Performance and Power Characterization of AI-enabled Applications on Heterogeneous System

H.E. Greenblatt*, Hunter Negron*, Melanie Cornelius, Romit Maulik, Stefan Muller, Zhiling Lan
Illinois Institute of Technology, Chicago, IL, USA

Xingfu Wu, Valerie Taylor
University of Chicago, Chicago, IL

Michael E. Papka
University of Illinois Chicago, Chicago, IL

***Abstract*—AI-enabled science—in which advanced machine learning technologies are used for surrogate models, autotuning, and *in-situ* data analysis—is quickly being adopted in science and engineering for tackling challenging computational problems. However, current research on efficient execution of AI-enabled science on high-performance computing (HPC) systems is far from sufficient. The wide adoption of heterogeneous systems comprising different types of processing devices (e.g., CPUs and GPUs) further complicates the execution of AI-enabled science on supercomputers. In this work, we present a performance and power profiling study of two AI-enabled proxy applications performing computational fluid dynamics on the production ThetaGPU system at Argonne National Laboratory.**

I. Introduction

The high-performance computing (HPC) community is rapidly embracing AI-enabled simulations, in domains from molecular dynamics simulation to environmental research [1]–[5]. The rise of AI-enabled science (and the increasing availability of large-scale curated datasets) has the potential to accelerate innovation in key areas in science and technology [6]. Meanwhile, a dominant trend in HPC is the deployment of heterogeneous processors (including CPUs, GPUs, and dedicated accelerators) to improve computing capabilities, boost performance, and meet energy efficiency goals.

While new AI/ML models are being explored for scientific discovery and innovation, little is known about AI-enabled simulations on a heterogeneous CPU-GPU computing environment. In addition, existing research on AI-enabled science has focused on canonical experiments, with little or no consideration of energy efficiency and scaling.

In this work, we present an experimental study of two AI-enabled proxy applications simulating incompressible fluid flow on the ThetaGPU production CPU-GPU machine at Argonne Leadership Computing Facility (ALCF). The objective is to provide insights into the execution time, scaling, and power consumption of AI-enabled science on heterogeneous systems. The study focuses on two proxy applications: a prototypical *Mini-app*, which embeds a Python/TensorFlow [7] machine learning model as a surrogate in a C++ computational physics workflow, and *PythonFOAM* [8], which employs the same ML tools for *in-situ* analysis of data generated by the popular and well-established OpenFOAM C++ computational fluid dynamics toolkit [9].

*Greenblatt and Negron contributed equally
+Maulik, Wu, Taylor, and Papka are also affiliated with Argonne National Laboratory

II. System and Application Description

A. ThetaGPU

ThetaGPU consists of twenty-four NVIDIA DGX A100 nodes, each node with two 64-core AMD Rome CPUs and eight NVIDIA A100 GPUs [10]. Jobs on ThetaGPU can be allocated either by node or by GPU. In this study, all experiments were conducted in full-node jobs, since sharing nodes may result in performance variation.

B. Mini-app

Our Mini-app is a proxy application in which scientific machine learning is deployed within a computational physics workflow. It is representative of a number of scientific machine learning workloads, in which a simple finite-difference calculation of a one-dimensional problem is performed using a legacy language such as C++ and an ML-based surrogate model in Python is used to advance the solution field. Specifically, snapshots of the solution field are linearly compressed using singular value decomposition (SVD), and the compressed representations are used as training data for a long short-term memory (LSTM) neural network which predicts compressed representations of the solution field in the future [11]. Fig. 1 presents the code structure integrating C++ and Python.

Mini-app has a set of hyperparameters which must be determined before running the algorithm. We focus on NX and DT, which control the problem size. NX specifies the number of points of the spatial discretization of the finite difference solver, while DT specifies the time step. These parameters are not independent; for numerical stability, they must scale approximately inversely. Table I lists the combinations of parameter values, covering a breadth of problem sizes, explored in the following experiments.

