Deep neural networks potentials for scalable molecular dynamics simulations on accelerator hardware

Andres Cruz

September 27, 2025

Abstract

Abstract ....

Contents

Chapter 1

Introduction

....

Chapter 2

Background

Molecular dynamics (MD) simulations rely on accurate and efficient descriptions of potential energy surfaces (PES), which are essential for predicting molecular properties and dynamics [1]. Traditional approaches either achieve high accuracy at prohibitive cost (*ab initio* methods) or high efficiency with limited transferability (empirical force fields) [2]. Machine learning force fields (MLFFs) have been proposed to bridge this gap by combining quantum accuracy with computational efficiency. Yet their practical use requires not only careful training but also efficient implementations [3]. The rise of computing power—particularly the use of specialized hardware such as graphics processing units (GPUs) with CUDA parallelism—has further accelerated their adoption. This chapter introduces the key concepts underlying MLFFs and neural networks, outlines the basics of GPU computing, and prepares the ground for the discussion of DeePMD, the neural network potential employed in this work.

2.1 Molecular Dynamics and Potential Energy Surfaces

Molecular dynamics (MD) simulations have become a standard tool in modern computational chemistry, physics, and materials science, enabling the investigation of molecular processes at the atomic level across diverse systems and phenomena [4]. Their versatility has made them valuable not only in fundamental studies of liquids [5, 6, 7], biomolecules [8, 9], and membranes [10, 11], but also in applied research areas such as drug discovery [12, 13], neuroscience [14], and geosciences [15, 16].

For example, in geoscience, MD simulations combined with *ab initio* methods and machine learning have been used to study hydrogen diffusion in garnet—a nominally anhydrous mineral found in the Earth's crust and mantle. This work revealed how cation vacancies and atomic vibrations affect hydrogen mobility and helped explain experimental discrepancies in diffusivity and activation energy [17].

A central quantity in MD is the *potential energy surface* (PES), which governs molecular properties such as structure, reactivity, spectroscopy, dipole moments, and polarizability. Since PESs cannot generally be obtained from experiments, they are defined theoretically using the Born-Oppenheimer (BO) approximation. The BO approximation assumes that electrons adjust instantaneously to nuclear motion, allowing the PES to be expressed as a function of nuclear coordinates. This is valid unless PESs corresponding to different electronic states approach each other too closely [1, 18].

Ab initio molecular dynamics (AIMD) is a highly accurate method that generates PESs and interatomic forces on the fly. Typically based on density functional theory (DFT), AIMD casts electron-electron interactions in many-electron systems into an effective one-electron potential dependent only on the electron density [19]. However, its applicability is limited to nanometer length scales and picosecond timescales [20, 21].

Constructing PESs requires a faithful representation of atomic environments, often called *descriptors* or *fingerprints*. Cartesian coordinates provide unambiguous atomic positions but are unsuitable for comparing molecular structures, since identical configurations may appear different after rotations, reflections, or translations. A good descriptor must therefore be invariant to these transformations while remaining *complete*, i.e., capable of distinguishing genuinely different environments [22].

The PES is a high-dimensional function, with $3N-6$ degrees of freedom for a system of N atoms. Its explicit construction is intractable, so PESs are usually approximated by physically motivated functional forms, also known as empirical force fields (EFFs). EFFs can effectively fit limited data but require intuition in functional choice and parameterization. They have mainly been reported for elemental or binary systems, with applications to multicomponent systems being rare. Although efficient, EFFs are generally restricted to the systems they were designed for and often fail to generalize [23, 24, 25].

These limitations motivated the development of more flexible, data-driven approaches to PES construction [26]. The use of machine learning will be discussed in the next section 2.2.

2.2 Machine learning in molecular simulations

To extend simulations to larger systems while retaining quantum-level accuracy, machine learning (ML) methods have emerged as a promising alternative. Many ML algorithms minimize a training objective by adjusting parameters to reduce deviations from reference data [27]. In molecular simulations, ML methods have been applied to accelerate sampling of equilibrium states [28, 29], rare events [30], reaction rates [31], and protein folding [3].

Machine learning methods have also been applied to the construction of force fields, giving rise to the so-called machine learning force fields (MLFFs). The first approach aimed to fit an empirical force field using vibrational spectra at the ground and excited states to determine the parameters of an anharmonic PES [32, 33]. One later work sought to predict the potential energy potential for CO/Ni(111) from the parameters that describe its configuration [26]. However, this latter approach faced significant challenges, including the absence of a systematic way to determine the number of required parameters and the use of internal coordinates that introduced arbitrary atom orderings, breaking permutation invariance.

Since then, several approaches have been proposed to address transferability and invariance issues. A general classification based on [3]:

- *High-dimensional neural networks potentials (HDNNPs)* are artificial neural networks (NN) based potentials proposed in [23]. In this framework, the system is decomposed into atom-centered environments, and the total energy is expressed as the sum of atomic contributions predicted by a neural network. The assumption is that each atom's energetic contribution depends primarily on its local chemical environment, which allows predictions to generalize to larger systems if enough sampling of the local environments are available.

  Two major variants of HDNNPs exist:

  descriptor-based NNPs, which rely on predefined mathematical functions (e.g., atom-centered symmetry functions, ACSFs) to encode local atomic environments into fixed-length vectors that serve as neural network inputs. This approach enables extension to systems with multiple atomic types, but accuracy depends heavily on descriptor choice and may require significant domain expertise. Moreover, high-dimensional descriptors increase the computational burden of both descriptor calculation and network evaluation.

  end-to-end NNs, inspired by graph neural networks and often implemented as message-passing neural networks (MPNNs). These models take nuclear charges and Cartesian coordinates directly as input and learn suitable representations from the data. By iteratively exchanging information along the graph edges (atoms as nodes, interactions as edges), MPNNs capture complex chemical interactions without the need for handcrafted descriptors and can generalize beyond Euclidean feature spaces [34, 35].

- *Gradient Domain Machine Learning (GDML)* represents a kernel-based alternative designed to reconstruct flexible force fields from relatively small datasets of high-level *ab initio* calculations. Instead of predicting energies and deriving forces, GDML is trained directly on forces, leveraging the fact that a dataset of M energies for N atoms naturally provides $3MN$ force components, thus yielding richer information. This makes the model more flexible and avoids the amplification of noise that would otherwise arise from differentiating predicted energies. However, the original GDML descriptor—based on inverse pairwise distances—was not invariant under atom permutations. To address this, the symmetric GDML extension introduced permutation-invariant descriptors, improving generalization across equivalent atomic orderings.

- *Gaussian approximation potential (GAP)* are another kernel-based approach, originally developed for materials but later extended to molecular systems. Like HDNNPs, GAP decomposes a system into atom-centered environments, but instead of neural networks, it uses kernel regression with carefully chosen descriptors. One common representation is the local atomic density $\rho(r) = \sum_j \delta(\mathbf{r} - \mathbf{r}_j)$, while a more widely adopted option is the smooth overlap of atomic positions (SOAP) kernel. SOAP compares atomic environments by integrating the overlap of their local densities over all possible rotations:

$$K(\rho, \rho') = \int d\mathcal{R} \left| \int \rho(\mathbf{r})\rho'(\mathcal{R}\mathbf{r})d\mathbf{r} \right|^n . \tag{2.1}$$

These descriptors are explicitly invariant to atom permutations and rotations, ensuring consistent treatment of equivalent environments.

While their ultimate accuracy is limited by the quality of the reference *ab initio* data, when properly trained they can deliver near-quantum accuracy at a fraction of the computational cost. Nevertheless, the significant time and resources required for data generation and training must be considered, as they may offset the efficiency gained during inference [3].

As this thesis focuses primarily on neural network potentials, the next section 2.3 will examine them in more detail.

2.3 Neural networks and optimization

Neural networks (NNs) are powerful universal function approximators, capable of representing highly nonlinear relationships. Their strength lies in learning complex patterns from data and adapting to an objective function [36], which makes them particularly well suited for modeling the nonlinearities of potential energy surfaces (PES).

A neural network can be seen as an interconnection of nodes, or *neurons*, whose collective behavior determines the mapping from inputs to outputs. Several properties characterize a network [37]:

- *node character*, which defines how each neuron processes information. This includes the number of inputs and outputs, the weights associated with each input, and the nonlinear activation function that generates the output.

- *topology*, describing how neurons are arranged and connected. Typically, nodes are organized into layers: an input layer, one or more hidden layers, and an output layer. Networks may be classified as *feedforward*, where information flows strictly forward without loops, or *feedback*, where recurrent connections allow outputs of one layer (or even the same layer) to influence earlier computations.

- learning rules, which specify how weights and biases are initialized and subsequently adjusted during training.

The use of nonlinear activation functions provides the flexibility required to approximate PES with high accuracy. However, NN performance depends not only on architecture but also on training. Optimization is typically carried out using gradient-based methods, such as gradient descent, which aim to minimize a *loss function* — a measure of the difference between predicted and reference values. The gradient of the loss with respect to network parameters (weights and biases) indicates how these parameters should be updated. By iteratively adjusting them in the opposite direction of the gradient, the network converges toward minimizing the prediction error [38, 39].

Although MLFF inference is significantly faster than ab initio molecular dynamics (AIMD), the evaluation of descriptors and neural networks in large systems remains a computational bottleneck. To address this, the next section 2.4 introduces GPU acceleration through CUDA, an NVIDIA platform designed for programming massively parallel computations.

2.4 GPU acceleration with CUDA

A *CUDA kernel* is a function executed on the GPU by thousands of lightweight *threads* running in parallel. Understanding how these threads are organized and how memory is managed is essential to exploit GPU parallelism efficiently. This section offers a summary of main concepts based on [40, 41, 42]. The CUDA execution hierarchy could be structured as follows:

1. *Thread*: the smallest execution unit, representing a single instruction stream.

2. *Warp*: a group of 32 threads executed in lockstep fashion by the hardware.

3. *Block*: a set of threads. Threads within a block can cooperate via fast shared memory. Each block is internally divided into warps.

4. *Streaming Multiprocessor (SM)*: the hardware unit responsible for executing entire blocks. Each SM can schedule multiple blocks concurrently but resource limits (registers, shared memory, and active warps) determine the level of concurrency. All threads in a warp are executed simultaneously.

5. *Grid*: the top-level structure, consisting of all blocks launched by a kernel.

Equally critical is the memory hierarchy, as performance strongly depends on where data are stored and how they are accessed:

1. **host memory**: CPU memory, where the main program runs. Data must be explicitly copied to the GPU before kernel execution and transferred back afterward. These transfers are relatively slow and can become bottlenecks.

2. **Global memory**: GPU device memory accessible by all threads. While in capacity it offers several GB of memory, it suffers from high latency, making coalesced access patterns essential.

3. **Shared memory**: on-chip memory shared among threads within a block. It is limited to 48 KB per block but fast accessed memory space. Efficient use of shared memory can improve performance, though its limited size constrains how many threads can run simultaneously on an SM.

4. **Registers**: private memory for individual threads. Excessive register usage reduces thread occupancy, limiting parallelism.

These concepts can be illustrated by the *neighbor-sorting* step, which is one of the earliest operations required to guarantee permutation invariance in local atomic environments (see chapter 3). Optimizations that reduce global memory accesses and exploit shared memory effectively can substantially improve performance.

The next chapter 4 introduces the DeePMD framework, the neural network potential of primary interest in this thesis.

Chapter 3

Parallel list ordering in GPU

One of the first steps performed in DeePMD potential calculations is sorting the neighbors of each particle by distance, so that only the closest ones contribute to its local environment [43]. In the sequential version, this was done with standard sorting algorithms, as each fitting network $\mathcal{N}_F$ was computed independently and the lists were small.

In the parallel version, however, all lists can be sorted simultaneously, since they are independent of each other. To achieve this, a parallel *Radix Sort* algorithm would be employed. HALMD already included a parallel *Radix Sort* implementation, thus, the idea was to adapt it to handle multiple independent lists.

The radix sort algorithm consists of three main stages — *Histogram-Keys*, *Scan-Buckets*, and *Rank-and-Permute* — repeated for each digit in the input data [44]. In this module, each stage corresponds to a separate kernel launch. Some observations from evaluating the original implementation are below:

- in *Histogram-Keys* inputs are distributed into buckets according to the current digit value (e.g., 10 buckets for decimal digits).

- In *Scan-Buckets* a prefix sum is computed to determine the position of each element based on its bucket count.

- In *Rank-and-Permute* elements are moved to their computed positions. Stability is required, so elements with the same digit must preserve their original relative order.

- For better performance, the input list must already be in global memory before sorting starts. Each kernel first copies data into shared memory, processes it, and writes results back to global memory only at the end.

- Instead of decimal digits, the algorithm uses binary representation, usually on groups of 8 bits, yielding 256 radix values per iteration.

- To preserve order, each bucket is divided into partitions (one per half-warp in a block). This ensures that elements processed later by a block do not overwrite earlier results.

- The Scan step then processes a list of buckets, including these partitions. For example, 4 blocks of 128 threads each would produce 8 partitions per bucket and a total of 8,192 buckets to scan.

- Finally, in the Permutation step, each thread finds its specific bucket and locate its own input values to the respective location in global memory.

- The Scan and Permutation steps both use the same number of blocks and threads (fixed to 128), while the Scan step distributes work so that each thread processes two elements of the buckets at a time.

An initial idea was to let each block handle multiple neighbor lists by increasing the number of threads per block. However, this approach would quickly exhaust shared memory, as each list required its own bucket space, limiting the number of blocks that could run concurrently. As a result, the design was simplified so that each block processes exactly one list.

With one list per block, the main change required was ensuring each block only accessed its own segment of the input and bucket arrays. Even so, performance was limited by the need to write intermediate results to global memory between the three stages and the evaluation of the next digit. To address this, the kernels were unified into a single kernel performing all three steps in shared memory and evaluating all digits in just one call. Accessing to global memory only twice — once at the beginning and once at the end — would reduce memory latency.

This approach brought new challenges related with shared memory. The prefix-sum (scan) step required additional memory and several iterations to propagate results. The algorithm for the scan step, followed the scheme proposed in [45, 44], where the input data (the buckets and partitions), would be treated as a balance binary tree. It involved an *up-sweep* phase (building the tree by summing children) and a *down-sweep* phase (propagating sums from the root to the leaves).

