OpenMM-Python-Force: Deploying Accelerated Python Modules in Molecular Dynamics Simulation

Zhi Wang* and Wen Yan*

ByteDance Research, Bellevue, Washington 98004, USA.

E-mail: zhi.wang1@bytedance.com; wen.yan@bytedance.com

Abstract

We present OpenMM-Python-Force, a plugin designed to extend OpenMM's functionality by enabling integration of energy and force calculations from external Python programs via a callback mechanism. During molecular dynamics simulations, data exchange can be implemented through `torch.Tensor` or `numpy.ndarray`, depending on the specific use case. This enhancement significantly expands OpenMM's capabilities, facilitating seamless integration of accelerated Python modules within molecular dynamics simulations. This approach represents a general solution that can be adapted to other molecular dynamics engines beyond OpenMM. The source code is openly available at https://github.com/bytedance/OpenMM-Python-Force.

1 Introduction

Modern computational scientific research is standing at the intersection of two distinct technical domains: molecular dynamics (MD) simulation and machine learning (ML). These fields have evolved along different technological paths, with MD engines predominantly built on C-family languages utilizing ahead-of-time (AOT) compilation, while the ML ecosystem has consolidated around

Python, particularly the PyTorch framework. This technological divergence creates significant challenges for researchers seeking to integrate these paradigms effectively.

The incorporation of pre-trained ML models into MD simulations has primarily relied on `torch.jit.script`, which generates TorchScript graphs through static analysis and Python Abstract Syntax Tree parsing. While this approach enables just-in-time (JIT) optimizations and provides essential C++ APIs—exemplified by implementations like OpenMM Torch[1,2]—it imposes considerable limitations by supporting only a restricted subset of Python syntax. These constraints are substantial in practice, with approximately 50% of real-world models failing to compile successfully using `torch.jit.script`.[3]

Recent advances have introduced various strategies to enhance both training and inference performance of ML models. CUDA 12's Graph functionality enables the recording and efficient replay of CUDA kernel sequences to reduce kernel launch overheads. For cases with well-identified computational bottlenecks, hand-optimized CUDA operators remain an effective optimization strategy. Projects like NNPOps[1,4] demonstrate this approach by providing specialized operators for tasks such as Particle Mesh Ewald calculations and neighbor list construction. PyTorch 2.0's `torch.compile`[3] marks a significant breakthrough, particularly valuable for non-obvious performance bottlenecks. The latter employs a frontend that extracts PyTorch operations from Python bytecode to construct FX graphs, which are then processed by a compiler backend generating Triton code for CUDA execution. This approach substantially relaxes Python syntax restrictions while enabling sophisticated optimizations such as kernel fusion. Moreover, it can modify the bytecode to incorporate AOT-compiled kernels. The efficacy of these optimizations is evidenced by frameworks such as TorchMD-NET,[5] which has achieved significant performance improvements in their training pipeline.

Despite its promising capabilities, `torch.compile` has not yet achieved widespread adoption in molecular dynamics simulations, primarily due to its lack of native C++ support. This paper addresses this limitation through three steps. First, we present a general callback mechanism that enables any Python module to serve as a gradient provider for MD simulations. Second, we validate this approach through rigorous numerical testing and performance profiling in gas-phase simulations, demonstrating both its numerical accuracy and computational efficiency. Finally, we demonstrate the broad applicability of this approach

through its implementation in ab initio molecular dynamics (AIMD) simulations and explore its portability to other MD engines, illustrating how our solution provides a generic and seamless integration between Python modules and MD simulations.

2 Methods

The fundamental basis for the callback mechanism lies in the C-API of the CPython interpreter. When Python code such as

```
results = model(*args, **kwargs)
```

is executed, it is internally translated into an equivalent C function call

```
PyObject *results, *model, *args, *kwargs;
PyObject *id = model;
results = PyObject_Call(id, args, kwargs);
```

This direct mapping between Python code and C function calls, as illustrated in Figure 1, extends to all Python operations, making it theoretically feasible to translate any Python code sequence into equivalent C function calls. While such manual translation would be impractical, the pybind11 library[6] substantially simplifies this process. The implementation of the callback mechanism requires two key components: (1) obtaining the unique identifier (a long integer) of the callable object in Python, which corresponds to the value of its `PyObject` pointer, achieved using Python's built-in `id()` function, and (2) transferring and storing this identifier from Python to C, accomplished through the CPython API or binding libraries such as pybind11 or SWIG.[7]

After capturing the `PyObject` pointer of the callable Python object in C and ensuring that the object remains resident in memory, model inference can proceed independently of the ML model's deployment method or optimization strategy. This approach establishes a robust and flexible foundation for integrating Python-based ML models with C-based MD simulations.

