Towards Native Execution of Deep Learning on a Leadership-Class HPC System

Srikanth B. Yoginath, Maksudul Alam, Arvind Ramanathan†, Debsindhu Bhowmik, Nouamane Laanait, Kalyan S. Perumalla
Oak Ridge National Laboratory, †Argonne National Laboratory, USA
{yoginathsb,alamm,bhowmikd,laanaitn,perumallaks}@ornl.gov, †ramanathana@anl.gov

***Abstract*—Large parallel machines generally offer the best parallel performance with "native execution" that is achieved using codes developed with the optimized compilers, communication libraries, and runtimes offered on the machines. In this paper, we report and analyze performance results from native execution of deep learning on a leadership-class high-performance computing (HPC) system. Using our new code called DEEPEX, we present a study of the parallel speed up and convergence rates of learning achieved with native parallel execution. In the trade-off between computational parallelism and synchronized convergence, we first focus on maximizing parallelism while still obtaining convergence. Scaling results are reported from execution on up to 15,000 GPUs using two scientific data sets from atom microscopy and protein folding applications, and also using the popular ImageNet data set. In terms of the traditional measure of parallel speed up, excellent scaling is observed up to 12,000 GPUs. Additionally, accounting for convergence rates of deep learning accuracy or error, a deep learning-specific metric called "learning speed up" is also tracked. The performance results indicate the need to evaluate parallel deep learning execution in terms of learning speed up, and point to additional directions for improved exploitation of high-end HPC systems.**

***Keywords*-Deep Learning, Massively Parallel Systems, Parallel Speedup, Learning Speedup**

I. INTRODUCTION

Deep Learning (DL) has been successfully applied in commercial applications and is now being evaluated in other domains such as scientific computing. Parallel execution of deep learning is important when dealing with large data sets and individual datum sizes. In some challenging scientific applications, such as in material sciences and biology, the image volumes are large. Image sizes to learn from can also be large. Traditional small-scale execution using desktops or small clusters are ineffective to ensure analysis of real-world data sets and desired execution speeds. Deep learning codes need to utilize larger parallel computing platforms to make learning feasible on the newer applications. In this paper, we advance this direction towards effective parallel execution of deep learning codes on massively parallel machines containing thousands of graphical processing units (GPUs). We specifically focus on the runtime dynamics and efficiency considerations in delivering efficient parallel execution of deep learning on the latest heterogeneous architectures of massively parallel supercomputing platforms. Towards this, we study the execution of our parallel deep learning code on the Summit supercomputer hosted at the Oak Ridge Leadership Computing Facility.

Summit is based on a CPU-GPU hybrid architecture and utilizes many GPUs to achieve an extremely large number of floating point operations. The GPU-accelerated architecture of Summit is well-suited to the problem of deep learning because GPU-based execution has been originally instrumental in enabling great leaps in deep learning. While GPUs had been known to perform extremely well on naturally parallel tasks, their adoption for training deep learning networks has been symbiotic, resulting in a phenomenal success of both GPUs and deep learning. Furthermore, several high-performance computing systems realized the computational capabilities and energy savings provided by GPUs and became early adopters of GPUs for addressing large-scale computational science and engineering problems. These powerful platforms of the new generation bring with them an interesting set of advantages and technical challenges.

A. Native Execution

Traditionally, HPC systems executed large-scale scientific computing codes that were developed and tuned over a long period of time (sometimes over several years) to obtain the best runtime performance of the scientific application or the library. The programs are specifically developed to suit the hardware and software of the HPC system. Codes are compiled and linked with the vendor-supplied libraries for both computation and communication services. There is only a thin (or nearly non-existent) layer that mediates between the application and the hardware. Sometimes also called (nearly) *bare-metal* execution, *native execution* is one in which the program runs directly on the machine hardware and operating system, rather than inside virtual machines, containers, jails, or sandboxes. In the case of deep learning on our target supercomputer, native execution would entail the learner running with no additional software layers (such as hypervisor or container interface)

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

†Research performed while author was at Oak Ridge National Laboratory.

between the program and the hardware (CPU, GPU, and the network).

The native execution approach to HPC continues to date for many applications. However, the arrival of deep learning (or machine learning or artificial intelligence, in general) to computing has resulted in a major shift in the popular approach to HPC execution.