This required additional memory equal to the bucket size. In total, shared memory was needed to store the original input, the associated values (for key-value pairs), and twice the bucket space. While this worked for small lists, tests performed on lists with more than 256 elements revealed a limitation caused by shared memory size and repeated iterations in the scan step, which hurt performance. In terms of performance, this approach was in fact less efficient than launching three separate kernels in certain cases, such as processing up to 32 lists of 256 elements or 1,000 lists of up to 16 elements, where runtimes increased instead of decreased. However, in other scenarios the method proved beneficial, achieving a speedup of up to 1.8× over the 3 kernels approach.

To address these limitations, the CUDA block-wide primitive *cub::BlockScan* function was evaluated, as it performs a parallel scan across all items in a block [46]. However, its use revealed inconsistencies. Since each thread was required to process multiple positions of the bucket array, correct results could only be obtained by hardcoding a limit on the number of elements per thread. Because the exact number of elements assigned to each thread was not known in advance, setting a high upper bound (e.g., 1,024 positions) was not feasible. Such large per-thread arrays would allocate a substantial number of registers per thread, significantly reducing occupancy and overall parallelism. In this second approach, a consistent reduction in runtime was observed across all test cases providing for example a 4.0× speedup for lists of 256 elements and a 3.4× speedup for 1,000 lists, over the implementation without *cub::BlockScan*.

Ultimately, the solution was to replace the custom approach with another CUDA block-wide primitive: *cub::BlockRadixSort*. This function performs the entire radix sort for all digits in a single call, using one block per list [47]. Each block reads its list from global memory, sorts it entirely on the GPU, and writes the result back. This approach eliminated the need for intermediate global memory operations and significantly simplified the implementation. Here, the number of threads per block is set according to the number of elements per list, with each thread handling one element. In this approach, a speedup of up to 2.9× for 256-element lists and 9.0× for 1,000 lists, over the second approach.

Figure 3.1 compares the runtime of three algorithm variants — the 3 separate kernels version, the unified 3-step kernel, the 3-step version using *cub::BlockScan*, and the version using *cub::BlockRadixSort*. They were tested on sets of 256-element lists and on 1000 lists of varying sizes. The results showed clear improvements from using *BlockScan*, likely due to avoiding several internal loops, and even better performance using *BlockRadixSort*. Both 3-step algorithms exhausted memory when lists exceeded 256 elements.

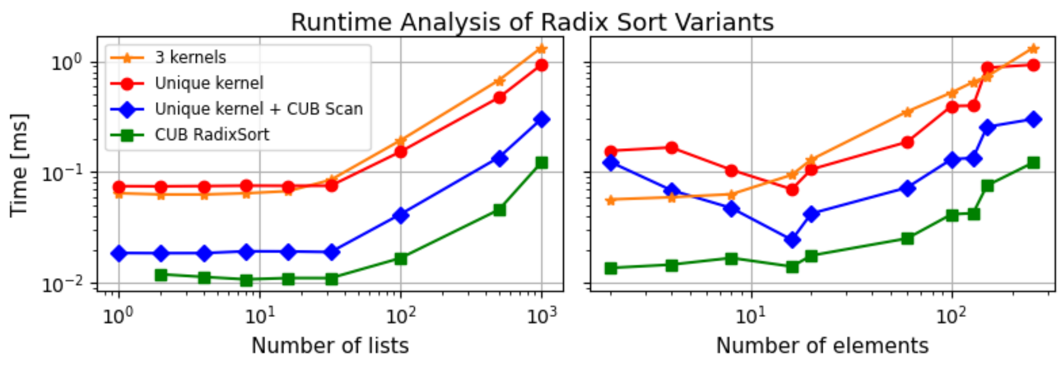

Figure 3.1. Runtime comparison of radix sort variants: 3 separate kernels using global memory, 3-step unique kernel using shared memory, 3-step using *cub::BlockScan*, and using *cub::BlockRadixSort*. Left: varying numbers of 256-element lists. Right: 1000 lists of varying sizes.

The final design using only *cub::BlockRadixSort* showed the best performance. As shown in Figure 3.2, the choice of variable type has a direct impact on runtime, since it determines the memory footprint during execution. In particular, using *double* keys—and especially key–value pairs with *double*—led to a substantial increase in runtime compared to the *unsigned int* case. The results are displayed in logarithmic scale to facilitate the comparison of lower values.

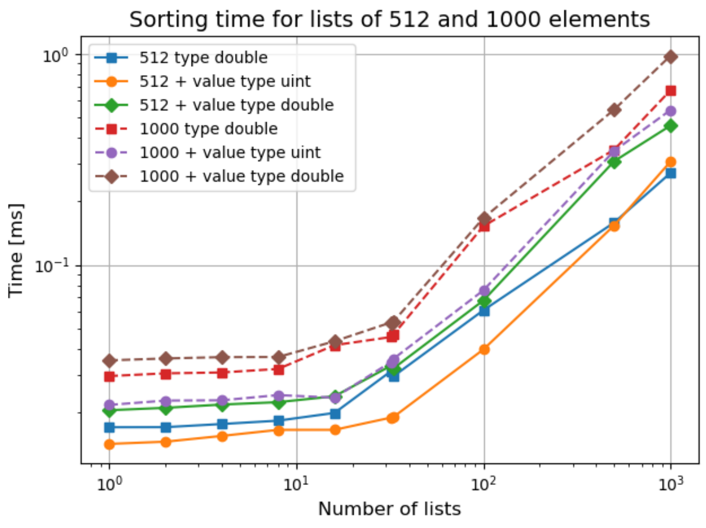

Figure 3.2. Runtime comparison for *unsigned int* and *double* keys/values in lists of 512 and 1000 elements.

Chapter 4

DeePMD Kit

DeePMD-kit is an open-source software that offers functionality for constructing descriptor-based NNPs (see section 2.2) and incorporating them into molecular dynamics workflows. According to reported applications [43], it has been applied to a range of atomistic systems, including inorganic solids[17, 48], liquids[49] or organic compounds[50], and interfaces, across various elements and thermodynamic conditions. Originally developed to provide an interface between deep learning frameworks—used for automated training of neural networks—and molecular dynamics packages [51], DeePMD-kit initially relied on TensorFlow as its backend. Support has since been extended to additional frameworks such as PyTorch and JAX [52].

Following the HDNNP scheme (see section 2.2), DeePMD-kit computes the total potential energy of a system as a sum of atomic contributions, where each atomic energy is predicted by a neural network based on a descriptor of its local chemical environment. These descriptors are constructed from atomic positions in a way that captures the local structural information while ensuring the potential energy surface (PES) respects key physical symmetries—translational, rotational, and permutational. This approach allows the model to assign distinct representations to different atomic environments, while ensuring that similar environments are mapped to similar representations, enabling a multi-body description of the system [51].

This chapter introduces the DeePMD-kit framework and its approach to energy prediction. It explains the construction of the Two-body embedding DeepPot-SE descriptor and provides an overview of TensorFlow to clarify the interaction between DeePMD-kit and its backend. The role of automatic differentiation in computing gradients is also discussed. The chapter concludes with the use of the DeePMD-kit API for practical energy predictions.

4.1 Fitting Networks

This section covers the way in which DeePMD-kit predicts the potential energy of a system. As propposed for HDNNP, the DeePMD-kit computes individual atomic energies using *fitting neural networks*, that map atomic descriptors to scalars (0-order tensors), as shown in eq. (4.1). These individual energies are computed for all particles in the system from their Cartesian coordinates and atomic species, which constitute the degrees of freedom of each atom. Let:

- $\mathbf{r_i}$ be the Cartesian coordinates of particle i,
- α_i be the atomic species of particle i,
- $\mathbf{x} = (\mathbf{r_i}, \alpha_i)$ represent the degrees of freedom of particle i,
- N_i be the set of neighboring atoms of atom i ,
- $\boldsymbol{\theta_f}$ be the trained parameters of the fitting network,
- $\boldsymbol{\theta_d}$ be the trained parameters of the descriptor network, if it exists

$$E^i = \mathcal{N}_F\Big(\mathcal{D}^i; \boldsymbol{\theta}_f\Big) = \mathcal{N}_F\Big(\mathcal{N}_D\Big(\mathbf{x}_i, \{\mathbf{x}_j\}_{j\in N(i)}; \boldsymbol{\theta}_d\Big); \boldsymbol{\theta}_f\Big). \tag{4.1}$$

Finally, the total energy is the sum of individual energies:

$$E = \sum_i^N E^i. \tag{4.2}$$

These neural network networks are defined as compositions of layers $\mathcal{L}^{(k)}$:

$$\mathcal{N} = \mathcal{L}^{(n)} \circ \mathcal{L}^{(n-1)} \circ \cdots \circ \mathcal{L}^{(1)}. \tag{4.3}$$

In DeePMD-kit, each layer $\mathcal{L}$ can take one of several forms, depending on whether a residual connection (ResNet) is used or not and on the dimensionality of the layers, where:

- $\mathbf{x} \in \mathbb{R}^{N_1}$ is the input vector.
- $\mathbf{y} \in \mathbb{R}^{N_2}$ is the output vector.
- $\mathbf{w} \in \mathbb{R}^{N_1 \times N_2}$ are trainable weights.
- $\mathbf{b} \in \mathbb{R}^{N_2}$ are trainable biases.
- $\hat{\mathbf{w}} \in \mathbb{R}^{N_2}$ is either a trainable vector for skip connections or a vector of ones.
- $\boldsymbol{\phi}$ is the activation function, such as hyperbolic tangent (tanh), rectified linear unit (ReLU) or identity.

$$\mathbf{y} = \mathcal{L}(\mathbf{x}; \mathbf{w}, \mathbf{b}) = \begin{cases} \hat{\mathbf{w}} \odot \boldsymbol{\phi}(\mathbf{x}^\top \mathbf{w} + \mathbf{b}) + \mathbf{x}, & \text{ResNet and } N_2 = N_1 \\ \hat{\mathbf{w}} \odot \boldsymbol{\phi}(\mathbf{x}^\top \mathbf{w} + \mathbf{b}) + \{\mathbf{x}, \mathbf{x}\}, & \text{ResNet and } N_2 = 2N_1 \\ \hat{\mathbf{w}} \odot \boldsymbol{\phi}(\mathbf{x}^\top \mathbf{w} + \mathbf{b}), & \text{otherwise} \end{cases}. \tag{4.4}$$

Residual Networks (ResNets) were suggested in [53] to address the difficulty of training very deep neural networks, where performance often degrades as layers are added. The key idea is to reformulate the learning objective so that each layer (or block of layers) learns a *residual function* rather than a complete transformation. Instead of learning a full mapping $H(x)$, the network learns the residual $F(x) = H(x) - x$, bypassing one or more layer through *skip connections* and adding directly the input of a block to its output. The final output of the block is then expressed as:

$$y = F(x) + x. \tag{4.5}$$

This structure allows the optimization process to focus on adjusting residual values, which should make it easier for gradient information to propagate through many layers. As a result, residual networks may enable an efficient training of much deeper models compared to standard feedforward architectures [53].

In DeePMD-kit, ResNet blocks are incorporated under specific conditions to improve training stability and efficiency when constructing DP models [54]:

- in the descriptor network: if the size of output neurons in a layer is twice the size of the input neurons, then the vector of input neurons is duplicated and concatenated to build a ResNet architecture between them.

- in the fitting network: if two consecutive layers have the same size, then a ResNet architecture is built between them.

Additionally, the user may specify whether a trainable vector is used in the skip connections. If enabled, these values are learned during the training process [43].

4.2 Descriptors

This study is primarily based on a model that employs the *Two-body embedding DeepPot-SE* descriptor with angular information. The specific components and characteristics of this descriptor will be reviwed in this section.

The descriptors, denoted as the matrix $\mathcal{D}^i \in \mathbb{R}^{M \times M_<}$, provide a multi-body of the local environment surrounding each atom i. These representations are constructed solely from pairwise distances between the central atom and its neighbors. The parameters M and $M_<$ are user-defined, corresponding respectively to the number of output neurons in the *filter* neural network and the dimensionality of a submatrix from the embedding matrix [43]:

- a user-defined maximum number of neighbors $N_c(a)$ must be specified for each atom type a, subject to the condition in eq. (4.6). This setting influences the model's memory usage, performance, and accuracy [55]. Let:

  - S be the set of all particles in the system.

  - N_i the set of neighbors of particle $i \in S$ within the cutoff radius r_c.

  - α_i denote the type of particle i.
  - $N_c(a)$ be the user-defined maximum number of neighbors of species a considered for each particle.

  Then, $N_c(s)$ must satisfy:

$$N_c(a) >= \max_{i \in S} |\{n \in N_i \mid \alpha_n = a\}|, \tag{4.6}$$

  i.e., for each species a, $N_c(a)$ must be greater than the maximum number of neighbors of species a found within the cutoff radius around any atom in the system.

- A smooth switching function $s(r)$ removes the discontinuity in the potential energy caused by the cutoff. This function transitions smoothly from 1 at r_s to 0 at r_c, and its second derivative is continuous:

$$s(r) = \begin{cases} 1, & r < r_s \\ x^3(-6x^2 + 15x - 10) + 1, & r_s \leq r < r_c \\ 0, & r \geq r_c \end{cases} \quad \text{where} \quad x = \frac{r - r_s}{r_c - r_s}. \tag{4.7}$$

- This switch function is applied to all pairwise distances between a central atom i and its N_c neighbors. The total number of neighbors of each particle in the system is:

$$N_c = \sum_a N_c(a). \tag{4.8}$$

  From these, a coordinate matrix $\mathcal{R}^i \in \mathbb{R}^{N_c \times 4}$ is constructed with rows of the form:

$$\mathcal{R}^i_j = s(r_{ij}) \begin{bmatrix} 1 & \frac{x_{ij}}{r_{ij}} & \frac{y_{ij}}{r_{ij}} & \frac{z_{ij}}{r_{ij}} \end{bmatrix}^\top. \tag{4.9}$$

  The rows in $\mathcal{R}^i$ not occupied are filled with zeros.