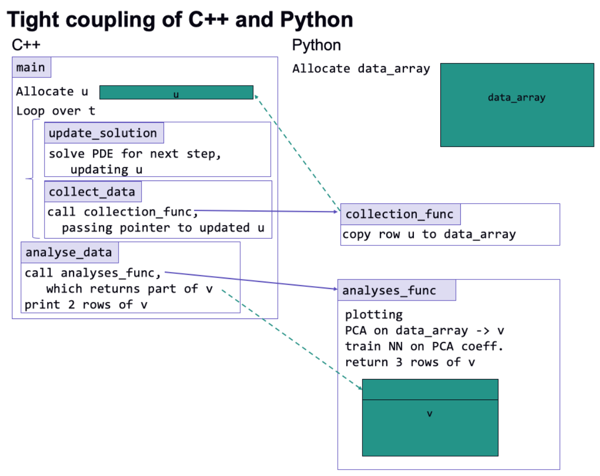

Fig. 1. Mini-app code structure coupling Python and C++ [11]

TABLE I
MINI-APP PROBLEM SIZES

Problem Size	NX	DT
Small (S)	2560	0.0001
Medium (M)	3840	0.0001
Large (L)	5120	0.00001

Our modifications to Mini-app, as well as details of the dependency management and environment configuration on ThetaGPU, are available as open source on GitHub [12]. Although this study focuses on single-GPU profiling, future plans include the use of multiple GPUs, so Mini-app was extended with Horovod [13], a distributed training framework that can be used to wrap a variety of Python ML platforms. Changes to Mini-app's machine learning code included making multiple GPUs available to TensorFlow, sharding the LSTM training data, and averaging the test loss among all workers.

C. *PythonFOAM*

PythonFOAM is a computational fluid dynamics toolkit based on OpenFOAM; it enhances OpenFOAM (which is written in C++) with embedded Python in order to add powerful, flexible *in situ* data analysis capabilities, including machine learning [8]. Its combination of distinct computational workloads—and its capacity for offloading its data analysis to dedicated hardware—make it a useful showcase for AI-enabled science.

OpenFOAM provides built-in capabilities for parallelization using MPI. *Domain decomposition* is used for parallelization: a normal OpenFOAM parallel workflow involves running an external utility (provided by OpenFOAM) to decompose the domain geometry and fields into subdomains; running an OpenFOAM solver in parallel mode on the decomposed case, with each subdomain assigned to a separate processor and processors exchanging data at subdomain boundaries; and running another external utility (also provided by OpenFOAM) to reconstruct the subdomains results into the original domain [14]. Since decomposition and reconstruction are outsourced to separate utilities rather than performed in the main solver, profiling the solver will not capture the parallelization overhead of this workload, but it is negligibly small.

PythonFOAM permits an OpenFOAM solver to call Python code using Python's C API. Each PythonFOAM process initializes its own Python interpreter, and can subsequently manipulate objects and call methods in that environment; this Python execution occurs serially, in the same process as the C++ in which it is embedded. In parallel mode, the multiple Python interpreters thus created are independent, but can exchange data using the same MPI communicator used by the host C++.

The specific application studied was the PythonFOAM example solver *AEFoam*, which uses an autoencoder (periodically replaced) trained on snapshots of the domain flow field to generate and output low-dimensional encodings thereof. A single cycle of its control flow is depicted in Fig. 2; this cycle repeats until the problem end time is reached. In parallel mode, each AEFoam process has a separate, independent autoencoder, trained on snapshots of its own subdomain only.