The changes to HPC execution brought in by the success of deep learning networks have been so overwhelming that the adoption of the new methodology has been relatively abrupt. The influence of rapidly developed deep learning codes has been so great that the execution mode on the largest computing systems has moved away from native executions to container-based executions, in order to rapidly cater to the urgent needs of executing popular deep learning libraries that depend on many third-party modules. Each third-party module presents its own challenges relating to compatibility and software restrictions. While this porting process solves the execution of deep learning networks on large systems, important runtime performance effects such as degraded device utilization or slow parallel convergence cannot be easily diagnosed and fixed. Further, backward compatibility is also rarely guaranteed by third-party libraries. The continual disruptions from backward incompatibilities and porting problems are generally unacceptable for scientific computing codes that are intended for long periods of usage. While scientific simulations are the mainstay of high performance computing systems, a large set of new deep learning-based solutions are currently being sought across scientific domains to address fundamental inefficiencies in large-scale/high-speed scientific experiments. They are also being championed to provide new innovative modalities of integrated simulation intelligently guided (on demand, sometimes in real time) by deep learning-driven workflows to expedite the process of scientific innovation.

To exploit the benefits of native HPC execution for deep learning, we developed a new deep learning library called DEEPEX. DEEPEX can be executed in a standalone mode or integrated into other native HPC application using its callable Application Programming Interface (API) for parallel execution.

B. Organization

In Section II, we introduce the software architecture of DEEPEX, our deep learning software library, which is used for runtime performance and scaling evaluation of benchmark applications in this paper. This is followed by Section III containing a discussion on the system-level details of the HPC platform and implementation. In Section IV, we introduce the benchmark applications, the datasets, and the deep learning networks used for performance evaluation. It is followed by a performance evaluation in Section V, where the runtime performance and the scaling behavior on various benchmarks are evaluated. We conclude the paper in Section VII with a summary and a brief overview of our future work.

II. SOFTWARE ARCHITECTURE

We based DEEPEX on NVIDIA's `CUDNN` library, which is the common denominator of many extant deep learning libraries. Currently, we are able to build any given convolutional neural network (CNN) and convolutional variational autoencoder (CVAE). The salient data structures of DEEPEX software are illustrated in Figure 1. For CNN, DEEPEX incorporates internal support for implementations of LeNet, AlexNet, VGGNet, and ResNet50, in addition to a customized CVAE network developed for a scientific application. We applied these implementations on two real life scientific problems. One arises from material sciences and it deals with simulated atom microscopy data generated by a multi-slice algorithm. The second is from the field of biological sciences using data from molecular dynamics-based protein folding simulations.

We optimized the performance of deep learning training for single-node execution before performing parallel computing evaluations. DEEPEX was developed to efficiently utilize the GPU compute power by minimizing the transfer of data between the host memory and the device memory. This is achieved by overlapping the reading of data with computation, and by caching in memory the data read from files. Also, the parallel software is coded in such a way that software overhead does not creep into the execution of CNN or CVAE in the parallel runs, as compared to single-node runs.

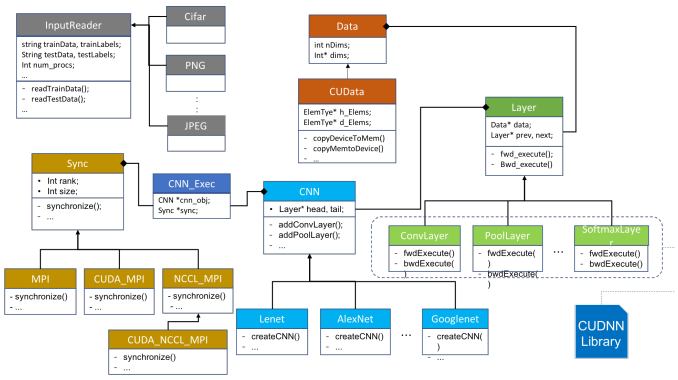

Fig. 1. DEEPEX software elements

a) Asynchronous Data Transfers: The input data was converted from the application's formats (JPEG, PNG, HDF5, etc.) to a standard single-precision NCHW (N-number of images, C-channels, H-height, W-width) format. We realized this using a `CuData` class object, which uses helper functions to move data from host memory to device memory. *CUDA streams* were used for each such object to realize asynchronous data transfers between main and device memories. Further, the data transfers between the host and device memories were also minimized. Only the input batch data is copied from host memory to device memory at the beginning of every batch and, the error and loss values are copied back from the device to main memory once at the end of each epoch.

