Computer Physics Communications 259 (2021) 107624

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

86 PFLOPS Deep Potential Molecular Dynamics simulation of 100 million atoms with *ab initio* accuracy☆,☆☆

Denghui Lu [a], Han Wang [b], Mohan Chen [a], Lin Lin [c,d], Roberto Car [e], Weinan E [e], Weile Jia [c,*], Linfeng Zhang [e,*]

[a] *CAPT, HEDPS, College of Engineering, Peking University, China*
[b] *Laboratory of Computational Physics, Institute of Applied Physics and Computational Mathematics, China*
[c] *University of California, Berkeley, United States of America*
[d] *Lawrence Berkeley National Laboratory, United States of America*
[e] *Princeton University, United States of America*

ARTICLE INFO

Article history:
Received 13 July 2020
Received in revised form 31 July 2020
Accepted 6 September 2020
Available online 21 September 2020

Keywords:
Deep potential
Molecular dynamics
GPU
Heterogeneous architecture
DeePMD-kit

ABSTRACT

We present the GPU version of DeePMD-kit, which, upon training a deep neural network model using *ab initio* data, can drive extremely large-scale molecular dynamics (MD) simulation with *ab initio* accuracy. Our tests show that for a water system of 12, 582, 912 atoms, the GPU version can be 7 times faster than the CPU version under the same power consumption. The code can scale up to the entire Summit supercomputer. For a copper system of 113, 246, 208 atoms, the code can perform one nanosecond MD simulation per day, reaching a peak performance of 86 PFLOPS (43% of the peak). Such unprecedented ability to perform MD simulation with *ab initio* accuracy opens up the possibility of studying many important issues in materials and molecules, such as heterogeneous catalysis, electrochemical cells, irradiation damage, crack propagation, and biochemical reactions.

Program summary

Program Title: DeePMD-kit
CPC Library link to program files: https://doi.org/10.17632/phyn4kgsfx.1
Developer's repository link: https://doi.org/10.5281/zenodo.3961106
Licensing provisions: LGPL
Programming language: C++/Python/CUDA
Journal reference of previous version: Comput. Phys. Commun. 228 (2018), 178–184.
Does the new version supersede the previous version?: Yes.
Reasons for the new version: Parallelize and optimize the DeePMD-kit for modern high performance computers.
Summary of revisions: The optimized DeePMD-kit is capable of computing 100 million atoms molecular dynamics with *ab initio* accuracy, achieving 86 PFLOPS in double precision.
Nature of problem: Modeling the many-body atomic interactions by deep neural network models. Running molecular dynamics simulations with the models.
Solution method: The Deep Potential for Molecular Dynamics (DeePMD) method is implemented based on the deep learning framework TensorFlow. Standard and customized TensorFlow operators are optimized for GPU. Massively parallel molecular dynamics simulations with DeePMD models on high performance computers are supported in the new version.

1. Introduction

In recent years, there has been a surge of interest in using *ab initio* simulation tools for a microscopic understanding of various macroscopic phenomena in many different disciplines, such as chemistry, biology, and materials science. One of the most powerful tools has been the *ab initio* molecular dynamics (AIMD) scheme [1]: By generating on-the-fly the potential energy surface (PES) and the interatomic forces from first-principles density

☆ The review of this paper was arranged by Prof. D.P. Landau.
☆☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding authors.
E-mail addresses: jiaweile@berkeley.edu (W. Jia), linfengz@princeton.edu (L. Zhang).

https://doi.org/10.1016/j.cpc.2020.107624
0010-4655/

functional theory (DFT) [2,3] during molecular dynamics (MD) simulations, it is possible to obtain an accurate description of the dynamic behavior of the system under study at the atomic level. However, due to the complexity associated with DFT, the spatial and temporal scales accessible by AIMD have been limited. Most routine AIMD calculations can only deal with systems with hundreds of atoms on the time scale of picoseconds. Although many linear-scaling DFT methods have been developed [4,5] and some of them have been implemented on high performance computing (HPC) architectures for large-scale atomic simulation with tens of thousands of atoms [6,7], they are mostly limited to insulating systems with relatively large band gaps.

For many problems of practical interests, such as heterogeneous catalysis, electrochemical cells, irradiation damage, crack propagation in brittle materials, and biochemical reactions, etc., a system size of thousands to millions of atoms, or even larger, is often required. In these cases, one usually has to resort to empirical force fields (EFFs), currently the main driving force of large-scale MD. In the past two decades, tremendous efforts have been made to develop parallel algorithms and softwares for EFF-based MD (EFFMD) on general purpose HPC machines [8–24]. Representative examples include the optimization of the long-range electrostatic interaction [25–28] and adapted MD algorithms for accelerators like GPU [26,29–31] and FPGA [32,33]. Besides general-purpose HPCs, there have also been constant attempts to build special-purpose hardware to boost the performance of MD simulation [34–38]. These attempts have made it possible to perform EFFMD for systems up to a spatial scale of sub-millimeters (twenty trillion atoms) [22] or a temporal scale of up to milliseconds [38]. Unfortunately, the practical significance of these efforts is hindered by the limitation of the accuracy and transferability of the EFFs. For example, it has been hard, if not impossible, to develop accurate and general-purpose EFF models for multi-element alloys.

Recent development of machine learning (ML) methods has brought new hope to addressing this problem and there have been a flurry of activities on ML-based models of the PES [39–47]. Despite the growing importance of the ML-based MD (MLMD), publicly available softwares are still rare in comparison to the EFFMD. The few existing ones are mainly designed for MLMD running on desktop GPU workstations or on CPU-only clusters [48–54]. For example, Lee et al. reported an implementation of the Behler–Parrinello neural network potential interfaced with LAMMPS package. In a SiO_2 system with 13,500 atoms, it scaled up to 80 CPU cores on a cluster with Intel Xeon E5-2650v2 CPUs [52]. Singraber et al. implemented the Behler–Parrinello neural network potential as a library and interfaced it with LAMMPS for molecular dynamics simulation. By using a water system with 2880 molecules (8640 atoms) as the test case, it was demonstrated that the implementation scaled to 512 CPU cores on a cluster with Intel Xeon Gold 6138 CPUs [53]. To the best of our knowledge, no attempt has been made to implement and optimize MLMD to fully utilize the computational resources of modern heterogeneous supercomputers like Summit. As a consequence, although in principle MLMD makes it possible to achieve AIMD accuracy with EFFMD efficiency, this has not been realized in practice.

Among the various ML models proposed in the past few years, the Deep Potential (DP) scheme [45–47] stands out as an end-to-end way of constructing accurate and robust PES models for a wide variety of systems. This was made possible due to the smooth symmetry-preserving embedding sub-net in DP (in addition to the fitting net), as well as the adaptive data generating scheme (in the framework of concurrent learning [55]) Deep Potential Generator (DP-GEN) [56]. DP-based molecular dynamics (DeePMD) can reach the accuracy of AIMD while reducing its cost by several orders of magnitude. Generalizations of the DP scheme have also made it possible to represent the free energy of coarse-grained particles [57] and various electronic properties [58–60]. In addition, an open-source implementation of DeePMD, named DeePMD-kit [50], has attracted researchers from various disciplines. DP models have been used to study problems like first-order phase transitions [61], infrared spectroscopy and Raman spectroscopy [58,59], nuclear quantum effects [62], and various phenomena in chemistry [63–65] and materials sciences [66–69].

Nevertheless, the performances of DeePMD-kit and other DeePMD-based codes are limited by their sub-optimal implementation. Although the training of DP models is rather efficient (typically less than one day on a single GPU card for most systems), extensive optimizations are required for model inference, namely to predict the energy and forces on-the-fly during an MD run, and to truly boost AIMD to large system size and long time scale.

To perform large-scale MD simulations, DeePMD-kit interfaces with LAMMPS [13] and TensorFlow [70]. LAMMPS provides the basic infrastructure for MD, while TensorFlow provides a flexible toolbox for the deep learning part of DeePMD. In each MD step, DeePMD-kit retrieves atomic coordinates from LAMMPS that maintains the atomic information and the spatial partitioning of the system. Then environment matrices that describe the relative positions of atoms are computed from the coordinates. In this step, the memory is accessed in a random order, which cannot be efficiently implemented by standard TensorFlow operators, so it is implemented by DeePMD-kit as a customized TensorFlow operator. Next, the environment matrices are converted to descriptors that describe the neighboring environment of atoms, and the descriptors are passed to a standard deep neural network (DNN) to produce atomic energies. This step is implemented by standard TensorFlow operators. Finally, the atomic energies and forces (obtained by back propagation) are returned to LAMMPS to update the atomic coordinates and momenta by numerical schemes.

The Summit supercomputer, which has a peak performance of 200 PFLOPS (Peta floating point operations per second), provides us with an unprecedented opportunity to speedup DeePMD. However, the original DeePMD-kit is not suitable for the heterogeneous architecture of Summit for the following reasons: (1) The environment matrix is only implemented on CPUs, this becomes the computational bottleneck when the descriptors and atomic energies are computed on GPUs. (2) Although standard TensorFlow operators support GPU computation, the original DeePMD-kit cannot assign multiple GPUs to multiple MPI processes in a massively parallel environment, thus only single GPU serial computation or multiple CPUs parallel computation are feasible. (3) The sizes of the DNNs in DP are relatively small, and the efficiency of the standard TensorFlow computational graph is relatively low.

To fully harness the power of Summit and future supercomputers, we need to address the following questions: (1) What is the best parallelization scheme for DeePMD-kit on a heterogeneous supercomputer like Summit? (2) How can we improve the efficiency of DeePMD-kit on a GPU supercomputer for both customized and standard TensorFlow operators? (3) What is the scaling bottleneck of DeePMD-kit and how can we further improve its efficiency on architectures of future supercomputers? Furthermore, we would also like to understand: (1) What is the limit of DeePMD-kit on Summit both in terms of system size and computational speed (time-to-solution)? (2) What is the maximal achievable speedup factor of the GPU version of DeePMD-kit versus the CPU version by using the same number of nodes or the same power consumption?

