End-to-end Differentiable Reactive Molecular Dynamics Simulations using JAX

Mehmet Cagri Kaymak[1], Samuel S. Schoenholz[4⋆], Ekin D. Cubuk[3], Kurt A. O'Hearn[1], Kenneth M. Merz, Jr.[2], and Hasan Metin Aktulga[1]

[1] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA {kaymakme,ohearnku,hma}@msu.edu
[2] Department of Chemistry, Michigan State University, East Lansing, MI 48824, USA merz@chemistry.msu.edu
[3] Google Research, Mountain View, CA, USA
[4] OpenAI, San Francisco, CA, USA

Abstract. The reactive force field (ReaxFF) interatomic potential is a powerful tool for simulating the behavior of molecules in a wide range of chemical and physical systems at the atomic level. Unlike traditional classical force fields, ReaxFF employs dynamic bonding and polarizability to enable the study of reactive systems. Over the past couple decades, highly optimized parallel implementations have been developed for ReaxFF to efficiently utilize modern hardware such as multi-core processors and graphics processing units (GPUs). However, the complexity of the ReaxFF potential poses challenges in terms of portability to new architectures (AMD and Intel GPUs, RISC-V processors, etc.), and limits the ability of computational scientists to tailor its functional form to their target systems. In this regard, the convergence of cyber-infrastructure for high performance computing (HPC) and machine learning (ML) presents new opportunities for customization, programmer productivity and performance portability. In this paper, we explore the benefits and limitations of JAX, a modern ML library in Python representing a prime example of the convergence of HPC and ML software, for implementing ReaxFF. We demonstrate that by leveraging auto-differentiation, just-in-time compilation, and vectorization capabilities of JAX, one can attain a portable, performant, and easy to maintain ReaxFF software. Beyond enabling MD simulations, end-to-end differentiability of trajectories produced by ReaxFF implemented with JAX makes it possible to perform related tasks such as force field parameter optimization and meta-analysis without requiring any significant software developments. We also discuss scalability limitations using the current version of JAX for ReaxFF simulations.

Keywords: reactive molecular dynamics · HPC/ML software · auto-differentiation · hardware portability

⋆ Work done while at Google.

1 Introduction

Molecular dynamics (MD) simulations are widely used to study physical and chemical processes at the atomistic level in fields such as biophysics, chemistry and materials science. Compared to quantum mechanical (QM) MD simulations, which involve solving the Schrodinger's equation, classical MD simulations are cost-effective. They enable the study of large systems over significantly long time frames by making certain approximations. In this approach, the atomic nucleus and its electrons are treated as single particle. The atomic interactions are governed by a force field (FF), a set of parameterized mathematical equations that capture known atomic interactions such as bonds, angles, torsions, van der Waals, and Coulomb interactions. To ensure the predictive power of empirical force fields, they must be fitted to reference data obtained through high-fidelity quantum mechanical computations and/or experimental studies.

Classical MD models typically adopt static bonds and fixed partial charges which make them unsuitable for studying reactive systems. To remedy these limitations, different reactive force fields have been developed [26,29,10]. In this paper, we focus on the ReaxFF, which is one of the most impactful and widely used reactive force fields [29,25]. It allows bonds to form and break throughout the simulation and dynamically calculates partial charges using suitable charge models. Due to the dynamic nature of bonds and partial charges, ReaxFF is significantly more complex and computationally expensive than classical force fields.

1.1 Related Work

To enable large-scale and long duration simulations, several ReaxFF implementations with different features and architectural support have been developed over the past couple decades. PuReMD has shared and distributed-memory versions for both CPUs and GPUs (CUDA-based), all of which are maintained separately [3,2,14], and several of these versions have been integrated into LAMMPS and AMBER [18]. More recently, to ensure hardware portability and simplify code maintenance and performance optimizations, a Kokkos-based implementation of ReaxFF has been developed in LAMMPS [28]. Kokkos is a performance portable programming model and allows the same codebase implemented using its primitives to be compiled for different backends. The current ReaxFF/Kokkos software also supports distributed-memory parallelism. In addition to the above open-source software, SCM provides a commercial software that includes ReaxFF support [21].

The success of ML techniques in fields such as computer vision and natural language processing has triggered its wide-spread use also in scientific computing. Specifically, in molecular modeling and simulation, a new class of force fields called machine learning potentials (MLP) such as SNAP [27], the Behler/Parrinello potential [7], SchNet [24], OrbNet [**?**], and NequIP [6] has emerged. More recently, we started witnessing an increase in the number of scientific applications adopting ML libraries such as Tensorflow [1], PyTorch [17],

and JAX [9], not only for ML approaches but as a general purpose programming model even when using conventional techniques. This can be attributed to the convenience of advanced tools developed around these programming models and libraries such as auto-differentiation, auto-vectorization, and just-in-time compilation. Such tools have enabled fast prototyping of new ideas as well as hardware portability without sacrificing much computational efficiency.

Intelligent-ReaxFF [12] and JAX-ReaxFF [13] implementations both leverage modern machine learning frameworks. However, they are both primarily designed for force field fitting, and as such they are designed to work with molecular systems typically containing tens of atoms, and they cannot scale beyond systems with more than a couple hundred atoms. More importantly, they both lack molecular dynamics capabilities.

1.2 Our Contribution

The aforementioned features of ML cyber-infrastructure are highly attractive from the perspective of MD software, considering the fact that existing force field implementations are mostly written in low-level languages and tuned to the target hardware for high performance. As such, we introduce a portable, performant, and easy-to-maintain ReaxFF implementation in Python built on top of JAX-MD [23]. This new implementation of ReaxFF is

- **easy-to-maintain** because it only requires expressing the functional form of the potential energy for different atomic interactions in Python. MD simulations require calculation of forces which are calculated by taking the gradient of the potential energy with respect to atom positions at each time step. This can simply be accomplished with a call to the `grad( )` function in JAX,
- **hardware portable** because for its functional transformations, JAX uses XLA (Accelerated Linear Algebra) [22], which is a domain specific compiler for vector and matrix operations. Since XLA has high performance implementations across different CPUs (x86_64 and ARM) as well as GPUs (Nvidia and AMD), porting our ReaxFF implementation does not require any additional coding,
- **performant** because we ensure that our underlying ReaxFF interaction lists are suitable for vectorization, and we leverage just-in-time compilation effectively through a carefully designed update/reallocation scheme,
- **versatile** because we designed our implementation such that the same interaction kernels can be re-used in either a single high performance run (needed for long MD simulations) or multiple small single-step runs (needed for parameter optimization) settings. This allows our implementation to be suitable for force field training as well. Also, it simplifies the study of new functional forms for various interactions in the ReaxFF model.

