they require more epochs for training, which leads to slow convergence rate. Therefore, there is an urgent need for new fast training algorithms. Research on second-order

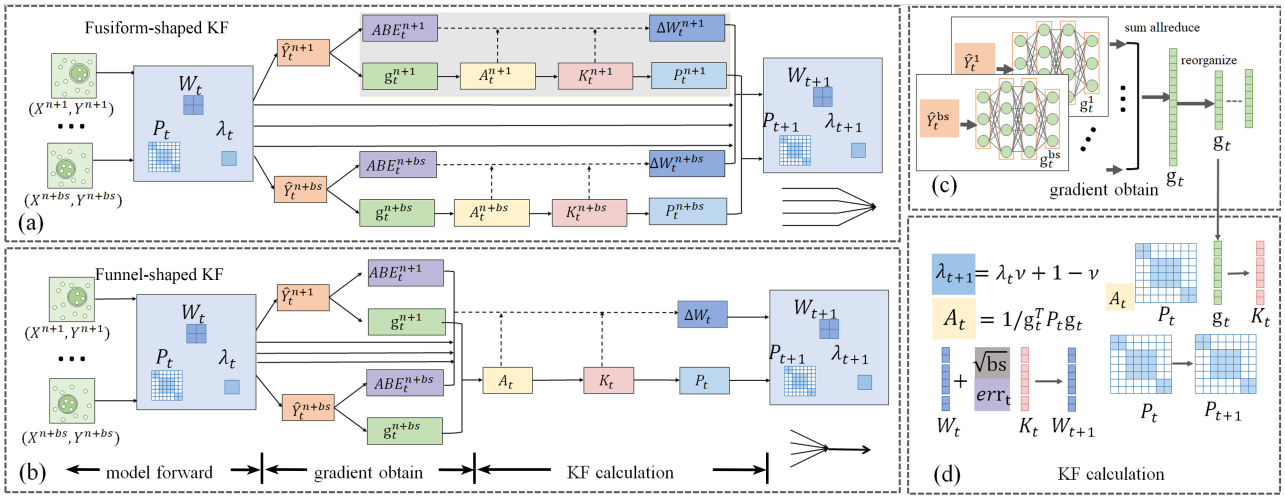

Figure 3. Parallelled multi-sample minibatch FEKF algorithms. (a) Fusiform-shaped EKF-based neural network training dataflow. It is a naive parallel EKF. (b) Funnel-based EKF-based neural network training dataflow. We call it Fast Extended Kalman Filter(FEKF). (c) Sum-Allreduce the gradient. (d) The formula of Kalman Filter updating.

methods is still scarce. Although the Newton method converges faster than first-order methods, its computational complexity is too high, and the algorithm's stability is poor due to the need to solve the inverse of the Hessian matrix. Luckily, Kalman filter has computational complexity and convergence speed that are both between those of the Newtonian methods and first-order methods, providing a tool for mitigating the conflict between the convergence speed and computational complexity of the algorithm. One of the typical Extended Kalman Filter-based methods is RLEKF [23] by organizing the expensive error covariance matrix into blocks to reduce the computational complexity.

2.2 RLEKF method

The DeePMD neural network can be represented as a stochastic system

$$\begin{cases} \boldsymbol{\theta}_t = \lambda_t^{-1/2}\boldsymbol{\theta}_{t-1}, \ \boldsymbol{\theta}_1 = \boldsymbol{w}, \\ y_t = h(\boldsymbol{\theta}_t, x_t) + \eta_t, \quad \eta_t \sim \mathcal{N}(0, L/\alpha_t^2), \end{cases}$$

in the form of a Kalman filter targeting on $\hat{\boldsymbol{\theta}}_t$ an estimator of $\boldsymbol{\theta}_t$, where L is the number of blocks dependent on RLEKF splitting strategy, $\boldsymbol{w}$ is the initialization of trainable parameters, $\boldsymbol{\theta_t}$ is the vector of all trainable parameters in the network $h(\cdot,\cdot)$, $\{(x_t, y_t)\}_{t\in\mathbb{N}}$ are pairs of feature and label, $\{y_t\}_{t\in\mathbb{N}}$ can also be seen as measurements of the system, $\{\eta_t\}_{t\in\mathbb{N}}$ are noise terms subject to normal distribution with mean 0 and variances $\{L/\alpha_t^2\}_{t\in\mathbb{N}}$ correspondingly, $\alpha_t := \Pi_{i=1}^{t}\lambda_i^{-1/2}$, $\alpha_0 := 1$, $0 < \lambda_i \leq 1$, $\lambda_i \to 1$.

The weights error covariance matrix of RLEKF is $\mathbf{P} = \mathrm{diag}(P_1, \ldots, P_L)$ a block diagonal matrix of shape $\{n_1^2, \ldots, n_L^2\}$ dependent on RLEKF splitting strategies, where n_i and $N := \sum_i n_i$ are the number of weights of the ith block and that of the whole neural network respectively.

For any given block, we recover the estimator of w via that of θ divided by the factor α_t, define $\forall t \in \mathbb{N}, \boldsymbol{w}_t := \alpha_t^{-1}\hat{\boldsymbol{\theta}}_t$, and then obtain

$$\begin{aligned} \mathbf{H}_t &= \left.\frac{\partial h(\boldsymbol{\theta}, x_t)}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}_{t|t-1}}, \\ \hat{\boldsymbol{\theta}}_{t|t-1} &:= \mathbb{E}[\boldsymbol{\theta}_t | y_{t-1}, y_{k-2}, \ldots, y_1], \forall t \in \mathbb{N}, \\ \mathbf{a}_t &= \lambda_t^{-1}\mathbf{H}_t^{\mathrm{T}}\mathbf{P}_{t-1}\mathbf{H}_t + L, \\ \mathbf{K}_t &= \lambda_t^{-1}\mathbf{P}_{t-1}\mathbf{H}_t\mathbf{a}_t^{-1}, \\ \mathbf{P}_t &= \left(\mathbf{I} - \mathbf{K}_t\mathbf{H}_t^{\mathrm{T}}\right)\lambda_t^{-1}\mathbf{P}_{t-1}, \\ \epsilon_t &= y_t - h(\boldsymbol{w}_{t-1}, x_t), \\ \boldsymbol{w}_t &= \boldsymbol{w}_{t-1} + \mathbf{K}_t\epsilon_t. \end{aligned}$$

We remark that RLEKF is a second-order method and it can converge to local minimum with a couple of epochs. Similar to other second-order methods like BFGS, Kalman filter gives $\mathbf{P}$ as an estimator of the inverse of optimization function's Hessian matrix with second-order derivative of optimization function term omitted. Compared with Adam, RLEKF requires more memory footprint in training since diagonal blocks of the matrix P are stored and updated in the training process.