The main contributions of this paper are:

- We find that DeePMD can use the same data distribution scheme of EFFMD, and parallelization is highly scalable on heterogeneous supercomputers.
- By carefully optimizing the CUDA customized TensorFlow operators and re-constructing the architecture of the standard TensorFlow operators, DeePMD-kit can reach 43% peak performance (86 PFLOPS) on Summit.
- By carefully analyzing the scaling of DeePMD-kit, we identify the latency of both the GPU and network as the bottleneck of the current heterogeneous platform, which requires future improvements to push the limit of scales and applications that DeePMD-kit can handle.
- Weak scaling shows that the GPU version of DeePMD-kit can scale up to the entire Summit supercomputer, on a copper system with 113 million atoms. The strong scaling of a water system shows that DeePMD-kit can reach 110 MD steps per second for a 4 million molecular water system with *ab initio* accuracy.
- Our test results show that the GPU DeePMD-kit can be 39 times faster compared to the CPU version when using the same number of nodes, and 7 times faster under the same power consumption on Summit.

The rest of this paper is organized as follows: The Deep Potential algorithm is introduced in Section 2, with implementation details provided in Section 3. The physical system and testing platform are presented in Sections 4 and 5, respectively. Results are discussed in Section 6, followed by a performance analysis in Section 7. Conclusions are drawn in Section 8.

2. The deep potential model

The central quantity of an MD simulation is the PES E, a function of the atomic coordinates $(r_1, \dots r_N) \in \mathbb{R}^{3N}$. The DP model expresses E as a sum of atomic contributions, i.e., $E = \sum_i E_i$. The contribution E_i from the atom i depends only on $\mathcal{R}_i$, the local environment of i: $\mathcal{R}_i = \{r_{ij} : j \in L(i)\}$, where $r_{ij} = r_j - r_i$. Here the neighbor index set $L(i)$ is defined by $\{j : |r_{ij}| \le r_c\}$, and r_c is a predefined cutoff radius. In the DP model, $\mathcal{R}_i$ is first mapped via an embedding net onto a symmetry-preserving descriptor $\mathcal{D}$, and then $\mathcal{D}$ is mapped via a fitting network $\mathcal{N}$ to give E_i, i.e.,

$$E_i = \mathcal{N}(\mathcal{D}(\mathcal{R}_i)). \tag{1}$$

Here the fitting net $\mathcal{N}$ is chosen to be a fully connected DNN with l hidden layers:

$$\mathcal{N}(x) = \mathcal{L}_l^f \circ \cdots \circ \mathcal{L}_1^f(x), \tag{2}$$

where $\circ$ denotes the function composition. Within each hidden layer, a skip connection between the input and the output is used,

$$\mathcal{L}_k^f(x) = x + \tanh(x \cdot W_k^f + b_k^f), \tag{3}$$

with the weight W_k^f being a square matrix and the bias b_k^f being a vector with the same size as the input x. The activation function tanh is applied component-wise.

The descriptor $\mathcal{D}$, which is required to preserve the translational, rotational and permutational symmetries, has the form

$$\mathcal{D}(\mathcal{R}_i) = (\mathcal{G}_i^<)^T \tilde{\mathcal{R}}_i (\tilde{\mathcal{R}}_i)^T \mathcal{G}_i, \tag{4}$$

where $\tilde{\mathcal{R}}_i \in \mathbb{R}^{N_m \times 4}$ is the environment matrix, and N_m is the largest number of neighbors for all the atoms. Each row of the environment matrix is a four dimensional vector:

$$s(r_{ij}) \times \left(1, x_{ij}/|r_{ij}|, y_{ij}/|r_{ij}|, z_{ij}/|r_{ij}|\right), \tag{5}$$

where $s(r_{ij}) = w(|r_{ij}|)/|r_{ij}|$ and $w(|r_{ij}|)$ is a gating function that decays smoothly from 1 to 0 at $|r_{ij}| = r_c$. The gating function ensures the smoothness of the environment matrix. (x_{ij}, y_{ij}, z_{ij}) are the Cartesian coordinates of r_{ij}. If the number of neighbors of atom i is less than N_m, the empty entries of $\tilde{\mathcal{R}}_i$ will be filled by zeros. $\mathcal{G}_i \in \mathbb{R}^{N_m \times M}$ is called the embedding matrix, with each row being an M dimensional vector

$$\left(G_1(s(r_{ij})), \dots, G_M(s(r_{ij}))\right). \tag{6}$$

Here for each neighbor j, the input scalar $s(r_{ij})$ is mapped to the output M dimensional vector $G = (G_1, \dots, G_M)$ via the so-called embedding net G, a DNN with the form

$$G(x) = \mathcal{L}_m^e \circ \cdots \circ \mathcal{L}_1^e \circ \mathcal{L}_0^e(x). \tag{7}$$

The first hidden layer is a standard feed forward network taking a scalar as input and outputting a vector of size s_1:

$$\mathcal{L}_0^e(x) = \tanh(x \cdot W_0^e + b_0^e), \tag{8}$$

where $W_0^e \in \mathbb{R}^{s_1}$ and $b_0^e \in \mathbb{R}$ denote the weight and bias, respectively. The rest of the hidden layers are expressed as

$$\mathcal{L}_k^e(x) = (x, x) + \tanh(x \cdot W_k^e + b_k^e). \tag{9}$$

Here the output size is twice of the input size, i.e., $s_k = 2s_{k-1}$. The weight is a matrix of size $s_{k-1} \times s_k$ and the bias is a vector of size s_k. $(x, x) \in \mathbb{R}^{s_k}$ denotes the concatenation of two $x \in \mathbb{R}^{s_{k-1}}$. The only restriction imposed on the sizes of hidden layers is that the output size of the final layer should be identical to M, i.e., $s_m = M$. In Eq. (4), the matrix $\mathcal{G}_i^< \in \mathbb{R}^{N_m \times M^<}$ with $M^< < M$ is a sub-matrix of $\mathcal{G}_i$ formed by taking the first $M^<$ columns of $\mathcal{G}_i$.

Remark 1. The DP formulation (1) can be easily generalized to multi-component (with atoms of multiple chemical species) systems. In this case, a fitting net $\mathcal{N}$ is built for each chemical species in the system, i.e., $E_i = \mathcal{N}_{\alpha_i}(\mathcal{D}(\mathcal{R}_i))$, where α_i denotes the chemical species of the atom indexed with i. The chemical species of the neighbors of the atom i are encoded in the descriptor (4) by separate embedding nets built for all possible combinations of the chemical species of two neighboring atoms, i.e., $G^{\alpha_i,\alpha_j}(s(r_{ij}))$. For example, for a system with 3 chemical species, 3 fitting nets and 9 embedding nets will be constructed.

Remark 2. The force on atom i is defined as the negative gradient of the total energy with respect to r_i:

$$F_i = -\nabla_{r_i} E = -\sum_j \nabla_{r_i} E_j, \tag{10}$$

where the force components $\nabla_{r_i} E_j$ are calculated by the back propagation.

Remark 3. For one evaluation of the DP model, the fitting net is evaluated for each atom, so the computational cost is of $\mathcal{O}(N)$. The embedding net is evaluated for each pair of neighbors, so the computational cost is of $\mathcal{O}(N \times N_m)$. The value of N_m depends on the density of the system and r_c. Usually N_m is of the order $100 \sim 1000$. Therefore, the evaluation of the embedding net is roughly two to three orders of magnitude more expensive than the fitting net.

3. Implementation

3.1. Parallelization

The DeePMD-kit takes advantage of the LAMMPS software package [13] by replacing the short-range EFF with the energy and forces derived from DP. Therefore, the data structure and parallelization strategy of LAMMPS are inherited by

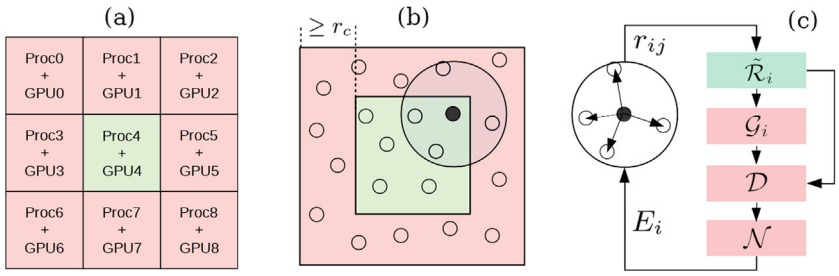

Fig. 1. Data distribution and workflow of the DeePMD-kit. (a) Spatial subdivision of a system and the associated allocation of computational resources. Each sub-region is represented by a square. (b) The ghost region (red) for one sub-region (green). The open particles in the big circle with radius r_c are the neighbors of the solid particle. The width of the ghost region should be equal to or larger than r_c. (c) The single-atom DP workflow. The green step, i.e., the environment matrix $\tilde{\mathcal{R}}_i$, is implemented by a customized TensorFlow operator, while the red steps are implemented by standard TensorFlow operators (see the text for details). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the DeePMD-kit code. More specifically, we provide a new `pair_style` named `deepmd` by implementing DP as a new derived class of the base class `Pair`. It is worth noting that although DP is invoked by a "`pair_style`", it is a multi-body interaction, which can be easily seen from the construction of DP in Section 2.

A two dimensional illustration of the data distribution in DeePMD-kit is shown in Fig. 1(a). The physical system is divided into sub-regions, and then distributed among different computing units. For each sub-region, an extra ghost region of size larger than or equal to r_c is needed to search neighbors of atoms, as shown in Fig. 1(b). In each MD step, the single-atom DP workflow (Fig. 1(c)) is conducted for each atom: First, the neighbor list is updated from the sub-region on the current computing unit, then the environment matrix $\tilde{\mathcal{R}}_i$ is computed from the neighbor list via a customized TensorFlow operator; Next, the atomic energy and force increments are evaluated through the DP model to update the atoms in the sub-region; Finally, the force increments in the ghost region are communicated among adjacent MPI processes, and global properties, such as energy, are communicated globally by MPI_Allreduce operations. All of the positions and velocities of atoms are updated using the resulting forces according to a certain numerical scheme.