2 Background

2.1 ReaxFF Overview

ReaxFF uses the bond order concept to determine the interaction strength between pairs of atoms given their element types and distances, and then applies corrections to these initial pairwise bond orders based on the information about all surrounding atoms. The corrected bond order is used as the main input for the energy terms such as bond energy (E_{bond}), valence angle energy (E_{val}), and torsion angle energy (E_{tors}). To account for atoms that may not attain their optimal coordination, additional energy terms such as under-/over-coordination energies, coalition, and conjugation energies are used, which we denote as E_{other} for simplicity. The van der Waals energy (E_{vdWaals}) and electrostatic energy terms (E_{Coulomb}) constitute the non-bonded terms. Since bond orders are dynamically changing, an important pre-requisite for calculation of electrostatic energy is the charge equilibration procedure which dynamically assigns charges to atoms based on the surroundings of each atom. For systems with hydrogen bonds, a special energy term ($E_{\text{H-bond}}$) is applied. Bonded interactions are typically truncated at 5 Å, hydrogen bonds are effective up to 7.5 Å, and the non-bonded interaction cutoff is typically set to 10–12 Å. Eq. (1) sums up the various parts that constitute the ReaxFF potential energy, and we summarize the dependency information between them in Fig. 1.

$$\begin{aligned} E_{\text{system}} = E_{\text{bond}} + E_{\text{val}} + E_{\text{tors}} + E_{\text{H-bond}} \\ + E_{\text{vdWaals}} + E_{\text{Coulomb}} + E_{\text{other}}. \end{aligned} \tag{1}$$

2.2 JAX and JAX-MD Overview

Since the new ReaxFF implementation is developed in JAX-MD, important design and implementation decisions were based on how JAX and JAX-MD work. As such, we first briefly describe these frameworks.

JAX [9] is a machine learning framework for transforming numerical functions. It implements the Numpy API using its own primitives and provides high order transformation functions for any Python function written using JAX primitives. The most notable of these transformation functions are automatic differentiation (*grad*), vectorization on a single device to leverage SIMD parallelism (*vmap*), parallelization across multiple devices (*pmap*), and just-in-time compilation (*JIT*). These transformations can be composed together to enable more complex ones. JAX uses XLA, a domain specific compiler for linear algebra, under the hood to achieve hardware portability. This allows any Python code written in terms of JAX primitives to be seamlessly compiled for CPUs, GPUs, or TPUs. Since XLA is also used extensively to accelerate Tensorflow models, XLA is supported for almost all modern processors, including GPUs by Nvidia and AMD. With

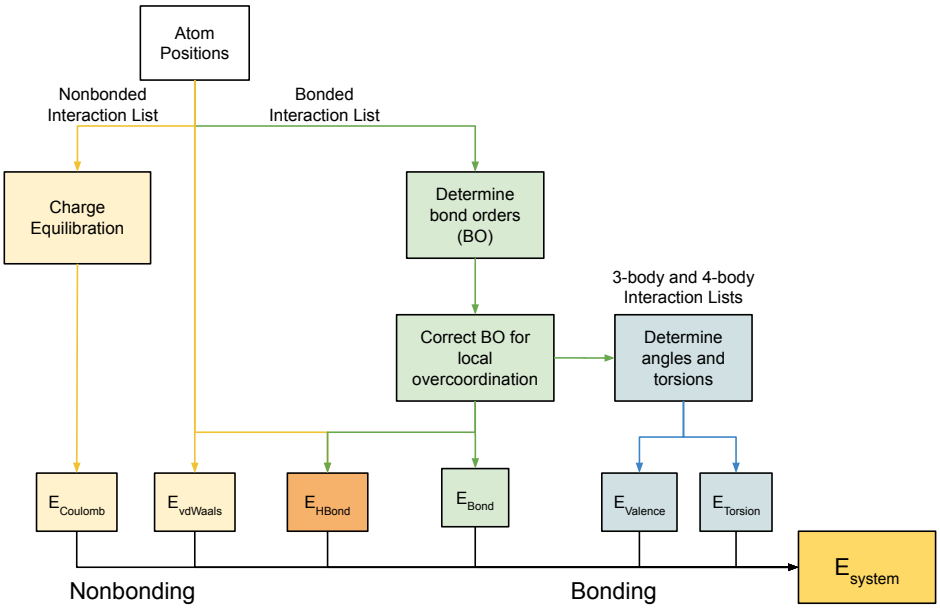

Fig. 1. Task dependency graph for calculations performed in ReaxFF.

JIT, XLA could apply performance optimizations targeted specifically for the selected device. The main limitation of JAX is that it expects the input data to the transformed functions to have fixed sizes. This allows XLA to adopt more aggressive performance optimizations during compilation, but when the size of the input data changes, the code needs to be recompiled.

JAX-MD [23] is an MD package built in Python using JAX. It is designed for performing differentiable physics simulations with a focus on MD. It supports periodic and non-periodic simulation environments. JAX-MD employs a scalable $3D$ grid-cell binning based algorithm to construct the neighbor list for atoms in a given system. It includes integrators for various kinds of ensembles as well as Fast Inertial Relaxation Engine (FIRE) Descent [8] and Gradient Descent based energy minimizers. Various machine learning potentials such as the Behler-Perrinello architecture [7] and graph neural networks including the Neural Equivariant Interatomic Potentials (NequIP) [6], based on the GraphNet library [5], are also readily available. When combined with the capabilities of JAX, this rich ecosystem enables researchers to easily develop and train hybrid approaches for various chemistry and physics applications.