- Each row of the embedding matrix $\mathcal{G}^i \in \mathbb{R}^{N_c \times M}$ contains an M-dim output vector of a *filter* neural network evaluated at $s(r_{ij})$:

$$\mathcal{G}^i_j = \mathcal{N}_D\Big(s(r_{ij}); \boldsymbol{\theta_d}\Big). \tag{4.10}$$

- Finally, the descriptor $\mathcal{D}^i \in \mathbb{R}^{M \times M_<}$ is computed as

$$\mathcal{D}^i = \frac{1}{N_c^2}(\mathcal{G}^i)^T \mathcal{R}^i (\mathcal{R}^i)^T \mathcal{G}^i_<. \tag{4.11}$$

  where $\mathcal{G}^i_< \in \mathbb{R}^{N_c \times M_<}$ indicates that only the first $M_<$ columns of $\mathcal{G}^i$ are used, which is an user-defined parameter used to reduce the size of the descriptors.

With this construction, the total number of neighbors, rather than the total number of atoms, governs the complexity of the model, keeping the model physically constrained by the cutoff radius [50]. In addition, it guarantees that the descriptors $\mathcal{D}^i$ preserve *translational and rotational symmetries* of the PES. This is due to the transformation via $\mathcal{R}^i(\mathcal{R}^i)^T$, which behaves as a *symmetry matrix* [50]. Similarly, the operation $(\mathcal{G}^i)^T\mathcal{R}^i$ represents a *permutation-invariant transformation*, as proposed in [56], ensuring that permutation symmetries are also preserved.

4.3 Forces computation

Since the potential energy of the system is defined as a function of the atomic coordinates, the forces acting on each particle can be obtained as the negative gradient of the energy with respect to its Cartesian coordinates, as will be reviwed in this section. Starting from the energy expression defined in eq. (4.2) and consider the specific descriptor introduced in section 4.2, the analytical expression for forces is derived by applying the chain rule to compute energy gradients with respect to atomic coordinates.

This yields to the following expression for the force acting on particle i along the Cartesian direction $\alpha \in x, y, z$:

$$\begin{aligned}
F_{i,\alpha} &= -\frac{\partial E}{\partial r_{i,\alpha}} \\
&= -\sum_{j}^{N} \frac{\partial E^j}{\partial r_{i,\alpha}} \\
&= -\sum_{j}^{N} \frac{\partial \mathcal{N}_F(\mathcal{D}^j)}{\partial \mathcal{D}^j} \cdot \frac{\partial \mathcal{D}^j}{\partial r_{i,\alpha}} \\
&= -\frac{\partial \mathcal{N}_F(\mathcal{D}^i)}{\partial \mathcal{D}^i} \cdot \frac{\partial \mathcal{D}^i}{\partial r_{i,\alpha}} - \sum_{j \in N_i} \frac{\partial \mathcal{N}_F(\mathcal{D}^j)}{\partial \mathcal{D}^j} \cdot \frac{\partial \mathcal{D}^j}{\partial r_{i,\alpha}} \\
&= -\frac{\partial \mathcal{N}_F(\mathcal{D}^i)}{\partial \mathcal{D}^i} \cdot \frac{\partial \mathcal{D}^i}{\partial r_{ij}} \cdot \frac{\partial r_{ij}}{\partial r_{i,\alpha}} - \sum_{j \in N_i} \frac{\partial \mathcal{N}_F(\mathcal{D}^j)}{\partial \mathcal{D}^j} \cdot \frac{\partial \mathcal{D}^j}{\partial r_{ji}} \cdot \frac{\partial r_{ji}}{\partial r_{i,\alpha}}.
\end{aligned} \tag{4.12}$$

As seen in eq. (4.12), evaluating the forces requires computing the derivatives of the descriptors $\mathcal{D}$ with respect to interatomic distances $r_{ij} = |\mathbf{r}_j - \mathbf{r}_i|$ involving each central atom i and its neighbors j. Since the embedding matrices $\mathcal{G}^i$ and coordinate matrices $\mathcal{R}^i$ are based on these distances (see eqs. (4.9) and (4.10)) one finds:

$$\begin{aligned}
\frac{\partial \mathcal{D}^i}{\partial r_{ij}} = \frac{1}{N_c^2} &\left[\left(\frac{\partial \mathcal{G}^i}{\partial r_{ij}} \right)^\top \cdot \mathcal{R}^i + \mathcal{G}^i \cdot \frac{\partial \mathcal{R}^i}{\partial r_{ij}} \right] \cdot (\mathcal{R}^i)^\top \cdot \mathcal{G}^i_< \\
&+ \frac{1}{N_c^2} (\mathcal{G}^i)^\top \cdot \mathcal{R}^i \cdot \left[\left(\frac{\partial \mathcal{R}^i}{\partial r_{ij}} \right)^\top \cdot \mathcal{G}^i_< + (\mathcal{R}^i)^\top \cdot \frac{\partial \mathcal{G}^i_<}{\partial r_{ij}} \right].
\end{aligned} \tag{4.13}$$

When computing the derivative of a descriptor $\mathcal{D}^i$ with respect to the distance from a neighbor j to its central particle i, the resulting matrices (e.g., $\partial \mathcal{G}^i / \partial r_{ji}$), have all-zero rows except for the j^{th}. This structure mirrors the derivative with respect to the central atom and allows simplification of matrix products:

$$\begin{aligned}\frac{\partial \mathcal{D}^i}{\partial r_{ji}} &= \frac{1}{N_c^2}\left[\left(\frac{\partial \mathcal{G}^i}{\partial r_{ji}}\right)^\top \cdot \mathcal{R}^i + \mathcal{G}^i \cdot \frac{\partial \mathcal{R}^i}{\partial r_{ji}}\right] \cdot (\mathcal{R}^i)^\top \cdot \mathcal{G}^i_< \\ &\quad + \frac{1}{N_c^2}(\mathcal{G}^i)^\top \cdot \mathcal{R}^i \cdot \left[\left(\frac{\partial \mathcal{R}^i}{\partial r_{ji}}\right)^\top \cdot \mathcal{G}^i_< + (\mathcal{R}^i)^\top \cdot \frac{\partial \mathcal{G}^i_<}{\partial r_{ji}}\right] \\ &= \frac{1}{N_c^2}\left[\frac{\partial \mathcal{G}^i_j}{\partial r_{ij}} \otimes \mathcal{R}^i_j + \mathcal{G}^i_j \otimes \frac{\partial \mathcal{R}^i_j}{\partial r_{ij}}\right] \cdot (\mathcal{R}^i)^\top \cdot \mathcal{G}^i_< \\ &\quad + \frac{1}{N_c^2}(\mathcal{G}^i)^\top \cdot \mathcal{R}^i \cdot \left[\frac{\partial \mathcal{R}^i_j}{\partial r_{ij}} \otimes \mathcal{G}^i_{<j} + \mathcal{R}^i_j \otimes \frac{\partial \mathcal{G}^i_{<j}}{\partial r_{ij}}\right], \end{aligned} \tag{4.14}$$

where the subscript j in $\mathcal{G}_j$ and $\mathcal{R}_j$ refers to the j^{th} row of each matrix.

For the embedding matrices $\mathcal{G}^i$, the derivative with respect to the distance from its central atom i, produces a matrix of the same shape. Each row j corresponds to the derivative of the filter network output:

$$\frac{\partial \mathcal{G}^i_j}{\partial r_{ij}} = \frac{\partial \mathcal{N}_F(r_{ij})}{\partial r_{ij}}. \tag{4.15}$$

According to eq. (4.9), the derivative of the coordinate matrices $\mathcal{R}^i$ reads:

$$\frac{\partial \mathcal{R}^i_j}{\partial r_{ij}} = \frac{\partial s(r_{ij})}{\partial r_{ij}} \cdot \begin{bmatrix} 1 & \frac{x_{ij}}{r_{ij}} & \frac{y_{ij}}{r_{ij}} & \frac{z_{ij}}{r_{ij}} \end{bmatrix}^\top + s(r_{ij}) \cdot \frac{\partial}{\partial r_{ij}} \begin{bmatrix} 1 & \frac{x_{ij}}{r_{ij}} & \frac{y_{ij}}{r_{ij}} & \frac{z_{ij}}{r_{ij}} \end{bmatrix}^\top, \tag{4.16}$$

where each row j contains the derivative of the scaled coordinate vector.

4.4 TensorFlow

The computation of forces involves evaluating a variety of derivatives, a process analogous to backpropagation in machine learning frameworks. This section discusses how this is handled within TensorFlow, one such framework used for training neural networks. TensorFlow is a software library for numerical computation based on dataflow graphs, designed to support scalable machine learning workflows. A typical TensorFlow program consists of two main stages: the *construction phase* and the *execution phase* [57]:

- **construction phase:**

  in this stage, a computational graph representing the machine learning model is defined. In the graph, *edges* correspond to data in the form of tensors (vectors, matrices, or higher-dimensional arrays), while *nodes* represent operations that act on tensors. Each operation can take zero or more tensors as input and produce zero or more tensors as output.

  A crucial aspect of this phase is the computation of gradients during the backward pass. Because models often involve a large number of parameters, manually deriving gradients of the loss function is error-prone and computationally inefficient. TensorFlow addresses this through *automatic differentiation (autodiff)*, which computes these gradients algorithmically. These gradients are then used in *gradient-based optimization algorithms*, such as stochastic gradient descent (SGC), to update the model parameters.

- **execution phase:**

  once the graph is constructed, it is executed over a series of training steps to optimize the model parameters. TensorFlow can distribute operations across CPUs, GPUs, and remote devices to support scalable execution.

Finally, when deploying a trained model, a **TensorFlow session** is created to execute the required subgraph of operations and evaluate tensors. After receiving a *fetch list* (a collection of tensors or operations to evaluate), the session identifies the minimal set of dependent operations and executes them to compute the requested outputs.

To support model development and debugging, TensorFlow includes TensorBoard, a visualization tool to inspect computational graphs, monitor training metrics, and track parameter values.

With the basics of TensorFlow in place, the next step is to focus on **Automatic Differentiation**, a central mechanism that enables efficient gradient computations for neural networks training and is equally critical in the formulation of DP models.

4.5 Automatic differentiation

Automatic differentiation (AD) is the algorithmic backbone of gradient-based optimization in machine learning frameworks. Beyond deep learning, it also plays a crucial role in molecular simulations, where quantities such as **forces** can be derived directly from the energy of a DP model via differentiation. This section provides a concise overview of the fundamentals of AD and its two main computational modes: *forward* and *reverse*, mainly based on [58].

Unlike numerical approximation methods such as finite differences, or symbolic differentiation, AD evaluates derivatives by systematically applying the chain rule at each step of the computation. The key insight is that any numerical function can be decomposed into a finite sequence of elementary operations—such as addition, multiplication, and standard mathematical functions—whose derivatives are known. These operations form a *computational graph*, through which both values and derivatives can be propagated [59].

As an illustration, consider a scalar function g of a single independent variable x:

$$g(x) = \sin(x) \cdot x^2. \tag{4.17}$$

This function can be decomposed into a sequence of intermediate operations:

$$\begin{aligned} f^{(1)}(x) &= x \\ f^{(2)}(f^{(1)}) &= \sin(f^{(1)}) \\ f^{(3)}(f^{(1)}) &= f^{(1)} \cdot f^{(1)} \\ f^{(4)}(f^{(2)}, f^{(3)}) &= f^{(2)} \cdot f^{(3)} = g(x). \end{aligned} \tag{4.18}$$

When evaluating g at a specific point, intermediate values are computed as in eq. (4.18), and their corresponding derivatives can also be obtained:

$$\begin{aligned} \dot{f^{(1)}}(x) &= 1 \\ \dot{f^{(2)}}(f^{(1)}) &= \cos(f^{(1)}) \cdot \dot{f^{(1)}} \\ \dot{f^{(3)}}(f^{(1)}) &= 2 \cdot f^{(1)} \cdot \dot{f^{(1)}} \\ \dot{f^{(4)}}(f^{(2)}, f^{(3)}) &= \dot{f^{(2)}} \cdot f^{(3)} + f^{(2)} \cdot \dot{f^{(3)}} = \frac{\partial g(x)}{\partial x}. \end{aligned} \tag{4.19}$$

This example generalizes naturally. For a function $g : \mathbb{R}^n \to \mathbb{R}^m$, its Jacobian matrix $\mathbf{J} = Dg$ contains all partial derivatives of the outputs with respect to the inputs:

$$\mathbf{J}_{ij} = \frac{\partial g_i}{\partial x_j}. \tag{4.20}$$

Each column of the Jacobian corresponds to the derivatives of the outputs with respect to a single input variable. If g is the composition of L functions f,

$$g = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)}, \tag{4.21}$$

then, by the chain rule, the Jacobian satisfies:

$$Dg = \mathbf{J}^{(L)} \cdot \mathbf{J}^{(L-1)} \cdot \cdots \cdot \mathbf{J}^{(1)}, \tag{4.22}$$

where $\mathbf{J}^{(k)}$ denotes the Jacobian of the k-th function in the sequence, with entries

$$\mathbf{J}^{(k)}_{ij} = \frac{\partial f^{(k)}_i(y_{k-1})}{\partial x_j}, \quad \text{where } y_k = f^{(k)}(y_{k-1}),\ y_0 = x. \tag{4.23}$$

Forward-Mode AD In forward-mode, a vector $\mathbf{u} \in \mathbb{R}^n$ is chosen in the input space. A single application, or *sweep*, propagates the Jacobian–vector product:

$$\mathbf{J} \cdot \mathbf{u} = \mathbf{J}^{(L)} \cdot \mathbf{J}^{(L-1)} \cdot \cdots \cdot \mathbf{J}^{(1)} \cdot \mathbf{u}, \tag{4.24}$$

where each $\mathbf{J}^{(k)} = Df^{(k)}(y_{k-1})$ is the Jacobian of the k^{th} operation as in eq. (4.23). This product can be written recursively in terms of intermediate vectors:

$$\begin{aligned} &\mathbf{u}^{(1)} = \mathbf{J}^{(1)} \cdot \mathbf{u}, \\ &\mathbf{u}^{(k)} = \mathbf{J}^{(k)} \cdot \mathbf{u}^{(k-1)}, \quad k = 2, \ldots, L, \end{aligned} \tag{4.25}$$

so that $\mathbf{u}^{(L)} = \mathbf{J} \cdot \mathbf{u}$ is obtained alongside the forward evaluation of $f(x)$.