Remark 4. The GPU version of LAMMPS takes advantage of the identity $\nabla_{r_i} E_j = -\nabla_{r_j} E_i$, which holds for most of the EFFs, so that the force of atom i is given by $F_i = \sum_j \nabla_{r_j} E_i$ according to Eq. (10). Therefore, all components of F_i are computed on the computing unit that holds atom i, and the communications of force components are avoided. By contrast, the relation $\nabla_{r_i} E_j = -\nabla_{r_j} E_i$ dose not hold for DP as $i \neq j$, so we have to fallback to the CPU version of LAMMPS that is able to transfer back the component $\nabla_{r_i} E_j$ computed on a non-native computing unit holding atom j if it is in the ghost region.

3.2. GPU implementation and optimization

3.2.1. Naive GPU implementation

The implementation of the DP model in DeePMD-kit is based on TensorFlow, a popular open-source software library for machine learning applications with GPU support [70]. One common practice is to link the GPU supported TensorFlow to build the executable, so all DP operations implemented by standard TensorFlow operators (red boxes in Fig. 1(c)) are accelerated by GPU without additional effort. The testing results indicate that an overall 26.53 times of speedup can be achieved using a single NVIDIA V100 GPU compared to a single Intel Xeon Gold 6132 CPU core for a typical water system consisted of 12,288 atoms (4096 molecules). We remark that throughout this section, the naive GPU implementation serves as the baseline of optimization. The same water system is used for benchmarking purposes, and one MPI process using one thread is bound to a single GPU.

3.2.2. Customized tensorflow operators

Algorithm 1 Formatting the neighbor list

Input: Atomic position $\{r_i\}$, the corresponding neighbor list $L(i, j)$
Output: Formatted neighbor list $\tilde{L}(i, j)$

for each $i \in [0, N_l)$ **do**
  for each $k \in [0, L(i).\text{size})$ **do**
    $j = L(i, k)$
    $r_{ij} = r_j - r_i, \quad |r_{ij}| = \sqrt{r_{ij} \cdot r_{ij}}$
    $S(i, k, 0) = \alpha(j), \ S(i, k, 1) = |r_{ij}|, \ S(i, k, 2) = j$
  end for
  Sort the second dim with the third dim as key $S \to S^*$
  Pad the second dim $S^* \to S^{**}$
  $\tilde{L}(i, :) = S^{**}(i, :, 2)$
end for

The customized TensorFlow operator for the environment matrix $\tilde{\mathcal{R}}_i$ is denoted by "Environment" and dominates the computational cost after linking the GPU TensorFlow library, because unlike the standard TensorFlow operators that support GPU, it only supports CPU in the original version of DeePMD-kit. The operator Environment includes two steps, formatting the neighbor list and computing the environment matrix by using the formatted neighbor list.

The algorithm for formatting the neighbor list is shown in Alg. 1. The arbitrary ordered neighbor list of atom i shown in Fig. 2(a) is sorted first based on the type of neighboring atoms, and then on the atomic distances r_{ij}. In the case where two neighbors are of the same type and distance, the neighbor with a smaller atomic index is placed before the neighbor with a larger atomic index. The neighbors with different types in the neighbor list are then padded, so that they are aligned to the maximal number of neighbors of that type, as shown in Fig. 2(a). The reason for this operation is the following: in the computation of the embedding matrix, the neighbors of atoms i are scanned over, and each row of the embedding matrix $\mathcal{G}_i$ is computed by passing $s(r_{ij})$ (the first element of the corresponding row of $\tilde{\mathcal{R}}_i$) to the embedding net G^{α_i,α_j}, which introduces a conditional branching according to the type of atom j. Sorting and padding of the neighbor list avoids this unfavorable branching. In our GPU code, the construction of the neighbor list is still on CPU due to the problem of GPU LAMMPS for DP, as explained by Remark 4 of Section 3.1. In practice, the neighbor list is usually updated every 10 to 50 steps in an MD simulation, so our current implementation results in a satisfactory performance.

In order to efficiently format the neighbor list on the GPU, we perform the following optimization steps:

1. *Naive CUDA customized kernels.* The first step of optimization is to write a single CUDA customized kernel to accelerate the

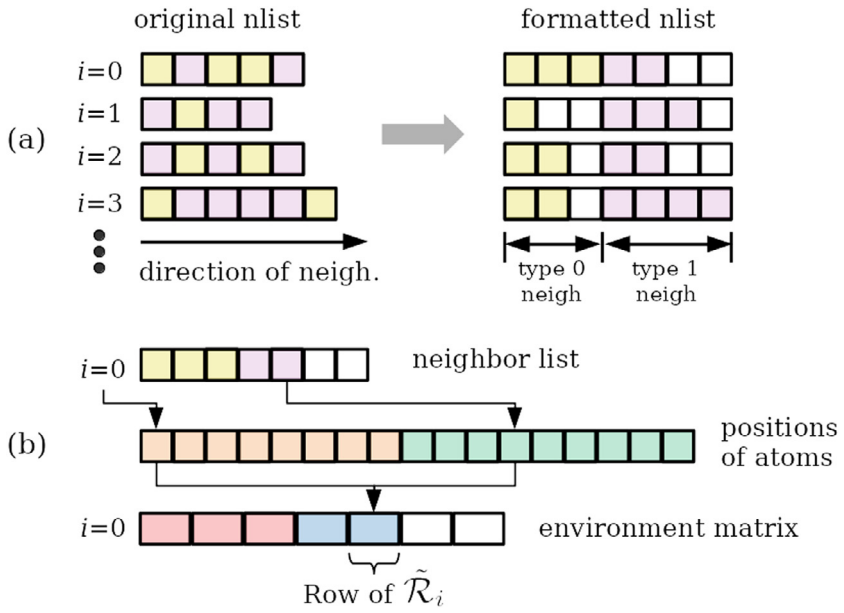

Fig. 2. Schematic plot of the computation of the environment matrix $\tilde{\mathcal{R}}_i$. (a) The first step: forming and formatting the neighbor list. The yellow squares stand for the neighbors of type 0, and the purple squares stand for the neighbors of type 1. The neighbors are first sorted according to their types (as shown in the figure), and then sorted according to their distance and their original atomic indices (not shown in the figure). The blank squares represent the padded positions in the neighbor list. (b) The second step: computing the environment matrix by using the formatted neighbor list, using the example of computing $\tilde{\mathcal{R}}_0$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

computation of Alg. 1. In this step, the first for loop (line 1) is unrolled with CUDA blocks and threads. Each CUDA thread is then responsible for calculating and sorting the neighbor list of a particular atom i.

2. *Converting array of structures (AoS) to structure of arrays (SoA).* A single element of the intermediate neighbor list S is expressed by a structure (see Alg. 1). For example the kth neighbor of the ith atom $S(i,k)$ is a structure of three elements $(\alpha(j), |r_{ij}|, j)$, where $\alpha(j)$ and j are integers and $|r_{ij}|$ is a floating point number. Thus the corresponding GPU memory is not coalesced during the sorting procedure. One way of improving the GPU performance is to store the neighbor list as SoA instead of AoS. The SoA can improve the memory coalescing significantly, thus improving the performance of the CUDA kernel.

3. *Unrolling of two for loops.* Two CUDA customized kernels are used to implement Alg. 1 in this step. The first kernel is used to construct the intermediate neighbor list S (line 1–6). In this implementation, the first and the second for loops are unrolled with CUDA blocks and threads respectively to further exploit the computing power of V100 GPU. Then the intermediate neighbor list is sorted and padded using a second kernel.

4. *Compressing elements of the neighbor list to a 64 bit integer.* The NVIDIA CUB library provides state-of-the-art and reusable software components for every layer of the CUDA programming model, including block-wide sorting. To efficiently use the CUB library, we compress $S(i,k)$ into an unsigned long long number with the following equation:

$$\tilde{S}(i,k) = \alpha(j) \times 10^{15} + \lfloor |r_{ij}| \times 10^{8} \rfloor \times 10^{5} + j \tag{11}$$

The 19 decimals of an unsigned long long integer is divided into 3 parts to store the neighbor list information: 4 decimal are used to store the atomic type of the neighbor atom ($\alpha(j)$), 10 decimals are used to store the distance of atom i and its neighbor atom ($|r_{ij}|$), 5 decimals are used to store the atomic index of the neighbor atom (j). The range of all the three parts are carefully chosen to fulfill the restrictions that the total number of atom types is smaller than 1843, the cut-off radius is smaller than 100 Å, and the number of neighbors is smaller than 100,000. These restrictions are rarely violated in typical MD simulations. The data compression is carried out before sorting, and a decompression procedure is needed afterwards. Both the compression and decompression are accelerated via CUDA customized kernels, and the corresponding computational time is negligible. We find that the compression reduces the total number of comparisons by half during the sorting procedure without deteriorating the accuracy of the result.

Fig. 3(a) shows the reduction of wall clock time associated with each stage of optimization. The baseline version is implemented on CPU, and tested with both single CPU core and single Xeon Gold 6132 socket with 14 cores by setting OpenMP threads to 14. We find that after all optimizations, the neighbor list formatting is accelerated by 141 times comparing to single CPU core and 12.25 times comparing to 14 CPU cores. We remark that when using both sockets with OpenMP threads set to 28, the performance is only slightly better than single socket due to the memory affinity.

