KUSP: PYTHON SERVER FOR DEPLOYING ML INTERATOMIC POTENTIALS

Amit Gupta
Aerospace Engineering and Mechanics
University of Minnesota
gupta839@umn.edu

Ellad B. Tadmor *
Aerospace Engineering and Mechanics
University of Minnesota
tadmor@umn.edu

Stefano Martiniani †
Center for Soft Matter Research, Department of Physics, New York University, New York, 10003, USA
Simons Center for Computational Physical Chemistry, Department of Chemistry, New York University, New York, 10003, USA
Courant Institute of Mathematical Sciences, New York University, New York, 10003, USA
stefano.martiniani@nyu.edu

ABSTRACT

The KIM Utility for Serving Potentials (KUSP) is a Python package designed to facilitate the rapid deployment of machine learning (ML) interatomic potentials (MLIPs) to arbitrary molecular simulation codes. KUSP imposes minimal restrictions on ML architecture and libraries, and is thus compatible with popular ML frameworks such as PyTorch, JAX, and TensorFlow, as well as utilities like PyTorch Geometric and the Deep Graph Library. By providing a simulator-agnostic interface via the KIM API, KUSP allows researchers to quickly prototype and deploy their models in molecular simulation codes such as LAMMPS and ASE. Moreover KUSP enables the validation and benchmarking of models through OpenKIM "tests" (molecular simulation-based material property calculations) and "verification checks" (basic physical consistency checks), allowing for direct comparison with other MLIPS and classical physics-based interatomic potentials within the Open Knowledgebase of Interatomic Models (OpenKIM). KUSP employs a client-server architecture where the Python server communicates with the KIM API using sockets after converting model output to a KIM API-compatible format. These innovations are poised to propel the computational materials research community towards more efficient, accurate, reproducible and effective MLIP development and deployment.

1 INTRODUCTION

Machine learning research demands rapid and flexible development, but such flexibility often comes at the cost of interoperability with existing software infrastructure. The deployment of machine learning (ML) interatomic potentials (MLIPs) to existing molecular simulation codes ("simulators") is no exception. Considerable effort is spent making MLIPs available on various simulator packages. For instance, the LAMMPS (Thompson et al., 2022) simulator offers custom "pair_style" potentials for several MLIPs, including SNAP (Thompson et al., 2015), GAP (Csányi et al., 2007), HDNNP (Singraber et al., 2019), NequIP (Johansson et al., 2024), and MACE (MACE Development Team, 2024). The integration is not straighforward and some of these models (e.g., MACE (MACE Development Team, 2024)) only work with custom LAMMPS builds.

To address this challenge, we recently introduced the TorchML model driver (Gupta, 2024) to the OpenKIM repository. This driver leverages the PyTorch C++ API (libtorch) to deploy MLIPs to widely used simulation codes such as LAMMPS and ASE via the KIM API (Elliot, Ryan Elliot, Ellad Tadmor, et al.; OpenKIM Development Team, 2024b). Models deployed through the TorchML driver benefit from highly efficient performance, including parallelism across CPU and GPU architectures, and seamless deployment to production level codes. However, this approach is limited to PyTorch models, specifically those compatible with TorchScript, and it requires the models' inputs and outputs to comply with a specific format.

In order to support initial rapid model development and testing for arbitrary MLIPs without any resrtrictions, we introduce the KIM Utility for Serving Potentials (KUSP), a Python-only server and KIM API compliant protocol inspired by TorchServe (PyTorch Foundation, 2024) and the i-Pi universal force engine (Kapil et al., 2019). KUSP takes advantage of Python's dynamism and flexibility offering the following standout features: (i) compatibility with multiple ML frameworks; (ii) ability to interface with the KIM API, enabling users to deploy their models to arbitrary simulation codes, as well as validate and benchmark their

*Equal contribution.
†Equal contribution.

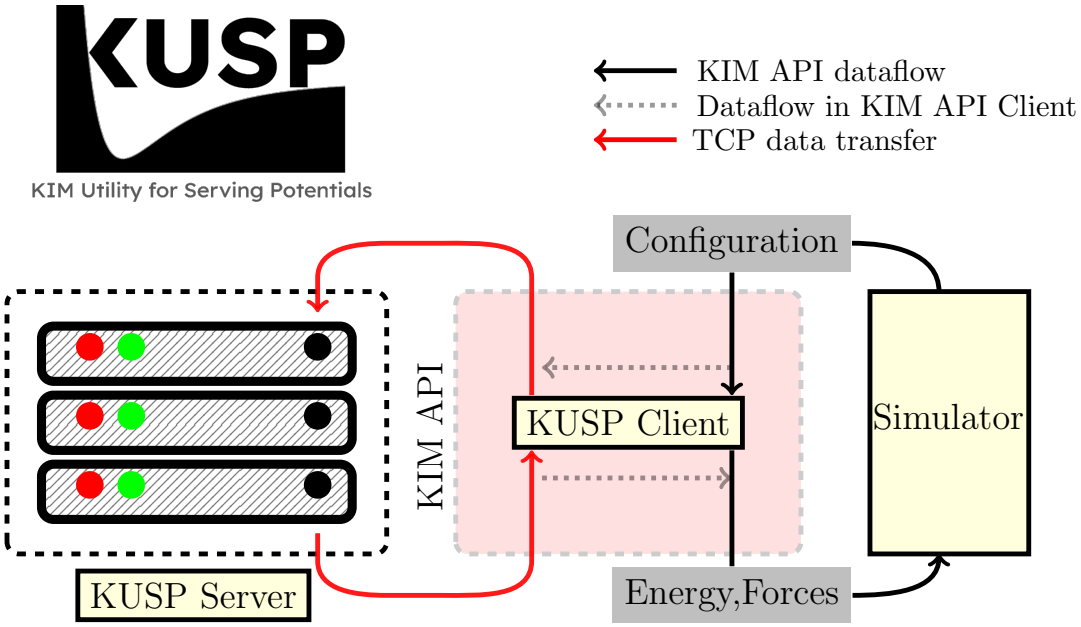

Figure 1: KUSP design schematic, showing the KUSP server and KIM API client (`KUSP__MO_000000000000_000`). The client interfaces with the simulation code using the KIM API, and the transfers the data to the server using the KUSP protocol over the sockets interface. The server then deploys the ML model and returns the results to the client, which the client then passes to the simulation code again using the KIM API.

models using OpenKIM "tests" (molecular simulation-based material property calculations) and "verification checks" (basic physical consistency checks); and (iii) ease of use.