3 Design and Implementation

In this section, we describe the overall design considerations and present the final design for our ReaxFF implementation in JAX-MD. To simplify the design and ensure modularity, generation of the interaction lists have been separated from the computation of partial energy terms. For overall efficiency and scalability, special consideration has been given to memory management.

3.1 Memory Management

To avoid frequent re-compilations, sizes of input to JAX's transforming functions must be known and fixed. As such, we separate the logic for handling the interaction list generation into `allocate` and `update` parts. The allocate function estimates the sizes of all interaction lists (see Fig. 2) and allocates the needed memory with some buffer space (default 20%). Due to its dynamic nature, JAX transformations such as *vmap* and *jit* cannot be applied to the allocate function. The update function works with the already-allocated interaction lists, and fills them based on atom positions while preserving their sizes. Since the update function works on arrays with static sizes, JAX transformations such as *vmap* and *jit* can be and are applied to this function. For effective use of *vmap*, the update function also applies padding when necessary. Finally, while filling in the interaction lists, it also checks whether the utilization of the space allocated for each list falls below a threshold mark (default 50%) where the utilization is the ratio of the true size to the total size. If it does, a call to the allocate function is triggered to shrink the interaction lists as shown in Algorithm 1, which in turn causes JAX to recompile the rest of the code since array dimensions change.

Algorithm 1 General structure of computations in an MD simulation.

```
interLists ← Create the interaction lists using the allocate function
for timestep = 1, 2, ... do
    Calculate forces
    Update positions using the calculated forces
    overflow ← Update the interaction lists
    if overflow then
        interLists ← Reallocate based on the most recent utilizations
    end if
end for
```

Another important aspect of our memory management scheme is the *filtering of interaction lists*. In ReaxFF, while bonds are calculated dynamically, not all bonds are strong enough to be chemically meaningful, and therefore they are ignored (a typical bond strength threshold is 0.01). This has ramifications for higher-order interactions such as 3-body, 4-body, and H-bond interactions as well because they are built on top of the dynamically generated bond lists. As

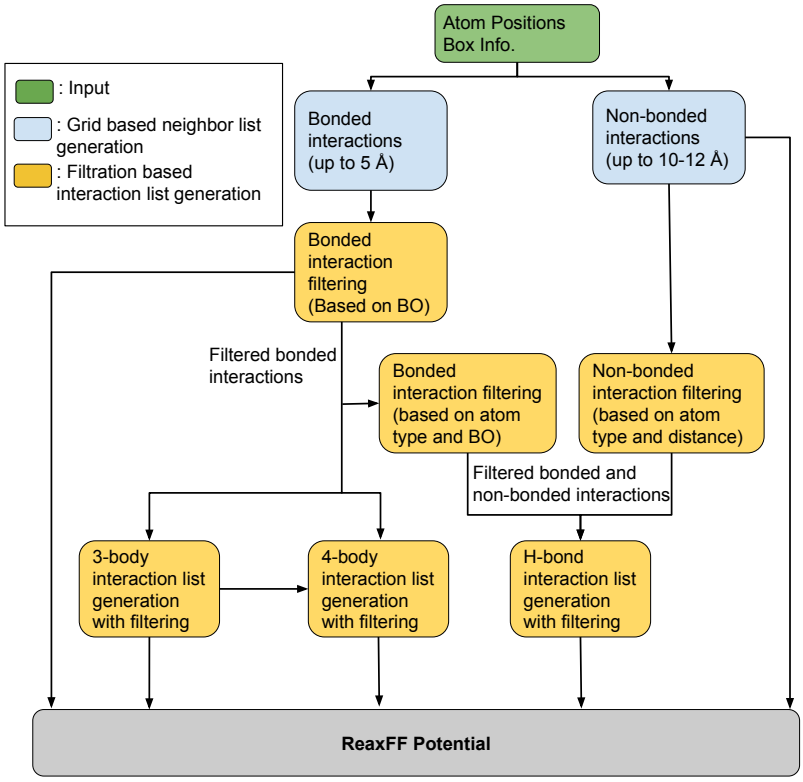

Fig. 2. Flow graph describing the generation of the interaction lists.

we discuss in more detail below, the acceptance criteria for each interaction is different. For 3-body and 4-body interactions, acceptance criteria depends on the strength of bonds among the involved atoms as well as force field parameters specific to that group of atoms; for H-bonds, it is a combination of acceptor-donor atom types and bond strengths. However, the steps for filtering all interaction lists are similar and can be implemented as a generic routine with a candidate interaction list and an interaction-specific acceptance criterion. The interactions that require filtering and their relevant input data are shown as yellow nodes in Fig. 2. First, the candidate interaction list is populated. Then, candidates get masked based on the predefined acceptance criterion. Finally, the candidate list is pruned and passed onto its corresponding potential energy computation function. While actually pruning the candidate list might be seen as an overhead, we note that the number of unaccepted 3-body and 4-body interactions are so high that simply ignoring them during the potential energy computations introduce a significant computational overhead. Also, the memory required to keep the

unfiltered 3-body and 4-body interaction lists would limit the scalability of our implementation for GPUs due to their limited memory resources.

The filtering logic discussed above is JAX-friendly because the shapes of the intermediate (candidate) and final (pruned) data structures are fixed. As such, *vmap* and *jit* transformations can be applied to the filtering procedure, too. As with un-pruned lists, filtered interaction list generation also keeps track of utilization of the relevant lists and sets the overflow flag, when necessary.

3.2 Generation of Interaction Lists

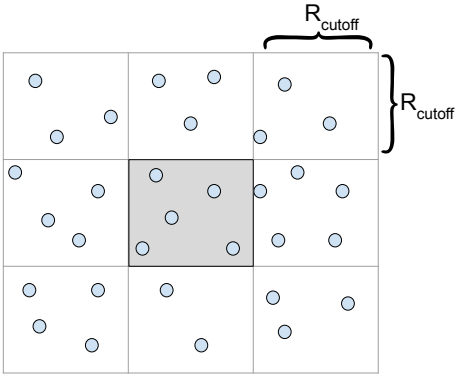

Fig. 3. Illustration of grid-cell neighbor search used to generate neighbor lists.