Algorithm 2 Computing the environment matrix $\tilde{\mathcal{R}}$

Input: Atomic position $\{r_i\}$, formatted neighbor list $\tilde{L}(i)$
Output: Environment matrix $\tilde{\mathcal{R}}_i$

```
for each i ∈ [0, N_l] do
  for each k ∈ [0, L̃(i).size) do
    j = L̃(i, k)
    if j is not a padded neighbor then
      r_ij = r_j − r_i,   |r_ij| = sqrt(r_ij · r_ij)
      R̃(i, k) = s(r_ij)(1, x_ij/|r_ij|, y_ij/|r_ij|, z_ij/|r_ij|)
    else
      R̃(i, k) = (0, 0, 0, 0)
    end if
  end for
end for
```

The algorithm of the second step of the operator Environment, computing the environment matrix, is shown in Alg. 2 and graphically illustrated by Fig. 2(b). The formatted neighbor list is taken

as input, and the corresponding environment matrix is built based on line 6 in Alg. 2. It is noted that the padded neighbors are skipped in the computation, and the corresponding places of the environment matrix are filled with zeros.

The optimization for the computation of the environment matrix follows the optimization steps 3 of formatting the neighbor list. The **for** loops in Alg. 2 (line 1 and 2) are unrolled with CUDA blocks and threads. Each thread only works on a specific i, j, k to fully exploit the computing power of V100 GPU. Two extra TensorFlow customized operators, ProdVirial and ProdForce, are also accelerated with the same fashion. These operators are used to calculate the force and virial outputs after the executions of embedding net and fitting net.

Fig. 3(b) shows the wall clock time of the customized TensorFlow operators. The testing results show that our GPU implementation achieves 120, 35 and 16 times of speedup for the Environment, ProdVirial, ProdForce operators, respectively. We remark that operators ProdVirial and ProdForce are not fully optimized in the OpenMP implementation in DeePMD-kit because of the atomic addition, thus only sequential results of these two operators are shown in Fig. 3(b). It is noted that the time for GPU memory allocations and the CPU–GPU memory copy operations are not included in the tests. For the water system consisting of 12,288 atoms, the total execution time of all three customized operators reduced from 363 to ∼6 ms, achieving a speedup of 60 times. Since the customized operators take 76% of the total time, the GPU version of DeePMD-kit gains a speedup of 4.0 compared to the baseline implementation.

3.2.3. Optimization of the embedding net

The environment matrix is used to compute the embedding matrix and assemble the descriptor, and finally the atomic energy contribution is computed by the fitting net, which takes the descriptor as input. All these steps are implemented by the standard TensorFlow execution graph. As discussed in Remark 2 in Section 2, the computational cost of the fitting net is of order $O(N_m)$, while the cost of the embedding net is of order $O(N_m \times N_l)$, where N_l being the number of atoms in the computing unit and N_m being the maximal number of neighbors of an atom. After optimizing the customized TensorFlow operators, about 85% of the total execution time is spent on the embedding net, while only 6% of the execution time is used in the fitting net in our benchmark system. Therefore, in this section, we benchmark and optimize the performance of the embedding net. The embedding net (Eq. (7)) is composed of several hidden layers. Except for the very first layer (8), the successive layers (9) output a vector that is twice as large as the input vector. Most of the computational cost is spend on the successive layers (Eq. (9)) rather than the first layer (Eq. (8)). Therefore, we focus our attention on the successive layers.

The execution graph of Eq. (9) with standard TensorFlow operators is presented in Fig. 4(a). The TensorFlow operators such as the MATMUL, component-wise SUM, TANH and CONCAT are executed to perform the operations of matrix–matrix multiplication, summation, activation function, and concatenation, respectively. MATMUL and TANH are two of the most computationally intensive operators, and they can reach 72% and 16% of the peak on the GPU, respectively. Other operators such as CONCAT and SUM are bandwidth intensive with little floating point operations. Although linking to the GPU supported TensorFlow library provides considerable speedup compared to the CPU code, as shown in Section 3.2.1, our profiling results show that the total computational time is still dominated by those bandwidth intensive operators. For example, the computational time of CONCAT and SUM operators contributes 43% of the total. Thus we identify these bandwidth-intensive operators as the ones that we make the greatest effort to optimize.

First, we notice that the summation and matrix–matrix multiplication are treated as two separated operators for evaluating $x \cdot W + b$ in the TensorFlow execution graph, as shown in Fig. 4(a). The MATMUL operator is invoked to calculate $x \cdot W$, where x is a matrix of size $376{,}832 \times 50$ (oxygen–hydrogen embedding) and w is the weight matrix of size 50×100 in the benchmark system. Next, the SUM operator is called to add the bias b to the resulting matrix $x \cdot W$. As shown in Fig. 4(b), the MATMUL and SUM operators can be replaced by a single CUBLAS GEMM call ($C = \alpha A \times B + \beta C$), which has both matrix–matrix multiplication and summation, thereby avoiding the corresponding SUM operator in the optimized implementation. It is noted that b is a vector, and it is converted to a matrix format by multiplying with a transpose of the vector *one*. The wall clock for performing the SUM and MATMUL operators is reduced by 28% after merging them into a single CUBLAS GEMM call.

Next, we move on to the optimization of the CONCAT operator shown in Fig. 4(a). The CONCAT operator is performed to concatenate two xs to form (x, x) in Eq. (9). The concatenation result, together with the result matrix of TANH operator, are summed up to produce the output of the embedding net. In the standard TensorFlow execution graph, the CONCAT operator is implemented via the EIGEN library, which is a C++ template library for linear algebra. In our optimized version, we replace the CONCAT operator with a matrix–matrix multiplication, so that the following SUM operator (with the result of TANH) can be merged into one GEMM operator:

$$(x, x) + \tanh(\cdots) \rightarrow \underbrace{x \times (I, I) + \tanh(\cdots)}_{\text{GEMM}}. \tag{12}$$

It is noted that, in terms of performance, the matrix–matrix multiplication is marginally better than the implementation of CONCAT by EIGEN, and the main benefit comes from the removal of the SUM operator. The wall clock time of the CONCAT and SUM operators is reduced by 55% after the optimization.

Last but not the least, we optimize the TANHGrad operator, which performs the derivation of $\tanh(x)$ in the backward propagation of the embedding net. It is noted that Fig. 4 only shows the forward propagation of the embedding net, and the TANHGrad operator is not included. However, in each MD step, both forward and backward propagation of the embedding net are executed. Noticing that the derivative of $\tanh(x)$ is also a function of $\tanh(x)$, i.e., $\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$, we merge the TANH and TANHGrad operators by implementing both functions in the same CUDA customized kernel. Our testing results show that 37% of the execution time is saved for the TANH and TANHGrad operators after optimization.

With all the optimizations above, an overall speedup factor of 1.18 is achieved compared to the results in Section 3.2.2, and the cost of the matrix–matrix multiplication changes from 30% to 61% of the total execution time in the benchmark system.

3.2.4. GPU memory accommodation

The memory footprint of the GPU version of DeePMD-kit sets the limit of the system size, since each NVIDIA V100 GPU on the Summit supercomputer only has 16 GB memory. In the GPU code, the most memory demanding part is the embedding matrix $\mathcal{G}$. The number of floating point numbers to store one embedding matrix is approximately $N_l \times N_m \times M$. Here N_l is the number of atoms residing on the GPU, N_m is the maximal number of neighbors, and M is the width of the output layer of the embedding net. Therefore, the GPU memory requirement is not only restricted by the size of the network (M), but also related to the number of neighbors included in the neighbor list. In the execution, three layers of embedding net are used, and the output matrix size is twice of the input matrix. In the last layer, the sizes of both

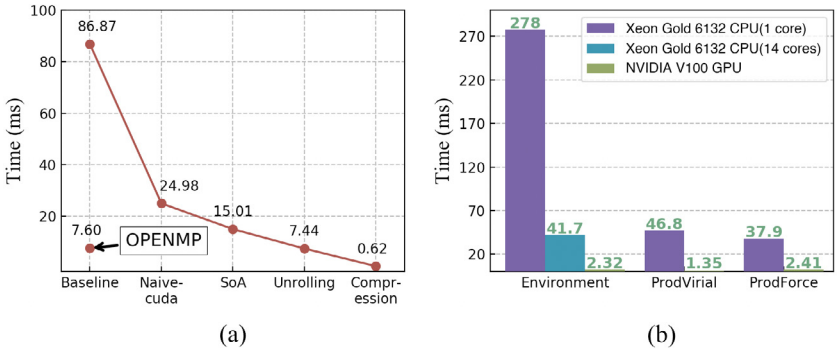

Fig. 3. (a) Wall clock time versus different levels of GPU optimization for formatting the neighbor list, and (b) Performance of customized TensorFlow operators for a water system of 12,288 atoms. The customized TensorFlow operators of the baseline code is implemented on the CPU. The wall clock time of the baseline is measured using both single CPU core and single Xeon Gold 6132 socket (14 cores with OpenMP threads set to 14). The GPU time is measured on single NVIDIA V100 GPU.