Server-client architectures have been widely employed for problems where the same data is requested in different formats by different clients (Microsoft, 2024). KUSP aims to provide such a solution for deploying MLIPs, decoupling entirely the development of simulators from that of models. The KUSP server wraps an MLIP model that computes energies and forces from a known set of inputs (atomic species and positions, and a boolean mask indicating whether an atom is contributing to the energy or provided as padding), and the client interfaces with the desired simulator via the KIM API, see Fig. 1 for a schematic.

The KUSP package includes a C++ client that leverages the KIM API. This client functions as a thin wrapper around KIM models: it transfers simulator data to the KUSP server and then returns the processed results back to it (Fig. 1). Consequently, users only need to implement the server to wrap their MLIP.

In the rest of the paper we discuss the design of the KUSP server and client in detail, and demonstrate KUSP usage across ML frameworks and simulator packages.

2 Design

KUSP is designed for ease of use while delivering high performance. To achieve these goals, the KUSP server is implemented as a Python-only object (`KUSPServer`), leveraging the KIM API on the client side in C++. A simulator employing an MLIP via KUSP uses the designation `KUSP__MO_000000000000_000` in its input script when specifying the interatomic potential being used. For example, in LAMMPS, the following commands are used (OpenKIM Development Team, 2024a):

```
kim init KUSP__MO_000000000000_000 <unit_system>
...
kim interactions <species_to_atom_types>
```

where `<unit_system>` is replaced by the LAMMPS unit system used by the MLIP, and `<species_to_atom_types>` is replaced by a list mapping atomic species to LAMMPS atom types.

The client hooks into KIM-compatible simulators (OpenKIM Development Team, 2024d) (e.g., LAMMPS, ASE) and passes the coordinates, species and contributing atom mask to the KUSP server, which deploys the ML model, Fig. 1. The connection of

the server to the MLIP is achieved by overloading the `KUSPServer` class and implementing the `prepare_model_inputs`, `prepare_model_outputs`, and `execute_model` methods. These methods recast the raw information received from the KIM API client into a format that the MLIP model understands, and vice versa. Data is transferred between the KIM API client and the KUSP server using the sockets protocol. The standard `socket` module in Python (Hunt, 2019) is adopted to keep dependencies to a minimum.

The KUSP communication protocol is simple, it assumes the byte (`b`) sequence to be formatted as follows,

```
<int_width:4b>
<n_atoms:(int_width)b>
<atomic_numbers:(n_atoms x int_width)b>
<positions:(n_atoms x 3 x 8)b>
<contributing_atoms:(n_atoms x int_width)b>
```

Here, `<var:width>` represents the expected variable, `var`, and its expected width in bytes. The `int_width` provides the integer size on the system and is utilized to convert bytes into integers. Next is the number of atoms in the configuration, the atomic numbers (species), positions, and contributing status of the atoms [1]. If no contributing atoms are provided, the server assumes all atoms are contributing to the energy and forces. The model outputs are provided in a similar format, with mandatory energy and forces, and optional virial stress.

The KUSP server expects integers to be of system integer size (specified by `int_width`), and all floating point values (inputs and outputs) to be in double precision. For single precision MLIPs, the positions must be converted to single precision in the `prepare_model_inputs` before being passed to the model, and the outputted energy, forces, and stress must be converted to double precision in `prepare_model_outputs`. For more details on the input and output data specifications, see the supplementary material (Section A1, Table 1 and Table 2).

For enhanced performance, `KUSPServer` offers the option of using a shared memory buffer to transfer data between the KUSP client and server. This method allows the client and server to exchange only the memory buffer addresses for positions, forces, and other data. This reduces the overhead for data transfer and is particularly useful when simulating large systems. When using this option only the memory buffer name is transferred over sockets followed by a `memcpy` operation to copy the data from the non-shared simulator memory to the shared buffers [2]. Currently, this option has an additional dependency (Boost C++ library) and requires a valid environment setup.

While the KUSP server-client design may appear inefficient for running molecular simulations (e.g., molecular dynamics), it is important to note that for typical production-level MLIPs the bottleneck is model inference, and not the data transfer. Consequently, while KUSP may not be the method of choice for integrating classical potentials, it offers a good trade-off between flexibility and performance for MLIPs. In Fig. 2 we show the time taken for model evaluation and data transfer using the NequIP model (Batzner et al., 2022), as it is implemented in the official `nequip` package (Group, 2024) and deployed using the KUSP server. The results clearly demonstrate that the data transport time is ≈ 0.5 % of the total time.

2.1 Configuration File

KUSP requires a configuration file to correctly launch the server and client. KUSP looks for this file in the current working directory or in the environment variable `KUSP_SERVER_CONFIG` if provided. The configuration file is a YAML file with mandatory and optional fields, which is divided into two sections: `server` and `global`.

```yaml

server:
    host: 127.0.0.1
    port: 12345
    optional:
      mode: IP
      max_connections: 1 # Maximum number of connections to the server
      buffer_size: 1024 # Buffer size for the server

global:
  elements:
    - Si
    - O
```

[1] Contributing atoms are those included in the energy calculation. Non-contributing atoms are determined by the boundary conditions and are provided as padding atoms to contributing atoms (OpenKIM Development Team, 2024b).

[2] An explicit `memcpy` is needed for shared memory buffers because existing memory allocations in a simulator cannot be re-cast as shareable, and any shared buffer has to be declared as such before requesting allocation.