Pair-wise Bonded Interactions: In ReaxFF, bond order (BO) between atom pairs are at the heart of all bonded potential energy computations. The BOs are computed in two steps. First, uncorrected BOs are computed according to Eq. (2), where r_{ij} is the distance between the atom pair i-j, and r_o^σ, r_o^π, and $r_o^{\pi\pi}$ are the ideal bond lengths for σ-σ, σ-π and π-π bonds, respectively.

$$\begin{aligned} \mathrm{BO}'_{ij} &= \mathrm{BO}^{\sigma}_{ij} + \mathrm{BO}^{\pi}_{ij} + \mathrm{BO}^{\pi\pi}_{ij} \\ &= \exp\left[p_{\mathrm{bo}_1} \cdot \left(\frac{r_{ij}}{r_o^\sigma}\right)^{p_{\mathrm{bo}_2}}\right] + \exp\left[p_{\mathrm{bo}_3} \cdot \left(\frac{r_{ij}}{r_o^\pi}\right)^{p_{\mathrm{bo}_4}}\right] \\ &\quad + \exp\left[p_{\mathrm{bo}_5} \cdot \left(\frac{r_{ij}}{r_o^{\pi\pi}}\right)^{p_{\mathrm{bo}_6}}\right]. \end{aligned} \tag{2}$$

After uncorrected bond orders are computed, the strength of BO'_{ij} is corrected based on the local neighborhood of atoms i and j. The corrected BO (BO_{ij}) represents the coordination number (i.e., number of bonds) between two atoms. Corrected bonds below a certain threshold get discarded as they do not correspond to chemical bonds. Hence, they do not contribute to the total energy.

To calculate uncorrected BO, for each atom in a given system, their neighbors are found using a grid-cell binning based neighbor search algorithm (Fig. 3). This allows us to generate the bonded neighbor lists in $O(Nk)$ where N is the number of atoms and k is the average number neighbors per atom. The side length of the grid cell is set to 5.5 Å, as a buffer space of 0.5 Å is added to the 5 Å actual bonded interaction cutoff to avoid frequent updates to the neighbors list. Since the cell size is almost the same as the bonded interaction cutoff, neighbor search only requires checking the nearby 3^3 grid cells. Neighbor information is stored in a $2D$ format where the neighbors of atom i are located on ith row with padding and alignment, as necessary. This format which is very similar to the ELLPACK format [31] is highly amenable for vectorization and memory coalescing on modern GPUs. It also simplifies bond order corrections because the neighbor indices for a given atom are stored consecutively. As will be discussed later, it also helps creating 3-body (for valency) and 4-body (for torsion) interactions since they use BOs as the main input. After creating the $2D$ neighbor array, BO terms are calculated and pairs with small BOs are filtered out as described above.

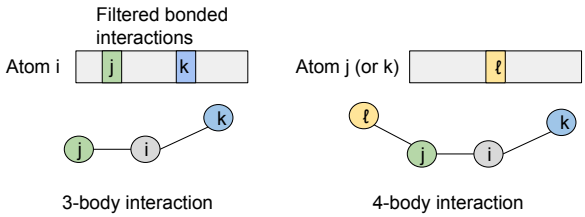

Fig. 4. Atoms and their interactions involved in formation of the 3-body and 4-body interactions.

Higher Order Bonded Interactions: After pruning the bonded interactions, 3-body and 4-body interaction lists are generated (Fig. 4). For each atom, every two neighbor pairs are selected to form the candidate list for 3-body interactions. In a system with N atoms and k neighbors per atom, there will be $O\left(Nk^2\right)$ candidates. Then the candidates are masked and filtered based the involved BO terms to form the final array with shape $M \times 3$ where M is the total number of interactions and columns are atom indices. After that, the finalized 3-body interaction list is used to generate the candidates for the 4-body interactions. For each 3 body interaction i-j-k, neighbors of both j and k are explored to form the 4-body candidate list and then the candidates get filtered based on the 4-body specific mask.

When the molecule involves hydrogen bonds, the hydrogen interaction list is built using the filtered bonded and non-bonded interactions. A hydrogen bond can only be present if there are hydrogen donors and acceptors. While the accep-

tor and the hydrogen are covalently bonded (short range), the acceptor bonds to the hydrogen through a dipole-dipole interaction, therefore it is long ranged (up to 7.5 Å). Hence, to find all possible hydrogen bonds involving a given hydrogen atom, both its bonded neighbors and non-bonded neighbors are scanned. Using the appropriate masking criterion, the final interaction list is formed to be used for potential energy calculations.

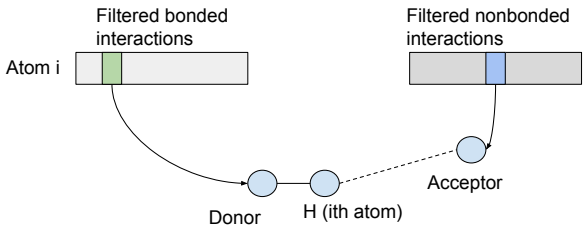

Fig. 5. Atoms and their interactions involved in formation of hydrogen bonds.

Non-Bonded Interactions: In ReaxFF, non-bonded interactions are effective up to 10–12 Å , and they are smoothly tapered down to 0 beyond the cutoff. Similar to the pair-wise bonded interactions, the long range neighbor lists are also built using the grid-cell binning approach, this time using a buffer distance of 1 Å to avoid frequent neighbor updates. The neighbors are again stored in a $2D$ array similar to the ELLPACK format. This simplifies accessing the long range neighbors of a given atom while building the Hydrogen bond interactions list (as shown in Fig. 5). Also, the sparse matrix-vector multiplication kernel (SpMV) required for the dynamic charge calculation becomes simpler and more suitable for GPUs [30].