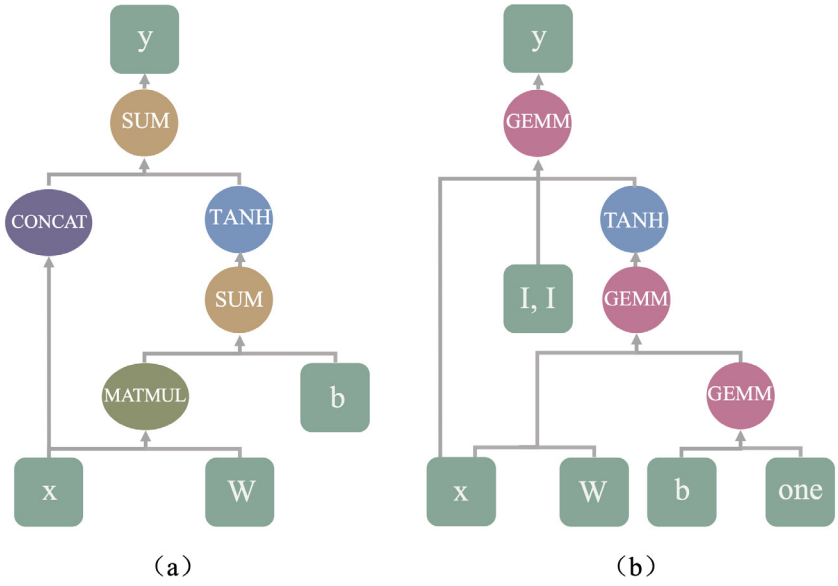

Fig. 4. Schematic plot of the execution graph of Eq. (9). (a) Implementation with the standard TensorFlow operators. (b) Implementation with our optimized TensorFlow operators.

the output matrix and its derivative are $N_l \times N_m \times M$. An extra matrix of size $N_l \times N_m \times M$ is used to perform the concatenation operation. Therefore, a total of 4.5 copies of the embedding matrix are needed in the DeePMD-kit calculations as the equation shown below:

$$4.5 \times N_l \times N_m \times M \times \texttt{sizeof}(data\_type) \tag{13}$$

For a typical system, such as the water and copper systems that will be discussed later, the memory requirement grows linearly with the number of atoms. Note that N_m is usually of the order of a hundred, and M is usually 100 in practice. For example, if we take $N_l = 25{,}000$, $N_m = 138$, $M = 100$ and *data_type = double*, the memory usage of $\mathcal{G}$ reaches 12.42 GB. This estimate can be verified by the numerical results in Section 6.

4. The physical system

As shown in Fig. 5, we use two representative examples, water and copper, to investigate the performance of the GPU DeePMD-kit software package. Water, despite its simple molecular structure, has an unmatched complexity in the condensed (liquid) phase, as a result of the delicate balance between weak non-covalent intermolecular interactions, e.g. the hydrogen bond network and van der Waals dispersion, thermal (entropic) effects, and nuclear quantum effects. Copper represents an important and yet relatively simple metallic system, well suited as a benchmark. The training data of the water and copper systems are describe in Refs. [46,47], and [55], respectively. The DP models for both systems share almost the same architecture: sizes of the embedding and fitting nets are 25 × 50 × 100 and 240 × 240 × 240, respectively. The cut-off radii of water and copper systems are 6 Å and 8 Å, respectively, and the maximal numbers of neighbors are 138 and 500, respectively. Extensive benchmarks and theoretical studies have been conducted using DeePMD-kit, thus the accuracy of the model is reasonably assured. As a result, we can focus on the computational performance of the MD simulations.

The strong scaling of GPU DeePMD-kit is tested using the water system composed of 12,582,912 atoms (4,194,304 water molecules), while the weak scaling is investigated using the copper system with 4139 atoms per GPU card. The configuration of the water system is made by replicating a well equilibrated liquid water system of 192 atoms for 64 × 32 × 32 times. The configurations of the copper system are generated as perfect face-centered-cubic (FCC) lattice with the lattice constant of 3.634 Å. The FCC unit cell is replicated by 384 × 384 × 192 times to generate the largest copper system (113,246,208 atoms) tested in this work. The MD equations are numerically integrated by

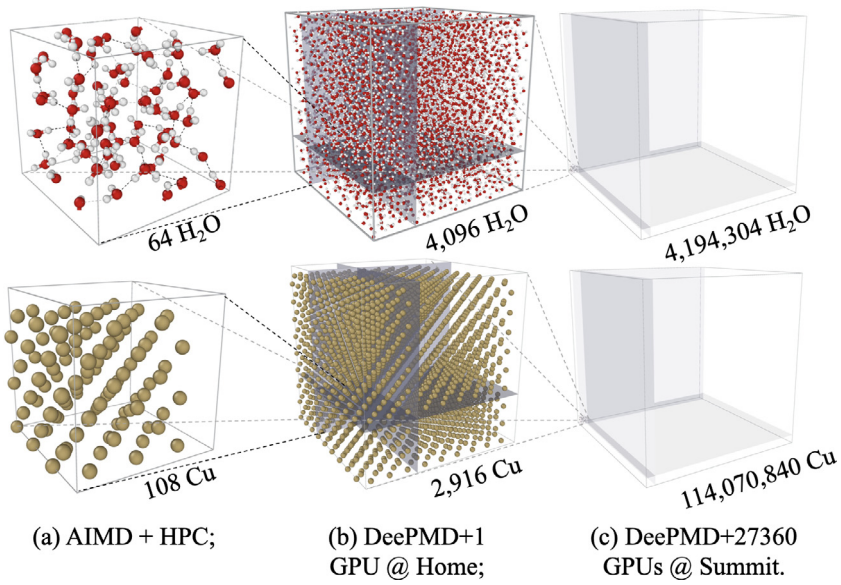

Fig. 5. Schematic illustration of two systems, water and copper, tested in this study. Shown in the figure are the system sizes accessible on different machines using different methods. (a) Sizes accessible by a typical AIMD simulation on HPC; (b) Sizes accessible by a typical DeePMD simulation with a single GPU; (c) Sizes accessible by the current study, the simulations using 27,360 GPUs on summit.

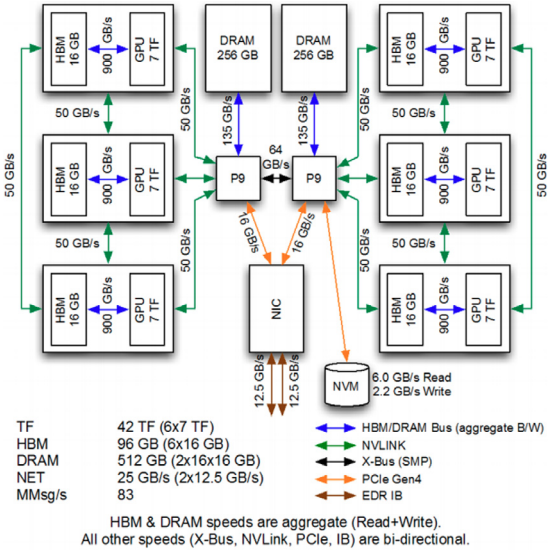

Fig. 6. The architecture of a computational node on Summit.

the Velocity-Verlet scheme for 500 steps (the energy and forces are evaluated for 501 times) at time-steps of 0.5 fs and 1.0 fs, respectively. The velocities of the atoms are randomly initialized subjected to the Boltzmann distribution at 330 K. The neighbor list with a 2 Å buffer region is updated every 50 time steps. The thermodynamic data including the kinetic energy, potential energy, temperature, pressure are collected and recorded in every 20 time steps.

5. Machine configuration

All numerical tests are performed on the Summit supercomputer. Fig. 6 shows the architecture of one of the 4608 Summit computing nodes. Each computing node consists of two identical groups, and each group has one IBM POWER 9 socket and 3 NVIDIA Volta V100 GPUs connected via NVLink with a bandwidth of 50 GB/s. Each POWER socket has 22 physical CPU cores and share 256 GB DDR4 CPU main memory, and each V100 GPU has its own 16 GB high bandwidth memory. The CPU bandwidth is 135 GB/s and GPU bandwidth is 900 GB/s. Each GPU has a theoretical peak performance of 7 TFLOPS double precision operations. The two groups of hardware are connected via X-Bus with a 64 GB/s bandwidth. The computing nodes are interconnected with a non-blocking fat-tree using a dual-rail Mellanox EDR InfiniBand interconnect with a total bandwidth of 25 GB/s.

In this paper, we utilize the MPI+CUDA programming model. In all the GPU tests, we use 6 MPI tasks per computing node (3 MPI tasks per socket to fully take advantage of both CPU–GPU affinity and network adapter), and each MPI task is bound to an individual GPU.

6. Numerical results

We compare the efficiency of the GPU version of DeePMD-kit to its CPU version for the water system with 12,582,912 atoms. In the CPU calculations, we utilize 42 MPIs per node to take full advantage of the Power 9 CPU sockets. In Fig. 7, we measure the wall clock time per MD step by averaging over 500 MD steps using both the CPU and GPU versions of DeePMD-kit. All numerical experiments in this paper are performed using double precision due to the high accuracy nature of the DeePMD-kit code.

First, we compare the performance of the GPU version of DeePMD-kit to its CPU version with the same number of nodes. Note that the CPU version can accommodate bigger physical systems because the size of the CPU memory per node (512 GB) is 5 times bigger than that of the GPU (96 GB) as shown in Fig. 6. However, in terms of computational speed, testing results indicate that the GPU version can be 39 times faster on 80 Summit nodes (480 V100 NVIDIA GPUs against 3360 POWER 9 CPU cores). The speedup factor decreases to 16 when 4560 nodes are used (27,360 GPUs against 191,520 CPU cores). The decrease of the speedup factor is due to the fact that, as shown in Fig. 8, the CPU code has a better strong scaling compared to the GPU code. It is also worth noting that the GPU version is already much faster than the CPU version in the baseline implementation: as shown in Fig. 7, the GPU baseline is 39 times faster than that of the CPU code when using 3360 CPU cores, and even faster than that of the

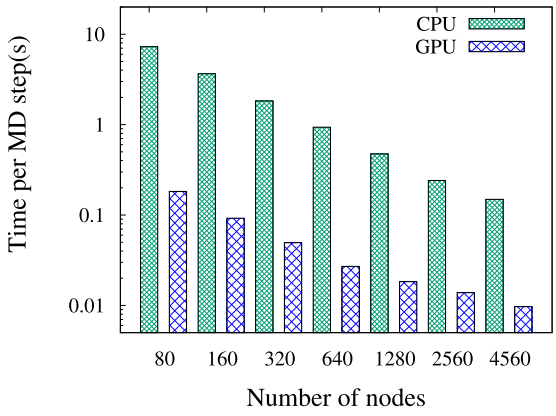

Fig. 7. Average wall clock time (log-scaled) of single MD step for a water system with 12,582,912 atoms using both CPU and GPU versions of the DeePMD-kit.