The implementation is streamlined through a custom `Callable` class that encapsulates the model identifier, the return variables, and the function parameters. This class serves as the primary initialization parameter for the new OpenMM

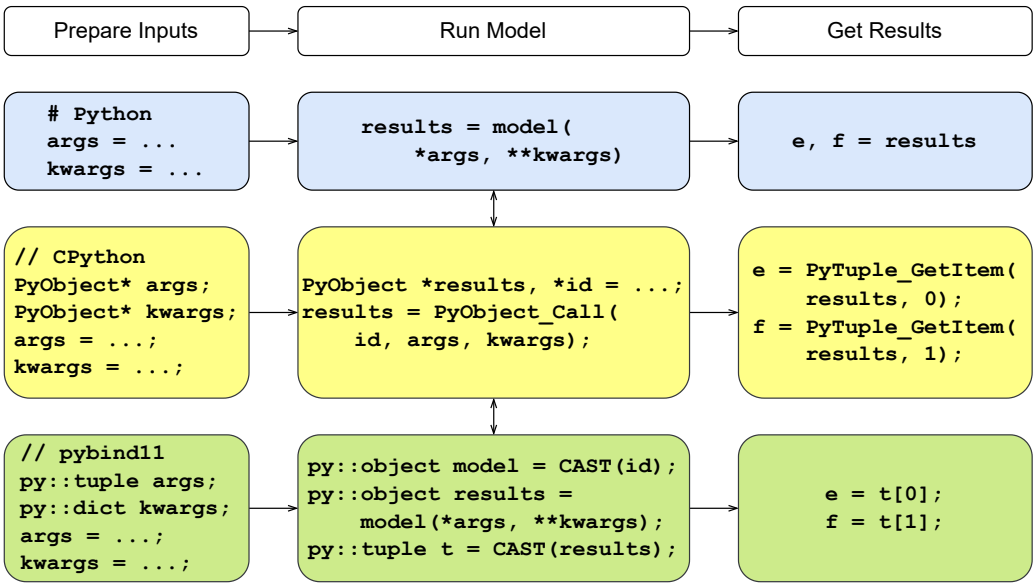

Figure 1: Illustration of the Python callback mechanism, demonstrating the translation between a Python function call and its corresponding pseudo C/C++ implementations using either the CPython API or pybind11 (with the C++ namespace `pybind11` abbreviated as `py`).

force provided in the plugin. Integration of this mechanism into an existing OpenMM Python script is straightforward, as demonstrated below:

```
from CallbackPyForce import Callable, TorchForce
class Model42(torch.nn.Module):
    def forward(self, positions):
        return torch.sum(positions**2)
model42 = Model42()
model42 = torch.compile(model42)
call = Callable(id(model42), Callable.RETURN_ENERGY)
force = TorchForce(call)
openmm_system.addForce(force)
```

The Torch library handles force calculations through backpropagation when the forward pass does not explicitly compute forces. Additionally, this mechanism seamlessly supports model optimization through PyTorch's compilation tools: models can be enhanced with `torch.jit.script` or `torch.compile` simply by adding these statements to the existing code, requiring minimal additional modifications.

3 Results and Discussion

3.1 Example: Ethanol

We evaluated the numerical accuracy and computational performance using a single ethanol molecule in vacuum, implementing eight distinct deployment and compilation strategies (Table 1). For each strategy, we performed independent NVE simulations for 100 steps with a 1 fs time-step. All simulations utilized identical random seeds and initial velocities corresponding to 300 K. We employed the BAMBOO[8] MLFF throughout, configuring OpenMM simulations with the mixed-precision CUDA platform and setting the BAMBOO model's internal data type to fp32. All computations were executed on a single NVIDIA L4 GPU. The numerical accuracy analysis encompassed three evaluations.