3 Parallel Algorithm

The single-sample minibatch RLEKF [23] has demonstrated robustness while maintaining accuracy, effectively preventing gradient explosion. This results in properties similar to those of Fast Extended Kalman Filter (FEKF), as they are all variants of the Kalman Filter based Algorithms. This section

Table 2. The general weights increment formulation of the first-order method, second-order method, and quasi-Newtonian method(EKF) based method. The second row to the fourth row represents the updating rule under the single-sample x_i, the naive multi-sample minibatch (a set B), the approximated algorithm under multi-sample minibatch(a set B).

First-order	Second-order	quasi-Newtonian(EKF)
$-\eta \cdot g$	$-H^{-1}g$	$K \cdot ABE$; RLEKF
$\mathbb{E}(-\widehat{\eta} \cdot g)$	$\mathbb{E}(-H^{-1}g)$	$\mathbb{E}(K \cdot ABE)$; Naive-EKF
$-\widehat{\eta} \cdot \mathbb{E}(g)$	$-\mathbb{E}(H)^{-1}\mathbb{E}(g)$	$K(\mathbb{E}(g)) \cdot \mathbb{E}(ABE)$; Fast-EKF

is organized as follows: In Section 3.1, we will first introduce two multi-sample minibatch KF parallel algorithms(fusiform-shaped and funnel-shaped dataflow respectively). Our proposed FEKF adopts the funnel-shaped updating flow. Section 3.2 provides intuitive guidance on hyperparameter settings. Section 3.3 analyzes the memory footprint and communication overhead of the two KF parallel algorithms. To exploit the computational power provided by the heterogeneous architecture, we apply system optimizations such as derivative refinement and customized kernels, as discussed in Section 3.4.

3.1 Algorithmic Innovation

Neural network training is a process of iteratively adjusting neural network weights. Each iteration can be represented by $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \boldsymbol{\delta}^*$. Optimizers differ in their own unique ways in giving $\boldsymbol{\delta}^*$. The $\boldsymbol{\delta}^*$ of first-order and second-order methods is summarized in Table. 2. First-order method updates for each individual entry in the parameter vector with its own gradients. Second-order method adopts some matrices (preconditioners, such as Hessian) to transform the gradient. The taken expectation "$\mathbb{E}$" over the samples in one batch is empirically performed when we extend the update procedure of a single-sample x_i to a multi-sample set B – a set of randomly selected samples, shown in the second and third rows of Table. 2. The approximation algorithm is then empirically proposed to reduce computational time and resources compared to exact algorithms, transit from the third row to the fourth row of Table. 2. In summary, the per-sample gradient g and Hessian H change to the mean gradients and Hessian of B, denoted as $\mathbb{E}(g)$ and $\mathbb{E}(H)$, respectively. Their definition is shown in eq. 1 where L denotes the loss function.

The EKF-based optimizer is a quasi-Newtonian method and is depicted in the last column of Table. 2. A typical work is RLEKF by using one instance x_i in neural network weights updating. The weights' increment of sample x_i at the timestep t is the product of Kalman gain vector (denoted by K) and the Absolute error of predictions and label (denoted as ABE).

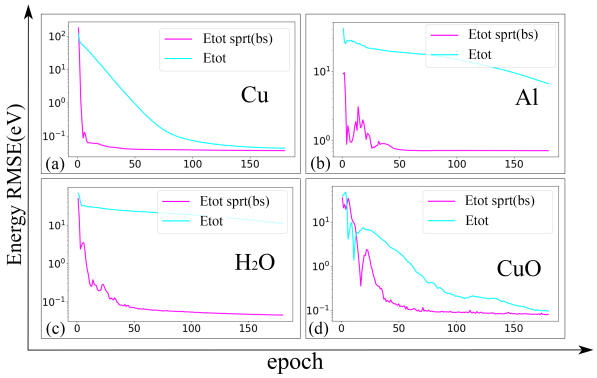

Figure 4. Effect of quasi-learning-rate Factor on Convergence Rate of Energy.

$$
\begin{aligned}
g &= \nabla L(f(x_i; w_t), y_i) \\
\mathbb{E}(g) &= \frac{1}{|B|} \sum_{x_i \in B} \nabla L(f(x_i; w_t), y_i) \\
H &= \nabla^2 L(f(x_i; w_t), y_i) \\
\mathbb{E}(H) &= \frac{1}{|B|} \sum_{x_i \in B} \nabla^2 L(f(x_i; w_t), y_i)
\end{aligned}
\tag{1}
$$

Fusiform-shaped dataflow: Averaging over samples in one batch is favored and empirically employed by first-order and second-order optimizers when batch size increases from 1 to a larger one. Theoretically, the weights' increment $\boldsymbol{\delta}^*$ of EKF is approximated by the mean value of the product of Kalman gain and the Absolute error, written as $\mathbb{E}(K \cdot ABE)$, namely, a Naive-EKF, shown in the last column of the third row in the Table. 2. It can be derived by statistically averaging over each sample $\Delta W_t^{n+i}, i = 1, \cdots, bs$. The brace of individual ΔW_t^{n+i} in Figure. 5(a) means a reduction is performed. We view it as an **"computing-then-aggregation"** mode since the KF computation is performed on each individual sample.

Funnel-shaped dataflow: An "early reduction" is adopted (i.e. approximating $\mathbb{E}(K \cdot ABE)$ by $K(\mathbb{E}(g))\mathbb{E}(ABE)$). This is inspired by the first and second-order approximating methods, When differentiating of samples first occurs in the backward pass, the emerge operation will be taken. A typical example of the second-order method is reducing over the Hessian matrix and gradients, denoted as $\mathbb{E}(H)$ and $\mathbb{E}(g)$ respectively in Table. 2. In our Fast Extended Kalman Filter, we perform reduction operations at the very initial point. More specifically, the reduction is applied to the Absolute errors ABE_t^{n+i} and the gradients g_t^{n+i}, where $i \in \{1, \cdots, bs\}$, shown in the last column of the fourth row in the Table. 2. The workflow of FEKF is in **"aggregation-then-computing"** mode which is presented in Figure. 5(b). The gradient reduction

Algorithm 1 Fast Extended Kalman Filter(FEKF)

1: Input:$\{\hat{Y}\}, \{Y\}, \{w_t\}, \{P_t\}, \{\lambda_t\}$
2: Initialization: $P_{t=0}$ is a $n \times n$ identity matrix, λ_0, ν
3: **if** $\hat{Y} \geq Y$ **then**
4: $\quad \hat{Y} = -\hat{Y}$
5: **end if**
6: $ABE_t = AbsoluteError(Y, \hat{Y})$
7: $g_t = \hat{Y}.sum().backward()$
8: $A_t = 1/(\lambda + (g_t)^T P_t g_t)$
9: $K_t = A_t P_t g_t$
10: $P_t = (1/\lambda) \times (P_t - (1/A) K_t K_t^T)$
11: $P_{t+1} = (P_t + (P_t)^T)/2$
12: $\lambda_{t+1} = \lambda_t \nu + 1 - \nu$
13: $w_{t+1} = w_t + \sqrt{bs} \times ABE_t \times K_t$
14: Output:$\{w_{t+1}\}, \{P_{t+1}\}, \{\lambda_t\}$

is performed across samples within a batch, illustrated in Figure. 5(c).