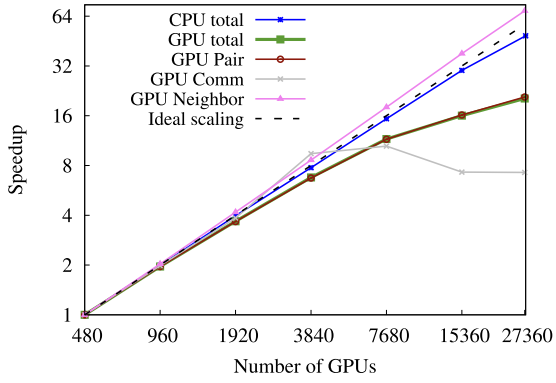

Fig. 8. Strong scaling results of simulating a 12,582,912-atom water system with both CPU and GPU versions of the DeePMD-kit. The speedup is computed by setting the wall clock time for 80 nodes as the baseline. It is noted that the GPU baseline is 39 times faster compared to the CPU baseline as shown in Fig. 7.

CPU code on 4560 nodes. A detailed discussion of the scaling will be presented in Section 7.

Next, we compare the GPU version to the CPU version under the same power consumption, which is particularly important for the upcoming exascale computing era. The power consumption of a single POWER 9 socket is 190 W, and 300 W for a single NVIDIA V100 GPU. Hence, the power consumption of a single CPU node with 2 POWER 9 CPU sockets is 380 W, while the power consumption of each GPU node with 6 NIVIDA V100 GPUs and 2 POWER 9 CPU sockets is 2180 W. 80 GPU nodes on Summit has a power consumption of 174,400 W, and that is equivalent to the power consumption of 459 CPU nodes. In our tests, the GPU version of the DeePMD-kit can be 7 times faster compared to the CPU version under the same power consumption.

Fig. 8 demonstrates the strong scaling of a 12,582,912-atom water system with respect to the number of nodes. For this system, we find our GPU implementation can perfectly scale up to 640 nodes (3840 GPUs) with 3276 atoms per GPU, and continue to scale up to the entire Summit supercomputer (4560 nodes with 27,360 GPUs) with 455 atoms per GPU. We remark that the strong scaling defines the speed of the MD simulation, i.e., the GPU code can finish 110 MD steps per second for the water system of 12,582,912 atoms (4,194,304 molecules) when scaled to 4560 Summit nodes. This delivers a capability of simulating the water system for 4.8 ns (with a time steps of 0.5 fs) in one day.

Fig. 9 shows the weak scaling of the GPU version of the DeePMD-kit for the copper systems. In the test, each MPI holds

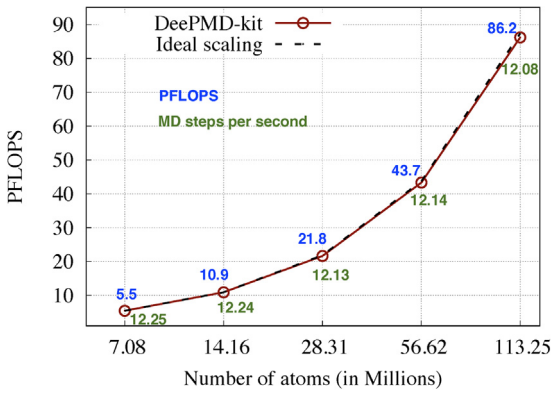

Fig. 9. Weak scaling of the copper system. Each GPU holds 4139 atoms, and the corresponding number of nodes scales from 285 to 4560 throughout the tests.

4139 copper atoms on average. The number of GPUs scales from 1710 to 15,360, and the corresponding number of atoms varies from 7,077,888 to 113,246,208, respectively. Our tests show that the GPU version of DeePMD-kit can achieve perfect scaling up to 4560 nodes. For the 113,246,208 systems with copper atoms, we achieve 86.2 PFLOPS with 4560 Summit nodes, reaching 43% of the peak performance of Summit. Each MD step only takes 83 milliseconds, therefore enabling one nanosecond simulation in one day. A detailed discussion on the floating point operations per second (FLOPS) will be presented in the next section.

Refs. [46,55] incorporate the "baseline implementation" of DeePMD-kit for MD simulations of water and copper systems, respectively, and have demonstrated that the accuracy of DeePMD is comparable to that of the AIMD simulation. In this work, the optimized GPU DeePMD-kit uses the same floating point precision as the baseline implementation. The energy, force and virial predictions are found to be consistent with the baseline implementation up to 15, 10 and 13 digits, respectively. Therefore, the *ab initio* accuracy of the GPU DeePMD-kit is warranted.

7. Performance analysis

In this section, we provide a detailed analysis for the GPU version of DeePMD-kit. The total number of floating point operations (FLOP) for the 12,582,912 atoms water system is 1.2483×10^{17}. This is collected from the CUDA profiling tool NVPROF. Although NVPROF only collects the FLOP number on the GPU, in our implementation, the CPU is only in charge of constructing and communicating the neighbor list and the corresponding FLOP number only accounts for less than 1% of the total FLOP. The FLOPS is calculated by (total FLOP)/(total time) and the corresponding efficiency is calculated via $\frac{\text{FLOPS}}{(\text{number of nodes}) \times 43 \text{ TFLOPS}}$ (each V100 GPU has 7.0 TFLOPS, and each IBM Power 9 socket 515 GFLOPS, thus $7 \times 6 + 0.515 \times 2 = 43$ TFLOPS in total). The efficiency of GPU version of DeePMD-kit is 38% when using 480 GPUs, and decreases to 13% when using 27,360 GPUs for the water system with 12,582,912 atoms, as shown in Fig. 10. On the other hand, the weak scaling of the copper system shows the GPU version of DeePMD-kit achieves a peak performance of 86 PFLOPS in double precision with 4560 nodes on Summit (43% of the peak) when calculating 113,246,208 copper atoms.

We notice that the GPU version of DeePMD-kit shows better performance (43% of the peak) on the copper system than the water system (36.8% of the peak). This is mainly due to two reasons: first, the average numbers of neighbors for each atom are 500 and 138 for the copper and water systems, respectively. Thus, the corresponding GEMM operation takes a larger proportion

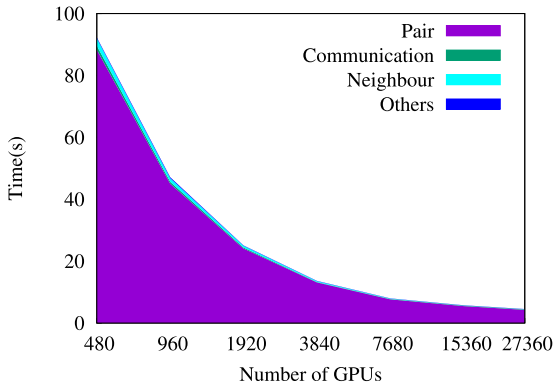

Fig. 10. Wall clock time of 500 MD steps for a water system of 12,582,912 atoms.

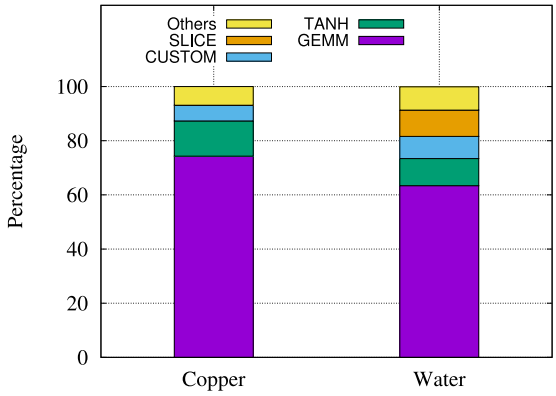

Fig. 11. Percentage of computational time by different TensorFlow operators for copper and water systems using the GPU version of DeePMD-kit.

in the copper system compared to that of the water system. Secondly, since copper is a mono-species atomic system, no extra sorting and slicing in the computation of the embedding matrix is needed as discussed in Section 3. Fig. 11 shows the proportion of different operations for both water and copper systems on the GPU. We find that the GEMM operator takes 92% and 64% of the GPU time for the copper and water system, respectively.

The total computational time of the MD simulation can be divided into four parts: Pair, MPI Communication, Neighbor, and Others. The wall clock time for 500 steps of MD for each part is listed in Table 1 and shown in Fig. 10. The corresponding strong scaling of the GPU DeePMD-kit is shown in Fig. 8. The total wall clock time is dominated by the evaluation of the atomic energies and forces, and denoted as "Pair". Therefore, the scaling of the Pair part is nearly the same as that of the total time in Fig. 8. The "Comm" part denotes the time used in updating the ghost region between adjacent MPI tasks. It scales with the number of GPUs when using less than 640 computing nodes, then becomes nearly a constant afterwards. This is because the communication time is gradually dominated by the network latency as the ghost region scales. The wall clock time for constructing the neighbor-list is labeled as "Neighbor". We notice that this operation shows superlinear scaling, which is attributed to both the reduction of the data size and better Cache hit ratio when more CPUs are used. The "Others" part includes all other calculations such as the IO and the computations invoked by fixes, and it only contributes to less than 1% of the total time, thus is negligible.

We focus on analyzing the Pair part, which takes more than 93% of the total time throughout the strong scaling tests for the 12,582,912 atom water system. This part includes the CPU–GPU memory copy operations and computation of the atomic energies and forces as discussed in Section 3.2. The efficiency of the Pair part is measured by the percentage of the peak performance as shown in Table 2. We find that the performance of this part highly depends on the data size. Note that when scaled up to 4560 computing nodes, each GPU only holds 459 atoms on average, the total GPU memory usage is around 227 MB (from Eq. (13)). The resulting data size cannot fully exploit the 7 TFLOPS computing power of the V100 GPU, which downgrades the efficiency. However, the GPUs are efficiently utilized when each GPU holds more than 3200 atoms. The efficiency drops dramatically with less than 1000 atoms per GPU. We also notice that the CPU–GPU memory copy slows down from 23.2 GB/s with 480 GPUs to 4.7 GB/s with 27,360 GPUs. Another reason for the drop of efficiency is the load imbalance when a large number of GPUs are used. For example, when scaled to 27,360 GPUs (4560 nodes), the minimum number of atoms per GPU is 407 while the maximum is 505. The load imbalance leads to waits in the execution, and reduces the efficiency of the GPU. The load balancing problem can be alleviated by dynamically redistributing the atoms onto the MPI tasks in the MD calculation [71–74].