Hamiltonian Conservation The time evolution of the Hamiltonian, shown in Figure 2, exhibited excellent consistency in all eight simulations. The system maintained an average total energy of -2545.1 kJ/mol with a small standard deviation of 0.22 kJ/mol.

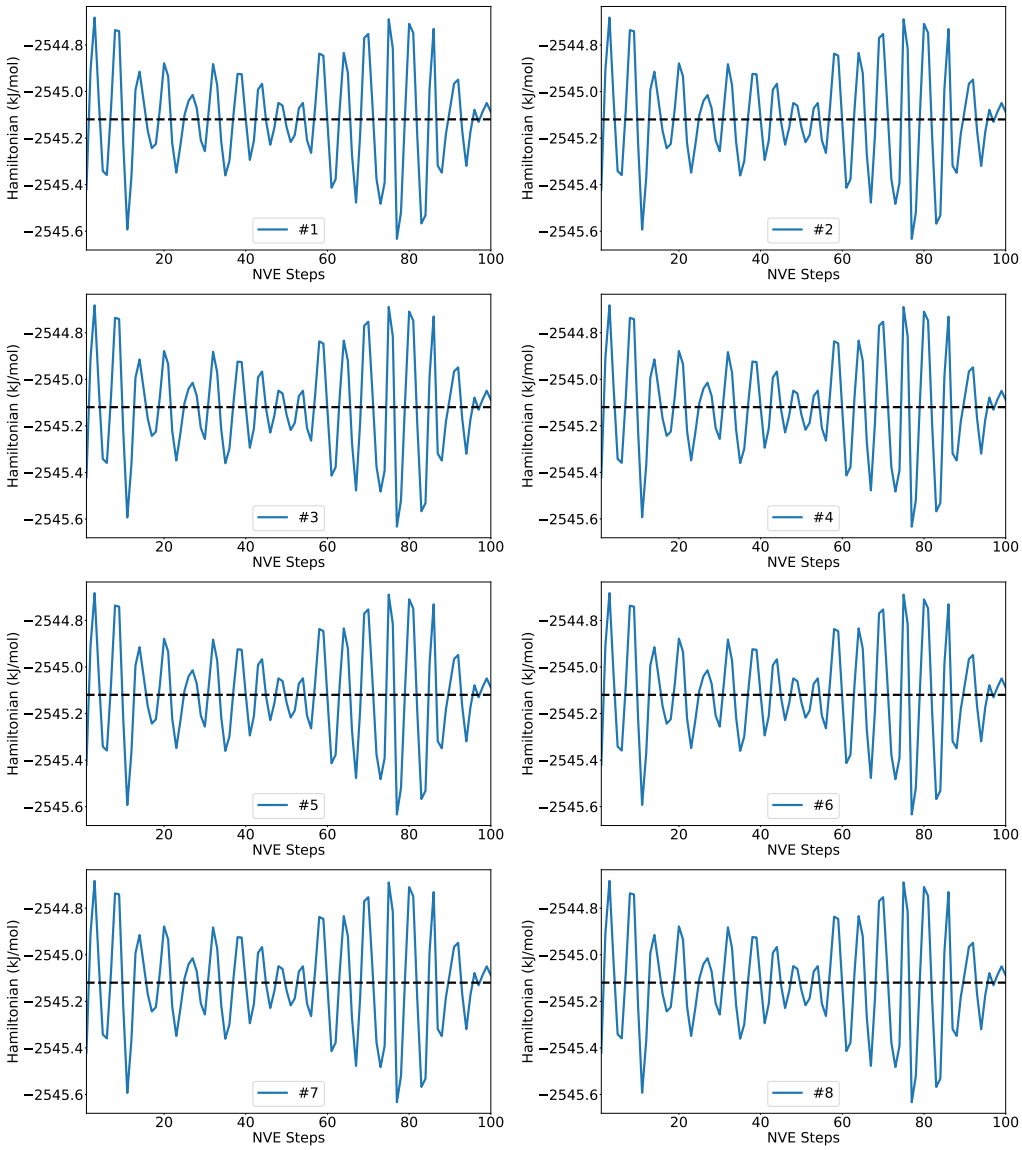

Figure 2: Evolution of system Hamiltonians over 100 time-steps for different deployment strategies. Each subfigure corresponds to a specific strategy as detailed in Table 1. Dashed lines indicate the mean Hamiltonian value.