The FEKF calculation is described in Figure. 5(d). The corresponding description of FEKF update process is provided in Algorithm 1. Algorithm 1 Line 1 denotes the input of FEKF optimizer where $\hat{Y}$ and Y denote the model predictions and labels of the current batch, w_t, P_t, λ_t represent the weights, weights' error covariance matrix and memory factor at the t*th* iteration. Algorithm 1 Line 2 introduces that the $P_{t=0}$ is initialized as an identity matrix, λ_0, ν are hyper-parameters. Line 3-6 of Algorithm 1 is the *ABE* calculation process. Line 6 represents the batched gradients aggregation. The standard EKF updating rule is shown in Line 8-12 of Algorithm 1. In order to obtain a faster convergence, weights are updated heuristically, shown in Algorithm 1 Line 13. The last Line of Algorithm 1 indicates the obtained(post-update) weights, weights' error covariance matrix, and memory factor at time step $t + 1$.

3.2 Tuning for Fast Convergence

Finding the quasi-learning-rate: In the FEKF, the square root of batch size is applied in the weights updating. It can be viewed as the expected length of the sum of several gradients if these gradients are chosen randomly and uniformly in all possible directions with unit length. Hence, the weights update rule of FEKF is set as Eq. 2. We verify the effectiveness of Eq. 2 and can easily draw a conclusion from Figure. 4 that the square root of batch size leads to faster convergence.

$$w_{t+1} = w_t + \sqrt{bs} \cdot K_t(\mathbb{E}(g_t)) \cdot \mathbb{E}(ABE_t) \tag{2}$$

Hyper-parameters setting: The second-order method requires less hyper-parameters tuning than the first-order method. We provide a task-independent parameter setting guideline that can be applied across various training systems. In FEKF, the hyper-parameter adjustment is only related to the training batch size. Hence, the practicability of FEKF

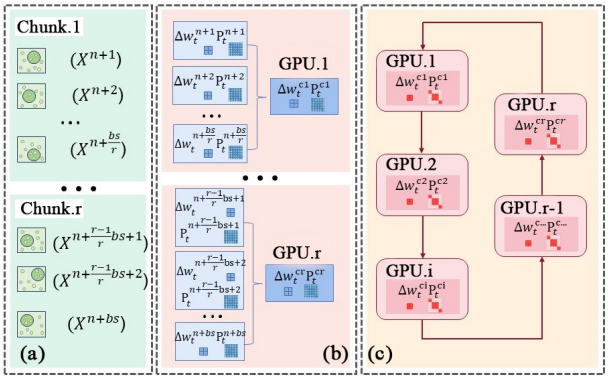

Figure 5. The data distribution, calculation, and communication of the naive fusiform-shaped multi minibatch EKF. (a) The samples are first divided into multiple chunks. (b) The data in different chunks are calculated on corresponding GPUs. (c) The communication overhead under Ring-Allreduce operation.

is significantly enhanced. Next, we will examine the effect of training batch size on the memory factor λ. The memory factor λ updates via Line 12 of Algorithm. 1, coupling with decay factor ν. The default settings are $\lambda = 0.98$ and $\nu = 0.9987$. As $t \to \infty$, λ is getting arbitrarily close to 1. Let's rewrite it to Eq. 3 to better analyze λ. The increment of λ consists of two parts. The $(1 - \nu)$ can be taken as a constant once ν is first determined. $1 - \lambda_t$ decreases to 0 as t increases. Hence, the increment of λ gets smaller as t goes infinite. As batch size increases, a larger step size $(1 - \nu)$ is usually adopted. Hence, ν is supposed to set a smaller value. The lower initial λ is required to make sure a wider range of λ where t changes from 0 to infinite. In our test, when the training batch size exceeds 1024, the recommended λ, ν are set to 0.90 and 0.996 respectively. We remark λ and ν are the only adjusted hyper-parameters when using large batch sizes under various systems.

$$\lambda_{t+1} = \lambda_t + (1 - \nu)(1 - \lambda_t) \tag{3}$$

3.3 Tuning for Data Movement

Memory footprint reduction: As the training batch size increases, the FEKF will not suffer as substantial memory footprint overhead as the Naive-EKF, because samples within a batch share a uniformed P matrix. In the Naive-EKF, each sample updates independently and has an individual P matrix. This becomes unbearable when a large batch is adopted in training.

Communication avoidance: In the FEKF, averaging over gradients and errors ensures an identical replica of the P matrix among different GPUs. Hence, the communication overhead of P is eliminated. To illustrate the communication of FEKF, we will first introduce Figure. 5. Assuming that one minibatch data is first separated in r Chunks. The

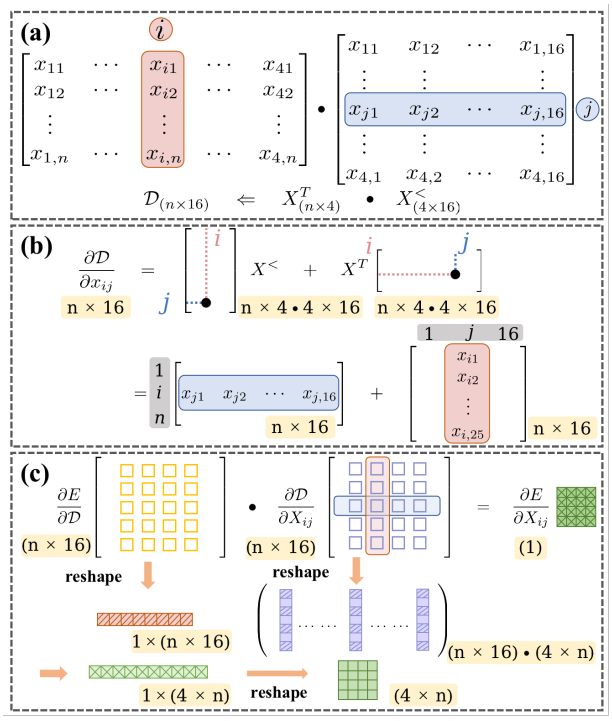

Figure 6. The computation of the first order derivatives of energy E to R^TG. (a) The abstraction of the symmetry-preserving operation. Briefly denoted as $D = X^TX^<$ where $X = R^TG$. (b) The derivative of descriptor D with respect to x_{ij}. (c) The derivative of Energy E with respect to x_{ij}. (b) and (c) shows the calculation process of Eq. 4.

training is then paralleled among r GPUs. Under a typical ring-Allreduce communication mode, the communication volume is $(r-1)\cdot N\times N$ where r is the number of GPUs and N is the number of weights parameters. Even in a smart decouple strategy(decoupling P into blocks; an even-splitting strategy [23]), the order of memory occupation for each block is $O(N_b^2)$ and the number of the block is of order $O(N/N_b)$. Hence the total communication of FEKF is of order $O((r-1)(N/N_b)N_b^2) = O((r-1)NN_b)$, where N_b is the threshold of splitting blocksize. The communication of P of Naive-EKF still hinders large-scale distributed training.