b) Parallel Input Data Reads: Due to the availability of memory across many nodes in distributed execution and also due to a relative abundance of host memory on each node, we read the data from files only once and keep it in the host memory throughout execution. We maintained a hash table for the data read from files and used them to quickly access the data during the execution of successive epochs. Further, we also overlapped the reading of data from the input files (or from host memory) to device memory using `pthreads`. To accomplish this, we reserve the device memory that is twice the size of batch data for the input layer. As the `CUDNN` batch computations execute, the next batch size of data is copied asynchronously to the buffer input. By toggling between the two buffer inputs at the start of every batch execution, we ensured an overlap of asynchronous data read operations with batch training computations.

c) Stackable Layered Architecture: We realized the feed-forward networks in DEEPEX as a composition of layered objects. Each layer performs a well defined task; for example, convolution computations to learn the filters were performed in a convolution layer. Each layer composes an object of type `CUData` to store its intermediate results, weights, biases, and differentials. We used the corresponding service in the `CUDNN` API to realize the functionality of each layer.

d) Network Traversal: The individual layers can be composed together and traversed as a doubly-linked list. Each layer object in between the input layer (head) and the output layer (tail) is linked to its previous and the next layers. Hence, during the forward and backward computations, the device memories of the previous and next layers are directly invoked to read or write data, thus minimizing the data duplication. Feed forward execution is realized by carrying out the computations layer-by-layer as we traverse through the doubly-linked list of layer objects. Similarly, traversal backward from tail to head realizes the passage of differentials and computations of gradients. We used `CUDA streams` to perform certain independent asynchronous computations. For example, the weights and biases are asynchronously updated after the corresponding gradient computations.

e) Domain Decomposition: To realize parallel execution across distributed memory, we divide the computations in parallel such that the overall workload is shared among parallel computational units. We used a data-parallel approach, which, with minimal communication and synchronization requirements, offers a greater potential to improve runtime performance and scaling. We divide the input data files almost equally among the parallel processes. Each process reads its sequence of input data files and learns only from that data. However, at every epoch, the sequence of the file reads is varied by permuting them by random shuffling. At the beginning of the execution and after synchronization, we ensure that the weight and bias values across all the relevant layers in each of these parallel processes are exactly the same.

f) Data Communication: As a result of our domain decomposition approach, each process learns its variables separately while training on its input data set. The weight and the bias values that are learned by the deep learning networks can be on the order of several million, depending on the network being trained. To have a single consistent learned model after training would require the parallel processes to communicate their learned information periodically with each other. Hence, the variables from several layers of the network need to be transferred over the network, which can be either performed layer-wise or as one whole set. We use a synchronous method for updating weights and bias values of the entire network at once and not layer-by-layer. Our synchronization method is close to *local SGD* [1]. We use parallel *all-reduce* algorithm for synchronization. More sophisticated mechanisms for background synchronization and inter-network synchronization are also supported in DEEPEX but are not discussed in this paper due to their complexity.

g) Feed Forward Network Execution: The execution of feed forward networks happens in 3 phases: (1) forward phase, (2) backward phase, and (3) update phase. The initial weight and bias values of the convolution and the fully connected layers are random values (from standard distributions such as Gaussian, uniform, and Xavier). Mathematical operations are performed on the input data using these weights and bias values in the forward phase. The forward phase predicts the classification group and compares it with the known ground truth. The errors at the end layer are estimated. The differentials calculated from the errors are percolated backward layer-by-layer through the network in the backward phase. These differentials are used to compute weight and bias gradients. The mini-batch stochastic gradient descent method of learning with a momentum-based optimizer is used in the backward propagation. In the update phase, the weight and bias gradients are used to update the variable values in the relevant layers of the network. These three steps are iteratively performed many times on the same input data set during training. A loss function is employed to measure the classification error. When the error rate diminishes to a user-specified threshold, this iterative learning procedure is terminated. For validation, only the forward phase is performed.

III. IMPLEMENTATION AND EXECUTION

In this section, a brief overview is presented regarding the hardware of the massively parallel HPC system used in our performance study along with some implementation details of our software implementation.