Table 1: Deployment and compilation strategies evaluated in ethanol simulations.

	No.	Description	No.	Description
C++	1	OpenMM Torch (Baseline)	5	1 + CUDA Graph
Python	2	native `torch.nn.Module`	6	2 + CUDA Graph
Python	3	`torch.jit.script`	7	3 + CUDA Graph
Python	4	`torch.compile`	8	4 + CUDA Graph

Trajectory Convergence Figure 3 depicts the differences per time-step in potential energy (U), kinetic energy (K), and Hamiltonian (H) between the baseline and other implementations. The energies maintained convergence throughout the 100 time-steps, with numerical differences approaching the theoretical limit of fp32 precision (approximately the 6th or 7th significant figure). The spatial coordinates, recorded in the PDB format, exhibited consistency up to three decimal places with a maximum deviation of 0.001 Å on all trajectories.

Errors in Forces Using the baseline trajectory, we recalculated the potential energies and forces. Figure 4 illustrates the unsigned error in potential energies and root mean square deviation (RMSD) of the 27 force components relative to the baseline values. These results corroborate that the differences are minimal, reaching the inherent precision limit of fp32 arithmetic.

The comparative performance of the eight strategies is summarized in Table 2. Performance benchmarks were conducted using a Langevin integrator with a friction of 0.1 ps^{-1} over 1000 time-steps, generating 10 trajectory frames. For small-scale test cases such as a single ethanol molecule, where kernel launch overhead dominates computational cost, CUDA Graph significantly enhanced performance. Comparisons between implementations 1 and 3, as well as 5 and 7, demonstrate that direct inference via C++ APIs achieves superior performance with reduced launch overhead compared to their Python counterparts. However, these benefits of reduced overhead are expected to diminish with increasing system size, a phenomenon previously observed in TorchMD-NET. Notably, `torch.compile` exhibited significant performance advantages, as evidenced by comparisons between implementations 1 and 4, and between 5 and 8.