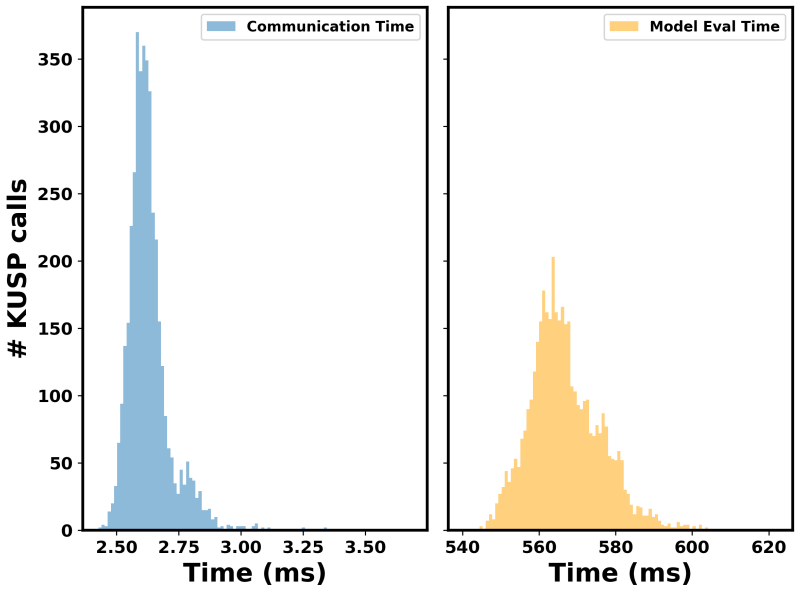

Figure 2: Histograms showing distribution of time taken in socket data transfer (left), compared with time taken by KUSP to evaluate energy and forces (right) for a 5000 step simulation of 13824 Si atoms. (MLIP evaluations were performed on a single A100 GPU, configurations were transferred from a CPU-only version of LAMMPS.)

```
influence_distance: 6.0
```

The `server` section contains information about the server, data transfer protocol, buffer sizes, etc. The required fields are `host` and `port` defining the host URL and the port for the KUSP server. Users should choose these values carefully as not all ports are open for use, and some might require root access. Optional fields include connection `mode`, `max_connections` and `buffer_size`. The connection mode can have one of three values: `IP`, `UNIX`, or `SHM`, for TCP/IP socket, UNIX sockets, and shared memory buffer, respectively. Currently only the `IP` and `SHM` modes are supported. UNIX sockets are planned for a future release. Choosing appropriate values for the maximum connections and buffer size can be crucial for the performance of the server. In working environment, with large network activity, and with low-latency networks (e.g. infiniband), smaller buffer size might show better performance, whereas in environments with low network usage, and high-latency (e.g. ethernet), larger buffer size will help in achieving higher performance.

The `global` section contains global information about the system, required by the server and/or client. The `elements` and `influence_distance` fields are mandatory, and arbitrary additional fields can be included as needed by the MLIP. The `elements` field is a list of chemical elements (species) that the MLIP supports; the KUSP client will enumerate the species using this list (e.g., in above configuration, silicon (`Si`) is index 0, and oxygen (`O`) is index 1). It is important to list all supported elements, as most MLIPs internally map these elements to a fixed index number. If an incomplete list of elements is provided, then these indices might not match, resulting in wrong results. For example, if a model supports `Si`, `Ti`, and `O`, the model might index them as (0, 1, 2) respectively. However, for simulating TiO_2 if only `Ti` and `O` are included in the element list, KUSP will index these two elements as (0, 1) respectively, which will result in incorrect values.

The `influence_distance` field is used to determine the cutoff distance for the model. For graph convolution MLIPs, it is advisable to use an influence distance calculated as the number of convolution layers multiplied by the cutoff distance of the model. This ensures that the model has all the information to compute the forces and energies correctly. Users can also provide additional information required by their model, such as cell size, energy scaling factors, inference mode, and so on.

3 EXAMPLES

In this section, we provide a simple example of deploying an MLIP for Si using KUSP. We chose the default NequIP equivariant GNN model implementation for the example, and compared its results with OpenKIM TorchML NequIP model, which has been validated extensively (Gupta et al., 2024).