Each choice of $\mathbf{u}$ corresponds to probing the derivative along a particular input direction. For example, setting $\mathbf{u} = e_j$, the j^{th} standard basis vector, isolates the j^{th} column of the Jacobian. Consequently, computing the full $m \times n$ Jacobian requires n forward sweeps—one for each input dimension. Forward mode thus naturally computes directional derivatives in tandem with the forward trace of the function.

In terms of computational cost, when $n \leq m$, forward-mode is efficient: its complexity is linear in the complexity of f. Using fused-multiply-add operations (OPS) as a metric, it has been shown that [60]:

$$OPS(f(x), \mathbf{J} \cdot \mathbf{u}(x)) \leq 2.5 \cdot OPS(f(x)). \tag{4.26}$$

Reverse-Mode AD In reverse-mode, a vector $\overline{\mathbf{w}} \in \mathbb{R}^m$ is chosen in the output space. A sweep computes the action of the transpose of the Jacobian on $\overline{\mathbf{w}}$, i.e. $\mathbf{J}^\top \cdot \overline{\mathbf{w}}$. Like forward-mode, this process is decomposed into elementary operations.

The central objects here are the *adjoints*. The adjoint of a variable x with respect to another variable z is defined as:

$$\overline{x} = \frac{\partial z}{\partial x}. \tag{4.27}$$

To compute adjoints of all inputs with respect to the k^{th} output, $\overline{\mathbf{w}}$ is initialized with zeros everywhere except for a one at the k^{th} position. Thus, the full Jacobian can be recovered in m reverse sweeps. Reverse-mode requires a forward evaluation (function values only), where partial numerical values are stored, followed by a backward pass that propagates adjoints from outputs to inputs.

In terms of complexity, reverse-mode is also linear in the complexity of f, but it is more efficient than forward-mode when $n > m$, and particularly advantageous when $n \gg m$, i.e., when the number of inputs greatly exceeds the outputs. Specifically, [60] showed:

$$OPS(f(x), \mathbf{J}^\top \cdot \overline{\mathbf{w}}(x)) \leq 4 \cdot OPS(f(x)). \tag{4.28}$$

In both forward- and reverse-mode, the decomposition of the target function in eq. (4.21) can be strategically chosen to enable more efficient implementations of automatic differentiation. Naïve implementations may otherwise result in inefficient code.

4.6 DeePMD-Kit TensorFlow Implementation

With this basic architecture in mind, this section shows a closer look at the DeeMD-kit Python API, which demonstrates how the discussed components come together in actual energy calculations.

To understand how DeePMD-kit computes the energy for a given atomic configuration, we used a model trained by [61], a simulation was run using its Python API. This model focuses on copper surface systems in the 500–700 K temperature range, slightly above the Hüttig temperature for Cu (447 K), where atoms should have enough energy to

Figure 4.1. Energy calculation using DeePMD-kit Python API

```python
import numpy as np
from deepmd.infer import DeepPot
a0 = 3.615 # Cu lattice constant (Å)
positions, box = get_configuration(a0) # FCC box
species = np.array([0] * len(positions)) # Single species
model_path = "Cu_model.pb" # Frozen model file
dp = DeepPot(model_path) # Instance of DeePMD library
positions = positions.reshape([1, -1])
box = box.reshape([1, -1])
# Get energy, forces and virials from the model
e, f, v = dp.eval(positions, box, species)
print("Total␣Energy:␣", e) # Printing energy computed
```

diffuse along the surface, defined empirically as 30% the melting temperature [62]. This required sufficient accuracy in the treatment of Cu particles.

Cioni et al. [61] trained their model for a single atomic species (Cu), thereby avoiding complications associated with handling multiple species. The model was validated against DFT results and experimental data for both bulk and surface properties of copper. The *fitting* neural network employed a topology of 1600-240-240-1, based on a ResNet architecture with skip connections. Hyperbolic tangent functions were used as activation functions in all hidden layers, while the identity function was applied in the output layer. In contrast, the *filter* network followed a simpler 1-25-50-100 topology without ResNet architectures or skip connections, again using hyperbolic tangent activations throughout, which also defined the M parameter of the model in 100. The cutoff radius was set to 6.0, Å, with a smooth switching function applied beginning at 0.5, Å. Finally, the maximum number of neighbors expected within the cutoff (N_c) and the number of columns to truncate the embedding matrix ($M_<$) were fixed at 100 and 16, respectively.

For testing we used a face-centered cubic (FCC) Cu structure with 256 atoms and box length of 14.46 Å (interatomic distance of 1.8075 Å along each axis). The resulting total energy was approximately $-427,527.79$ eV. The Python snippet used is shown in fig. 4.1.

Validating the DeePMD GitHub repository [63], it was confirmed that the Python function *DeepPot.eval* in fig. 4.1 follows a structured evaluation procedure. The function first prepares the fetch list, which includes the Cartesian coordinates, box coordinates, and species of the system. It then launches a TensorFlow session to load the frozen model and evaluates the model using the TensorFlow computational graph. Finally, the outputs are converted back to physical quantities, namely the energy, forces, and virials.

To further examine the internal mechanics, I explored the computational graph of the model evaluation using TensorBoard. As summarized in fig. 4.2, the process begins with the **ProdEnvMatA** operation, which computes pairwise distances and applies the switch function $s(r)$ to generate coordinate matrices $\mathcal{R}$ (eq. (4.9)). The resulting data are then passed to the **Filter NN**, which evaluates the descriptor network $\mathcal{N}_D$ to obtain $\mathcal{G}$ (eq. (4.10)). From here, the **Descriptors** step constructs $\mathcal{D}^i$ from the combined contri-

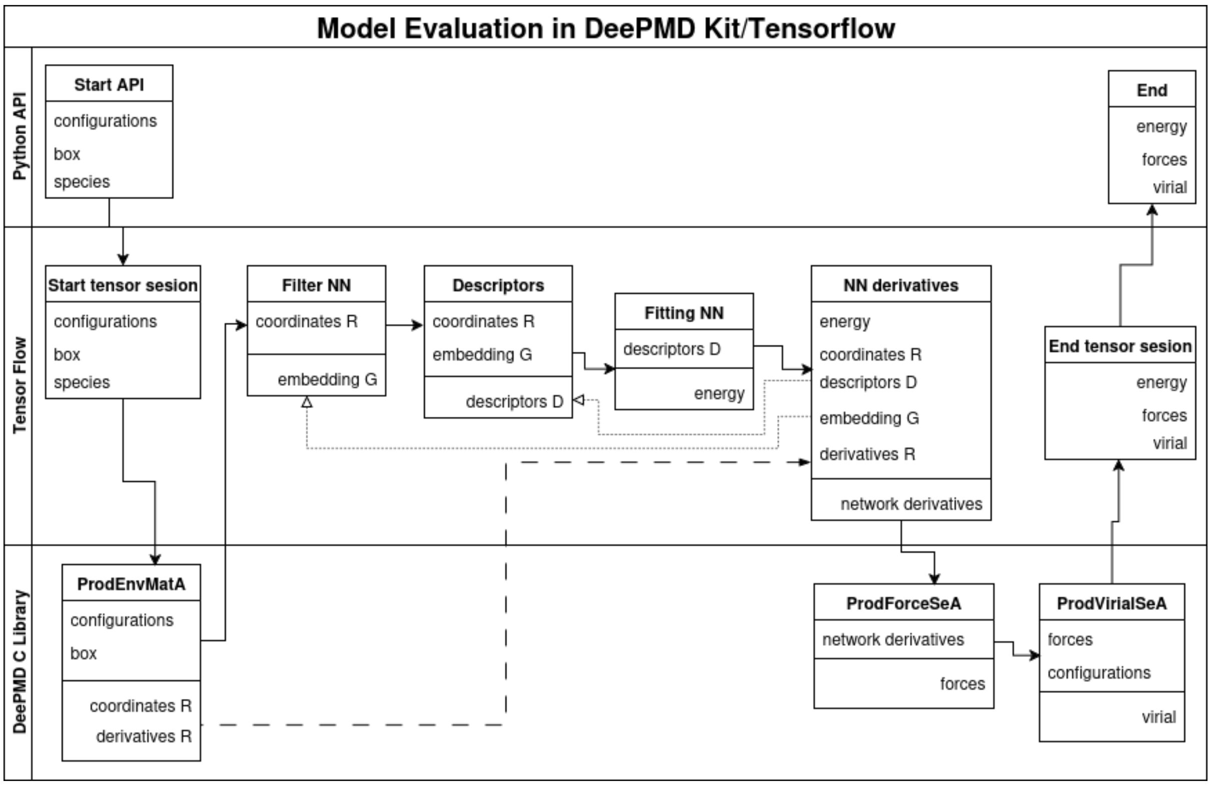

Figure 4.2. Flowchart of the model evaluation process in DeePMD-kit, derived from the TensorBoard graph of the implementation described in [61]. The diagram is organized into three horizontal sections corresponding to the DeePMD-kit Python API, the TensorFlow runtime, and the DeePMD-kit C++ library with its custom TensorFlow operations. Each operation is shown with its associated input (left-aligned) and output (right-aligned) tensors, in the section responsible of its execution. Solid arrows indicate tightly coupled operations that occur in sequence, or close in proximity, dashed arrows denote dependencies across non-adjacent steps, and dotted arrows represent the flow of gradients generated through TensorFlow's automatic differentiation.

butions of $\mathcal{R}^i$ and $\mathcal{G}^i$ (eq. (4.11)). These descriptors are subsequently fed into the **Fitting NN**, which applies $\mathcal{N}_F$ to compute atomic energies (eq. (4.1)). TensorFlow's automatic differentiation framework then generates the required gradients in the step labeled **NN Derivatives**. The gradients are used by **ProdForceSeA** to compute the atomic forces (eq. (4.12)), and finally by **ProdVirialSeA**, which calculates the virials from the atomic positions and forces.

Chapter 5 will focus on reproduce those results appart from the DeeMD-kit package and TensorFlow core functions.

Chapter 5

Implementation

Once the DeePMD implementation was reverse-engineered via TensorBoard, the next step was to replicate the energy computation without relying on DeePMD or TensorFlow functions. To this end, a Python prototype was developed, following the procedure discussed earlier (see section 4.6). Using TensorFlow libraries, the trained model parameters were extracted. Analysis of the model revealed that it employed the *two-body embedding DeepPot-SE* descriptor. Consequently, this was the only descriptor included in the implementation of the present work, and the preceding discussion focuses exclusively on it.

5.1 Python prototype

The prototype included the computation of normalized distances and embedding matrices, as described for the specific descriptor (see section 4.2):

$$\mathcal{G}^i_n = \mathcal{N}_D\Big(s(r_{in})\Big), \qquad \mathcal{G}^i = \Big(\mathcal{N}_D(s(r_{in}))\Big)_n, \qquad \mathcal{G}^i = \mathcal{N}_D\Big((s(r_{in}))_n\Big). \tag{5.1}$$

NumPy was used to efficiently handle tensors via its broadcasting property, which allows operations between arrays of different shapes to be applied element-wise without explicit loops [64]. The embedding matrices $\mathcal{G}^i$ for each particle were computed simultaneously for all n neighbors, by applying the linear combinations of the filter network $\mathcal{N}_D$ to a vector of scalars obtained from the switch function applied to the distances between each central particle i and its neighbors.

An energy calculation was performed using the same configuration as in the API test. With the cutoff defined in the model (see section 4.6), each particle had a neighbor list of 78 atoms holding the condition fixed in the model.

Algorithm 1 shows the pseudocode for the prototype, providing a high-level overview of the energy calculation. Some important details are summarized below:

- the model contained two groups of trained parameters: one for the filter network ($\boldsymbol{\theta}_d$) and one for the fitting network ($\boldsymbol{\theta}_f$), as it was trained for a single atomic species.

- Distances r_{pn} were computed using the minimum image convention to account for periodic boundary conditions.
- The subscript n in $\mathcal{R}_n$ and $\mathcal{G}_n$ refers to specific rows in the corresponding matrices.

Algorithm 1: Energy calculation

```
Input: Frozen model $F$, Particles $S$
Output: Energy $\epsilon$
$\boldsymbol{\theta}_d \leftarrow$ parameters of filter NN from $F$
$\boldsymbol{\theta}_f \leftarrow$ parameters of fitting NN from $F$
foreach $p \in S$ do
    $N(p) \leftarrow$ list of $N_c$ neighbours inside cutoff of $p$
    foreach $n \in N(p)$ do
        $r_{pn} \leftarrow$ pair distance from $p$ to $n$
        Coordinate matrix $\mathcal{R}_n \leftarrow$ coordinates composition for $r_{pn}$
        Embedding matrix $\mathcal{G}_n \leftarrow \mathcal{N}_D(s(r_{pn}), \boldsymbol{\theta}_d)$
    end
    Descriptor $\mathcal{D} \leftarrow N_c^{-2} \cdot \mathcal{G}^\top \cdot \mathcal{R} \cdot \mathcal{R}^\top \cdot \mathcal{G}_<$
    Individual energy $\epsilon_p \leftarrow \mathcal{N}_F(\mathcal{D}, \boldsymbol{\theta}_f)$
end
Total energy $\epsilon \leftarrow \sum_{p \in S} \epsilon_p$
return $\epsilon$
```

The prototype achieved a relative error in the total energy of 6.74×10^{-4} eV compared to DeePMD-Kit. Having validated the method through the Python prototype, attention was then directed toward embedding the approach into HALMD, to enable efficient large-scale simulations with CPU and GPU support.

5.2 HALMD Implementation

Since the prototype showed good agreement with DeePMD-Kit energy computations, the next step was to implement the method within the **HALMD project**. The primary requirement was to access the trained data stored in the frozen model. Once this was achieved, modules could start to be developed to perform the computations on both CPU and GPU.

HALMD is interfaced by Lua, a language for extending applications that combines procedural features with data description facilities designed to be used as a general purpose extension language, allowing to extend applications with user defined functions based on primitives provided by main applications, who provide the basic classes and objects of the system without making them to deal with configuration decisions taken during execution time [65]. In that sense, along with the main components developed in *C++*, a lua module was also included.