The communication of the ghost region is performed with the adjacent MPI tasks, and the data size is listed in Table 2. The received size of a ghost region for each GPU from its neighboring MPI tasks is 25,566 (613 kB) when using 480 GPUs, and decreases to 3039 (73 kB) when using 27,360 GPUs. Table 1 shows that the communication time decreases as the data size becomes smaller from 480 to 7680 GPUs. Eventually, the communication time of the ghost region is dominated by the latency of the network, thus it stops scaling when using 15,360 and 27,360 GPUs in Table 2.

Collective MPI communication is also needed in obtaining the global properties for data IO during the simulation. Properties such as total energy, the stress, and the temperature, etc. are collected via MPI_Allreduce. Since each of those properties is merely one double precision number, the MPI_Allreduce operations are dominated by network latency. However, these latency can be a bottleneck in the extreme scale run if the physical properties are collected at every time step. By setting the output of the above mentioned properties to every 20 time steps, we find that the latency only accounts for less than 1% of the total time. Since the latency is mainly caused by the implicit MPI_Barrier, it can be further avoided by using the asynchronous MPI_Iallreduce operation.

Table 1
Wall clock time of the computationally intensive components for calculating 500 MD steps of 12,582,912 atoms of water system. All the testing results are in seconds.

#GPUs	480	960	1920	3840	7680	15,360	27,360
Pair	88.4	45.03	24.08	13.13	7.66	5.46	4.25
Comm	1.18	0.59	0.31	0.13	0.11	0.16	0.16
Neighbor	2.39	1.18	0.57	0.28	0.13	0.06	0.03
Others	0.32	0.30	0.12	0.09	0.08	0.08	0.09
Total time	92.3	47.1	25.1	13.6	8.0	5.8	4.5
#CPU cores	3360	6720	13,440	26,880	53,760	107,520	191,520
Total time	3632.8	1824.5	914.3	468.3	237.0	120.8	74.5

Table 2
Average number of atoms (per GPU), ghost atom number (per GPU) and FLOPS for a 12,582,912 atom water system.

#GPUs	480	960	1920	3840	7680	15 360	27 360
#atoms	26 214	13 107	6553	3276	1638	819	459
#ghosts	25 566	16 728	11 548	7962	5467	3995	3039
PFLOPS	1.35	2.65	4.98	9.16	15.63	21.66	27.51
% of Peak	38.54	37.76	35.46	32.64	27.85	19.30	13.75

8. Conclusion

In this work, we propose the GPU adapted algorithms and re-implement the DeePMD-kit package on the heterogeneous supercomputer Summit.

The weak scaling tests show that DeePMD-kit can scale up to 99% of the Summit supercomputer, reaching a peak performance of 86.2 PFLOPS (43% of the peak). For this particular system, each MD step only takes 83 ms, thereby enabling nanoseconds time scale simulation with *ab initio* accuracy for the first time. For a typical water system consisting of 12,582,912 atoms, our GPU code can scale up to 27,360 GPUs and run MD for 110 steps in one second. Compared to the CPU version, the GPU code is 16 — 39 times faster when using the same number of nodes, and 7 times faster under the same power consumption.

To study problems like heterogeneous catalysis, electrochemical cells, irradiation damage, crack propagation in brittle materials and biochemical reactions, the required system size for molecular simulation ranges from thousands to hundreds of millions of atoms. Traditionally these systems would be investigated with EFFMDs, from which scientific conclusions could not be solidly derived due to the limited accuracy of EFFs. The unprecedented accuracy and efficiency of GPU DeePMD-kit, as realized in this work by integrating physical-based modeling, deep learning and optimized implementation on the world's largest supercomputer, open a new era of large scale molecular simulation, and will lead to groundbreaking scientific discoveries and innovations.

The success of our GPU code relies on: (1) adapting the data distribution of the classical MD software, (2) carefully optimizing the customized TensorFlow operators on GPU (3) optimizing the standard TensorFlow operators on GPU. We remark that all these optimization techniques can be employed by other DP MD packages. We also analyze the scaling, and identify that the latency of both GPU and the network is the key for future improvement of exascale supercomputers to further accelerate the DP MD codes.

Although we only demonstrate the optimization on the GPU Summit supercomputer, such strategies can also be applied to other heterogeneous architectures. For example, it can be easily converted to the Heterogeneous-compute Interface for Portability (HIP) programming model to run on the next exascale supercomputer Frontier, which will be based on AMD GPUs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Numerical tests were performed on the Summit supercomputer located in the Oak Ridge National Laboratory. The work of H. W. is supported by the National Science Foundation of China under Grant No. 11871110, the National Key Research and Development Program of China under Grant Nos. 2016YFB0201200 and 2016YFB0201203, and Beijing Academy of Artificial Intelligence (BAAI), China. This work was partially supported by the National Science Foundation, China under Grant No. 1450372, No. DMS-1652330 (W. J. and L. L.), and by the Department of Energy, United States of America under Grant No. DE-SC0017867 (L. L.). We thank a gift from iFlytek to Princeton University and the ONR, United States of America grant N00014-13-1-0338 (L. Z. and W. E), and the Center Chemistry in Solution and at Interfaces (CSI) funded by the DOE Award, United States of America DE-SC0019394 (L. Z., R. C. and W. E). The authors would like to thank Junqi Yin, Chao Yang, Lin-Wang Wang for helpful discussions.

References

[1] Roberto Car, Michele Parrinello, Phys. Rev. Lett. 55 (22) (1985) 2471.
[2] P. Hohenberg, W. Kohn, Phys. Rev. 136 (1964) 864B.
[3] Walter Kohn, Lu Jeu Sham, Phys. Rev. 140 (4A) (1965) A1133.
[4] S. Goedecker, Rev. Modern Phys. 71 (1999) 1085–1123.
[5] D.R. Bowler, T. Miyazaki, Rep. Progr. Phys. 75 (2012) 036503.
[6] Lin-Wang Wang, Byounghak Lee, Hongzhang Shan, Zhengji Zhao, Juan Meza, Erich Strohmaier, David H Bailey, SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE, 2008, pp. 1–10.
[7] Markus Eisenbach, C-G Zhou, Don M Nicholson, Greg Brown, Jeff Larkin, Thomas C Schulthess, A scalable method for ab initio computation of free energies in nanoscale systems, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009, pp. 1–8.
[8] H.J.C. Berendsen, D. Van der Spoel, R. Van Drunen, Comput. Phys. Comm. 91 (1–3) (1995) 43–56.
[9] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, J. Chem. Theory Comput. 4 (3) (2008) 435–447.
[10] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, K. Schulten, J. Comput. Phys. 151 (1) (1999) 283–312.
[11] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, K. Schulten, J. Comput. Chem. 26 (16) (2005) 1781–1802.
[12] D.A. Case, T.E. Cheatham III, T. Darden, H. Gohlke, R. Luo, K.M. Merz Jr, A. Onufriev, C. Simmerling, B. Wang, R.J. Woods, J. Comput. Chem. 26 (16) (2005) 1668–1688.
[13] S. Plimpton, J. Comput. Phys. 117 (1) (1995) 1–19.
[14] Louis Lagardère, Luc-Henri Jolly, Filippo Lipparini, Félix Aviat, Benjamin Stamm, Zhifeng F Jing, Matthew Harger, Hedieh Torabifard, G Andrés Cisneros, Michael J Schnieders, et al., Chem. Sci. 9 (4) (2018) 956–972.
[15] Blake G Fitch, Aleksandr Rayshubskiy, Maria Eleftheriou, TJ Christopher Ward, Mark Giampapa, Yuri Zhestkov, Michael C Pitman, Frank Suits, Alan Grossfield, Jed Pitera, et al., International Conference on Computational Science, Springer, 2006, pp. 846–854.
[16] Kevin J Bowers, David E Chow, Huafeng Xu, Ron O Dror, Michael P Eastwood, Brent A Gregersen, John L Klepeis, Istvan Kolossvary, Mark A Moraes, Federico D Sacerdoti, et al., SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, IEEE, 2006, p. 43.
[17] James N Glosli, David F Richards, KJ Caspersen, RE Rudd, John A Gunnels, Frederick H Streitz, Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, ACM, 2007, p. 58.
[18] Timothy C. Germann, Kai Kadau, Internat. J. Modern Phys. C 19 (09) (2008) 1315–1319.
[19] Markus Höhnerbach, Ahmed E. Ismail, Paolo Bientinesi, SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2016, pp. 69–81.
[20] Kuang Liu, Subodh Tiwari, Chunyang Sheng, Aravind Krishnamoorthy, Sungwook Hong, Pankaj Rajak, Rajiv K Kalia, Aiichiro Nakano, Ken-ichi Nomura, Priya Vashishta, et al., 2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), IEEE, 2018, pp. 41–48.
[21] Ada Sedova, John D Eblen, Reuben Budiardja, Arnold Tharrington, Jeremy C Smith, 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2018, pp. 1–13.
[22] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin W Glass, Christoph Niethammer, Nicolay Hammer, et al., Int. J. High Perform. Comput. Appl. 33 (5) (2019) 838–854.
[23] Tingjian Zhang, Yuxuan Li, Ping Gao, Qi Shao, Mingshan Shao, Meng Zhang, Jinxiao Zhang, Xiaohui Duan, Zhao Liu, Lin Gan, et al., SW_GROMACS: accelerate GROMACS on sunway taihulight, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–14.
[24] Kun Li, Honghui Shang, Yunquan Zhang, Shigang Li, Baodong Wu, Dong Wang, Libo Zhang, Fang Li, Dexun Chen, Zhiqiang Wei, OpenKMC: a KMC design for hundred-billion-atom simulation using millions of cores on Sunway Taihulight, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–16.
[25] David F Richards, James N Glosli, Bor Chan, MR Dorr, Erik W Draeger, J-L Fattebert, William D Krauss, T Spelce, Frederick H Streitz, MP Surh, et al., Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis, IEEE, 2009, pp. 1–12.
[26] Romelia Salomon-Ferrer, Andreas Walter Goetz, Duncan Poole, Scott Le Grand, Ross C Walker, J. Chem. Theory Comput. 9 (9) (2013) 3878–3888.
[27] Yoshimichi Andoh, Noriyuki Yoshii, Kazushi Fujimoto, Keisuke Mizutani, Hidekazu Kojima, Atsushi Yamada, Susumu Okazaki, Kazutomo Kawaguchi, Hidemi Nagao, Kensuke Iwahashi, et al., J. Chem. Theory Comput. (2013).