The Summit machine is made of approximately 4,600 compute nodes containing a mixture of CPUs and GPUs. Each node contains two IBM POWER9 processors (two CPU sockets) and six NVIDIA Volta V100 accelerators (six GPUs) connected by high speed NVLINK connections. Each node has 512 GB of DDR4 memory for use by the CPUs, 96 GB of High Bandwidth Memory (HBM2) for use by the GPUs, and 1.6TB of non-volatile memory. Nodes are connected by dual-rail EDR InfiniBand network with a node injection bandwidth of 23 GB/s. The inter-node network has a non-blocking fat tree topology, which is implemented by a combination of a three-level tree of cabinet switches and inter-cabinet director

switches. Each IBM POWER9 CPU has 22 physical cores. Each physical core supports (multiplexes) 4 hardware threads, giving 88 hardware threads per core. One physical core per CPU is reserved for system operation, and the rest are available for user applications. Thus, each node has 176 hardware threads of which 168 hardware threads are usable by applications. Each CPU and 3 GPUs are connected via NVLINK2 links. Additional detail may be obtained from the online documentation of the machine at `www.olcf.ornl.gov`.

The DEEPEX code is written in C/C++ and CUDA, with calls to the native implementation of MPI (Spectrum MPI). CUDA-aware MPI is enabled using the `--smpiargs="-gpu"` options to the jobs. Jobs are launched via the `jsrun` job control system. On Summit, DEEPEX is launched with 6 MPI tasks per node, which maps 3 MPI tasks per CPU. Each task exclusively uses a single GPU. Therefore, the 6 MPI tasks use all the 6 GPUs on each node, one per task. Input image files are accessed through a GPFS high performance storage system.

Communication of synchronized data across deep learning networks spanning intra-node GPUs as well as inter-node GPUs is achieved via a combination of the Message Passing Interface (MPI) natively supported by the HPC system, and also via the NVIDIA Collective Communications Library (NCCL) interface natively supported by the GPUs. We have implementations to support direct MPI, CUDA-aware MPI and the NCCL library. We have exercised GPU-Direct MPI by explicitly managing all data structures on the device memory to avoid copying from/to host memory to/from device memory before/after MPI communication. We also exploit CUDA-aware MPI features that pipeline copy operations by merging them through one unified thread.

The data structures are also carefully organized to minimize the overheads in intra-node and inter-node communication. The memory layout is designed such that the number of MPI calls (for periodic synchronization of weights across nodes) is minimized. Specifically, the convolutional network elements are laid out in a way that makes it possible to invoke a single call per epoch to `MPI_Allreduce()` to synchronize all weights across all GPUs.

At initialization, we compute the memory size required for all weights in the learning network. We then allocate a single contiguous block of memory for all weights (on both host and device memories). Offsets into that single block are used to set up device pointers for the weight vectors with `CUDNN`. This scheme enables us to pass a single pointer to MPI to transfer all weights since they are contiguously placed as one vector in physical memory. Moreover, instead of host pointers, device pointers are directly passed to the MPI calls to bypass unnecessary device-to-host copy operations.

IV. BENCHMARK DATASETS AND DL NETWORKS

In this section, we introduce the benchmark applications, their data sets, and their corresponding deep learning networks. Three benchmarks are used: (1) Atom Microscopy, (2) Protein Folding, and (3) ImageNet.

A. Atom Microscopy

Deep learning has introduced a great amount of new enthusiasm in the scientific community to uncover the unknowns hidden in extremely large data sets from simulations as well as from sophisticated scientific tools such as electron microscopes. At the heart of Electron Microscopy (EM) lies the difficult inverse problem of inferring the three-dimensional atomic distributions that produce an electron diffraction image. It has been established that a single scanning convergent beam electron diffraction (CBED) image encodes complete information regarding the three-dimensional atomic distribution (that is, the 3-D positions of atoms and their chemical elements) [2]. In essence, by recovering this full information from CBED, most material properties of interest will be accessible at sub-nanometer spatial resolutions. Similarly, accurate numerical simulations of how electrons interact with a material to produce CBED are available [3]. The major caveat here is that the 3-D atomic potential of the material must be specified a priori as input. Moreover, a direct comparison between simulations and experimental images requires a large number of parameters to be included that characterize a particular electron microscope. The combination of the unknown 3-D atomic potentials and unknown instrumental parameters is where the difficulty of the inverse problem of CBED lies. The objective of the scientific problem is the prediction of the 3-D atomic potential using deep learning networks on the simulation data that also accommodates the experimental imaging conditions.