Figure 5.1 shows the proposed class design for HALMD, which integrates many-body force potentials. An independent module reads the trained neural network data to be used by the *deepmd* and *embedded_neural_network* classes. Two additional interfaces,

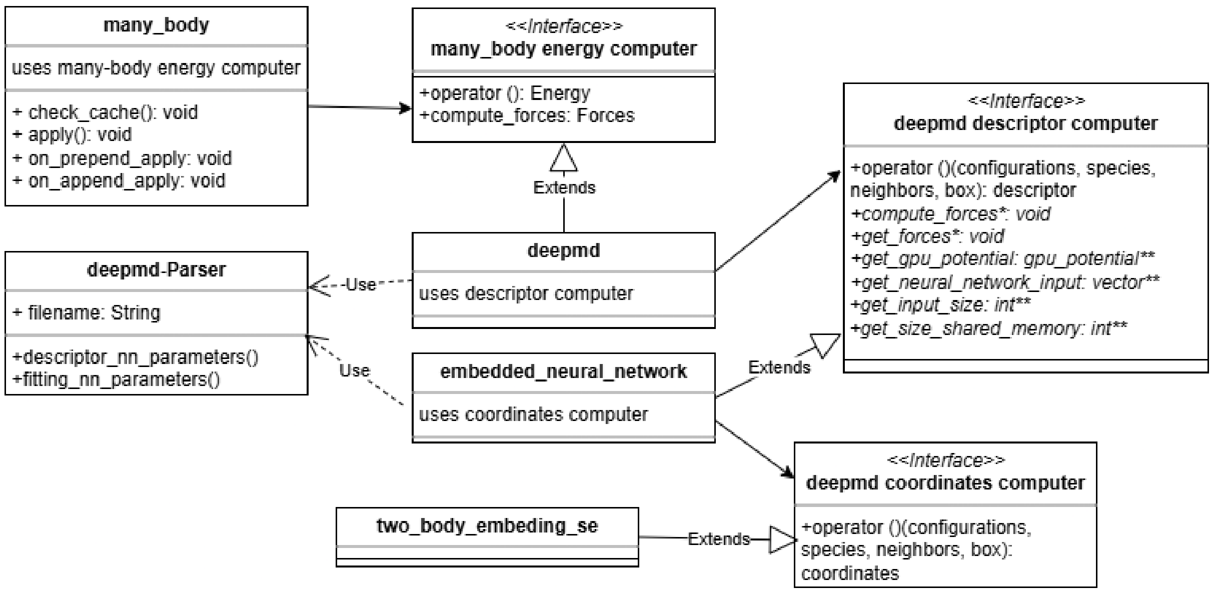
many_body
uses many-body energy computer
+ check_cache(): void
+ apply(): void
+ on_prepend_apply: void
+ on_append_apply: void
<<Interface>>
many_body energy computer
+operator (): Energy
+compute_forces: Forces
Extends
<<Interface>>
deepmd descriptor computer
+operator ()(configurations, species, neighbors, box): descriptor
+compute_forces*: void
+get_forces*: void
+get_gpu_potential: gpu_potential**
+get_neural_network_input: vector**
+get_input_size: int**
+get_size_shared_memory: int**
deepmd-Parser
+ filename: String
+descriptor_nn_parameters()
+fitting_nn_parameters()
Use
deepmd
uses descriptor computer
Use
embedded_neural_network
uses coordinates computer
Extends
<<Interface>>
deepmd coordinates computer
+operator ()(configurations, species, neighbors, box): coordinates
two_body_embeding_se
Extends

Figure 5.1. Class model of the proposed integration of the many-body DeePM potential into HALMD. As HALMD is structured into modular components [66], the new potential is incorporated within the module responsible for many-body interactions, allowing it to be used via a *many-body-force* object during simulations. The same interface-based design was extended to the coordinate and descriptor coordinate components required by DeePMD, facilitating the future integration of alternative implementations. A * symbol after a method name indicates that it is required only for the CPU version, while ** denotes methods required exclusively for the GPU version.

deepmd descriptor computer and *deepmd coordinates computer*, were proposed to provide flexibility when using the *deepmd* many-body energy computer.

To streamline access to trained DeePMD models, a new class was added to HALMD's *input/output* module. This class retrieves information from the model's HDF5 file and makes it available to other components. It is accessible through the lua module *halmd.io.readers.deepmd*. For compatibility, the HDF5 file must follow the structure shown in Listing 5.1.

Listing 5.1. Structure of the DeePMD model HDF5 file

```
general
|- rcut (double): cutoff radius
|- rcut_smth (double): smoothing cutoff radius
|- axis_neuron (int): number of neurons in the axis column
|- species (int): number of atomic species
|- species_info
   |- <i> (group for species i, where i = 0 to species-1)
      |- descriptors
      |  |- general
      |  |  |- sel (int): max number of neighbors considered
      |  |  |- layers (vector<int>): number of layers
      |     |- <j> (group for layer j)
      |        |- neurons (int): number of neurons in this layer
      |        |- activation_function (int): activation (ASCII)
      |        |- weights (vector<double>): COLUMN-major order
      |        |- bias (vector<double>): bias for each neuron
      |        |- timesteps (vector<double>): ResNet Timestep
      |        |- residual_net (bool, optional): is ResNet layer
      |- main
         |- layers (int): number of layers in fitting NN
         |- data
            |- <j> (group for layer j)
               |- neurons (int)
               |- activation_function (int)
               |- weights (vector<double>)
               |- bias (vector<double>)
               |- timesteps (vector<double>, optional)
               |- residual_net (bool, optional)
```

Additionally, a Python module was created to extract the trained data from DeePMD models and generate the HDF5 file required by HALMD. This tool relies on the same TensorFlow utilities used during prototype construction.

With the general structure defined and the core module to allow access to the trained data, it was possible to turn to the sequential implementation for simulations with CPU resources.

5.3 CPU implementation

The existing CPU module in HALMD for evaluating neural networks computes derivatives with respect to the inputs during forward evaluation, following the same idea described for forward-mode automatic differentiation (see section 4.5). Building on this principle, the modules implemented here return both the expected values and their derivatives. Next, they are briefly described:

- **coordinates computer**: responsible for constructing the coordinate matrix $\mathcal{R}$ for each particle, given the atomic configuration, box dimensions, and maximum number of neighbors per species.

  In this implementation, a Lua module *mdsim.potentials.many_body.two_body_embe ding_se* was developed. It produces:

  1. the coordinate matrix $\mathcal{R}$ as defined in eq. (4.9),
  2. a neighbor map to identify the row in $\mathcal{R}$ associated with each neighbor, and
  3. the derivatives of $\mathcal{R}$ with respect to the position of the central particle i along each axis, according with eq. (4.16).

  item **Descriptor computer**: responsible for computing the descriptors $\mathcal{D}$ for each particle, using the atomic configuration, box limits, neighbor list, and a coordinates computer.

  For this task, the Lua module *mdsim.potentials.many_body.embedded_neural_network* was implemented. It evaluates the embedding matrix $\mathcal{G}$ as in eq. (4.10), constructs the descriptors as defined in eq. (4.11), and stores in memory both the descriptors and their derivatives with respect to the central particle (eq. (4.13)) and its neighbors (eq. (4.14)).

- **DeepMD**: responsible for computing the energies for each particle, using the atomic configuration, box limits, neighbor list, a descriptor computer and a reader.

  In this implementation, a Lua module mdsim.potentials.many_body.deepmd was created. It evaluates the fitting network $\mathcal{N}_F$ as in eq. (4.1) and once the energy is computed, uses the derivatives of the network with respect to the descriptor to finish the computation of the forces, calling a function *compute_forces*, of the descriptor computer, to store temporarily the individual forces until all energies are computed to obtain the total forces in the system, as in eq. (4.12).

Once the sequential implementation was developed, the focus shifted to the parallel implementation, in order to be able to use GPU resources in the energies computation.

GPU implementation

In the parallel implementation using GPU resources, the same module structure as in the sequential version was adopted. The key difference is that most computations are no longer performed particle by particle but instead in parallel across all particles simultaneously, thanks to the CUDA architecture

As it was seen in section 2.4, one of the main bottlenecks that can be found in a GPU implementation, are related with the process of transfering information from host memory to global memory [40]. In this sense, most of the objects created work transfering pointers to data that is already charged in global memory. Here some other specific characteristics of each object are described:

- **coordinates computer**: responsible for constructing all coordinate matrices $\mathcal{R}$ from given configurations, species and box. For this implementation, the object *two_body_embeding_se* was created. It was divided in three individual kernels, one to compute the pairwise distances, other to order those distances and a final one to organize the respective matrices.

  compute_distances: on each thread, computes the pair distances between a different central particle and all its neighbors and stores those pair distances in global memory as well as the component-wise distances (in another space in global memory) and a neighbouring map in order the next kernel orders the euclidean distances paired with the respective neighbor. Currently this kernel works only in global memory.

  radix_direct_sort_list: orders neighbors lists using a separate block for a different central particle as was described in chapter 3.

  compute_coordinate_matrix: scales the component-wise distances by the relation between the switch function and the distance, and creates the $\mathcal{R}$ matrices as referred in eq. (4.9). Each matrix will have their own portion in a vector global memory, ordered in row-major order, meaning that information of each row of each matrix is aside. Currently all work is divided evenly in all threads launched, then each thread will compute one or more rows of one or more matrices, using only global memory.

Finally, it returns a pointer to $\mathcal{R}$ matrices in global memory.

- **Descriptor computer**: responsible for computing all descriptors $\mathcal{D}$, from given configurations, species, box and a coordinates computer. For this implementation the object *embedded_neural_network* was created. In its constructor, it transfers to global memory the parameters of neural networks taken from the parser module. Once the $\mathcal{R}$ matrices are computed, it is divided in two sections: the construction of the $\mathcal{G}$ matrices and the construction of the actual descriptors.

  compute_filter_neural_networks: takes the values of the first column of all $\mathcal{R}$ matrices and reads the filter networks $\mathcal{N}_D$ to obtain the rows of all $\mathcal{G}$ matrices. It launches a block for each different row of the $\mathcal{G}$ matrices, reads the neural network for each value in shared memory and then registers in global memory the results, creating a $\mathcal{G}$ matrix for each particle in column-major order, as it would help the next step. It requires shared memory space equal to two times the largest layer in the network and if it uses ResNet architecture, another chunk of the same size.

  Descriptors $\mathcal{D}$ construction: constructs individual descriptors using *cublasSgemm* function to do the matrices multiplications described in eq. (4.11). In this step, descriptors are constructed sequentially through individual calls to the *cublasSgemm* function, using the respective information in global memory and registering the results also in global memory in the specific section of each particle.

  Finally, it returns a pointer to the descriptors $\mathcal{D}$ in global memory.

- **DeepMD**: responsible for preparing for the computation of energies through neural networks, from given configurations, species, box and a descriptors computer. In its constructor loads to global memory the parameters of neural networks $\mathcal{N}_F$ taken from the parser module. Next are functions needed from the GPU many-body module, when it tries to compute energies:

  get_gpu_potential: returns the information of the neural networks in global memory.

  get_neural_network_input: returns all inputs to read the neural networs. In this case the descriptors $\mathcal{D}$.

  get_input_size: returns the size of each descriptor: $M \times M_<$.

  get_size_shared_memory: returns the space needed to evaluate the neural networks. In this case 2 times the descriptor size and in case it uses ResNet architecture, an additional chunk of the same size.

Table 5.1 summarizes the register count and memory requirements of the principal CUDA kernels used in this implementation. This overview highlights the resources involved in the construction process and provides context for the computational load of each stage.

Table 5.1. Summary of resource usage per CUDA kernel. Register counts and memory usage are reported from the compiler flag `-Xptxas -v` when available. Constant memory (`cmem`) typically stores kernel arguments, while shared memory (`smem`) refers to statically allocated block-local memory. Global memory sizes are expressed in terms of the number of memory positions of type `float`, unless otherwise indicated. These sizes refer to memory usage per particle, except when explicitly marked as Total, which denotes the memory required for the entire simulation. The variables D, N_c, M, and $M_<$ correspond to the number of spatial dimensions, neighbors, filter network output size, and the number of columns used in the descriptors, respectively, as defined in Section 4.2. The symbol H_{Nc} refers to the HALMD neighbor list size. Entries marked with "–" indicate that the usage could not be determined statically at compile time.

Kernel	**Regs.**	**Const. Mem.**	**Shared Mem.**	**Input Global Memory**	**Output Global Mem.**
compute_ distances	32	384 b	0	H_{Nc}	H_{Nc}(int) + $D \cdot (H_{Nc}+1)$
radix_ direct_ sort_ list	30	364 b	4640 b	H_{Nc} + H_{Nc}(int)	H_{Nc} + H_{Nc}(int)
compute_ coordinate_ matrix	19	364 b	0	H_{Nc} + H_{Nc}(int)	$N_c \cdot (D+1)$
compute_ filter_ neural_ networks	–	–	$2M$	$N_c \cdot (D+1)$	$N_c \cdot M$
compute_ filter_ neural_ networks (ResNet)	–	–	$3M$	$N_c \cdot (D+1)$	$N_c \cdot M$
Descriptors construction $\mathcal{G}^\top \mathcal{R}$	–	–	–	$N_c \cdot (M+D+1)$	Total: $M \cdot (D+1)$
Descriptors construction $\mathcal{R}^\top \mathcal{G}_<$	–	–	–	$N_c \cdot (D+1+M_<)$	Total: $M_< \cdot (D+1)$
Descriptors construction $\mathcal{D}$	–	–	–	Total: $M \cdot M_< \cdot (D+1)$	$M \cdot M_<$
compute_ fitting_ neural_ networks	–	–	$2M \cdot M_<$	$M \cdot M_<$	1
compute_ fitting_ neural_ networks (ResNet)	–	–	$3M \cdot M_<$	$M \cdot M_<$	1

Having established the details of the kernel design and resource usage, chapter 6 will present an evaluation of the implementation.