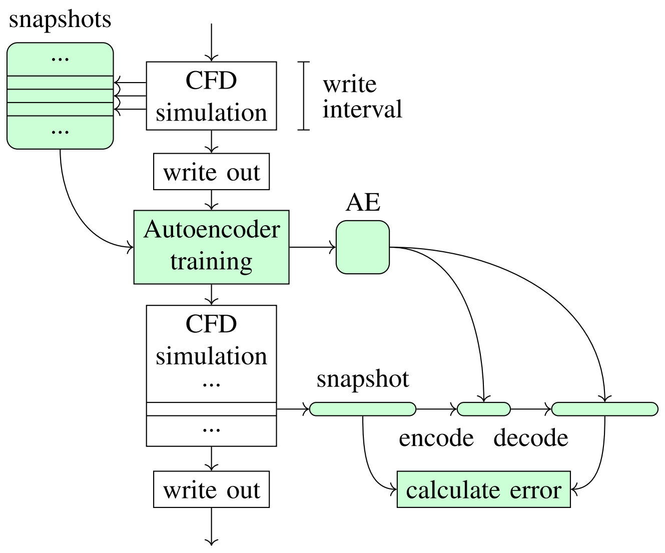

Fig. 2. PythonFOAM (AEFoam solver) control flow on a single subdomain. Operations (rectangular nodes) and data (rounded) in white occur in C++; those in green, Python.

Normally, OpenFOAM calculates the value of every field at every cell in its mesh for every iteration, but only writes those values to disk at long intervals; values from the iterations between write intervals are discarded (since the storage and I/O required would otherwise soon grow prohibitive). By training an autoencoder *in situ* on snapshots of all iterations, rather than from data saved to disk, large volumes of training data can be collected quickly, and by using the trained autoencoder to generate latent representations of the field of interest, good approximations of any or all iterations can be saved—in both cases without excessive I/O. This opens up new space in simulation design tradeoffs.

AEFoam was studied using the case directory supplied in the supplementary repository of [8], except that the problem end time was reduced to 0.035s to limit runtime under profiling. This does not affect solver behavior in preceding timesteps. TensorFlow memory growth was enabled via environment variable in order to permit more than one autoencoder to train on the same GPU. Some bugs in AEFoam relating to ownership of Python objects, which could cause segfaults under memory pressure, were fixed. Finally, in order to study multi-GPU performance, a short block was added to AEFoam's Python module which restricted the physical devices visible to TensorFlow if the solver was running in parallel mode and multiple GPUs were available, effectively assigning each autoencoder to a particular GPU. Parallel performance was studied in powers of two, from 1 rank to the machine maximum of 128. The detailed PythonFOAM code, along with its configuration for ThetaGPU, is available as open source on GitHub [12].

III. Performance and Power Characterization

Both applications were studied across two compute modes: *CPU-only* and *CPU-1GPU* (CPU plus a single GPU). PythonFOAM was additionally studied under *CPU-8GPU* (CPU plus 8 GPUs). All experiments were conducted on one ThetaGPU node with 64 cores. Each application in each compute mode was profiled with multiple tools: CPU and other host-side performance metrics were collected with Linux perf [15], while GPU metrics were collected with the NVIDIA Systems Management Interface (nvidia-smi) [16] and the NVIDIA Nsight Systems CLI (nsys) [17]. The NVIDIA SMI was used to measure overall system metrics such as power and GPU-side memory use, while the Nsight Systems CLI was used to gather detailed statistics on GPU activities.

These diverse profiling methods were orchestrated and consolidated with a tool called Mantis [18], which simplifies multi-configuration application profiling on a heterogeneous system. For instance, Mini-app includes three different sizes (see Table I); given appropriate configuration, Mantis can automatically run all these sizes under each profiling tool. In addition, although these tools output various CSV, JSON, SQLite, and proprietary "qdrep" files, Mantis can unify the formats into a single file for convenient analysis. This greatly simplifies the profiling and data analysis workflow.

A. *Mini-app Study*

Full analysis of Mini-app is still a work in progress. While the Python part of Mini-app has been extended with Horovod for distributed learning, the C++ part is still sequential; therefore, the results presented below focus on CPU-only and single-GPU configurations.

While the profiling tools used in this study are considered lightweight, they did have non-trivial overheads. Experiments show that the discrepancy between an uninstrumented and a profiled run of Mini-app ranges from just a few minutes (with NVIDIA SMI) to over an hour (with NVIDIA Nsight), with an overhead of up to 146%. This effect on overall performance, particularly for the larger sizes, meant that Mantis had to profile only a few configurations in each run in order to avoid exceeding ThetaGPU's 12-hour maximum job time [19].