[28] William McDoniel, Markus Höhnerbach, Rodrigo Canales, Ahmed E Ismail, Paolo Bientinesi, International Supercomputing Conference, Springer, 2017, pp. 61–78.
[29] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, J. Comput. Chem. 28 (16) (2007) 2618–2640.
[30] J.A. Anderson, C.D. Lorenz, A. Travesset, J. Comput. Phys. 227 (10) (2008) 5342–5359.
[31] Szilárd Páll, Berk Hess, Comput. Phys. Comm. 184 (2013) 2641–2650.
[32] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, Kota Kasahara, 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), IEEE, 2016, pp. 1–5.
[33] Chen Yang, Tong Geng, Tianqi Wang, Rushi Patel, Qingqing Xiong, Ahmed Sanaullah, Chunshu Wu, Jiayi Sheng, Charles Lin, Vipin Sachdeva, et al., Fully integrated FPGA molecular dynamics simulations, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–31.
[34] Tetsu Narumi, Ryutaro Susukita, Takahiro Koishi, Kenji Yasuoka, Hideaki Furusawa, Atsushi Kawai, Toshikazu Ebisuzaki, SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, IEEE, 2000, p. 54.
[35] Makoto Taiji, Tetsu Narumi, Yousuke Ohno, Noriyuki Futatsugi, Atsushi Suenaga, Naoki Takada, Akihiko Konagaya, Protein explorer: A petaflops special-purpose computer system for molecular dynamics simulations, in: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, 2003, pp. 15.
[36] Tetsu Narumi, Yousuke Ohno, Noriaki Okimoto, Takahiro Koishi, Atsushi Suenaga, Noriyuki Futatsugi, Ryoko Yanai, Ryutaro Himeno, Shigenori Fujikawa, Makoto Taiji, et al., Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, ACM, 2006, p. 49.
[37] David E Shaw, Martin M Deneroff, Ron O Dror, Jeffrey S Kuskin, Richard H Larson, John K Salmon, Cliff Young, Brannon Batson, Kevin J Bowers, Jack C Chao, et al., Commun. ACM 51 (7) (2008) 91–97.
[38] David E Shaw, JP Grossman, Joseph A Bank, Brannon Batson, J Adam Butts, Jack C Chao, Martin M Deneroff, Ron O Dror, Amos Even, Christopher H Fenton, et al., Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, 2014, pp. 41–53.
[39] Jörg Behler, Michele Parrinello, Phys. Rev. Lett. 98 (14) (2007) 146401.
[40] Albert P Bartók, Mike C Payne, Risi Kondor, Gábor Csányi, Phys. Rev. Lett. 104 (13) (2010) 136403.
[41] Matthias Rupp, Alexandre Tkatchenko, Klaus-Robert Müller, O Anatole VonLilienfeld, Phys. Rev. Lett. 108 (5) (2012) 058301.
[42] Stefan Chmiela, Alexandre Tkatchenko, Huziel E Sauceda, Igor Poltavsky, Kristof T Schütt, Klaus-Robert Müller, Sci. Adv. 3 (5) (2017) e1603015.
[43] Kristof Schütt, Pieter-Jan Kindermans, Huziel Enoc Sauceda Felix, Stefan Chmiela, Alexandre Tkatchenko, Klaus-Robert Müller, Advances in Neural Information Processing Systems, 2017, pp. 992–1002.
[44] Justin S. Smith, Olexandr Isayev, Adrian E. Roitberg, Chem. Sci. 8 (4) (2017) 3192–3203.
[45] Jiequn Han, Linfeng Zhang, Roberto Car, Weinan E, Commun. Comput. Phys. 23 (3) (2018) 629–639.
[46] Linfeng Zhang, Jiequn Han, Han Wang, Roberto Car, Weinan E, Phys. Rev. Lett. 120 (2018) 143001.
[47] Linfeng Zhang, Jiequn Han, Han Wang, Wissam Saidi, Roberto Car, E. Weinan, in: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems 31, Curran Associates, Inc., 2018, pp. 4441–4451.
[48] Alireza Khorshidi, Andrew A. Peterson, Comput. Phys. Comm. 207 (2016) 310–324.
[49] Kun Yao, John E Herr, David W Toth, Ryker Mckintyre, John Parkhill, Chem. Sci. 9 (8) (2018) 2261–2269.
[50] Han Wang, Linfeng Zhang, Jiequn Han, E. Weinan, Comput. Phys. Comm. 228 (2018) 178–184.
[51] Adam S Abbott, Justin M Turney, Boyi Zhang, Daniel GA Smith, Doaa Altarawy, Henry F Schaefer, J. Chem. Theory Comput. (2019).
[52] Kyuhyun Lee, Dongsun Yoo, Wonseok Jeong, Seungwu Han, Comput. Phys. Comm. 242 (2019) 95–103.
[53] Andreas Singraber, Jörg Behler, Christoph Dellago, J. Chem. Theory Comput. 15 (3) (2019) 1827–1840.
[54] QUIP - quantum mechanics and interatomic potentials, 2020, https://github.com/libAtoms/QUIP (Accessed: 2020-03-03).
[55] Yuzhi Zhang, Haidi Wang, Weijie Chen, Jinzhe Zeng, Linfeng Zhang, Han Wang, E. Weinan, Comput. Phys. Comm. (2020) 107206.
[56] Linfeng Zhang, De-Ye Lin, Han Wang, Roberto Car, E. Weinan, Phys. Rev. Mater. 3 (2) (2019) 023804.
[57] Linfeng Zhang, Jiequn Han, Han Wang, Roberto Car, E. Weinan, J. Chem. Phys. 149 (3) (2018) 034101.
[58] Linfeng Zhang, Mohan Chen, Xifan Wu, Han Wang, E. Weinan, Roberto Car, Phys. Rev. B 102 (2020) 041121.
[59] Grace M. Sommers, Marcos F. Calegari Andrade, Linfeng Zhang, Han Wang, Roberto Car, Phys. Chem. Chem. Phys. 22 (2020) 10592–10602.
[60] Leonardo Zepeda-Núñez, Yixiao Chen, Jiefu Zhang, Weile Jia, Linfeng Zhang, Lin Lin, Deep density: circumventing the Kohn-Sham equations via symmetry preserving neural networks, 2019, arXiv preprint arXiv:1912.00775.
[61] Luigi Bonati, Michele Parrinello, Phys. Rev. Lett. 121 (26) (2018) 265701.
[62] Hsin-Yu Ko, Linfeng Zhang, Biswajit Santra, Han Wang, Weinan E, Robert A DiStasio Jr, Roberto Car, Mol. Phys. 117 (22) (2019) 3269–3281.
[63] Marcos F. Calegari Andrade, Hsin-Yu Ko, Linfeng Zhang, Roberto Car, Annabella Selloni, Chem. Sci. 11 (2020) 2335–2341.
[64] Jinzhe Zeng, Liqun Cao, Mingyuan Xu, Tong Zhu, John ZH Zhang, Neural network based in silico simulation of combustion reactions, 2019, arXiv preprint arXiv:1911.12252.
[65] Wen-Kai Chen, Xiang-Yang Liu, Wei-Hai Fang, Pavlo O Dral, Ganglong Cui, J. Phys. Chem. Lett. 9 (23) (2018) 6702–6708.
[66] Fu-Zhi Dai, Bo Wen, Yinjie Sun, Huimin Xiang, Yanchun Zhou, J. Mater. Sci. Technol. (2020).
[67] Aris Marcolongo, Tobias Binninger, Federico Zipoli, Teodoro Laino, ChemSystemsChem (2019).
[68] Hao Wang, Xun Guo, Linfeng Zhang, Han Wang, Jianming Xue, Appl. Phys. Lett. 114 (24) (2019) 244101.
[69] Qianrui Liu, Denghui Lu, Mohan Chen, J. Phys.-Condens. Matter 32 (14) (2020).
[70] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al., 12th {*USENIX*} Symposium on Operating Systems Design and Implementation ({*OSDI*} 16), 2016, pp. 265–283.
[71] Abhinav Bhatelé, Laxmikant V. Kalé, Sameer Kumar, Dynamic topology aware load balancing algorithms for molecular dynamics applications, in: Proceedings of the 23rd International Conference on Supercomputing, 2009, pp. 110–116.
[72] Long Chen, Oreste Villa, Sriram Krishnamoorthy, Guang R Gao, 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), IEEE, 2010, pp. 1–12.
[73] W Michael Brown, Peng Wang, Steven J Plimpton, Arnold N Tharrington, Comput. Phys. Comm. 182 (4) (2011) 898–911.
[74] Jens Glaser, Trung Dac Nguyen, Joshua A Anderson, Pak Lui, Filippo Spiga, Jaime A Millan, David C Morse, Sharon C Glotzer, Comput. Phys. Comm. 192 (2015) 97–107.