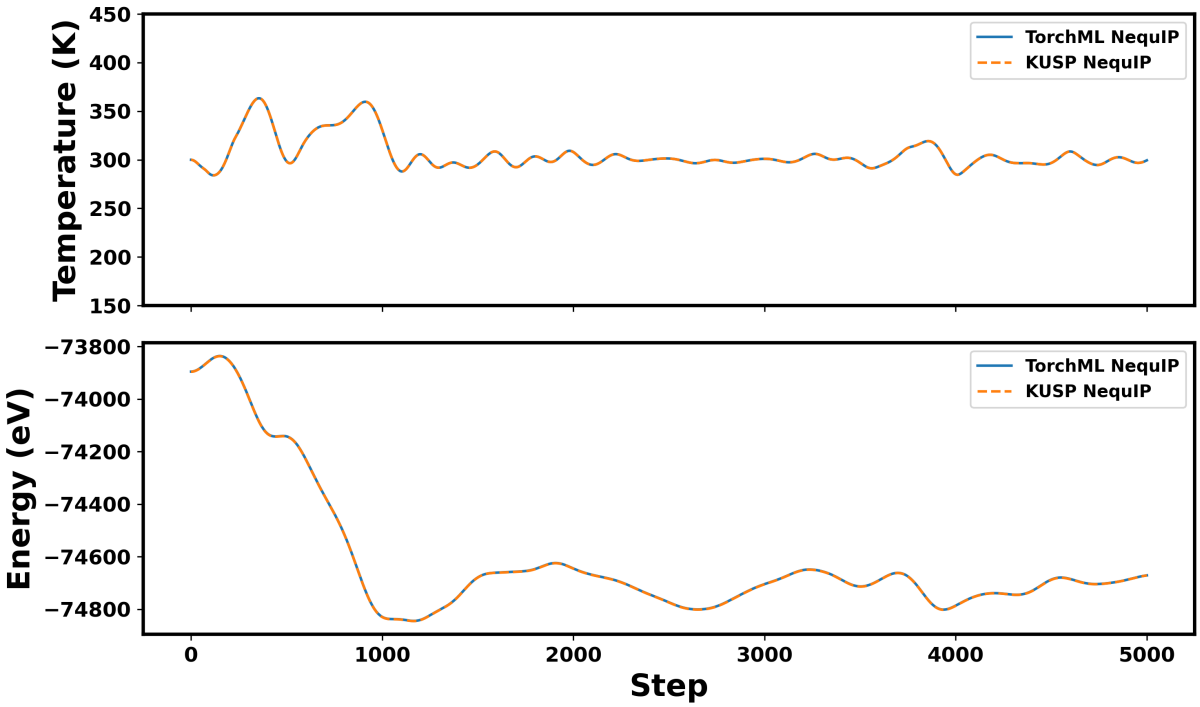

Figure 3: LAMMPS molecular simulation of 13,824 Si atoms for 5000 time steps under NVT conditions, using the NequIP MLIP deployed via the KIM TorchML model driver (Gupta et al., 2024), and KUSP. The KUSP server deploys the TorchScript model file, as obtained thought `nequip-deploy` command, without any modification. Both simulations were run using LAMMPS, and ML model was evaluated on a single A100 GPU.

To highlight the generalizability of KUSP, we also demonstrate running an MD simulation using JAX-MD with a Stillinger–Weber (SW) interatomic potential for Si, and compare obtained results with the OpenKIM SW model.

3.1 NEQUIP-KUSP SERVER

NequIP (Batzner et al., 2022) creates advanced equivariant MLIPs using spherical tensor products. We chose it for KUSP due to its popularity and complex design, making it ideal to showcase KUSP's simplicity. Below, we explain how to use a pre-trained NequIP potential in KUSP.

3.1.1 INITIALIZATION __INIT__

The NequIP server is created by extending the `KUSPServer` class and passing the YAML configuration file and the MLIP model to the `__init__` method.

```
class NequIPServer(KUSPServer):
def __init__(self, model, configuration):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    self.device = device
    model = model.double()
    model = model.to(device)

    super().__init__(model, server_config)

    self.cutoff = self.global_information['influence_distance']/3
    self.species = self.global_information['elements']
```

Here the model is transferred to the GPU first, and then the GPU copied model is provided as in input to the parent class. The element supported by this model (Si) is defined in the `species` field. All of the information in the `global` section of the KUSP configuration file is accessible through the `global_information` attribute.

3.1.2 FORMATTING MODEL INPUTS AND OUTPUTS

When developing a KUSP server for a new MLIP, two methods must be implemented: `prepare_model_inputs` and `prepare_model_outputs`.

The `prepare_model_inputs` function converts KIM API client output (atomic numbers, atom positions, contributing atoms) into valid structured data in the MLIP's expected format. By default the model in KUSP expects inputs as a dictionary that is then fed to the MLIP as unrolled keyword arguments. If the model inputs require additional processing, this can be done by extending the `execute_model` method that executes the MLIP and returns the model outputs.

```
def prepare_model_inputs(self, atomic_numbers, positions, contributing_atoms):
    species = [self.species[atomic_number] for atomic_number in atomic_numbers]

    graph = graph_generator(species, positions, )

    # NequIP input dictionary
    # required inputs: "pos" "edge_index" "edge_cell_shift" "cell" "atom_types"
    input_dict = {
        "pos": torch.tensor(positions, dtype=torch.float64, requires_grad=True, device=self.
    device),
        "cell": torch.tensor(graph.cell, dtype=torch.float64, device=self.device),
        "atom_types": torch.tensor(atomic_numbers, dtype=torch.long, device=self.device),
        "edge_index": torch.tensor(graph.edge_index0, dtype=torch.long, device=self.device),
        "edge_cell_shift": torch.zeros((graph.edge_index0.shape[1] ,3), dtype=torch.float64,
    device=self.device),
        "_contributing_atoms": torch.tensor(contributing_atoms, dtype=torch.float64, device=
    self.device), # for later use
    }
    return {"input_dict": input_dict}
```

Here the `graph_generator` is a function that maps the configurations to the edge graphs that NequIP takes as input. Several popular libraries, such as `ase` (Bahn & Jacobsen, 2002), `pymatgen` (Ong et al., 2013), and `kliff` (Wen et al., 2022) provide routines that can be used to that end.

The `prepare_model_outputs` function converts MLIP output (energy, forces, virial stress) to KIM API compatible format. In most cases, this simply involves a conversion from the model outputs to numpy arrays. In the provided NequIP example, this function is also used to compute the gradients for force calculation (`backward()` function).

```
def prepare_model_outputs(self, output):
    energy = ((output['atomic_energy'].squeeze()) * output["_contributing_atoms"]).sum()
    energy.backward()
    # Extract the gradients
    forces = -output["pos"].grad

    return {"energy": energy.detach().cpu().numpy(),
            "forces": forces.detach().cpu().numpy()}
```

3.1.3 RUNNING THE NEQUIP-SERVER

The NequIP server uses the deployed TorchScript model as obtained from the `nequip` (Batzner et al., 2022; Geiger et al., 2022) python package, using the `nequip-deploy` command. The NequIP model was trained using the PRX GAP Si dataset (Bartók et al., 2018), with 3 convolution layers, and spherical tensors of maximum order 1.

```
model = torch.jit.load("trained_si_model.pt")

# Only evaluate first children of model, for atomwise energies
model = list(list(model.children())[0].children())[0]
server = NequIPSever(model=model, configuration="kusp_config.yaml")
server.serve()
```

In the above example, the MLIP is loaded from the deployed trained model file `trained_si_model`, and the server configuration is loaded from the `kusp_config.yaml` file.

3.1.4 RUNNING SIMULATIONS AND COMPUTATIONS

Once the model is deployed via the KUSP server, it can be accessed as a regular KIM portable model (OpenKIM Development Team, 2024c) using the KIM ID `KUSP__MO_000000000000_000`, which points to the KUSP client. The KUSP client is bundled with the KUSP python package and can be installed using the `kusp.install_kim_model()` method.