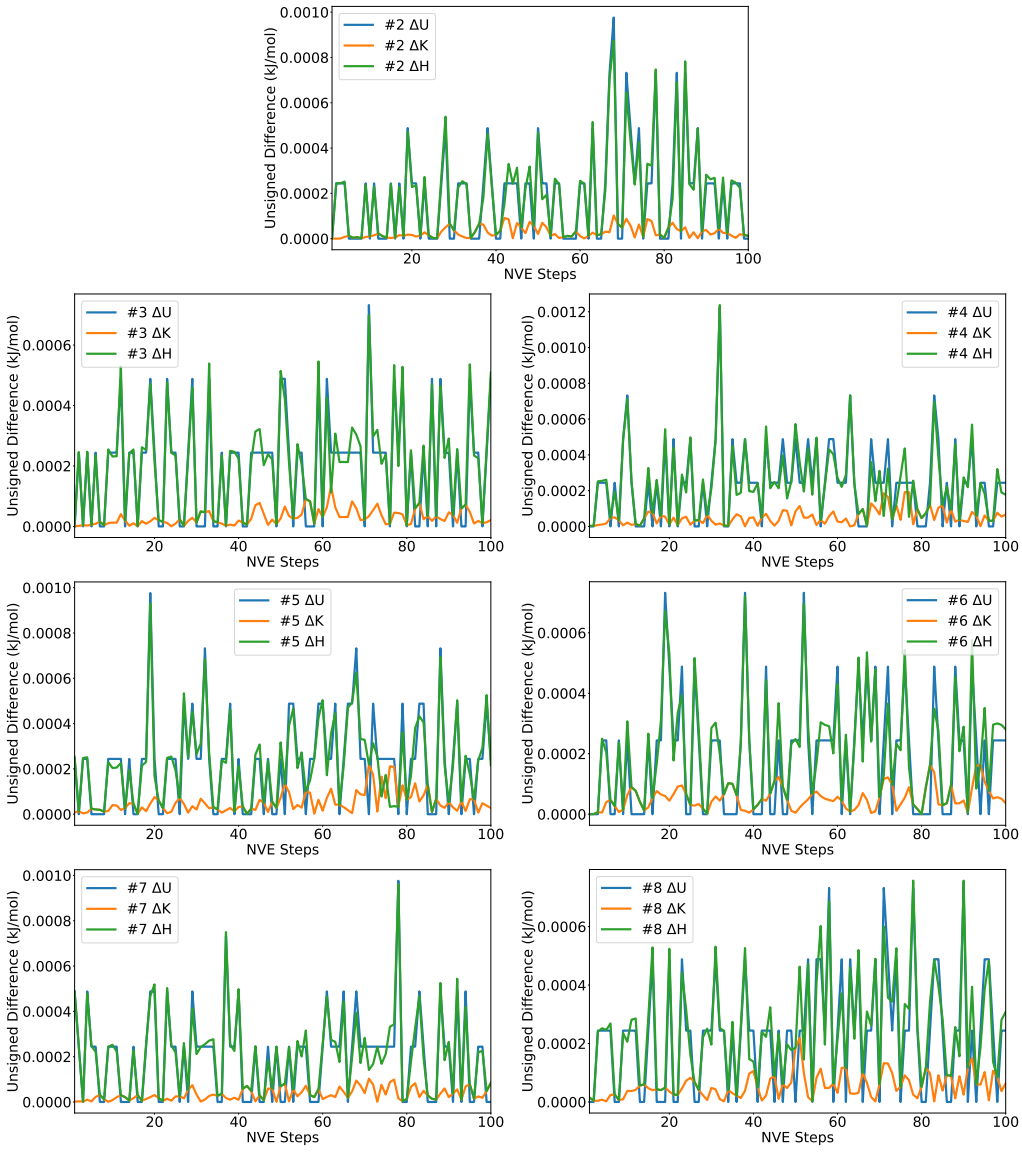

Figure 3: Comparison of energies across deployment strategies: unsigned differences in potential energy (U), kinetic energy (K), and Hamiltonian (H) relative to the baseline simulation over 100 time-steps. The number in each subfigure indicates the deployment method as defined in Table 1.