Fig. 3 shows scaling results with respect to growing Mini-app problem sizes. The data were extracted from perf's "instructions" counter. As expected, the application runtime grows with increasing problem size, especially for the large size, while GPU acceleration provides some performance benefit over the corresponding CPU-only problem (shown as percent speedup). However, as the problem size increases, the relative improvement decreases. There are two reasons for this scaling behavior. First, as the problem size increases, so does the amount of data being moved between CPU and GPU. The increasing overhead of data movement leads to decreasing benefit of running Mini-app on CPU-1GPU over CPU-only. Second, Mini-app is CPU-bound.

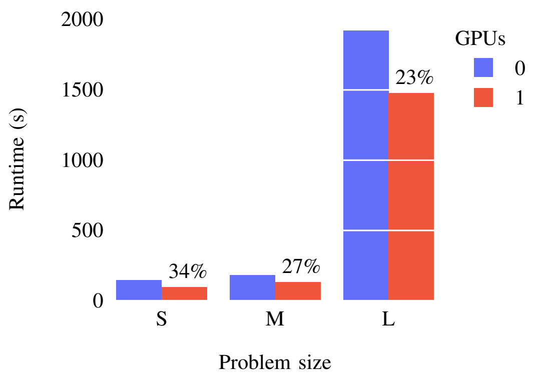

Fig. 3. Mini-app wall-clock runtime on CPU-only (blue) and CPU-1GPU (red) vs. problem size. Relative performance improvement of CPU-1GPU over CPU-only is indicated in percent.

Fig. 4 presents side-by-side profiling results for instructions on the CPU and utilization of the GPU. CPU behavior is very similar between problem sizes. (Although there is an apparent difference in "spike" frequency, this is merely a result of the relative timescale; in fact, the spikes occur at approximately the same absolute intervals.) More importantly, as the problem size increases, the relative duration of the machine learning phase—the portion of time that the GPU is utilized—actually decreases. Starting from size M, the GPU works for less than half of the total runtime of the application. This demonstrates that the CPU bottlenecks the performance of Mini-app. While efforts to accelerate the machine learning phase—through GPU offloading, parallelization, or both—*can* certainly improve the performance of Mini-app, these benefits will decrease as problem size increases and the computational physics phase becomes dominant.

Fig. 5 shows the power draw over time. As with GPU memory used, the GPU power draw is relatively low for a time, and then jumps up when the machine learning algorithms run. The minimum and maximum power draws are somewhat less correlated between problem sizes here than is memory usage; the amount of data being processed does not necessarily contribute to more power consumed per unit of time.

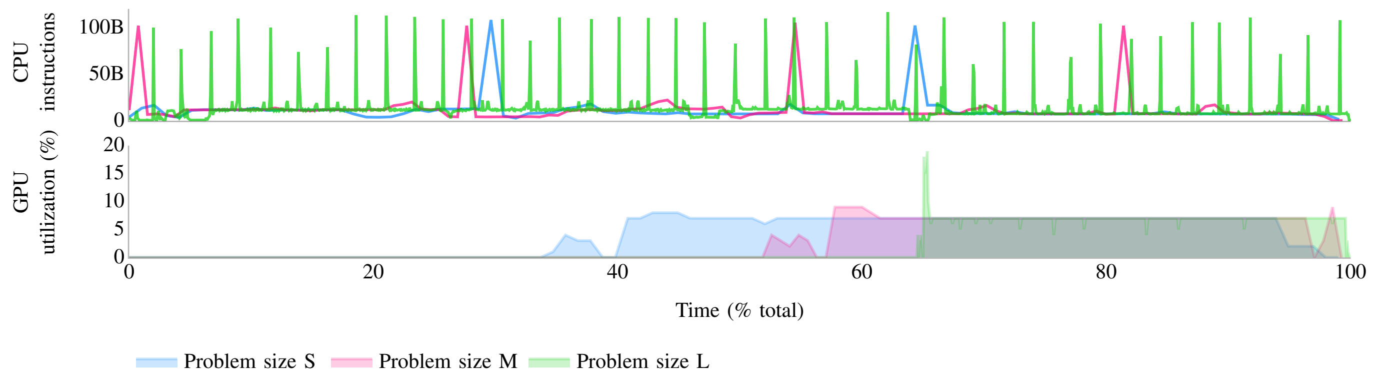

Fig. 4. CPU instructions (top) and GPU utilization (bottom) vs. relative time, CPU-1GPU. CPU and GPU data were collected in separate profiling runs.