Figure 3 shows the simulation results for 13,284 Si atoms integrated for 5000 MD steps (0.1fs timestep) under NVT conditions using a Nosé-Hoover thermostat. Temperature was fixed at 300K. Two simulations were run using identical MLIPs weights and LAMMPS random seed, using both KUSP and the OpenKIM TorchML model driver (Gupta et al., 2024) for comparison. The TorchML-deployed NequIP model (Model id: `MO_196181738937_000` in OpenKIM) has been extensively validated, and hence forms the perfect baseline to identify any issues in KUSP. Our experiments show that both simulations run identically, barring negligible differences originating from floating point arithmetic. Excerpts from the input script are given below.

```
# Initialize KIM Model
kim init KUSP__MO_000000000000_000 metal
kim interactions Si

# Fix thermostat and run simulation
fix 1 all nvt temp 300.0 300.0 $(100.0*dt)
run    5000
```

As the KUSP client supports all KIM API compatible simulators, the same model can be executed in ASE, by simply calling the KIM model pointing to the KUSP client in an ASE calculator.

```
from ase.calculators.kim import KIM

# Initialize KIM Model
model = KIM("KUSP__MO_000000000000_000")

# Set it as calculator
config.set_calculator(model)

# Compute energy/forces
energy = config.get_potential_energy()
```

3.2 JAX-MD SERVER

As model evaluation is performed completely in a Python environment, KUSP does not depend on any specific ML package or model architecture, rather it supports all packages and libraries that provide a Python API.

As an example, we demonstrate a KUSP server based deployment of the SW interatomic potential for Si, using the JAX framework based differentiable MD package, JAX-MD (Figure 4). The potential has been minimally modified to provide per-particle energy (Supplementary Information Section A2).

For validation purposes, the same simulation was also run with the same random seed using the OpenKIM SW potential (Singh, 2021). The simulations were run with 64 Si atoms, for 5000 steps (timestep = 0.1fs), using Nosé-Hoover thermostat (T=300K). Both simulations (OpenKIM and JAX-MD KUSP) are in excellent agreement. This highlights the flexibility and the utility of our approach.

3.3 GPU SUPPORT

KUSP is a platform agnostic tool, and can be run on any system with minimal dependencies. KUSP does not enable or hinder any accelerated hardware support, such as GPUs. KUSP provides the input data as numpy arrays in CPU memory. Users can use any library that supports GPU acceleration to perform the computation on the GPU. For GPU computations, users will typically need to transfer the model to the GPU before running the `serve()` method, and manually transfer the input data to the GPU in the `prepare_model_inputs()` or `execute_model()` methods. The output of the models then needs to be transferred back to the CPU memory in the `prepare_model_outputs()` method, and converted to the numpy array before returning the output.

3.4 OPENKIM TESTS AND VERIFICATION CHECKS

OpenKIM project provides an extensive suite of tests (evaluation of a standard material property) and verification checks (VCs) (physical correctness of the models). These tests and VCs are crucial for developing ML models, as the vast majority of MLIPs

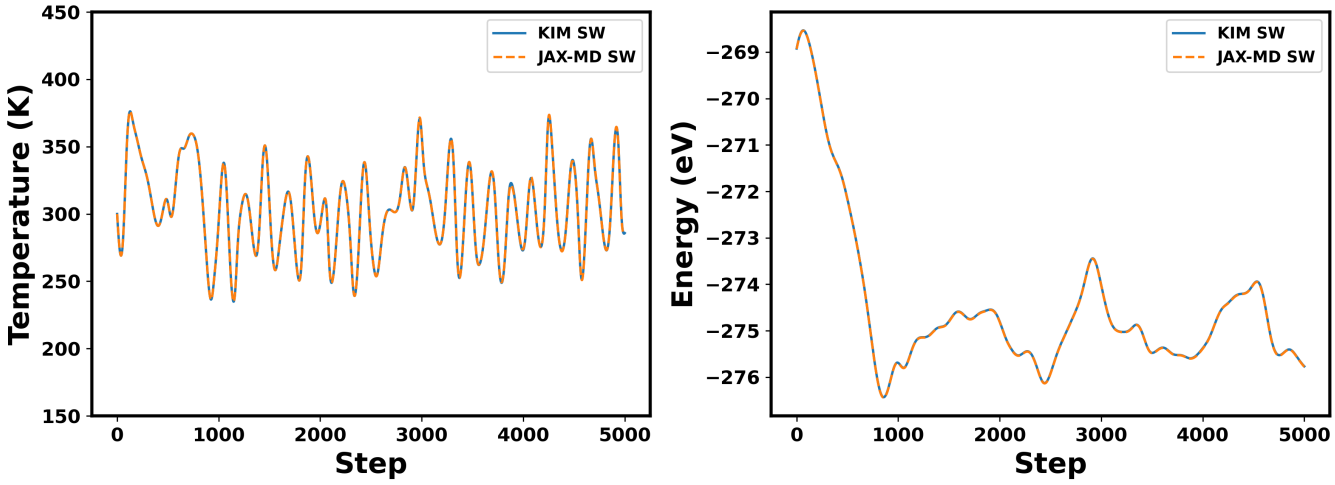

Figure 4: MD simulation of a 64 atom Si cell using the JAX-MD SW potential, deployed via KUSP. The results are compared to an OpenKIM SW potential.

are only benchmarked against energy and forces of the validation dataset. KUSP provides a convenient approach for employing these tests and VCs. As the KUSP client is fully compatible with the KIM API, any potential deployed through KUSP can be tested like a standard KIM model.

To demonstrate this, we ran the OpenKIM *Ojectivity VC* (Tadmor, 2019) using the JAX-MD server for deploying the model. The Objectivity VC verifies that a potential is invariant with respect to rigid-body translation and rotation. The VC calculations were ran inside the KIM Developer Platform (KDP) docker image provided by the OpenKIM project (Karls et al., 2020) modified to include a `species` segment in the specification file. The VC is run using the KDP command: `pipeline-run-pair Objectivity__VC_813478999433_002 KUSP__MO_000000000000_000 -v`. The output of the VC is appended in the Supplementary Information Section A3.