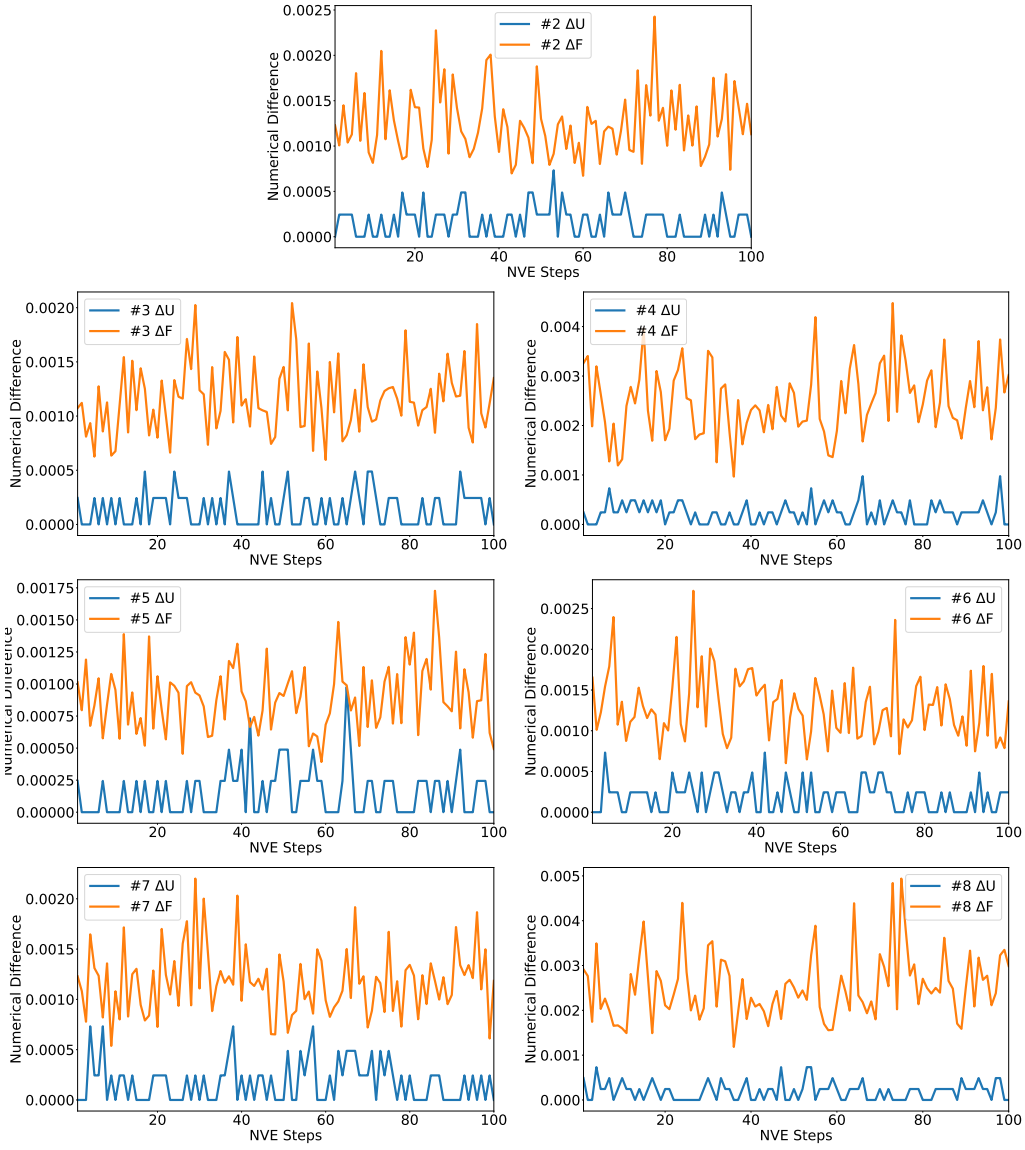

Figure 4: The unsigned differences in potential energies and root mean square deviation of forces (in kJ/mol and nm) compared to the baseline, obtained by recalculating an identical PDB trajectory across deployment methods. The number in each subfigure denotes the deployment method as defined in Table 1.

Table 2: Performance comparison of different deployment and compilation strategies in ethanol simulations. The relative speeds are normalized to the baseline implementation (#1).

No.	ms/step	10^6 steps/day	Relative Speed
1	3.97	21.8	1
2	7.40	11.7	0.54
3	5.58	15.5	0.71
4	3.07	28.1	1.3
5	0.668	129	5.9
6	1.15	75.4	3.5
7	0.915	94.5	4.3
8	0.486	178	8.2

3.2 Example: AIMD Simulation

The versatility of the callback mechanism extends beyond PyTorch tensors to accommodate diverse data containers. To demonstrate this flexibility, we implemented an AIMD simulation using PySCF/GPU4PySCF[9–12] with the B3LYP functional and D3BJ dispersion correction. The simulation utilizes *NumPyForce*, an OpenMM Force plugin we developed alongside *TorchForce* to establish a bridge between OpenMM and quantum chemistry software packages. All relevant implementation files are available on GitHub.

In this quantum mechanical context, where automatic differentiation via backpropagation is not available, forces or gradients must be explicitly provided to the MD engine. While NumPy lacks native CUDA support, the overhead from device-to-device data transfer and floating-point conversion (between fp32 and fp64) proves negligible in AIMD simulations, where quantum chemical force calculations typically dominate the computational cost.

3.3 Extensibility to Other MD Engines

The proposed callback mechanism demonstrates broad compatibility with other molecular dynamics engines, such as Tinker[13] and LAMMPS,[14] even when these packages do not natively initialize a Python interpreter. Based on our implemen-

tation experience, incorporating a "callback Python energy term" would require comparable code modifications in their respective source files. Furthermore, the initialization of the Python interpreter would necessitate only minimal additional changes to other components of the codebase, making it a straightforward extension.