Chapter 6

Results

In order to verify the results of the implementation, a series of random configurations for different numbers of particles and box sizes were generated, and their respective energies were computed using the model mentioned previously, trained by [61]. These energy predictions were performed with the DeePMD-kit Python API and both CPU and GPU implementations in HALMD.

All simulations were performed on a workstation running *Debian GNU/Linux 12 (Bookworm)* with Linux kernel version *6.1.0-39-amd64*. The system is equipped with an *Intel® Core^{TM} i7-14700K* processor (20 physical cores / 28 threads, base frequency up to 5.6 GHz) and 31 GiB of RAM. For GPU-accelerated executions, an *NVIDIA GeForce RTX 4070* graphics card with 12 GiB of dedicated memory was used, running under driver version *535.247.01* with *CUDA* toolkit version *11.2*. The code was compiled with the *GNU Compiler Collection* (*gcc/g++* 12.2.0). The DeePMD-kit simulations were carried out using *DeePMD-kit* v*3.1.0* with a *TensorFlow* backend v*2.14.1*.

Section 6.1 describes the performance of the CPU implementation, then the accuracy of the potential energy per particle resulting from the HALMD simulations compared with the predictions obtained from the API is validated. Finally, results and performance of the GPU implementation in HALMD are reviewed.

6.1 CPU Implementation

In order to verify the CPU implementation of the model in HALMD, a simulation of 32 particles was executed, inside a cubic box of 13Å with periodic boundary conditions. In this system particles were randomly distributed inside the box. The minimum distance was ~ 0.3907Å, the maximum ~ 10.5486Å, and the mean ~ 6.0227Å± 1.9139Å consistent with the distances and cutoff defined by the model.

This simulation ran for 3.271 timesteps, taking about 24 hours in total. Each step requiered 23.758 ± 0.429 seconds.

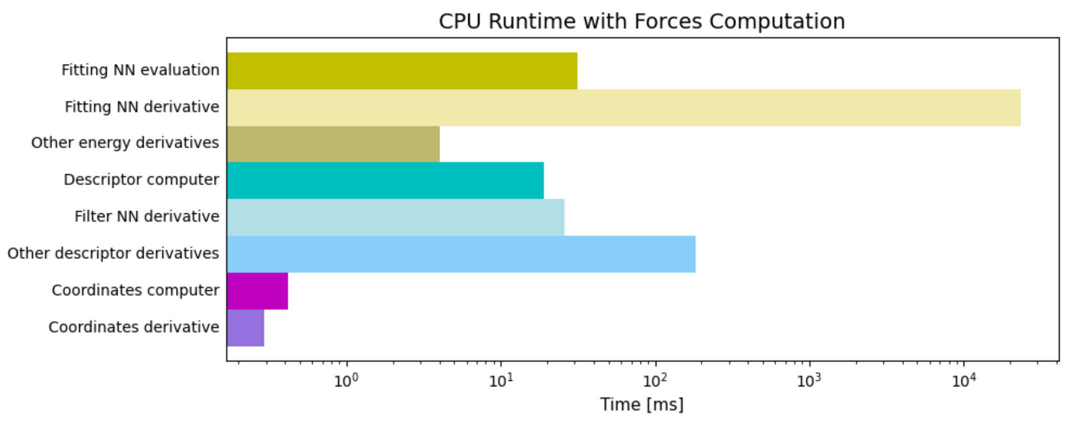

Figure 6.1. Log-scaled runtime breakdown of program sections. The chart displays the execution time of three main computational steps — Coordinates computation, Descriptors computation, and Energy computation — each subdivided into their primary calculations and associated derivatives. Due to the dominance of neural network derivative computations, time (y-axis) is presented on a logarithmic scale to highlight the relative contributions of smaller components. For Descriptors and Energy, derivative times are further separated into those arising from neural network evaluations and other derivative operations.

It was observed that the computation of derivatives in each section completely dominated the execution time. Figure 6.1 shows the runtime breakdown of the three main sections of the implementation. The most expensive operation was the computation of derivatives of the fitting network ($\mathcal{N}_F$), representing about 99% of the total runtime.

In general, the computation of derivatives dominated all segments. This reveals an important issue in the implementation and emphasizes the need for an efficient way to compute these gradients. Table 6.1 summarizes the percentage contributions of each section in the total runtime.

Section	Contribution (%)
Energy computer	**98.9016**
Fitting NN	0.1318
Fitting derivative	98.8848
Other derivatives	0.0168
Descriptor computer	**0.9636**
Main computations	0.0801
Filter derivative	0.1087
Other derivatives	0.7748
Coordinates computer	**0.003**
Main computations	0.0018
Derivative	0.0012

Table 6.1. Percentage contribution of each computation step to the total runtime for a CPU simulation of 32 particles in a cubic box of 13Å, with both energies and forces computed using HALMD.

Since the main goal of this study is to verify the impact of parallel evaluation of the model, and the current implementation only includes the energy computation, force computations and all derivatives were disabled to allow energy predicting and obtaining potential energies for comparison.

As computations related to forces represented about 99.79% of the runtime, disabling them allowed simulations of much larger systems, making it possible to verify scaling behavior. Seventy-six configurations were each evaluated five times to predict energies and analyze the runtime behavior of the CPU implementation. These systems ranged from 284 particles in a box of ~ 20.048Å to 149,132 particles in a box of ~ 161.742, Å aiming to simulate many independent atomic environments.

Figure 6.2 shows the average runtime per system with error bars indicating variation. Although the Coordinate step is the cheapest, its cost increased faster than linearly. Further analysis revealed that its main operations contributed little to the runtime; most of the cost came from matrix initializations, suggesting an implementation issue.

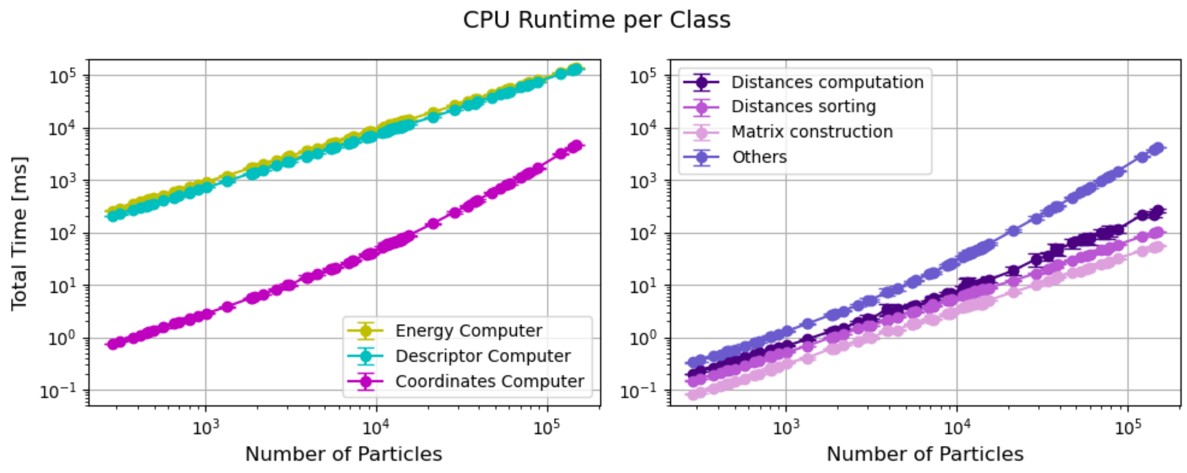

Figure 6.2. Average runtime per system in the CPU implementation. Each system was evaluated five times, and results were averaged. Left: comparison of the three main computational steps. Right: breakdown of the Coordinate step, which exhibits a non-linear growth due to additional operations not directly related to the construction of the $\mathcal{R}$ matrix.

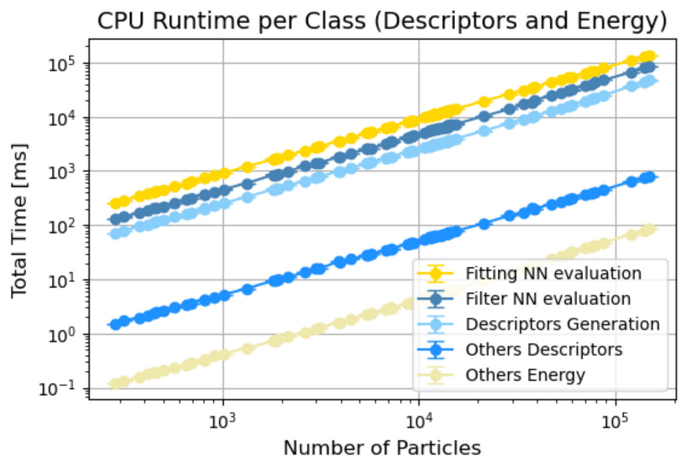

Figure 6.3. Average runtime of descriptor and energy components in the CPU implementation. Results are averaged per system, with deviations shown as error bars.

In contrast, the Descriptor computation and Energy prediction steps showed nearly linear scaling, but both remained the most expensive operations in similar proportions (see fig. 6.2). Figure 6.3 details the main segments of each. The most costly operation was the evaluation of the fitting networks (eq. (4.1)), probably related to the number of operations required. Without activation functions, the model requires 883,680 FLOPs per particle evaluated.

Finally, for the Descriptor computation, the main operations—evaluating the filter networks to construct the $\mathcal{G}$ matrices (eq. (4.10)) and computing the $\mathcal{D}$ descriptors (eq. (4.11))—were the main contributors to runtime. The former required 12,550 FLOPs plus activation functions per neighbor inside the cutoff radius, i.e. up to 1,255,000 FLOPs per particle. Descriptor construction required 103,536 FLOPs per particle. This number of operations explains the behavior of the program.

While the runtime profiling clarified how computations are distributed in the CPU version, it is equally important to confirm that the implementation preserves numerical accuracy. To this end, section 6.2 evaluates the potential energy deviation between HALMD and DeePMD-kit.

6.2 Accuracy of potential energy

The same configurations described previously were evaluated with the DeePMD-kit Python API. However, it ran out of memory after $13,981$ particles in a box of ~ 73.476Å, so only 55 systems were validated against its results. In these systems, the minimum distance between particles averaged ~ 0.149Å $\pm\, 0.066$Å, ranging from ~ 0.099Å in a system of $15,432$ particles to ~ 0.464Å in a system of 313 particles.

On average, the relative error in the potential energy per particle was $\sim 6.088 \times 10^{-4} \pm 1.159 \times 10^{-5}$ in the CPU execution. For the GPU execution, the error was nearly identical: $\sim 6.087 \times 10^{-4} \pm 1.159 \times 10^{-5}$. The minimum relative error of $\sim 5.623 \times 10^{-4}$ was observed in a system of 691 particles in a box of ~ 26.964Å, for both CPU and GPU simulations. Meanwhile, the maximum relative error of $\sim 6.284 \times 10^{-4}$ occurred in a system of 654 particles in a box of ~ 26.474Å.

An increase in the relative error of the potential energy was observed in larger systems, as shown in fig. 6.4. The 55 systems were grouped by size, and the relative errors were averaged within each group. This revealed both a systematic increase in relative error for larger systems and a reduction in error variation. Table 6.2 summarizes these results, showing for each group the systems with the minimum and maximum number of particles, as well as the cases with the lowest and highest relative error.

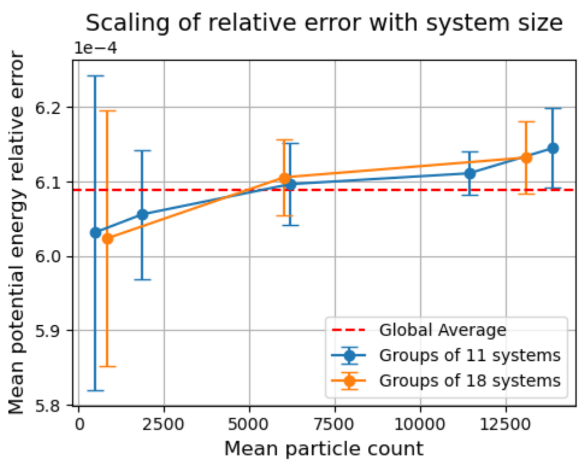

Figure 6.4. Scaling of the mean relative error of the potential energy per particle with system size. Each point shows the average relative error across systems of similar size, with error bars representing the standard deviation. The red dashed line indicates the global mean relative error.

It is likely that these differences are related to normalizations applied in the DeePMD-kit package, which were not fully documented during the evaluation described in chapter 4. For instance, parameters related to the mean and standard deviation of distances appeared in the TensorBoard graph, but their role was not explained in the program or documentation.

Table 6.2. Comparison of potential energy per particle obtained with the DeePMD-Kit Python API and HALMD for the model trained in [61]. Systems are grouped by particle number. For each group, the table shows the smallest and largest system, as well as the cases with minimum and maximum relative error. The lowest relative error in each group is shown in bold. Averages are reported for CPU and GPU executions.

Particles	Box length (Å)	Min. distance (Å)	PE DeePMD	PE HALMD	Rel. Error
First Group					
284	20.048	0.343	-1,668.010	-1,669.017	6.036e-04
654	26.474	0.192	-1,667.971	-1,669.019	6.284e-04
691	26.965	0.251	-1,668.078	-1,669.016	**5.624e-04**
1,923	37.928	0.137	-1,667.995	-1,669.015	6.116e-04
		Average Rel. Error	CPU	6.024e-04 ± 1.718e-5	
			GPU	6.023e-04 ± 1.718e-5	
Second Group					
2,161	39.432	0.166	-1,667.988	-1,669.017	6.170e-04
5,482	53.779	0.102	-1,668.009	-1,669.016	**6.037e-04**
7,306	59.182	0.151	-1,667.975	-1,669.013	6.223e-04
10,758	67.330	0.117	-1,667.997	-1,669.013	6.089e-04
		Average Rel. Error	CPU	6.106e-04 ± 5.034e-6	
			GPU	6.105e-04 ± 5.034e-6	
Third Group					
11,367	68.578	0.135	-1,667.992	-1,669.011	6.107e-04
11,507	68.858	0.150	-1,668.000	-1,669.013	**6.069e-04**
13,962	73.443	0.116	-1,667.967	-1,669.008	6.241e-04
13,981	73.476	0.126	-1,667.975	-1,669.010	6.203e-04
		Average Rel. Error	CPU	6.132e-04 ± 4.838e-6	
			GPU	6.132e-04 ± 4.838e-6	

Having established that the CPU implementation is comparable to the DeePMD-kit, the focus shifts toward performance. In particular, section 6.3 assesses scalability and benchmarks runtime improvements for the GPU implementation.