4 Remaining Challenges

The main challenge in adopting KUSP arises from the purely local formalism adopted by the KIM API. Most interatomic potentials, including MLIPs and ab-initio methods, follow the principle of "near-sightedness" (Kohn, 1996), whereby an atom's properties are influenced only by its neighboring environment. This approximation simplifies the computation of the potential energy and allows for parallelization through domain decomposition. The KIM API utilizes this near-sightedness for its simulator-agnostic model implementations, requiring compliant models to accept a cluster of atoms with non-contributing padding atoms carrying boundary condition information.

This KIM API requirement does not work well with all MLIP implementations. For instance, Spookynet (Unke et al., 2021) aggregates a global feature vector to compute the energy, requiring additional information about system periodicity and cell size. Even some local MLIPs require cell vectors and atom positions to compute unwrapped atomic distances internally, thus requiring the specification of cell size and cell vectors that KUSP does not provide. Workarounds like using larger cell sizes and contributing atom information is possible, but getting these models to work can be complex.

5 Conclusion

KUSP is a Python package designed to facilitate the rapid deployment of MLIPs to arbitrary simulation codes. It imposes minimal restrictions on ML architectures and libraries, and as we have shown, it is compatible with popular ML frameworks such as PyTorch, JAX, and TensorFlow, as well as utilities like PyTorch Geometric and the Deep Graph Library. By providing a simulator-agnostic interface via the KIM API, KUSP allows researchers to quickly prototype and deploy their models to production level simulation platforms.

In upcoming coming releases we plan to improve the performance of the KUSP server by adding multithreading support for parallel model evaluation, and expanding to more efficient data transfer protocols like UNIX sockets. Additionally, we plan to

support the publication of self-contained KUSP models (using the Python C++ API) on the OpenKIM repository (Ope, 2024) for evaluation, benchmarking, and reuse. We will also provide examples for more ML frameworks and simulation codes.

CODE AVAILABILITY

Source code for KUSP is available at `https://github.com/openkim/kusp`, and the documentation is available at `https://kusp.readthedocs.io/`. Users can install KUSP using `pip install kusp`.

ACKNOWLEDGEMENTS

The authors acknowledge partial support by the National Science Foundation, USA (NSF) under grants No. DMR-1834251, OAC-2039575, OAC-2311632, DMR-2333411.

REFERENCES

Sune P. Bahn and Karsten W. Jacobsen. The Atomic Simulation Environment—a Python library for working with atoms. *Computational Science & Engineering*, 3(2):15–25, 2002.

Albert P Bartók, James Kermode, Noam Bernstein, and Gábor Csányi. Machine learning a general-purpose interatomic potential for silicon. *Physical Review X*, 8(4):041048, 2018.

Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E Smidt, and Boris Kozinsky. E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature communications*, 13(1):2453, 2022.

Gábor Csányi, Steven Winfield, J R Kermode, A De Vita, Alessio Comisso, Noam Bernstein, and Michael C Payne. Expressive programming for computational physics in fortran 95+. *IoP Comput. Phys. Newsletter*, pp. Spring 2007, 2007.

Ryan Elliot. KIM Application Programming Interface (API). `https://github.com/openkim/kim-api`, Ryan Elliot, Ellad Tadmor, et al. [Accessed 03-07-2024].

Mario Geiger, Tess Smidt, Alby M., Benjamin Kurt Miller, Wouter Boomsma, Bradley Dice, Kostiantyn Lapchevskyi, Maurice Weiler, Michał Tyszkiewicz, Martin Uhrin, Simon Batzner, Dylan Madisetti, Jes Frellsen, Nuri Jung, Sophia Sanborn, jkh, Mingjian Wen, Josh Rackers, Marcel Rød, and Michael Bailey. e3nn/e3nn: 2022-12-12, December 2022. URL `https://zenodo.org/record/7430260`.

MIR Group. Github Repo - NequIP: E(3)-Equivariant Interatomic Potentials. `https://github.com/mir-group/nequip`, 2024. Accessed: 2024-07-24.

Amit Gupta. TorchML model driver v000. `https://openkim.org/id/TorchML__MD_173118614730_000`, 2024.

Amit Gupta, Ellad B. Tadmor, and Stefano Martiniani. TorchML NequIP for silicon. `https://doi.org/10.25950/d7a965ba`, 2024. URL `https://openkim.org/id/TorchML_NequIP_GuptaTadmorMartiniani_2024_Si__MO_196181738937_000`. Accessed: 2024-07-24.

John Hunt. Sockets in python. In *Advanced Guide to Python 3 Programming*, pp. 457–470. Springer, Cham, 2019. doi: 10.1007/978-3-030-25943-3_38. URL `https://doi.org/10.1007/978-3-030-25943-3_38`.

Anders Johansson, Albert Musaelian, and Lixin Sun. pair_nequip: LAMMPS pair style for NequIP. `https://github.com/mir-group/pair_nequip`, 2024.

Venkat Kapil, Mariana Rossi, Ondrej Marsalek, Riccardo Petraglia, Yair Litman, Thomas Spura, Bingqing Cheng, Alice Cuzzocrea, Robert H Meißner, David M Wilkins, et al. i-PI 2.0: A universal force engine for advanced molecular simulations. *Computer Physics Communications*, 236:214–223, 2019.

Daniel S Karls, Matthew Bierbaum, Alexander A Alemi, Ryan S Elliott, James P Sethna, and Ellad B Tadmor. The openkim processing pipeline: A cloud-based automatic material property computation engine. *The Journal of Chemical Physics*, 153 (6), 2020.

Walter Kohn. Density functional and density matrix method scaling linearly with the number of atoms. *Physical Review Letters*, 76(17):3168, 1996.

MACE Development Team. *MACE in LAMMPS*, 2024. https://mace-docs.readthedocs.io/en/latest/guide/lammps.html.

Microsoft. *Language Server Protocol*, 2024. https://microsoft.github.io/language-server-protocol/.