3.4 Specific Implementation Optimization

The network optimization: As detailed in section 2.1, DeePMD has a unique symmetry-preserving descriptor $\mathcal{D}$ that is constructed via the output of the embedding network and then trained by the fully connected fitting network. The system energy is fitted via a forward network, and atomic forces are derived by backward propagation to maintain energy conservation. Note that force field $F_i := -\nabla_{\mathbf{r}_i}E$ involves first-order derivative calculations and is realized by Autograd API of Machine Learning Framework. We observe a lot of fragmented kernels being launched by using Autograd API. To this end,

Table 3. Dataset description. Physical systems, the temperature at which the data was generated, time steps (frequency of yielding snapshots), the number of snapshots (images), and the atom number in one snapshot are indicated from left to right.

Systems	Temperature(K)	Time Step(fs)	# Snapshots	atom number
Cu	400-800	2	72102	108
Al	300,500,800,1000	2	24457	32
Si	300,500,800	3	40000	72
NaCl	300,500,800	2	40000	64
Mg	300,500,800	3	12800	36
H_2O	300,500,800,1000	1	28032	48
CuO	300, 500, 800	3	10281	64
HfO_2	-200-2400	1	28577	98

we implement the force calculation manually. In this section, we focus on the derivative of the symmetry-preserving operation. We remark that the symmetry-preserving principle is widely applied in physics simulation and the descriptor $\mathcal{D}$ is critical for accurately describing the force field.

Problem statement: The $\mathcal{D}$ is written as $X^TX^<$. Our goal is to calculate $\frac{\partial E}{\partial X}$.

Problem decomposition: The first step is to calculate the $\frac{\partial \mathcal{D}}{\partial X}$ by the "product derivative rule". The second step is to calculate the $\frac{\partial E}{\partial X}$ by the "chain rule".

Problem solution: To better understand, we degenerate the derivative of the tensor X to a scalar x_{ij} and by iterating over i and j to get the complete derivative of X. The decomposed two problems can be mathematically expressed in Eq. 4. The computation and actual tensor shape are illustrated in Figure 6(b) and (c) respectively.

$$\begin{aligned}\frac{\partial D}{\partial x_{ij}} &= \frac{\partial X^T}{\partial x_{ij}}X^< + X^T\frac{\partial X^<}{\partial x_{ij}}\\ \frac{\partial E}{\partial x_{ij}} &= \frac{\partial E}{\partial \mathcal{D}}\cdot\frac{\partial \mathcal{D}}{\partial x_{ij}}\end{aligned} \tag{4}$$

The optimizer optimization: (1) Rewrite P updating: According to the updating rule of P matrix, shown in Line 10 of Algorithm. 1. Note that the $K\cdot K^T$ operation is involved where K is with the shape $N\times 1$. In machine learning frameworks, the backend invokes CUDA GEMM kernels and they are highly optimized. For example, the tiling strategy is often used to efficiently utilize the GPU's memory hierarchy. A simple tiling method is introduced in Supplementary.I and $ktile \geq 8$. The number of multiply-add operations of $K\cdot K^T$ required is $2N^2$ but increased by a factor of $ktile$ by using torch.matmul() API. Hence, we substitute the pytorch implementation of Algorithm. 1 Line 10 with a handwritten kernel. The number of floating-point operations and memory access are all reduced. (2) Cache intermediate results: The A calculation in Line 8 of Algorithm. 1 involves Pg, which is required

Table 4. The convergence ratio of 32-sample minibatch FEKF with regard to single-sample minibatch Adam. The root mean square errors (RMSE) of the training(before slashes) and the testing set (after slashes). The testing is conducted on 8 systems.

Systems	Convergence ratio		Root Mean Square Error(RMSE)	
	Adam batch size 1	FEKF batch size 32	Adam batch size 1	FEKF batch size 32
Cu	17	0.118	0.0885 ±0.0010 /0.0851 ±0.0016	0.0862 ±0.0009 /**0.0736 ±0.0010**
Al	52	0.192	0.7592 ±0.0003 /0.1453 ±0.0007	0.7467 ±0.0085 /**0.1091 ±0.0023**
Si	31	0.097	0.1509 ±0.0094 /0.1547 ±0.0098	0.1264 ±0.0039 /**0.1365 ±0.0024**
NaCl	31	0.226	0.0170 ±0.0004 /0.0162 ±0.0001	0.0165 ±0.0000 /**0.0155 ±0.0001**
Mg	99	0.071	0.0360 ±0.0001 /0.0388 ±0.0001	0.0335 ±0.0000 /**0.0379 ±0.0002**
H_2O	45	0.089	0.1157 ±0.0002 /0.1215 ±0.0011	0.1119 ±0.0000 /**0.1128 ±0.0000**
CuO	105	0.105	0.1248 ±0.0000 /0.1367 ±0.0017	0.1219 ±0.0017 /**0.1236 ±0.0014**
HfO_2	45	0.222	0.7595 ±0.0010 /0.7782 ±0.0000	0.7473 ±0.0127 /**0.7396 ±0.0778**

in K updates (Line 9 of Algorithm. 1). Supplementary.II illustrates the calculation procedure. The intermediate results Pg are cached for K reuse, as Supplementary.II shows.

4 Experiments Setup

Hardware and software stacks. All numerical tests were conducted using the GPU cluster, which consists of 629 computing nodes. Each node is equipped with 2 64-core Kunpeng 920s (ARM architecture) and 4 NVIDIA A100 GPUs. Each node has 256GB host memory (8 channels × 32GB), and each A100 GPU has memory capacity of 40GB and a bandwidth of 900GB/s. The CPUs and GPUs are connected via a PCIe 4.0 with a bandwidth of 64 GB/s. The computing nodes are interconnected with a non-blocking fat-tree topology using RoCE interconnect, providing a total bandwidth of 25 GB/s. The Pytorch 2.0 framework and Horovod 0.27.0 are utilized in training the DeePMD model. Gcc 9.3 and CUDA 11.8 are used as our CPU and GPU compilers, respectively.