The non-bonded interaction list is used to compute van der Waals and Coulomb energy terms. While E_{vdWaals} computation is relatively simple as it only involves the summation of the pair-wise interaction energies, E_{Coulomb} requires charges to be dynamically computed based on a suitable charge model such as the charge equilibration (QEq) [20], electronegativity equalization (EE) [16], or atom-condensed Kohn-Sham density functional theory approximated to second order (ACKS2) method [32]. Our current JAX-based implementation relies on the EE method.

The EE method involves assigning partial charges to individual atoms while satisfying constraints for both the net system charge and the equalized atom electronegativities. For a given system with n atoms, let the charges and the positions be $\mathbf{q} = (q_1, q_2, \ldots, q_n)$ and $\mathbf{R} = (r_1, r_2, \ldots, r_n)$, respectively. The electronegativity constraint can be formalized as follows

$$\epsilon_1 = \epsilon_2 = \cdots = \epsilon_i = \underline{\epsilon},$$

where ϵ_i is the electronegativity of atom i and $\underline{\epsilon}$ is the average electronegativity. The net system charge constraint is expressed as

$$\sum_{i=1}^{n} q_i = q_{\text{net}},$$

where q_{net} is the net system charge. The constraints and the parameterized long range interactions can be expressed as a set of linear equations with the partial charges $\mathbf{q}$ being the solution to

$$\begin{bmatrix} \boldsymbol{H} & 1_n \\ 1_n^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{q} \\ \underline{\boldsymbol{\epsilon}} \end{bmatrix} = \begin{bmatrix} -\boldsymbol{\chi} \\ q_{\text{net}} \end{bmatrix},$$

where $\boldsymbol{\chi}$ is an $n \times 1$ vector of target electronegativities and $\boldsymbol{H}$ is a symmetric $n \times n$ matrix describing the interactions between atoms. $H_{i,j}$ is defined as

$$H_{i,j} = \delta_{i,j} \cdot \eta_i + (1 - \delta_{i,j}) \cdot F_{i,j}$$

where $\delta_{i,j}$ is the Kronecker delta operator and η_i is the idempotential. Lastly, $F_{i,j}$ is defined as

$$F_{i,j} = \begin{cases} \frac{1}{\sqrt[3]{r_{i,j}^3 + \gamma_{i,j}^{-3}}}, & r_{i,j} \leq R_{\text{cutoff}} \\ 0, & \text{otherwise} \end{cases}$$

where $r_{i,j}$ is the distance between atom i and j, $\gamma_{i,j}$ is the pair-wise shielding term, and R_{cutoff} is the long range cutoff.

Since the size of the above linear system is $(n + 1) \times (n + 1)$, it is prohibitively expensive to solve it with direct methods when n is becomes large (beyond a few hundred). Hence, we employ an iterative sparse linear solver. The iterative solvers available in JAX only expect a linear operator as a function pointer that can perform the matrix-vector multiplication. This allows us to define the SpMV operation directly using the non-bonded neighbor lists provided in an ELLPACK-like format described earlier without applying any transformations. Another optimization to accelerate the charge equilibration is to use initial guesses to warm start the iterative solver. Since the charges fluctuate smoothly as the simulation progresses, we use the cubic spline extrapolation to produce the initial guesses based on past history [2].

3.3 Force Field Training

Predictive capabilities of empirical force fields are arguably more important than their performance. For this, it is crucial for force field parameters to be optimized using high-fidelity quantum mechanical training data. In contrast to MD simulations involving a single system iterated over long durations, this optimization process typically involves executing several (on the order of hundreds to thousands, depending on the model and target systems) small molecular systems

Algorithm 2 Gradient-based parameter optimization.

1: $\theta \leftarrow$ Initialize the model parameters
2: training set $\leftarrow$ Align the training set by padding with dummy atoms
3: lossFunction $\leftarrow$ Create a loss function by utilizing vmap(energyFunction)
4: calculateGradients $\leftarrow$ jit(grad(lossFunction))
5: **while** stopping criterion not met **do**
6: $X_i, Y_i \leftarrow$ Sample a minibatch of data from the training set
7: Create the interaction lists for X_i
8: g $\leftarrow$ calculateGradients(θ, interLists, Y_i)
9: $\theta \leftarrow$ Update the model parameters using g
10: **end while**

for a single step using different parameter sets in a high-throughput fashion. While evolutionary algorithms have traditionally been used for Reax force field optimizations, as JAX-ReaxFF [13] and Intelligent-ReaxFF [12] have recently demonstrated, using gradient-based optimization techniques can accelerate the training process by two to three orders of magnitude. However, the gradient information needed for force field optimization is much more complex than that of MD simulation – one needs to calculate the derivative of the fitness function which is typically formulated as a weighted sum of the difference between predicted and reference quantities over all systems in the training dataset with respect to parameters to be optimized (which is usually on the order of tens of parameters for ReaxFF). While this would be a formidable task using analytical or numerical techniques, the auto-differentiation capabilities of JAX enable us to easily repurpose the above described ReaxFF MD implementation for parameter optimization. By composing different transformations, a simple loss function defined for a single sample can extended to work for a batch of training data as shown in Algorithm 2. To fully take advantage of SIMD parallelism, especially on GPUs, we ensure that different molecules in the training dataset are properly divided into small batches. To reduce the number of dummy atoms and the amount of padding within each batch, the training set could be clustered based on how much computation they require. Given the allocate/update mechanism described in Section 3.1, the different sizes of interaction lists for different molecular systems in a batch data does not cause additional challenges.

4 Experimental Results

4.1 Software and Hardware Setup

To verify the accuracy of the presented JAX-based ReaxFF implementation, simulations were performed using molecular systems shown in Table 1. The Kokkos-based LAMMPS implementation of ReaxFF was chosen for validation and benchmarking comparisons due to its maturity and maintenance. For this purpose, we used the most recent stable release of LAMMPS (git tag stable_23Jun2022_update3), and experimented on both Nvidia and AMD GPUs.