6.3 CPU vs GPU implementation

In order to evaluate the GPU implementation, each of the seventy-six configurations used in section 6.1 were used to predict energies in a total of 5 times per system. To verify the consistency between the two implementations, the relative error in potential energy was compared. A difference of $\sim 2.136 \times 10^{-8} \pm 6.249 \times 10^{-10}$ was found between CPU and GPU runs, which is consistent with floating-point precision. Figure 6.5 shows how this deviation scales with the number of particles, with results averaged over groups of 14 systems.

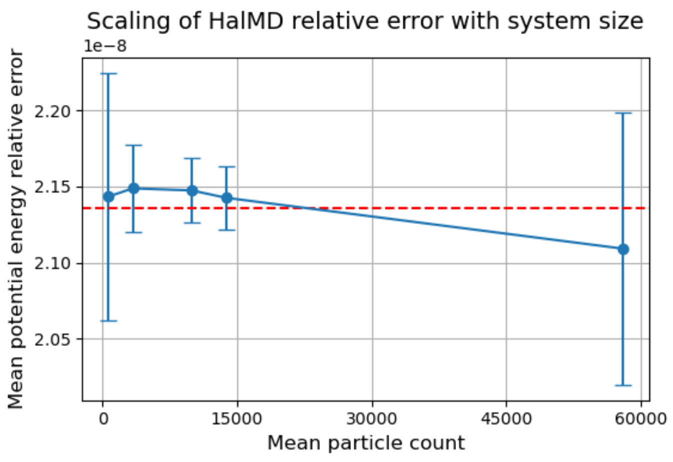

Figure 6.5. Mean relative error in potential energy per particle for CPU and GPU implementations in HALMD, grouped into 14 systems per data point. Error bars show the standard deviation within each group. The dashed red line represents the global average relative error.

As deviations in potential energy were within floating-point limits, the next step was to compare the runtime of both implementations. Figure 6.6 shows the scaling of runtime for potential energy evaluation. As expected, the CPU version requires significantly more time than the GPU version.

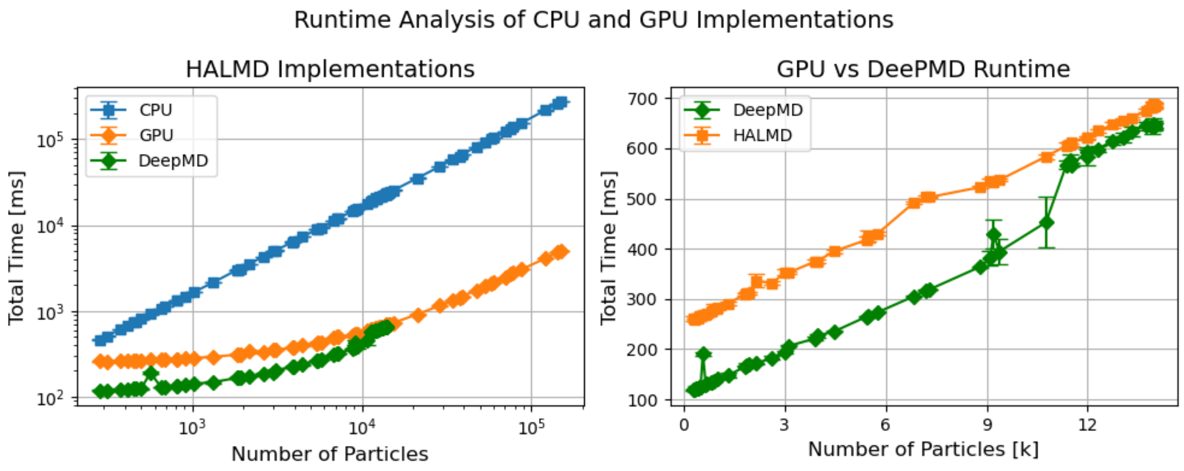

Figure 6.6. Scaling of runtime with system size, comparing CPU and GPU implementations in HALMD. Each data point represents the average of five runs per system, with error bars indicating the observed variation. The left panel presents a comparison between CPU and GPU implementations, alongside results obtained with the DeePMD API where available. The right panel focuses on the GPU implementation and the DeePMD API in linear scale, providing a clearer view of their scaling behavior.

In addition, GPU runtimes were compared with those obtained using the DeePMD-kit Python API, which includes energies, forces, and virials. Although HALMD computes only energies, this comparison provides insight into the competitiveness of the proposed implementation. Figure 6.6 also shows this comparison. A marked increase in DeePMD runtimes was observed for systems with $\sim 10,000$ particles, which can be related with the memory errors later encountered for systems of $\sim 14,000$ particles.

To analyze runtime behavior in more detail, each computation segment of the GPU implementation was measured. Figure 6.7 shows runtime scaling with particle number, broken down into individual components. Descriptor computation clearly dominates the overall runtime. The left graph highlights the Coordinate and Energy modules, showing that Coordinate computations grow faster than linearly. This could be explained by the fact that most kernels in this module access global memory directly, without using shared memory, which limits performance.

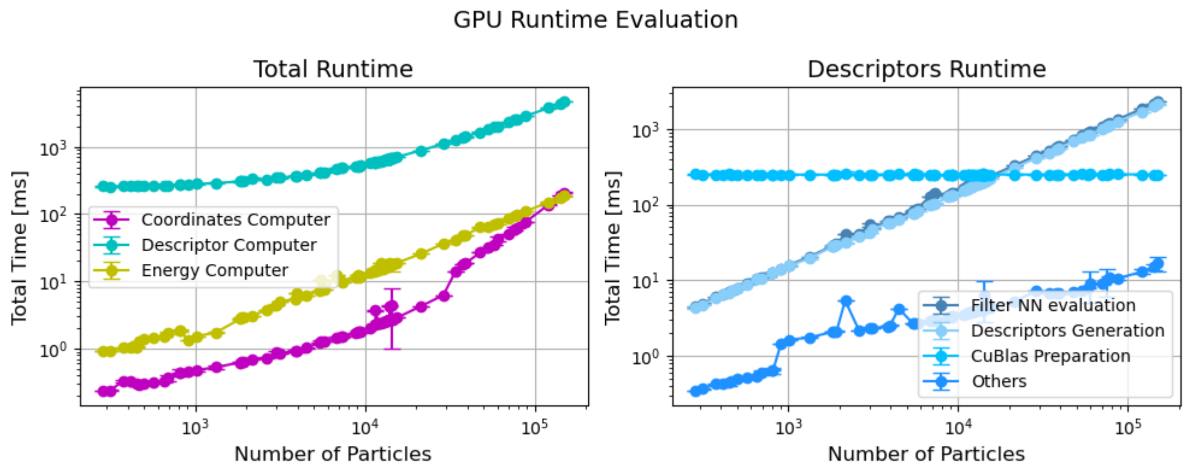

Figure 6.7. GPU runtime breakdown by computation segment in HALMD. Left: runtime of all segments, showing descriptor dominance. Right: Runtime scaling of GPU descriptor components in HALMD.

Within the descriptor computations, the most expensive segments were also profiled. Figure 6.7 also shows that the evaluation of filter networks ($\mathcal{N}_F$) and the descriptor generation (eq. (4.11)) dominate runtime, as expected. For the filter networks, N_c networks must be evaluated per particle, corresponding in this model to 6,275 scalar multiplications and 6,275 scalar additions ($12,550$ FLOPs per network), in addition to activation function evaluations. This explains their cost. The current GPU implementation for neural network evaluation, developed in [66], was already verified to perform efficiently, consistent with the good behavior of the fitting networks. However, in that case only one network per particle was required.

Descriptor generation involves several matrix–matrix multiplications. At present, each multiplication is executed in parallel via cuBLAS, but three such multiplications must be performed sequentially for each descriptor. Moreover, descriptors themselves are generated sequentially, further compounding the cost. A strategy enabling parallel execution of multiple matrix–matrix multiplications, similar to the matrix–vector approach in [66], could improve this part of the code.

Finally, fig. 6.8 provides an overview of the runtime decomposition for a selection of systems, summarizing the trends discussed throughout this section. For smaller systems, the CuBLAS handle segment, which prepares matrix–matrix multiplications on the GPU, dominates the runtime. It also illustrates the scaling of other key components, including the evaluation of the neural networks and the descriptor generation, where the matrix–matrix multiplications are performed. Even with the dominance of the descriptor segment, the runtime of the coordinate computations begins to increase for larger systems, highlighting potential memory management inefficiencies. These observations reinforce the patterns identified earlier and emphasize the importance of pursuing a more efficient strategy for matrix–matrix multiplications. Additionally, a closer examination of the neural network evaluation, particularly the filter networks, may reveal opportunities to further optimize performance and reduce runtime.

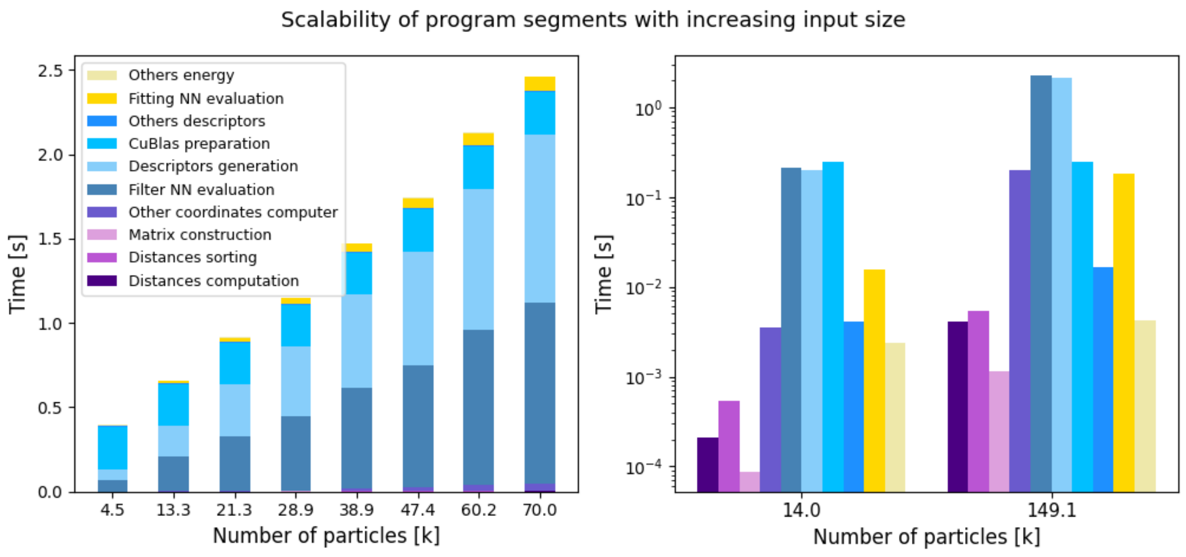

Figure 6.8. Execution time of GPU components in HALMD for different particle counts. The left panel shows how the runtime of each segment scales with increasing system size, while the right panel compares the detailed runtime breakdown of two selected particle counts on a logarithmic scale

Bibliography

[1] H. Bernhard Schlegel. "Exploring potential energy surfaces for chemical reactions: An overview of some practical methods". In: *Journal of Computational Chemistry* 24.12 (2003), pp. 1514–1527. DOI: `https://doi.org/10.1002/jcc.10231`.

[2] Jörg Behler. "Neural network potential-energy surfaces in chemistry: a tool for large-scale simulations". In: *Phys. Chem. Chem. Phys.* 13.40 (2011), pp. 17930–17955. DOI: `10.1039/C1CP21668F`.

[3] Frank Noé, Gianni De Fabritiis, and Cecilia Clementi. "Machine learning for protein folding and dynamics". In: *Current Opinion in Structural Biology* 60 (Feb. 2020), pp. 77–84. ISSN: 0959-440X. DOI: `10.1016/j.sbi.2019.12.005`.

[4] Jörg Behler. "Atom-centered symmetry functions for constructing high-dimensional neural network potentials". In: *The Journal of Chemical Physics* 134.7 (Feb. 2011). ISSN: 1089-7690. DOI: `10.1063/1.3553717`.

[5] Kun Zhang et al. "From Molecular Simulations to Experiments: The Recent Development of Room Temperature Ionic Liquid-Based Electrolytes in Electric Double-Layer Capacitors". In: *Molecules* 29.6 (Mar. 2024), p. 1246. ISSN: 1420-3049. DOI: `10.3390/molecules29061246`.

[6] Aaron R. Finney and Matteo Salvalaglio. "Molecular simulation approaches to study crystal nucleation from solutions: Theoretical considerations and computational challenges". In: *WIREs Computational Molecular Science* 14.1 (Nov. 2023). ISSN: 1759-0884. DOI: `10.1002/wcms.1697`.

[7] Xiaowei Cheng et al. "Solid-liquid equilibrium behavior and thermodynamic analysis of ivermectin using experiments and molecular simulations". In: *Journal of Molecular Liquids* 418 (Jan. 2025), p. 126701. ISSN: 0167-7322. DOI: `10.1016/j.molliq.2024.126701`.

[8] Ramón Garduño-Juárez et al. "Molecular Dynamic Simulations for Biopolymers with Biomedical Applications". In: *Polymers* 16.13 (June 2024), p. 1864. ISSN: 2073-4360. DOI: `10.3390/polym16131864`.

[9] Jan Philipp Bittner, Irina Smirnova, and Sven Jakobtorweihen. "Investigating Biomolecules in Deep Eutectic Solvents with Molecular Dynamics Simulations: Current State, Challenges and Future Perspectives". In: *Molecules* 29.3 (Feb. 2024), p. 703. ISSN: 1420-3049. DOI: `10.3390/molecules29030703`.

[10] Jinlong He et al. "Molecular simulations of organic solvent transport in dense polymer membranes: Solution-diffusion or pore-flow mechanism?" In: *Journal of Membrane Science* 708 (Aug. 2024), p. 123055. ISSN: 0376-7388. DOI: `10.1016/j.memsci.2024.123055`.