Dataset. The systems tested are bulk systems, which pose a much greater challenge for AIMD training than small molecular simulations. One of the reasons is the relatively larger number of total atoms in one sample, usually with over 50 atoms per image. Another reason for making the training task more complex than small molecules is the various configurations since the samples are mixed with different temperatures when generating. The sub-systems and their corresponding temperature are listed in the first two columns of Table. 3, ranging from -200 to 2400 Kelvins. For each dataset, snapshots are yielded based on solving *ab initio* molecular trajectories via PWmat [25] except for HfO_2 [47]. During this process, to enlarge the sampling span of configuration space, we fast generate a long sequence of the snapshot by a small time step (the third column of Table. 3) and choose one for every fixed number. The number of samples are ranging from 10k-70k in the fourth column.

Model parameters. The DeePMD network size is [25, 25, 25] and [400, 50, 50, 50, 1] for embedding net and fitting net respectively. The truncation value under the symmetry preserving operation $G^T RR^T G^<$ is set 16. The activation is tanh. The number of parameters is 26651. In Adam optimizer single-sample-batch training procedure, the learning rate is 0.001 and exponential decay by 0.95 for every 5000 steps. In the EKF optimizer training, the blocksize is set to 10240. The neural network weights are updated one time with total Energy and four times with atomic force.

5 Numerical Results

5.1 Convergence results

Accuracy: The Adam single-sample minibatch DeePMD shows the SOTA accuracy in many published papers. In the experiment part, we set the precision at which Adam single-sample minibatch training RMSE (the summation of Energy RMSE and Force RMSE) as the baseline. The RLEKF exhibits comparable accuracy with Adam-based DeePMD. We claim that FEKF can also reach a promising accuracy as the baseline accuracy. Table. 4 describes the accuracy results of Adam batch size 1 and FEKF with training batch size 32. For the eight physical cases under the Adam batch size 1 and FEKF batch size 32, the RMSE of FEKF is slightly lower than that of Adam (the lower the better).

Generalization gap: The FEKF with training batch size 32 does not suffer a generalization problem, shown in the last column of Table. 4. The training set RMSE (before slash) and testing set RMSE (after slash) are with fewer differences, ranging from 0.0009 to 0.0798.

Convergence ratio: The faster convergence is guaranteed by a factor (the square root of batch size) when updating weights. In Section 3.1, We have provided intuition to this strategy and experiments supported it well. The convergence (epoch) ratio is used in convergence measurement. The third column of Table. 4 is the ratio of single-sample minibatch Adam to 32-sample minibatch FEKF with regard to the converged epochs number. The lower convergence ratio means a smaller number of convergence epochs in FEKF, indicating a faster training procedure.

5.2 End-to-end Training Time

We unbiasedly choose 8 representative datasets and comprehensively test these systems. Figure. 7(a) describes the training wall clock time of training by single-sample minibatch

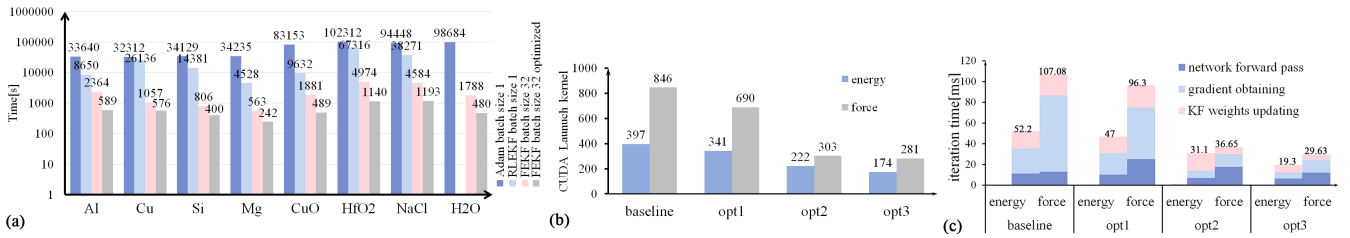

Figure 7. (a)End-to-end training time of Adam, RLEKF and EFKF optimizers.(b)The number of CUDA kernels launched under step-by-step system optimization.(c)The iteration time under step-by-step system optimization.

Table 5. Training wall time of Cu system under different configurations. Each column represents a configuration(the batch size and the number of GPUs in experiment testing) and the corresponding results(the end-to-end time and speedup).

Cu72102	RLEKF	FEKF		
batch size(#GPUs)	1(1)	32(1)	512(4)	4096(16)
Time(Speedup)	26136s(1x)	576s(54x)	360(72x)	281s(93x)

Adam, single-sample minibatch RLEKF, the 32-sample minibatch FEKF, and the 32-sample minibatch FEKF after system optimization. Without loss of accuracy, the speedup of 32-sample minibatch FEKF to single-sample minibatch RLEKF over the eight systems is from 3.66× to 24.69× and the average speedup is 11.61×. The optimized 32-sample minibatch FEKF can attain a further 2.02× to 4.36× speedup compared with the 32-sample-batch FEKF without optimization. The average speedup is 3.25× in terms of system optimization. The overall average speedup is 32.22×. The training wall clock time of Figure. 7(a) is measured under the accuracy referring Table. 4. FEKF can further reduce the wall time when employing a larger batch size. For a typical Copper system, as illustrated in Table. 5, when reaching 1.5× the baseline accuracy, the training can scale up to 16 GPUs with a batch size of 4096, resulting in a speedup of 93×.

5.3 Performance Analysis

We have conducted extensive optimizations (detailed in section 3.2) to fully utilize the computing power of GPU. The testing is performed on A100 GPU and chooses a typical Cu72102 dataset as an example. The training batch size is 64. Figure. 7(b) describes the number of CUDA launch kernels. Figure. 7(c) records iteration time. The x-axis of Figure. 7(b) and Figure. 7(c) represents the different optimization configurations. The listed baseline is the original version. Opt1 substitutes the Autograd with handwritten kernels. Opt2 adopts the torch.compile(model) API to automatically fuse the launched kernels. Opt3 optimizes the FEKF updating process by using a customized kernel(P_t calculation) and computation reuse($P_t g_t$). For each configuration, the left column is the FEKF updating by using Energy predictions and the right column is the FEKF updating by using Force predictions.

CUDA launched kernel: The system optimization greatly reduces the launched CUDA kernels as shown in Figure. 7(b). Upon initial observation, the launched CUDA kernels decreased from 397 to 174 and 846 to 281 in terms of FEKF updating by Energy and Force respectively. The training consists of one Energy-based RLEKF update and four Force-based updates. The overall number of CUDA launched kernels is from 3781(397+846x4) to 1298(174+281x4). The number of Kernels is reduced by 64% compared to the baseline.

Iteration time: In Figure. 7(c), we separate the whole updating process into three parts, represented by different shades on the bar. From bottom to top, it represents three independent processes. The first part is going through the network forward pass to get the predictions and errors. The second part is the gradient obtaining which is required in EKF updating. The third part is the FEKF calculation flow. The total iteration time is 3.48× faster when the systematic optimizations have been applied step by step. One of the major reasons for the wall time reduction is the decrease in the number of launched kernels.