Shyue Ping Ong, Shreyas Cholia, Anubhav Jain, Mirko Brafman, Dan Gunter, Gerbrand Ceder, and Kristin A. Persson. Pymatgen: A robust, open-source python library for materials analysis. *Computational Materials Science*, 68:314–319, 2013.

OpenKIM Repository. OpenKIM, 2024. https://openkim.org/.

OpenKIM Development Team. *kim command*. Sandia National Laboratories, 2024a. https://https://docs.lammps.org/kim_commands.html.

OpenKIM Development Team. *KIM API documentation*. OpenKIM, 2024b. https://kim-api.readthedocs.io/en/latest/.

OpenKIM Development Team. *Types of KIM Content*. OpenKIM, 2024c. https://openkim.org/doc/repository/kim-content/.

OpenKIM Development Team. *Software and Projects using KIM*. OpenKIM, 2024d. https://openkim.org/projects-using-kim/.

PyTorch Foundation. Torchserve. https://pytorch.org/serve/, 2024. Accessed: 2024-06-20.

Amit K. Singh. SW_StillingerWeber_1985_Si__MO_405512056662_006. https://openkim.org/id/SW_StillingerWeber_1985_Si__MO_405512056662_006, 2021.

Andreas Singraber, Jörg Behler, and Christoph Dellago. Library-based LAMMPS implementation of high-dimensional neural network potentials. *Journal of chemical theory and computation*, 15(3):1827–1840, 2019.

Ellad B. Tadmor. Verification check of invariance with respect to rigid-body motion (objectivity) v002. OpenKIM, https://doi.org/10.25950/7e538eba, 2019.

A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Comp. Phys. Comm.*, 271:108171, 2022. doi: 10.1016/j.cpc.2021.108171.

Aidan P Thompson, Laura P Swiler, Christian R Trott, Stephen M Foiles, and Garritt J Tucker. Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials. *Journal of Computational Physics*, 285:316–330, 2015.

Oliver T Unke, Stefan Chmiela, Michael Gastegger, Kristof T Schütt, Huziel E Sauceda, and Klaus-Robert Müller. Spookynet: Learning force fields with electronic degrees of freedom and nonlocal effects. *Nature communications*, 12(1):7273, 2021.

Mingjian Wen, Yaser Afshar, Ryan S. Elliott, and Ellad B. Tadmor. KLIFF: A framework to develop physics-based and machine learning interatomic potentials. *Computer Physics Communications*, 272:108218, 2022. doi: 10.1016/j.cpc.2021.108218.

A SUPPLEMENTARY MATERIAL

A.1 KUSP INPUT AND OUTPUT DATA

Field	Description	Required
`int_width`	Size of integer on the system	✓
`n_atoms`	Number of atoms	✓
`atomic_numbers`	Atomic numbers of atoms	✓
`positions`	Positions of atoms	✓
`contributing_atoms`	Atoms to compute energy for	x

Table 1: KUSP input data fields. These data items are received by the server from the client.

Field	Description	Required
`energy`	Energy of the system	✓
`forces`	Forces on the atoms	✓
`virial`	Virial tensor	x

Table 2: KUSP output data fields. These data items are sent by the server to the client.

A.2 JAX-MD SERVER

JAX function to get per-atom energies and forces from an SW potential,

```python
import jax_md
import jax_md.space as space
import jax_md.energy as energy

def stillinger_weber_per_atom(
        displacement: Callable,
        sigma: float = 2.0951,
        A: float = 7.049556277,
        B: float = 0.6022245584,
        lam: float = 21.0,
        gamma: float = 1.2,
        epsilon: float = 2.16826,
        three_body_strength: float = 1.0,
        cutoff: float = 3.77118) -> Callable[[Array], Array]:
    """
    Compute the per-atom energy of a Stillinger-Weber potential.
    This is the same function as jax_md.energy.stillinger_weber,
    but it returns the per-atom energy by not calling the
    jax_md.utils.high_precision_sum function.
    """
    two_body_fn = partial(energy._sw_radial_interaction, sigma, B, cutoff)
    three_body_fn = partial(energy._sw_angle_interaction, gamma, sigma, cutoff)
    three_body_fn = vmap(vmap(vmap(three_body_fn, (0, None)), (None, 0)))

    def compute_fn(R, **kwargs):
        d = partial(displacement, **kwargs)
        dR = space.map_product(d)(R, R)
        dr = space.distance(dR)
        two_body_energy = jnp.sum(two_body_fn(dr), axis=1) * A / 2.0
        three_body_energy = jnp.sum(jnp.sum(three_body_fn(dR, dR), axis=2), axis=1) * lam / 2.0
        per_atom_energy = epsilon * (two_body_energy + three_body_strength * three_body_energy)
        return per_atom_energy
    return compute_fn
```

A.2.1 JAX-MD KUSP SERVER IMPLEMENTATION

```python
def sum_per_atom_energy_and_force(energy_fn, positions, contributions):
    """Sum the per-atom energy and force."""
    per_atom_energy = energy_fn(positions)
    per_atom_energy *= contributions
    total_energy = jnp.sum(per_atom_energy)
    forces = -grad(lambda R: jnp.sum(energy_fn(R) * contributions))(positions)
    return total_energy, forces


class JAXMDServer(KUSPServer):
    def __init__(self, model, server_config):
        super().__init__(model, server_config)

    def prepare_model_inputs(self, atomic_numbers, positions, contributing_atoms):
        pos = jnp.array(positions)
        contributing_atoms = jnp.array(contributing_atoms)
        return {"pos": pos, "contributing_atoms": contributing_atoms}

    def execute_model(self, pos, contributing_atoms):
        e, f = sum_per_atom_energy_and_force(self.exec_func, pos, contributing_atoms)
        return {"energy": e, "forces": f}

    def prepare_model_outputs(self, e_and_f):
        # print(e_and_f)
        numpy_array = {"energy": np.array(e_and_f["energy"]),
                       "forces": np.array(e_and_f["forces"])}
        return numpy_array
```