LAMMPS was built using GCC v10.3.0, OpenMPI v4.1.1, and CUDA v11.4.2 for the Nvidia GPUs, and with ROCm v5.3.0, aomp v16.0, and OpenMPI v4.1.4 for the AMD GPUs (using device-specific compiler optimization flags for both). For the JAX experiments, Python v3.8, JAX v0.4.1, and JAX-MD v0.2.24 were paired with CUDA v11.4.2 for the Nvidia GPUs, and ROCm v5.3.0 for the AMD GPUs. Hardware details are presented in Table 2. The compute nodes at the Michigan State University High-Performance Computing Center (MSU-HPCC) and the AMD Cloud Platform are used for the experiments.

Name	Chem. Rep.	N	Sim. Box (Å)	Force Field
Water	H_2O	2400	$29.0 \times 28.9 \times 29.3$	[11]
Silica	SiO_2	6000	$36.9 \times 50.7 \times 52.5$	[11]

Table 1. Molecular systems used in the performance evaluation section, with the third column (N) indicating the number of atoms, the fourth one denoting the dimensions of the rectangular simulation box, and the last column showing the force field used to simulate the system.

GPU	CPU	Cluster
A100	Intel Xeon 8358 (64 cores)	MSU-HPCC
V100	Intel Xeon Platinum 8260 (48 cores)	MSU-HPCC
MI210	AMD EPYC 7742 (64 cores)	AMD Cloud Platform
MI100	AMD EPYC 7742 (64 cores)	AMD Cloud Platform

Table 2. Hardware details of the platforms used for performance experiments.

4.2 Validation of MD Capabilities

Fig. 6 shows that the JAX-based ReaxFF energies almost perfectly match those from LAMMPS in actual MD simulations. The deviation only becomes visible after 2000 MD steps which is inevitable due to machine precision limitations. The relative energy difference is around 10^{-7} for both the water and silica systems.

4.3 Performance and Scalability

We compare the performance of JAX-based ReaxFF to Kokkos/ReaxFF package in LAMMPS on both Nvidia and AMD GPUs. While Kokkos/ReaxFF supports

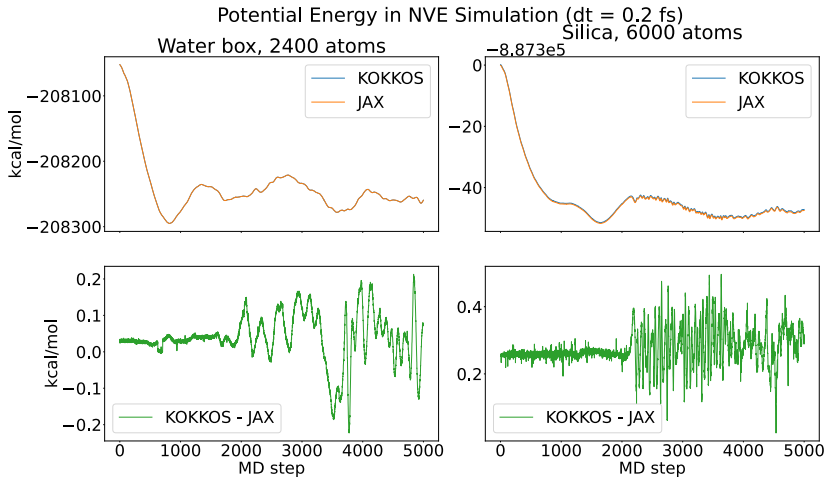

Fig. 6. Comparison of absolute (top plots) and relative difference (bottom plots) in potential energies for NVE simulations with a time step of 0.2 fs and a CG solver with 1e-6 tolerance for the charge calculation.

MPI parallelism, we use a single GPU for all tests. Kokkos/ReaxFF incurs minimal communication overheads when there is a single MPI process. The performance comparison on AMD GPUs is possible through Kokkos' ROCm backend support, as well as the availability of JAX/XLA on AMD GPUs.

To create systems with varying size, the molecular systems shown in Table 1 have been periodically replicated along the x, y, and z dimensions. The number of atoms vary from 2400 to 19200 for the water systems and from 6000 to 24000 for the silica systems. For each experiment, NVE simulations with a time step of 0.2 fs were run for 5000 steps, and the average time per step in ms was reported. For both the Kokkos and JAX-based implementations, the buffer distance for the non-bonded interactions was set to 1 Å. While reneighboring is done every 25 MD steps for Kokkos, the JAX implementation keeps track of how much atoms move since the last neighborhood update and only reneighbors when atoms move more than the buffer distance. As suggested by the Kokkos documentation, the half-neighbor list option is used.

While written in Python using JAX primitives, the proposed implementation is faster when the system size is small on all GPUs. As the number of atoms increases, while the time increases linearly for the JAX implementation, the Kokkos one increases sublinearly. The sublinear scaling for Kokkos indicates that it cannot fully utilize the resources when the problem size is small unlike JAX. As the problem size increases, Kokkos starts to utilize the GPU better and yield better performance. The Kokkos implementation achieves up to 3.2x speedup for the largest water systems on AMD GPUs (MI100 and MI210). On Nvidia

GPUs (V100 and A100), it is around 2.3x faster for the same water system with 19200 atoms. For the silica systems where there are no hydrogen bonds, Kokkos is around 2x faster on the AMD GPUs and 1.5x on the Nvidia GPUs. On the other hand, when the problem size is small, JAX achieves up to 1.8x speedup on an A100 GPU.

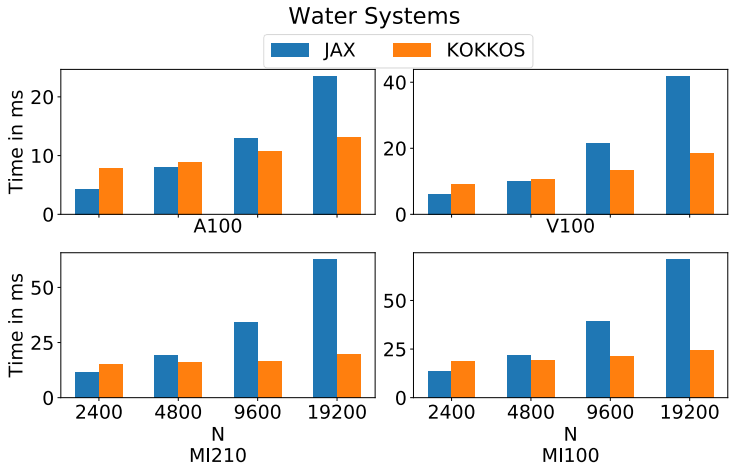

Fig. 7. Average time per MD step (in ms) for the water systems with varying sizes.