Memory reduction: The updating of the P matrix is substituted by a customized handwritten kernel. The memory footprint can be reduced compared with the Pytorch implementation of P updating. In the copper system with training batch size set to 1, the peak memory usage can be reduced from 3380MB to 1805MB. To figure out the peak memory usage, we have to introduce the following prerequisite: In the DeePMD network, the number of parameters is 26651. The blocksize used in FEKF method is 10240. The weights error covariance matrix of FEKF is $\mathbf{P} = \mathrm{diag}(P_1, P_2, P_3, P_4)$ a block diagonal matrix of shape $\{1350^2, 10240^2, 9760^2, 5301^2\}$ by the gather and splitting strategies of [23]. The P_1, P_2, P_3, P_4 have a memory consumption of 13.90, 800, 726.76, 214.39 in MB respectively. The $\mathbf{P}$ has a memory footprint of 1755MB. The peak memory usage of optimized FEKF is 1805MB, consisting of the $\mathbf{P}$, weights, and intermediate variables. During the real computation process, the peak memory usage will be larger than this allocation, for the reason that some additional intermediate variable results will be generated. The implemented P updates by Pytorch, inevitably bring a P_i matrix writing

when calculating the out product of $K_i K_i^T$. The subtraction operation occurs in P_i calculation introduces a $N_i \times N_i$ matrix memory read/write overhead. The dominant intermediate memory footprint of P is from P_2. Hence, the memory allocation is twice the P_2 footprint. Hence, the whole peak memory usage is 3405(2×800+1805)MB in theory. In summary, the handwritten kernel can reduce memory consumption to twice the memory footprint of $max\{P_i\}, i = 1, \cdots, L$.

Scalibilty Analysis: We remark that the communication overhead of FEKF is approaching that of first order optimizer when the number of GPUs(#GPUs) is far less than the number of neural network parameters. The distributed error covariance matrix Ps do not need to be communicated because they are uniform already. In the FEKF and Adam-based method, the reduction in gradients is all required for latter iteration usage. The gradient $g = \{g_1, g_2, g_3, g_4\}$ of FEKF is with a shape $\{1350 \times 1, 10240 \times 1, 9760 \times 1, 5301 \times 1\}$. Their memory footprints are 0.01, 0.08, 0.07, and 0.04 in MB respectively. The overall memory usage of the neural network gradient is 0.2MB(Mem(g)). The communication of gradients is $(\#GPUs-1)\times Mem(g)$. The only additional communication overhead of FEKF is from Absolute Error(ABE). The communication of ABE is the order of $O(\#GPUs)$. That is to say, if $\#GPUs \ll N$, the predominant communication overhead in FEKF arises from the gradient, and the communication of ABEs can be ignored.

6 Related work

Optimizers play a fundamental role in training neural networks. Adam [27], AdamW [32], SGD [18] are widely used in training deep neural networks. They are regular and standard first-order methods(Stochastic Gradient based methods). With the advent of large-scale datasets, training deep neural networks usually takes days [11, 21]. Due to the recent hardware advances, a feasible method to tackle the training issue is applying large mini-batches. Plenty of strategies [18, 22] have been proposed for effectively large mini-batch training. One of the straightforward strategies is that the learning rate multiplies the square root of the minibatch size [22]. This is based on the variance-keeping principle when using large mini-batches. The linear scaling is multiplying the learning rate with minibatch size [18]. By using linear scaling with LR warm-up, Resnet-50 is trained with batch size 8K without loss in accuracy [18]. However, these methods need hand-tuned warmup to avoid instability and can be detrimental beyond a certain batch size. Another strategy is then proposed for training larger minibatch by using adaptive learning rates mechanisms, such as LARS [50] and LAMB [51] The LARS [50] and LAMB [51] are representative methods while their performance differs in networks. More specifically, LARS works in ResNet50 and LAMB gains performance on attention models. Although the first-order methods based on large minibatch training have been deeply investigated, some recently sprung-up AI for Science networks (such as DeePMD [46], NequIP [3], etc) still adopt small mini-batches in the training procedure. The large minibatch training is not successfully applied to these NNMD networks due to the following reasons: The network training is strongly coupled with the physical systems. Each system will train a customized network that is tailored for the given system. The high cost of the necessary hand-tuning warmup prelude on all various training systems is not acceptable. Hence, the universal training method suitable for each system is more favored.

The second-order optimizers use second derivatives and/or second-order statistics of the data to speed up the iteration. The well-known second-order optimizers, such as Shampoo [20, 43], K-FAC [35], SP-NGD [39], BFGS [31, 54], AdaHessian [49], THOR [13], RLEKF [23] are proposed for its faster convergence speed. These second-order methods are formulated based on distinct theoretical foundations. In this paper, we focus on the Extended Kalman Filter Theory based method. The Kalman Filter method is robust and well-founded. While the implementations of KF used in neural network weight updating of real applications are seldom studied. [23] is one of the works that is a instance-by-instance weights updating implementation. In this paper, we propose a paralleled FEKF in neural network weights updating.

Kernel fusion aims at reducing the number of launched kernels. It enables better sharing of computation and eliminates intermediate allocations [30]. Kernel fusion is an indispensable Deep Learning oriented optimization supported by Deep Learning compilers, such as TVM [8], TensorRT [53], etc. However, their main effort is to make inferences more efficiently. XLA [28] compiler can be used in model training while only supporting Tensorflow [1] and JAX [14] Framework. Expanding the capability of Deep Learning compilers to support model training on more popular Deep Learning frameworks such as PyTorch [40], is imperative. In summary, the implemented kernel fusion in our paper is focused on model training and differentiation based refinement.

7 Conclusion

The slow convergence of first-order Adam and instance-by-instance RLEKF method have hindered the time-to-solution of the DeePMD model. In this paper, we propose a parallelled KF algorithm namely FEKF. FEKF can be used to accelerate the training process in low-batch with no sacrifice in accuracy. As a quasi-Newtonian method, FEKF requires less hyperparameter tuning compared to first-order methods. This naturally fits the requirements of repetitive training in real scenarios. We give a heuristic parameter-tuning strategy which is independent of the specific training tasks. Hence, the hand tuning in each training is eliminated. FEKF has benefits for memory allocation and data movement. Besides, system optimizations such as kernel fusion are adopted to

further exploit the computing power of the heterogeneous architecture. Comprehensive tests on eight physical systems demonstrate consistent effectiveness in convergency.

There are also two main issues with EKF methods: first, a suitable and theoretically proven loss function is needed for classification tasks. Second, the P decoupling strategy needs to be adjusted for different network architectures like CNN, RNN, GNN, and Transformers.

In the future, we will work on the rigorous theoretical proof of this FEKF and adapt FEKF to support model parallelism. AI-for-Science neural network training is growing more and more important. The proposed FEKF sheds light on other AI-for-Science models and other fields such as computer vision and natural language processing.