Taking Tinker as an example, initializing a Python interpreter conceptually resembles the initialization of a Fortran runtime library, as currently implemented in Tinker9.[15] During program initialization, Tinker9 calls compiler-specific functions: `_gfortran_set_args` for GFortran-compiled executables, or `for_rtl_init_` and `for_rtl_finish_` for initialization and cleanup with the Intel compiler, respectively. Similarly, implementing Python support would primarily involve incorporating CPython C-API functions such as `Py_Initialize` and `Py_Finalize`. For detailed implementation guidance, we refer readers to the official CPython documentation[16] and pybind11's *embedding the interpreter* documentation.[17]

4 Conclusion

In this work, we have presented OpenMM-Python-Force, a callback mechanism that seamlessly bridges molecular dynamics simulations with machine learning model inference. Our evaluation demonstrates that this approach is not only robust and computationally efficient but also remarkably versatile in its applications. The applications of this callback mechanism extend well beyond its initial implementation with PyTorch and OpenMM, encompassing both classical and ab initio molecular dynamics simulations. We anticipate that this work will substantially reduce the technical barriers for integrating various computational backends with MD simulations, thereby accelerating progress in relevant fields of research.

Acknowledgements

We extend our gratitude to Leyuan Wang and Siyuan Liu for their assistance in identifying the source of numerical discrepancies between results produced by `torch.compile` and other deployment strategies; Chi Xu for conducting the

code review; and Sheng Gong, Zhenliang Mu, Zhichen Pu, and Xu Han for their support in setting up the BAMBOO model. We also thank Xiaojie Wu for his contributions to the code interfacing GPU4PySCF with this work.

References

(1) Eastman, P.; Galvelis, R.; Peláez, R. P.; Abreu, C. R. A.; Farr, S. E.; Gallicchio, E.; Gorenko, A.; Henry, M. M.; Hu, F.; Huang, J.; Krämer, A.; Michel, J.; Mitchell, J. A.; Pande, V. S.; Rodrigues, J. P.; Rodriguez-Guerra, J.; Simmonett, A. C.; Singh, S.; Swails, J.; Turner, P.; Wang, Y.; Zhang, I.; Chodera, J. D.; De Fabritiis, G.; Markland, T. E. OpenMM 8: Molecular Dynamics Simulation with Machine Learning Potentials. *The Journal of Physical Chemistry B* **2024**, *128*, 109–116.

(2) OpenMM Torch. https://github.com/openmm/openmm-torch.

(3) Ansel, J.; Yang, E.; He, H.; Gimelshein, N.; Jain, A.; Voznesensky, M.; Bao, B.; Bell, P.; Berard, D.; Burovski, E.; Chauhan, G.; Chourdia, A.; Constable, W.; Desmaison, A.; DeVito, Z.; Ellison, E.; Feng, W.; Gong, J.; Gschwind, M.; Hirsh, B.; Huang, S.; Kalambarkar, K.; Kirsch, L.; Lazos, M.; Lezcano, M.; Liang, Y.; Liang, J.; Lu, Y.; Luk, C. K.; Maher, B.; Pan, Y.; Puhrsch, C.; Reso, M.; Saroufim, M.; Siraichi, M. Y.; Suk, H.; Zhang, S.; Suo, M.; Tillet, P.; Zhao, X.; Wang, E.; Zhou, K.; Zou, R.; Wang, X.; Mathews, A.; Wen, W.; Chanan, G.; Wu, P.; Chintala, S. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. La Jolla CA USA, 2024; pp 929–947.

(4) NNPOps. https://github.com/openmm/NNPOps.

(5) Pelaez, R. P.; Simeon, G.; Galvelis, R.; Mirarchi, A.; Eastman, P.; Doerr, S.; Thölke, P.; Markland, T. E.; De Fabritiis, G. TorchMD-Net 2.0: Fast Neural Network Potentials for Molecular Simulations. *Journal of Chemical Theory and Computation* **2024**, *20*, 4076–4087.