4.4 Training

To demonstrate the training performance of the described implementation, we trained the ReaxFF parameters on the public QM9 dataset of about 134k relaxed organic molecules made up of H, C, N, O, and F atoms, with each molecule containing up to nine non-hydrogen atoms [19]. All systems are calculated at the B3LYP/6-31G(2df,p) level of theory. To simplify the dataset, we removed the molecules that contain F atoms which resulted in around 130k molecules. During optimization, 80% of the data is used for training and the remaining 20% for testing. The training is done using the AdamW optimizer [15] from the Optax library [4] with a batch size of 512 and the learning rate is set to 0.001.

The ReaxFF model is typically fit to the training data containing relative energy differences between molecules with the same type of atoms (different conformations and configurations) and the energies of the individual atoms get canceled out. Since the QM9 dataset only contains the absolute energies, we added a new term to the ReaxFF potential to remedy the energy shifts caused

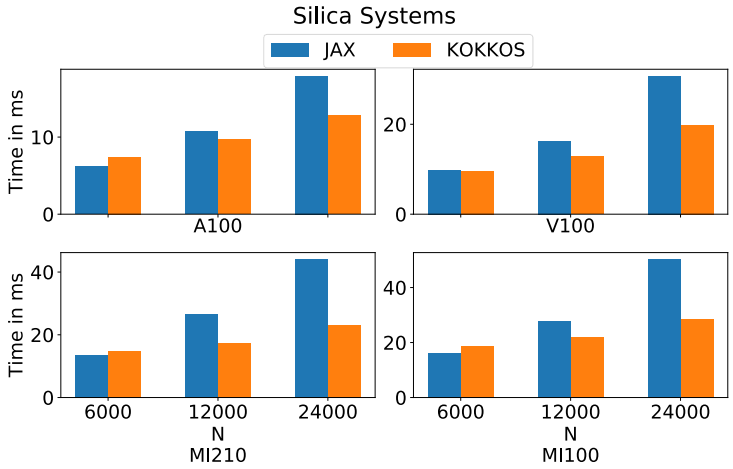

Fig. 8. Average time per MD step (in ms) for the silica systems with varying sizes.

by the self-energies of the individual atoms.

$$\begin{aligned} E_{\text{system}} &= E_{\text{ReaxFF}} + E_{\text{self-energy}} \\ E_{\text{self-energy}} &= \sum_{i=1}^{N} s_i \end{aligned} \tag{3}$$

In Eq. (3), E_{ReaxFF} is the original ReaxFF potential designed to capture the interaction related terms and $E_{\text{self-energy}}$ is the newly added parameterized self-energy term to capture the energy shifts, and s_i is the self energy of atom i solely determined by the atom type. Hence, the new term only contains 4 parameters as there are 4 atom types in the modified QM9 dataset. In total, around 1100 ReaxFF parameters are optimized during the training. The training is performed on an A100 GPU with each epoch taking approximately 8 seconds. Fig. 9 shows the mean absolute error (MAE) per epoch. Since the ReaxFF model has a relatively small number of parameters compared to most modern ML methods, the training and test MAE perfectly overlap throughout the training. The final MAE of the model on the test data is 3.6 kcal/mol. While this is higher than the ideal target of 1 kcal/mol error, we note that this is a straight optimization without any fine-tuning to demonstrate the capabilities of the new ReaxFF implementation.

5 Conclusion

With the accelerator landscape changing rapidly and becoming more complex, cross platform compilers gain more importance as they enable the same codebase to be used on different architectures. By leveraging modern machine learning

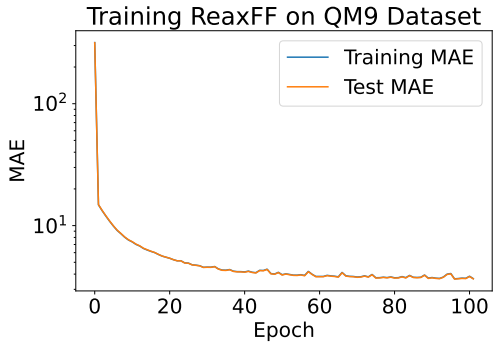

Fig. 9. Training progress of the ReaxFF model on the QM9 dataset, with the final MAE on the test data being 3.6 kcal/mol.

cyber-infrastructure, we developed a new JAX-based ReaxFF implementation that is easy-to-maintain, hardware portable, performant, and versatile. Using auto-differentiation, forces in MD simulations are computed directly from energy functions implemented in Python without requiring any extra coding. It also allows the same code to be used for both MD simulations and parameter optimization which are both essential to study any system of interest with ReaxFF. While Kokkos is an another cross-plotform solution, it lacks auto-differentiation and batching optimization capabilities. Although it is more performant for bigger molecules, the JAX implementation is faster for small ones while also providing new functionalities.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: 12th USENIX symposium on operating systems design and implementation (OSDI 16). pp. 265–283 (2016)
2. Aktulga, H.M., Fogarty, J.C., Pandit, S.A., Grama, A.Y.: Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques. Parallel Computing **38**(4-5), 245–259 (2012)
3. Aktulga, H.M., Pandit, S.A., van Duin, A.C., Grama, A.Y.: Reactive molecular dynamics: Numerical methods and algorithmic techniques. SIAM Journal on Scientific Computing **34**(1), C1–C23 (2012)
4. Babuschkin, I., Baumli, K., Bell, A., Bhupatiraju, S., Bruce, J., Buchlovsky, P., Budden, D., Cai, T., Clark, A., Danihelka, I., Fantacci, C., Godwin, J., Jones, C., Hemsley, R., Hennigan, T., Hessel, M., Hou, S., Kapturowski, S., Keck, T., Kemaev, I., King, M., Kunesch, M., Martens, L., Merzic, H., Mikulik, V., Norman, T., Quan, J., Papamakarios, G., Ring, R., Ruiz, F., Sanchez, A., Schneider, R., Sezener, E., Spencer, S., Srinivasan, S., Wang, L., Stokowiec, W., Viola, F.: The DeepMind JAX Ecosystem (2020), see http://github.com/deepmind/jax