Acknowledgments

Numerical experiments are performed at the supercomputer center in Wuhan. This work is supported by the following funding: National Key Research and Development Program of China (2021YFB0300600), National Science Foundation of China (92270206, T2125013, 62372435 62032023, 61972377, T2293702), CAS Project for Young Scientists in Basic Research (YSBR-005) and Network Information Project of Chinese Academy of Sciences (CASWX2021SF-0103), ShuGuang Foundation (ghfund202302024140) and Huawei Technologies Co., Ltd. We thank Dr. Haibo Li for helpful discussions. The numerical calculations in this study were partially carried out on the ORISE Supercomputer.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] RA Ainsworth. 1984. The assessment of defects in structures of strain hardening material. *Engineering Fracture Mechanics* 19, 4 (1984), 633–642.

[3] Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P. Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E. Smidt, and Boris Kozinsky. 2022. E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature Communications* 13, 1 (04 May 2022), 2453. https://doi.org/10.1038/s41467-022-29939-5

[4] J Behler. 2014. Representing potential energy surfaces by high-dimensional neural network potentials. *Journal of Physics: Condensed Matter* 26, 18 (apr 2014), 183001. https://doi.org/10.1088/0953-8984/26/18/183001

[5] Jörg Behler. 2017. First principles neural network potentials for reactive simulations of large molecular and condensed systems. *Angewandte Chemie International Edition* 56, 42 (2017), 12828–12840.

[6] Jörg Behler and Michele Parrinello. 2007. Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. *Phys. Rev. Lett.* 98 (Apr 2007), 146401. Issue 14. https://doi.org/10.1103/PhysRevLett.98.146401

[7] Luigi Bonati and Michele Parrinello. 2018. Silicon liquid structure and crystal nucleation from ab initio deep metadynamics. *Physical review letters* 121, 26 (2018), 265701.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[9] Albert Cohen, Wolfgang Dahmen, and Ronald DeVore. 2009. Compressed sensing and best -term approximation. *Journal of the American mathematical society* 22, 1 (2009), 211–231.

[10] Saaketh Desai, Samuel Temple Reeve, and James F. Belak. 2022. Implementing a neural network interatomic model with performance portability for emerging exascale architectures. *Computer Physics Communications* 270 (2022), 108156. https://doi.org/10.1016/j.cpc.2021.108156

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Ralf Drautz. 2019. Atomic cluster expansion for accurate and transferable interatomic potentials. *Phys. Rev. B* 99 (Jan 2019), 014104. Issue 1. https://doi.org/10.1103/PhysRevB.99.014104

[13] Patrick L Fitzgibbons, David L Page, Donald Weaver, Ann D Thor, D Craig Allred, Gary M Clark, Stephen G Ruby, Frances O'Malley, Jean F Simpson, James L Connolly, et al. 2000. Prognostic factors in breast cancer: College of American Pathologists consensus statement 1999. *Archives of pathology & laboratory medicine* 124, 7 (2000), 966–978.

[14] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[15] Mirza Galib and David T Limmer. 2021. Reactive uptake of N2O5 by atmospheric aerosol is dominated by interfacial processes. *Science* 371, 6532 (2021), 921–925.

[16] Johannes Gasteiger, Shankari Giri, Johannes T. Margraf, and Stephan Günnemann. 2020. Fast and Uncertainty-Aware Directional Message Passing for Non-Equilibrium Molecules. In *Machine Learning for Molecules Workshop, NeurIPS*.

[17] Johannes Gasteiger, Janek Groß, and Stephan Günnemann. 2020. Directional Message Passing for Molecular Graphs. In *International Conference on Learning Representations (ICLR)*.

[18] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[19] Zhuoqiang Guo, Denghui Lu, Yujin Yan, Siyu Hu, Rongrong Liu, Guangming Tan, Ninghui Sun, Wanrun Jiang, Lijun Liu, Yixiao Chen, Linfeng Zhang, Mohan Chen, Han Wang, and Weile Jia. 2022. Extending the Limit of Molecular Dynamics with Ab Initio Accuracy to 10 Billion Atoms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 205–218. https://doi.org/10.1145/3503221.3508425

[20] Vineet Gupta, Tomer Koren, and Yoram Singer. 2018. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*. PMLR, 1842–1850.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[22] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2018. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv:1705.08741 [stat.ML]

[23] Siyu Hu, Wentao Zhang, Qiuchen Sha, Feng Pan, Lin-Wang Wang, Weile Jia, Guangming Tan, and Tong Zhao. 2023. RLEKF: An Optimizer for Deep Potential with Ab Initio Accuracy. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 7910–7918.

[24] Yu-Peng Huang, Yijie Xia, Lijiang Yang, Jiachen Wei, Yi Isaac Yang, and Yi Qin Gao. 2022. SPONGE: A GPU-Accelerated

Molecular Dynamics Package with Enhanced Sampling and AI-Driven Algorithms. *Chinese Journal of Chemistry* 40, 1 (2022), 160–168. https://doi.org/10.1002/cjoc.202100456 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cjoc.202100456

[25] Weile Jia, Jiyun Fu, Zongyan Cao, Long Wang, Xuebin Chi, Weiguo Gao, and Lin-Wang Wang. 2013. Fast plane wave density functional theory molecular dynamics calculations on multi-GPU machines. *J. Comput. Phys.* 251 (2013), 102–115. https://doi.org/10.1016/j.jcp.2013.05.005

[26] Weile Jia, Han Wang, Mohan Chen, Denghui Lu, Lin Lin, Roberto Car, E Weinan, and Linfeng Zhang. 2020. Pushing the Limit of Molecular Dynamics with Ab Initio Accuracy to 100 Million Atoms with Machine Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00009

[27] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[28] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* 2, 3 (2017).

[29] Kyuhyun Lee, Dongsun Yoo, Wonseok Jeong, and Seungwu Han. 2019. SIMPLE-NN: An efficient package for training and executing neural-network interatomic potentials. *Computer Physics Communications* 242 (2019), 95–103.

[30] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[31] Dong C Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming* 45, 1-3 (1989), 503–528.

[32] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[33] Yury Lysogorskiy, Cas van der Oord, Anton Bochkarev, Sarath Menon, Matteo Rinaldi, Thomas Hammerschmidt, Matous Mrovec, Aidan Thompson, Gábor Csányi, Christoph Ortner, and Ralf Drautz. 2021. Performant implementation of the atomic cluster expansion (PACE) and application to copper and silicon. *npj Computational Materials* 7, 1 (28 Jun 2021), 97. https://doi.org/10.1038/s41524-021-00559-9

[34] Aris Marcolongo, Tobias Binninger, Federico Zipoli, and Teodoro Laino. 2020. Simulating diffusion properties of solid-state electrolytes via a neural network potential: performance and training scheme. *ChemSystemsChem* 2, 3 (2020), e1900031.