(6) Jakob, W.; Rhinelander, J.; Moldovan, D. Pybind11 – Seamless Operability between C++11 and Python. 2017; https://github.com/pybind/pybind11.

(7) SWIG. https://www.swig.org.

(8) Gong, S.; Zhang, Y.; Mu, Z.; Pu, Z.; Wang, H.; Yu, Z.; Chen, M.; Zheng, T.; Wang, Z.; Chen, L.; Wu, X.; Shi, S.; Gao, W.; Yan, W.; Xiang, L. BAMBOO: A Predictive and Transferable Machine Learning Force Field Framework for Liquid Electrolyte Development. 2024; https://arxiv.org/abs/2404.07181.

(9) Sun, Q.; Zhang, X.; Banerjee, S.; Bao, P.; Barbry, M.; Blunt, N. S.; Bogdanov, N. A.; Booth, G. H.; Chen, J.; Cui, Z.-H.; Eriksen, J. J.; Gao, Y.; Guo, S.; Hermann, J.; Hermes, M. R.; Koh, K.; Koval, P.; Lehtola, S.; Li, Z.; Liu, J.; Mardirossian, N.; McClain, J. D.; Motta, M.; Mussard, B.; Pham, H. Q.; Pulkin, A.; Purwanto, W.; Robinson, P. J.; Ronca, E.; Sayfutyarova, E. R.; Scheurer, M.; Schurkus, H. F.; Smith, J. E. T.; Sun, C.; Sun, S.-N.; Upadhyay, S.; Wagner, L. K.; Wang, X.; White, A.; Whitfield, J. D.; Williamson, M. J.; Wouters, S.; Yang, J.; Yu, J. M.; Zhu, T.; Berkelbach, T. C.; Sharma, S.; Sokolov, A. Y.; Chan, G. K.-L. Recent Developments in the PySCF Program Package. *The Journal of Chemical Physics* **2020**, *153*, 024109.

(10) Wu, X.; Sun, Q.; Pu, Z.; Zheng, T.; Ma, W.; Yan, W.; Yu, X.; Wu, Z.; Huo, M.; Li, X.; Ren, W.; Gong, S.; Zhang, Y.; Gao, W. Enhancing GPU-acceleration in the Python-based Simulations of Chemistry Framework. 2024; http://arxiv.org/abs/2404.09452.

(11) Li, R.; Sun, Q.; Zhang, X.; Chan, G. K.-L. Introducing GPU-acceleration into the Python-based Simulations of Chemistry Framework. 2024; https://arxiv.org/abs/2407.09700.

(12) Lehtola, S.; Steigemann, C.; Oliveira, M. J.; Marques, M. A. Recent Developments in Libxc — A Comprehensive Library of Functionals for Density Functional Theory. *SoftwareX* **2018**, *7*, 1–5.

(13) Rackers, J. A.; Wang, Z.; Lu, C.; Laury, M. L.; Lagardère, L.; Schnieders, M. J.; Piquemal, J.-P.; Ren, P.; Ponder, J. W. Tinker 8: Software Tools for Molecular Design. *Journal of Chemical Theory and Computation* **2018**, *14*, 5273–5289.

(14) Thompson, A. P.; Aktulga, H. M.; Berger, R.; Bolintineanu, D. S.; Brown, W. M.; Crozier, P. S.; In 'T Veld, P. J.; Kohlmeyer, A.; Moore, S. G.; Nguyen, T. D.; Shan, R.; Stevens, M. J.; Tranchida, J.; Trott, C.; Plimpton, S. J. LAMMPS - a Flexible Simulation Tool for Particle-Based Materials Modeling at the Atomic, Meso, and Continuum Scales. *Computer Physics Communications* **2022**, *271*, 108171.

(15) Wang, Z.; Ponder, J. W. Tinker9: Next Generation of Tinker with GPU Support. https://github.com/TinkerTools/tinker9.

(16) Python/C API Reference Manual. https://docs.python.org/3/c-api/index.html.

(17) Pybind11 Documentation: Embedding the Interpreter. https://pybind11.readthedocs.io/en/stable/advanced/embedding.html.