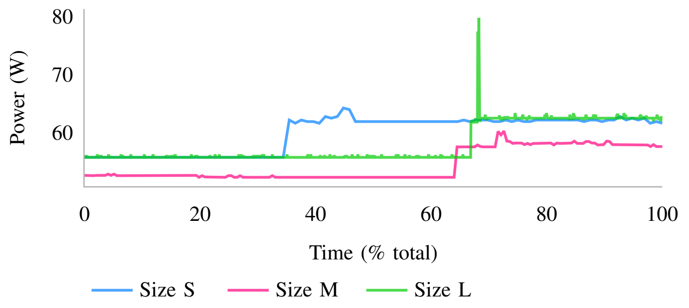

Fig. 5. Power draw vs. relative time, CPU-1GPU.

B. PythonFOAM Study

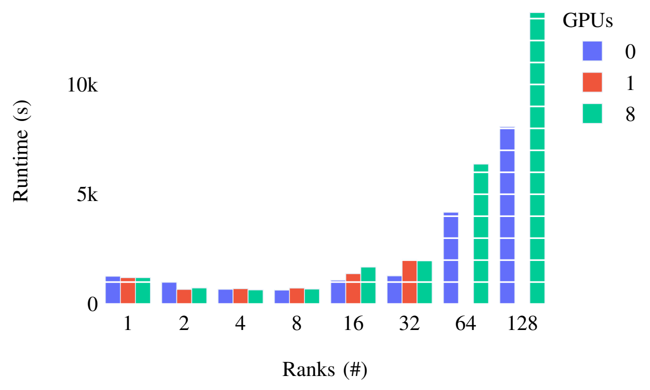

Fig. 6. PythonFOAM wall-clock runtime on CPU-only (blue), CPU-1GPU (red), and CPU-8GPU (green) vs. number of MPI ranks.

Scaling results for PythonFOAM are illustrated in Fig. 6. (CPU-1GPU data were not collected for 64 or 128 ranks because the amount of memory required to run 64 or 128 autoencoders exceeded that available on a single GPU.) A slight benefit from GPU offloading is evident at 1–2 ranks.

Overall, PythonFOAM scaled very poorly, implying that the case directory used (with its 20,540 cells) described too small a problem to be appropriate for a scaling analysis. Inspecting the output from the 128-rank decomposition revealed that at this size multiple subdomains *contained no cells*. Re-running the scaling analysis with a larger problem size is an obvious future direction for this work.

Nevertheless, the pathological behavior here is interesting in itself. In particular, The increased runtime at 16 ranks and above on one or more GPUs suggests that host/device memory movement swamps any computational advantage gained on the GPU. Theoretically, it should be possible to confirm or refute this by examining the "GPU Memory Operations Summary" collected by NVIDIA NSight—but in fact, AEFoam always nondeterministically hanged when run in parallel mode on at least one GPU under NSight (regardless of the number of GPUs made available, either through environment configuration or ThetaGPU's job allocator, and even with NSight configured to collect data only on one GPU). At this time, it is not clear what causes the hang. Nsight Systems's user guide specifies that profiling runs greater than five minutes are not officially supported [20]; although substantially longer runs completed successfully for both Mini-app and PythonFOAM, the suggested use of a fixed profiling duration, possibly combined with a reduced write interval (to decrease the training time of the autoencoders), may serve as a workaround.

Fig. 7 shows three CPU counters, the most important of which is number of instructions, collected with perf in the CPU-only configuration. There is an obvious periodicity. The 28 spikes correspond in number to the 28 write intervals completed at the end time, but because writing itself should not require more instructions than computation, and other behavior at the end of a write interval differs based on whether a new autoencoder should be trained, it is hard to convincingly attribute this to a particular task.