[35] James Martens and Roger Grosse. 2015. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*. PMLR, 2408–2417.

[36] Thomas V Mikosch, Sidney I Resnick, and Stephen M Robinson. 2006. Numerical Optimization (2nd Edition). (2006).

[37] Jorge Nocedal§. 2016. Optimization Methods for Large-Scale Machine Learning.

[38] Akira Onuki. 2002. *Phase transition dynamics*. Cambridge University Press.

[39] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Chuan-Sheng Foo, and Rio Yokota. 2020. Scalable and practical natural gradient for large-scale deep learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 1 (2020), 404–415.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[41] Jean-Yves Raty, François Gygi, and Giulia Galli. 2005. Growth of carbon nanotubes on metal nanoparticles: a microscopic mechanism from ab initio molecular dynamics simulations. *Physical review letters* 95, 9 (2005), 096103.

[42] Kristof Schütt, Pieter-Jan Kindermans, Huziel Enoc Sauceda Felix, Stefan Chmiela, Alexandre Tkatchenko, and Klaus-Robert Müller. 2017. Schnet: A continuous-filter convolutional neural network for modeling quantum interactions. *Advances in neural information processing systems* 30 (2017).

[43] Hao-Jun Michael Shi, Tsung-Hsien Lee, Shintaro Iwasaki, Jose Gallego-Posada, Zhijing Li, Kaushik Rangadurai, Dheevatsa Mudigere, and Michael Rabbat. 2023. A Distributed Data-Parallel PyTorch Implementation of the Distributed Shampoo Optimizer for Training Neural Networks At-Scale. arXiv:2309.06497 [cs.LG]

[44] A.P. Thompson, L.P. Swiler, C.R. Trott, S.M. Foiles, and G.J. Tucker. 2015. Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials. *J. Comput. Phys.* 285 (2015), 316–330. https://doi.org/10.1016/j.jcp.2014.12.018

[45] Oliver T. Unke, Stefan Chmiela, Michael Gastegger, Kristof T. Schütt, Huziel E. Sauceda, and Klaus-Robert Müller. 2021. SpookyNet: Learning force fields with electronic degrees of freedom and nonlocal effects. *Nature Communications* 12, 1 (14 Dec 2021), 7273. https://doi.org/10.1038/s41467-021-27504-0

[46] Han Wang, Linfeng Zhang, Jiequn Han, and E Weinan. 2018. DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics. *Computer Physics Communications* 228 (2018), 178–184.

[47] Jing Wu, Yuzhi Zhang, Linfeng Zhang, and Shi Liu. 2021. Deep learning of accurate force field of ferroelectric HfO_2. *Phys. Rev. B* 103 (Jan 2021), 024108. Issue 2. https://doi.org/10.1103/PhysRevB.103.024108

[48] Kun Yao, John E. Herr, Seth N. Brown, and John Parkhill. 2017. Intrinsic Bond Energies from a Bonds-in-Molecules Neural Network. *The Journal of Physical Chemistry Letters* 8, 12 (2017), 2689–2694. https://doi.org/10.1021/acs.jpclett.7b01072 arXiv:https://doi.org/10.1021/acs.jpclett.7b01072 PMID: 28573865.

[49] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. 2021. Adahessian: An adaptive second order optimizer for machine learning. In *proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 10665–10673.

[50] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888* 6, 12 (2017), 6.

[51] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962* (2019).

[52] Linfeng Zhang, Han Wang, Roberto Car, and E Weinan. 2021. Phase Diagram of a Deep Potential Water Model. *Physical Review Letters* 126, 23 (2021), 236001.

[53] Zheng-De Zhang, Meng-Lu Tan, Zhi-Cai Lan, Hai-Chun Liu, Ling Pei, and Wen-Xian Yu. 2022. CDNet: A real-time and robust crosswalk detection network on Jetson nano based on YOLOv5. *Neural Computing and Applications* 34, 13 (2022), 10719–10730.

[54] Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. 1997. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)* 23, 4 (1997), 550–560.

A Artifact Appendix

The source code, dataset, and scripts are available on Zenodo: https://zenodo.org/records/10213773.

The testing is performed on 8 systems(Cu, Al, Si, Mg, CuO, HfO_2, NaCl, H_2O). The test is conducted using the Cu system as an example, and the same procedure can be extended to other systems.

The testing is performed on Arm Server(CPU: KunPeng 920s, GPU: Nvidia A100-PCIe-40GB). They adopt Duonao Job Scheduler to submit jobs. We provide job scripts(both Duonao Job Scheduler and Slurm Job Scheduler) in our MLFF code.

A.1 Dependency

GCC 9.3.0; CUDA toolkits 11.8; cmake 3.20.1; Anaconda 2020.11; Horovod 0.27.0.

A.2 Installation

Download the code from Zenodo and then build feature generation tools and customized op;

```
cd RLEKF/src/
./build.sh
cd ../../FEKF_multi/Op/Src
pip install .
cd ../../../
```

A.3 Experiments

The feature generation has run successfully when the output log shows "Saving npy file done".

```
cd dataset/Cu72102
chmod +x cu_gen_data.sh
dsub -s cu_gen_data.sh
```

Table.1: Baseline Statement. Table 1 describes the consumed number of epochs among single-instance Adam, 32-instance Adam and 64-instance Adam when reaching a given Energy RMSE.

Figure.7: Single GPU Performance. The end-to-end training time of single-instance Adam, RLEKF, 32-instance FEKF, the optimized 32-instance FEKF under 8 comprehensive datasets.

Table.5: Distributed FEKF Performance. Table 5 represents the training time of RLEKF, larger-batch FEKF on multiple GPUs with a large number of training samples on Cu dataset.

```
cd dataset/Cu72102
bash cu_adam.sh
bash cu_fekf.sh
bash cu_distributed.sh
```

A.4 Evaluation

Table.1: to obtain epoch consumption when reaching a given Energy RMSE. The default generated directories are "table1_ adam_ bs1", "table1_ adam_ bs32", and "table1_ adam_ bs64".

```
cd dataset/cu72102
# python process.py the/generated/file/
    epoch_train.dat the_RMSE_error
```

Figure.7: to get the training wall time(s). The default generated directories are: "figure7_ rlekf_ bs1", "figure7_ fekf_ bs32", "figure7_ fekf_ bs32_ opt".

Table.5: to get the training wall time(s). The default generated directories are "table5_ rlekf_ bs1_ gpu1","table5_ fekf_ opt_ bs32_ gpu1", "table5_ fekf_opt_ bs512_ gpu4", "table5_ fekf_opt_ bs4096_ gpu16"

```
cd dataset/cu72102
# awk '{last=$NF} END {print last}' the/
    generated/file/epoch_train.dat
```