5. Battaglia, P.W., Hamrick, J.B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al.: Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261 (2018)
6. Batzner, S., Musaelian, A., Sun, L., Geiger, M., Mailoa, J.P., Kornbluth, M., Molinari, N., Smidt, T.E., Kozinsky, B.: E (3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. Nature communications **13**(1), 1–11 (2022)
7. Behler, J., Parrinello, M.: Generalized neural-network representation of high-dimensional potential-energy surfaces. Physical review letters **98**(14), 146401 (2007)
8. Bitzek, E., Koskinen, P., Gähler, F., Moseler, M., Gumbsch, P.: Structural relaxation made simple. Physical review letters **97**(17), 170201 (2006)
9. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., et al.: Jax: composable transformations of python+ numpy programs. Version 0.2 **5**, 14–24 (2018)
10. Brenner, D.W., Shenderova, O.A., Harrison, J.A., Stuart, S.J., Ni, B., Sinnott, S.B.: A second-generation reactive empirical bond order (rebo) potential energy expression for hydrocarbons. Journal of Physics: Condensed Matter **14**(4), 783 (2002)
11. Fogarty, J.C., Aktulga, H.M., Grama, A.Y., Van Duin, A.C., Pandit, S.A.: A reactive molecular dynamics simulation of the silica-water interface. The Journal of chemical physics **132**(17), 174704 (2010)
12. Guo, F., Wen, Y.S., Feng, S.Q., Li, X.D., Li, H.S., Cui, S.X., Zhang, Z.R., Hu, H.Q., Zhang, G.Q., Cheng, X.L.: Intelligent-ReaxFF: evaluating the reactive force field parameters with machine learning. Computational Materials Science **172**, 109393 (2020)
13. Kaymak, M.C., Rahnamoun, A., O'Hearn, K.A., Van Duin, A.C., Merz Jr, K.M., Aktulga, H.M.: JAX-ReaxFF: A Gradient-Based Framework for Fast Optimization of Reactive Force Fields. Journal of Chemical Theory and Computation **18**(9), 5181–5194 (2022)
14. Kylasa, S.B., Aktulga, H.M., Grama, A.Y.: PuReMD-GPU: A reactive molecular dynamics simulation package for GPUs. Journal of Computational Physics **272**, 343–359 (2014)
15. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 (2017)
16. Mortier, W.J., Ghosh, S.K., Shankar, S.: Electronegativity-equalization method for the calculation of atomic charges in molecules. Journal of the American Chemical Society **108**(15), 4315–4320 (1986)
17. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32** (2019)
18. Rahnamoun, A., Kaymak, M.C., Manathunga, M., Götz, A.W., Van Duin, A.C., Merz Jr, K.M., Aktulga, H.M.: Reaxff/amber—a framework for hybrid reactive/nonreactive force field molecular dynamics simulations. Journal of chemical theory and computation **16**(12), 7645–7654 (2020)
19. Ramakrishnan, R., Dral, P.O., Rupp, M., Von Lilienfeld, O.A.: Quantum chemistry structures and properties of 134 kilo molecules. Scientific data **1**(1), 1–7 (2014)
20. Rappe, A.K., Goddard III, W.A.: Charge equilibration for molecular dynamics simulations. The Journal of Physical Chemistry **95**(8), 3358–3363 (1991)

21. ReaxFF, S.: Theoretical chemistry (2020)
22. Sabne, A.: Xla: Compiling machine learning for peak performance (2020)
23. Schoenholz, S., Cubuk, E.D.: Jax md: a framework for differentiable physics. Advances in Neural Information Processing Systems **33**, 11428–11441 (2020)
24. Schütt, K., Kindermans, P.J., Sauceda Felix, H.E., Chmiela, S., Tkatchenko, A., Müller, K.R.: Schnet: A continuous-filter convolutional neural network for modeling quantum interactions. Advances in neural information processing systems **30** (2017)
25. Senftle, T.P., Hong, S., Islam, M.M., Kylasa, S.B., Zheng, Y., Shin, Y.K., Junkermeier, C., Engel-Herbert, R., Janik, M.J., Aktulga, H.M., et al.: The ReaxFF reactive force-field: development, applications and future directions. npj Computational Materials **2**(1), 1–14 (2016)
26. Tersoff, J.: Modeling solid-state chemistry: Interatomic potentials for multicomponent systems. Physical review B **39**(8), 5566 (1989)
27. Thompson, A.P., Swiler, L.P., Trott, C.R., Foiles, S.M., Tucker, G.J.: Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials. Journal of Computational Physics **285**, 316–330 (2015)
28. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., et al.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2021)
29. Van Duin, A.C., Dasgupta, S., Lorant, F., Goddard, W.A.: Reaxff: a reactive force field for hydrocarbons. The Journal of Physical Chemistry A **105**(41), 9396–9409 (2001)
30. Vazquez, F., Garzón, E.M., Martinez, J., Fernandez, J.: The sparse matrix vector product on GPUs. In: Proceedings of the 2009 International Conference on Computational and Mathematical Methods in Science and Engineering. vol. 2, pp. 1081–1092. Computational and Mathematical Methods in Science and Engineering Gijón, Spain (2009)
31. Vázquez, F., Fernández, J.J., Garzón, E.M.: A new approach for sparse matrix vector product on nvidia gpus. Concurrency and Computation: Practice and Experience **23**(8), 815–826 (2011)
32. Verstraelen, T., Ayers, P., Van Speybroeck, V., Waroquier, M.: ACKS2: Atom-condensed Kohn-Sham DFT approximated to second order. The Journal of chemical physics **138**(7), 074108 (2013)