In comparison, combined CPU-GPU analysis is much more illustrative. Fig. 8 shows AEFoam's characteristic profile on CPU and GPU: load alternates between CPU, where the CFD simulation is solved (as indicated by the rise in CPU instruction volume), and GPU, where the autoencoder is trained (as indicated by the rise in GPU power draw). The 14 autoencoder

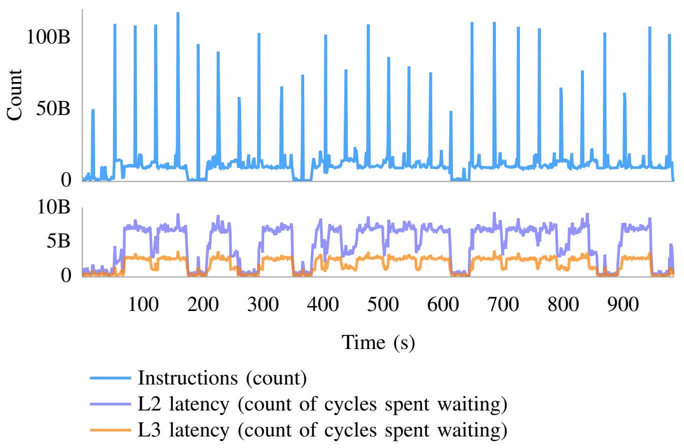

Fig. 7. CPU statistics vs. absolute time, CPU-only, 2 MPI ranks. All counters were collected in the same profliing run.

training intervals are clearly identifiable on both devices. CPU L2 total latency (l2_latency.l2_cycles_waiting_on_fills) and L3 total latency (xi_sys_fill_latency) both fall when instructions rise, indicating that the pimpleFoam solver on which AEFoam is based has good locality that tolerates load.

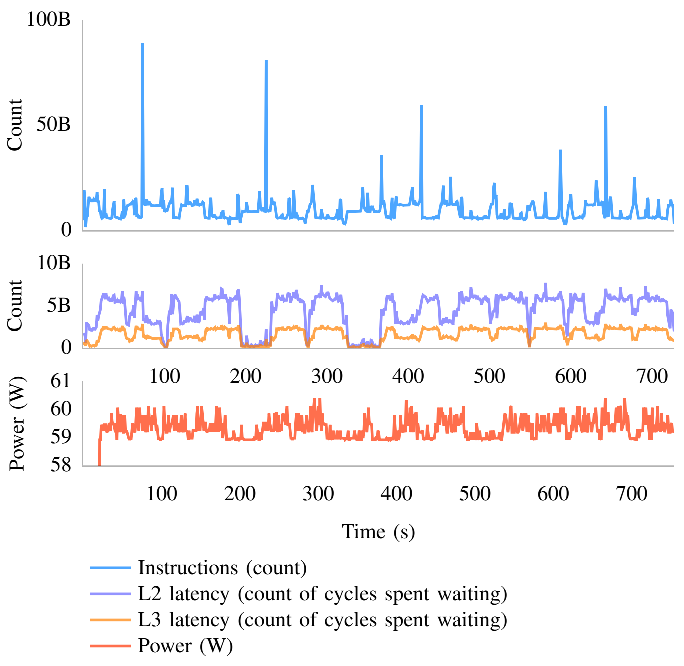

Fig. 8. CPU and GPU statistics vs. absolute time, CPU-8GPU, 2 MPI ranks. CPU and GPU data were collected in separate profiling runs; hence, time alignment is approximate, especially later in the run.

An unexpected feature of this trace is the irregular periodicity of the CPU/GPU alternation; although most GPU-heavy intervals are about the same length, suggesting that each autoencoder takes a similar time to train, the CPU-heavy intervals vary by almost an order of magnitude. This may be due to the turbulent nature of fluid flow over the backward-facing step in this problem, with some timesteps placing heavier demands on the solver than others.