[11] Jianhao Qian et al. “Molecular simulations reveal gas transport mechanisms in polyamide membranes”. In: *Journal of Membrane Science* 731 (July 2025), p. 124056. ISSN: 0376-7388. DOI: 10.1016/j.memsci.2025.124056.

[12] V. Spahn et al. “A nontoxic pain killer designed by modeling of pathological receptor conformations”. In: *Science* 355.6328 (Mar. 2017), pp. 966–969. ISSN: 1095-9203. DOI: 10.1126/science.aai8636.

[13] Hugo Verli and Chris Oostenbrink. “Status and Prospects of Molecular Simulations for Drug Discovery”. In: *Journal of the Brazilian Chemical Society* (2024). ISSN: 1678-4790. DOI: 10.21577/0103-5053.20240119.

[14] G. Brent Dawe et al. “Distinct Structural Pathways Coordinate the Activation of AMPA Receptor-Auxiliary Subunit Complexes”. In: *Neuron* 89.6 (Mar. 2016), pp. 1264–1276. ISSN: 0896-6273. DOI: 10.1016/j.neuron.2016.01.038.

[15] Andrew G. Stack and Paul R. C. Kent. “Geochemical reaction mechanism discovery from molecular simulation”. In: *Environmental Chemistry* 12.1 (2015), p. 20. ISSN: 1448-2517. DOI: 10.1071/en14045.

[16] W. F. van Gunsteren et al. “Validation of Molecular Simulation: An Overview of Issues”. In: *Angewandte Chemie International Edition* 57.4 (Dec. 2017), pp. 884–902. ISSN: 1521-3773. DOI: 10.1002/anie.201702945.

[17] Xin Zhong, Felix Höfling, and Timm John. “Hydrogen Diffusion in Garnet: Insights From Atomistic Simulations”. In: *Geochemistry, Geophysics, Geosystems* 26.2 (Feb. 2025). ISSN: 1525-2027. DOI: 10.1029/2024gc011951.

[18] Bastiaan J. Braams and Joel M. Bowman. “Permutationally invariant potential energy surfaces in high dimensionality”. In: *International Reviews in Physical Chemistry* 28.4 (Oct. 2009), pp. 577–606. ISSN: 1366-591X. DOI: 10.1080/01442350903234923.

[19] Jürgen Hafner. “Ab-initio simulations of materials using VASP: Density functional theory and beyond”. In: *Journal of Computational Chemistry* 29.13 (July 2008), pp. 2044–2078. ISSN: 1096-987X. DOI: 10.1002/jcc.21057.

[20] Denghui Lu et al. “86 PFLOPS Deep Potential Molecular Dynamics simulation of 100 million atoms with ab initio accuracy”. In: *Computer Physics Communications* 259 (Feb. 2021), p. 107624. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2020.107624.

[21] V. Botu et al. “Machine Learning Force Fields: Construction, Validation, and Outlook”. In: *The Journal of Physical Chemistry C* 121.1 (Dec. 2016), pp. 511–522. ISSN: 1932-7455. DOI: 10.1021/acs.jpcc.6b10908.

[22] Albert P. Bartók, Risi Kondor, and Gábor Csányi. “On representing chemical environments”. In: *Physical Review B* 87.18 (May 2013). ISSN: 1550-235X. DOI: 10.1103/physrevb.87.184115.

[23] Jörg Behler and Michele Parrinello. “Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces”. In: *Physical Review Letters* 98.14 (Apr. 2007), p. 146401. ISSN: 1079-7114. DOI: 10.1103/physrevlett.98.146401.

[24] Nongnuch Artrith and Jörg Behler. “High-dimensional neural network potentials for metal surfaces: A prototype study for copper”. In: *Physical Review B* 85.4 (Jan. 2012). ISSN: 1550-235X. DOI: 10.1103/physrevb.85.045439.

[25] Nongnuch Artrith, Björn Hiller, and Jörg Behler. “Neural network potentials for metals and oxides - First applications to copper clusters at zinc oxide”. In: *physica status solidi (b)* 250.6 (Nov. 2012), pp. 1191–1203. ISSN: 1521-3951. DOI: 10.1002/pssb.201248370.

[26] Thomas B. Blank et al. “Neural network models of potential energy surfaces”. In: *The Journal of Chemical Physics* 103.10 (Sept. 1995), pp. 4129–4137. ISSN: 0021-9606. DOI: 10.1063/1.469597.

[27] Yoshua Bengio. “Gradient-Based Optimization of Hyperparameters”. In: *Neural Computation* 12.8 (Aug. 2000), pp. 1889–1900. ISSN: 1530-888X. DOI: 10.1162/089976600300015187.

[28] Frank Noé et al. “Boltzmann generators: Sampling equilibrium states of many-body systems with deep learning”. In: *Science* 365.6457 (Sept. 2019). ISSN: 1095-9203. DOI: 10.1126/science.aaw1147.

[29] Shuxin Zheng et al. “Predicting equilibrium distributions for molecular systems with deep learning”. In: *Nature Machine Intelligence* 6.5 (May 2024), pp. 558–567. ISSN: 2522-5839. DOI: 10.1038/s42256-024-00837-3.

[30] Luigi Bonati, GiovanniMaria Piccini, and Michele Parrinello. “Deep learning the slow modes for rare events sampling”. In: *Proceedings of the National Academy of Sciences* 118.44 (Oct. 2021). ISSN: 1091-6490. DOI: 10.1073/pnas.2113533118.

[31] Markus Meuwly. “Machine Learning for Chemical Reactions”. In: *Chemical Reviews* 121.16 (June 2021), pp. 10218–10239. ISSN: 1520-6890. DOI: 10.1021/acs.chemrev.1c00033.

[32] Bobby G. Sumpter and Donald W. Noid. “Potential energy surfaces for macromolecules. A neural network technique”. In: *Chemical Physics Letters* 192.5 (May 1992), pp. 455–462. ISSN: 0009-2614. DOI: 10.1016/0009-2614(92)85498-y.

[33] Chris M. Handley and Paul L. A. Popelier. “Potential Energy Surfaces Fitted by Artificial Neural Networks”. In: *The Journal of Physical Chemistry A* 114.10 (Feb. 2010), pp. 3371–3383. ISSN: 1520-5215. DOI: 10.1021/jp9105585.

[34] Francesco Di Giovanni et al. “On Over-Squashing in Message Passing Neural Networks: The Impact of Width, Depth, and Topology”. In: *Proceedings of the 40th International Conference on Machine Learning*. Ed. by Andreas Krause et al. Vol. 202. Proceedings of Machine Learning Research. PMLR, 23–29 Jul 2023, pp. 7865–7885. URL: https://proceedings.mlr.press/v202/di-giovanni23a.html.

[35] Justin Gilmer et al. “Neural Message Passing for Quantum Chemistry”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 1263–1272. URL: https://proceedings.mlr.press/v70/gilmer17a.html.

[36] Enzo Grossi and Massimo Buscema. “Introduction to artificial neural networks”. In: *European Journal of Gastroenterology and Hepatology* 19.12 (Dec. 2007), pp. 1046–1054. ISSN: 0954-691X. DOI: 10.1097/meg.0b013e3282f198a0.

[37] Jinming Zou, Yi Han, and Sung-Sau So. “Overview of Artificial Neural Networks”. In: *Artificial Neural Networks*. Humana Press, 2008, pp. 14–22. ISBN: 9781603271011. DOI: 10.1007/978-1-60327-101-1_2.

[38] Shun-ichi Amari. “Backpropagation and stochastic gradient descent method”. In: *Neurocomputing* 5.4-5 (June 1993), pp. 185–196. ISSN: 0925-2312. DOI: 10.1016/0925-2312(93)90006-o.

[39] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: (Sept. 2016). DOI: 10.48550/ARXIV.1609.04747. arXiv: 1609.04747 [cs.LG].

[40] David Luebke. “Data parallel computing”. In: *Programming Massively Parallel Processors*. Elsevier, 2017, pp. 19–41. ISBN: 9780128119860. DOI: 10.1016/b978-0-12-811986-0.00002-9.

[41] Mark Ebersole. “Scalable parallel execution”. In: *Programming Massively Parallel Processors*. Elsevier, 2017, pp. 43–69. ISBN: 9780128119860. DOI: 10.1016/b978-0-12-811986-0.00003-0.

[42] David B. Kirk and Wen-mei W. Hwu. “Memory and data locality”. In: *Programming Massively Parallel Processors*. Elsevier, 2017, pp. 71–101. ISBN: 9780128119860. DOI: 10.1016/b978-0-12-811986-0.00004-2.

[43] Jinzhe Zeng et al. “DeePMD-kit v2: A software package for deep potential models”. In: *The Journal of Chemical Physics* 159.5 (Aug. 2023). ISSN: 1089-7690. DOI: 10.1063/5.0155600.

[44] Alexander Kammeyer. “Radix sort for massively parallel hardware: a multi-GPU implementation”. MA thesis. Freie Universität Berlin, 2018.

[45] Mark Harris, Shubhabrata Sengupta, and John D. Owens. “Parallel Prefix Sum (Scan) with CUDA”. In: *GPU Gems 3*. Available online: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda. Boston, MA: Addison-Wesley, Oct. 2007. Chap. 39.

[46] CUDA Core Compute Libraries. *cub::BlockScan CUDA Core Compute Libraries (CUB)*. https://nvidia.github.io/cccl/cub/api/classcub_1_1BlockScan.html. Accessed: 2025-09-11. NVIDIA, 2025.

[47] CUDA Core Compute Libraries. *cub::BlockRadixSort CUDA Core Compute Libraries (CUB)*. https://nvidia.github.io/cccl/cub/api/classcub_1_1BlockRadixSort.html. Accessed: 2025-09-11. NVIDIA, 2025.

[48] Jinjin Wang et al. “A deep learning interatomic potential developed for atomistic simulation of carbon materials”. In: *Carbon* 186 (Jan. 2022), pp. 1–8. ISSN: 0008-6223. DOI: 10.1016/j.carbon.2021.09.062.

[49] Jianhang Xu et al. “Isotope effects in molecular structures and electronic properties of liquid water via deep potential molecular dynamics based on the SCAN functional”. In: *Physical Review B* 102.21 (Dec. 2020). ISSN: 2469-9969. DOI: 10.1103/physrevb.102.214113.

[50] Linfeng Zhang et al. “End-to-end Symmetry Preserving Inter-atomic Potential Energy Model for Finite and Extended Systems”. In: NeurIPS (May 2018). DOI: 10.48550/ARXIV.1805.09003. arXiv: 1805.09003 [physics.comp-ph].

[51] Han Wang et al. “DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics”. In: *Computer Physics Communications* 228 (July 2018), pp. 178–184. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2018.03.016.

[52] Jinzhe Zeng et al. “DeePMD-kit v3: A Multiple-Backend Framework for Machine Learning Potentials”. In: *Journal of Chemical Theory and Computation* 21.9 (May 2025), pp. 4375–4385. ISSN: 1549-9626. DOI: `10.1021/acs.jctc.5c00340`.

[53] Kaiming He et al. *Deep Residual Learning for Image Recognition.* 2015. DOI: `10.48550/ARXIV.1512.03385`.

[54] DeePMD-kit developers. *Use DeePMD-kit.* `https://deepmd-kit.readthedocs.io/en/stable/use-deepmd-kit.html`. Accessed: 2025-09-01. 2025.

[55] DeepModeling Developers. *DeePMD-kit Documentation: `sel` Parameter.* `https://docs.deepmodeling.com/projects/deepmd/en/master/model/sel.html`. Accessed: 2025-09-05. 2025.

[56] Manzil Zaheer et al. *Deep Sets.* 2017. DOI: `10.48550/ARXIV.1703.06114`.

[57] Bo Pang, Erik Nijkamp, and Ying Nian Wu. “Deep Learning With TensorFlow: A Review”. In: *Journal of Educational and Behavioral Statistics* 45.2 (Sept. 2019), pp. 227–248. ISSN: 1935-1054. DOI: `10.3102/1076998619872761`.

[58] Charles C. Margossian. “A review of automatic differentiation and its efficient implementation”. In: *WIREs Data Mining and Knowledge Discovery* 9.4 (Mar. 2019). ISSN: 1942-4795. DOI: `10.1002/widm.1305`.

[59] Atilim Gunes Baydin et al. “Automatic differentiation in machine learning: a survey”. In: *Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. The Journal of Machine Learning Research, 18(153):1–43, 2018* (Feb. 2015). DOI: `10.48550/ARXIV.1502.05767`. arXiv: `1502.05767 [cs.SC]`.

[60] Andreas Griewank and Andrea Walther. “Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation”. In: *Evaluating Derivatives.* Society for Industrial and Applied Mathematics, Jan. 2008. Chap. Memory Issues and Complexity Bounds, pp. 61–90. ISBN: 9780898717761. DOI: `10.1137/1.9780898717761.ch4`.

[61] Matteo Cioni et al. “Innate dynamics and identity crisis of a metal surface unveiled by machine learning of atomic environments”. In: *The Journal of Chemical Physics* 158.12 (Mar. 2023). ISSN: 1089-7690. DOI: `10.1063/5.0139010`.

[62] J.A Moulijn, A.E van Diepen, and F Kapteijn. “Catalyst deactivation: is it predictable?” In: *Applied Catalysis A: General* 212.1–2 (Apr. 2001), pp. 3–16. ISSN: 0926-860X. DOI: `10.1016/s0926-860x(00)00842-5`.

[63] DeePMD-kit Developers. *DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics.* `https://github.com/deepmodeling/deepmd-kit`. Accessed: 2025-09-05. 2025.

[64] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. ISSN: 1476-4687. DOI: `10.1038/s41586-020-2649-2`.

[65] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. “Lua-An Extensible Extension Language”. In: *Software: Practice and Experience* 26.6 (June 1996), pp. 635–652. ISSN: 1097-024X. DOI: `10.1002/(sici)1097-024x(199606)26:6<635::aid-spe26>3.0.co;2-p`.

[66] Viktor Skoblin. “Neural Network Potentials for Hardware-Accelerated Molecular Simulations of Bulk Metals”. MA thesis. Freie Universität Berlin, 2023.