A.3 OPENKIM OBJECTIVITY VERIFICATION CHECK

Output of the OpenKIM Objectivity VC (Tadmor, 2019) for the SW potential deployed on JAX-MD via KUSP.

```
pipeline-run-pair Objectivity__VC_813478999433_002 KUSP__MO_000000000000_000  -v
+ Running pair (Objectivity__VC_813478999433_002, KUSP__MO_000000000000_000)

Model Extended KIM ID =
=== Verification check vc-objectivity start (2024-07-25 02:41:06) ===
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!                                         !!!!!
!!!!!  VERIFICATION CHECK: vc-objectivity    !!!!!
!!!!!                                         !!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Description: Check whether a model is invariant with respect to rigid-body
             motion (translation and rotation) as required by objectivity
             (material frame-indifference). This is expected to be true for any
             model that does not depend on an external field. The check is
             performed for a randomly distorted non-periodic body-centered cubic
             (BCC) cube base structure. Separate configurations are tested for
             each species supported by the model, as well as one containing a
             random distribution of all species.  The energy and forces of each
             configuration is compared with that of the same configuration
             rotated about a random axis by an irrational angle and translated
             in a random direction by an irrational distance. The verification
             check will pass if the energy of all configurations that the model
             is able to compute are invariant and the forces are mapped back by
             the inverse rotation. Configurations used for testing are provided
             as auxiliary files.

Author: Ellad Tadmor
```

```

--------------------------------------------------------------------------------------------------

Results for KIM Model       : KUSP__MO_000000000000_000
Supported species           : Si

random seed                 = 13
lattice constant (orig)     = 3.000
perturbation amplitude      = 0.300
number unit cells per side = 2
--------------------------------------------------------------------------------------------------



MONOATOMIC STRUCTURE -- Species = Si   (Configuration in file "config-Si.xyz")
--------------------------------------------------------------------------------------------------

Rotation matrix     = -8.10696825e-02  -9.63724213e-01   2.54289888e-01
                      -7.12053926e-01  -1.22522411e-01  -6.91351911e-01
                       6.97428787e-01  -2.37115793e-01  -6.76290757e-01

Translation vector = -1.47787265e+00  -1.76500948e+00  -2.13781159e+00

Energy requirement:

V(Q*r_1+c,...,Q*r_N+c) = V(r_1,...,r_N), where r_i is the position of atom i, V is the potential energy,
Q is a rotation, and c is a translation vector.

V(Q*r_1+c,...,Q*r_N+c) = -26.236466351752792
V(Q*r_1,...,Q*r_N)     = -26.236466351752792
V(r_1,...,r_N)         = -26.236466351752828

Forces requirement:

f_i(Q*r_1+c,...,Q*r_N+c) = Q*f_i(r_1,...,r_N), where r_i is the position of atom i, f_i is the force
on atom i, Q is a rotation matrix, and c is a translation vector.

  i                 f_i(Q*r_1+c,...,Q*r_N+c)                                    Q*f_i(r_1,...,r_N)
--------------------------------------------------------------------------------------------------

  0    -3.05472553e-01     1.59863626e+00    -6.82521580e-01    |  -3.05472553e-01     1.59863626e+00    -6.82521580e-01
  1     4.52265728e+00     8.40006941e+00    -5.31210163e+00    |   4.52265728e+00     8.40006941e+00    -5.31210163e+00
  2     3.37312585e+00     5.36869515e+00     7.23699808e+00    |   3.37312585e+00     5.36869515e+00     7.23699808e+00
  3    -2.29370169e+00    -6.46030167e+00     9.35730755e+00    |  -2.29370169e+00    -6.46030167e+00     9.35730755e+00
  4     1.56664671e+00     5.14701730e+00    -2.47428793e+00    |   1.56664671e+00     5.14701730e+00    -2.47428793e+00
  5    -1.24561557e+01     4.05701445e+00    -1.22131088e+01    |  -1.24561557e+01     4.05701445e+00    -1.22131088e+01
  6    -2.96961678e+00     2.48668813e+00     1.79109057e+01    |  -2.96961678e+00     2.48668813e+00     1.79109057e+01
  7    -2.28461642e+00    -3.05050457e+00     3.03460287e+00    |  -2.28461642e+00    -3.05050457e+00     3.03460287e+00
  8     1.26706235e+00     1.64418120e-01    -2.47374134e+00    |   1.26706235e+00     1.64418120e-01    -2.47374134e+00
  9     2.91804451e+00     5.78725426e-01    -2.96895503e+00    |   2.91804451e+00     5.78725426e-01    -2.96895503e+00
 10     9.25432325e+00    -5.49482452e+00    -1.26990575e+00    |   9.25432325e+00    -5.49482452e+00    -1.26990575e+00
```

```
 11     2.97783788e-03    -3.20681936e+00     3.17145812e-01    |    2.97783788e-03    -3.20681936e
    +00     3.17145812e-01
 12     1.23008899e+00     5.16105563e+00    -3.95721654e+00    |    1.23008899e+00     5.16105563e
    +00    -3.95721654e+00
 13    -2.82617532e+00    -3.89009683e+00    -3.57385364e+00    |   -2.82617532e+00    -3.89009683e
    +00    -3.57385364e+00
 14    -3.08122977e-01    -8.99691093e+00    -3.23336012e+00    |   -3.08122977e-01    -8.99691093e
    +00    -3.23336012e+00
 15    -6.91065346e-01    -1.86286200e+00     3.02092338e-01    |   -6.91065346e-01    -1.86286200e
    +00     3.02092338e-01
-----------------------------------------------------------------------------------------------------

PASS: Energies and forces are the same to within a relative error of 1e-08
-----------------------------------------------------------------------------------------------------


=====================================================================================================

To pass this verification check the model must be invariant with respect to
rigid-body motion (translation and rotation) for all configurations
it was able to compute.

Grade: P

Comment: Model energy and forces are invariant with respect to rigid-body motion (translation
    and rotation) for all configurations the model was able to compute.


=== Verification check vc-objectivity end (2024-07-25 02:41:16) ===
{"usertime":1.07,"memmax":58212,"memavg":0}

  Pair produced Verification Result VC_813478999433_002-and-MO_000000000000_000-1721875265-vr
    in 11.34522032737732 seconds
```