IV. Conclusion

In this short paper, we present a performance and power profiling study of two AI-enabled proxy applications—integrating legacy computation in C++ with ML modeling in Python—on a hybrid CPU-GPU machine. Since both applications were initially developed on CPU-only systems [8], [11], part of our work was to port these applications to the hybrid CPU-GPU environment where the C++ computation was executed on CPUs and the Python ML modeling was executed on GPUs. Both applications, along with their configuration for ThetaGPU and their profiling results, are made available on GitHub [12].

Because these AI-enabled applications were built with multiple programming models and executed on a hybrid CPU-GPU system, multiple profiling tools were needed for collecting performance and power data, which added substantial complexity to the profiling process. For both benchmarks, we observed the performance benefit of running these AI-enabled applications in the CPU-GPU mode over in the CPU-only mode. The scaling of these applications on hybrid systems depends on many factors, including application problem size, workload distribution over CPUs and GPUs, and data movement effects. For example, the overhead of data movement between CPU and GPU can diminish the benefit of using the CPU-GPU. Our on-going work includes porting and profiling these benchmarks on heterogeneous systems other than ThetaGPU (e.g., systems equipped with different CPU and GPU configurations). We are also interested in collecting more data movement statistics across the memory hierarchy as well as among CPU and GPU for a better understanding of heterogeneous computing.

Acknowledgment

This work is supported in part by US NSF grants CCF 2119294, 2119203, 2119056, and DOE Office of Science under contract DE-AC02-06CH11357.

References

[1] W. Jia, H. Wang, M. Chen, D. Lu, L. Lin, R. Car, W. E, and L. Zhang, "Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ser. SC '20. IEEE Press, 2020.

[2] J. Wang, P. Balaprakash, and R. Kotamarthi, "Fast domain-aware neural network emulation of a planetary boundary layer parameterization in a numerical weather forecast model," *Geoscientific Model Development*, vol. 12, 2019.

[3] M. Parashar, "CI2030: Future advanced cyberinfrastructure," *NSF Advisory Committee for Cyberinfrastructure*, 2018.

[4] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019.

[5] E. J. Parish and K. Duraisamy, "A paradigm for data-driven predictive modeling using field inversion and machine learning," *Journal of Computational Physics*, vol. 305, pp. 758–774, 2016.

[6] I. Foster, D. Parkes, and S. Zheng, “The rise of AI-driven simulators: Building a new crystal ball,” 2020.

[7] TensorFlow Developers, “Tensorflow,” May 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6574269

[8] R. Maulik, D. K. Fytanidis, B. Lusch, V. Vishwanath, and S. Patel, “PythonFOAM: In-situ data analyses with OpenFOAM and Python,” *Journal of Computational Science*, vol. 62, p. 101750, 2022.

[9] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, “A tensorial approach to computational continuum mechanics using object-oriented techniques,” *Computers in physics*, vol. 12, no. 6, pp. 620–631, 1998.

[10] ThetaGPU at argonne leadership computing facility. [Online]. Available: https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview

[11] Mini-app. [Online]. Available: https://github.com/argonne-lcf/sdl_ai_workshop/tree/master/05_Simulation_ML

[12] SEEr code repo on github. [Online]. Available: https://github.com/SPEAR-IIT/SEEr

[13] A. Sergeev and M. D. Balso, “Horovod: fast and easy distributed deep learning in TensorFlow,” *arXiv preprint arXiv:1802.05799*, 2018.

[14] Openfoam v8 user guide - 3.4: Running applications in parallel. [Online]. Available: https://doc.cfd.direct/openfoam/user-guide-v8/running-applications-parallel

[15] Linux perf. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[16] Nvidia smi. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface

[17] Nvidia nsight systems. [Online]. Available: https://developer.nvidia.com/nsight-systems

[18] Mantis. [Online]. Available: https://github.com/SPEAR-IIT/mantis

[19] Job and queue scheduling on thetagpu. [Online]. Available: https://www.alcf.anl.gov/support-center/theta-gpu-nodes/job-and-queue-scheduling-thetagpu

[20] Nsight systems v2022.2.1 release notes. [Online]. Available: https://docs.nvidia.com/nsight-systems/ReleaseNotes/index.html