a) Dataset: The first dataset (AtomAI) contains several single-channel images of resolution 120×85 along with 27 labels. The data is generated from a prismatic simulation tool and the output 3-D atomic potentials are binned into 27 clusters. The training sample contains 656,100 images and the test sample contains 72,900 images, which are in the portable network graphics (PNG) format.

b) DL Network: To solve the classification task in this application, we employ convolutional neural networks (CNN). Specifically, we built a ResNet-50 CNN using DEEPEX for this purpose. In contrast to other feed forward networks such as AlexNet or VGGNet, the ResNet-50 network tries to optimize the residual mapping [5]. To accomplish this, ResNet-50 uses shortcut connections that several layers. These shortcut connections, in turn, result in multiple transitions from a few layers of the network. To realize this non-linear flow within the layered architecture of DEEPEX, along with its traversal to realize forward and back propagation, we created another layer called `AddLayer`. During forward propagation, the `AddLayer` combines data from multiple inputs. During back propagation, the differentials stored within this layer are exposed to all the layers to which it is connected. The operation of the `AddLayer` becomes slightly complicated when the tensor dimensions of the source of the shortcut connection and its destination do not match. We addressed this mismatch by composing an additional block of *convolutional*, *batchnorm* and *activation* layers within the `AddLayer` and

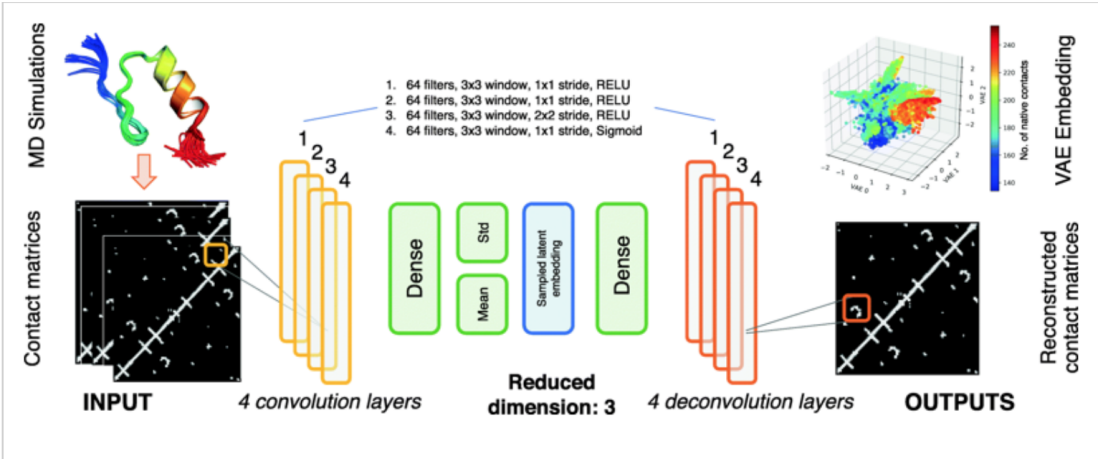

Fig. 2. Molecular dynamics simulation data converted to contact matrices are passed through the CVAE network that includes convolution, dense and latent layers. In the latent layer, a 3-D latent space is built using which the contact maps are reconstructed [4].

utilizing them under conditions of dimension mismatch.

B. Protein Folding

The next data set comes from a protein folding analysis application illustrated in Figure 2.

Molecular dynamics (MD) simulations are used to obtain insights on the events driving complex biological phenomena such as protein folding, ligand binding, and membrane formation. MD simulations are governed by a potential energy function that includes both bonded and non-bonded terms whose gradient defines a force-field applied to every atom in the bio-molecular system [6] [7] [8] [9]. These simulations apply Newtons laws of motion on every atom in the system and advances typically in femtosecond (10^{-15} second) time-steps. The molecular events of interest typically occur in the microsecond to millisecond timeframes. Machine learning is employed to obtain statistical insights on the time-dependent structural changes of a biomolecule in the simulation, to identify events that characterize large-scale conformational changes at multiple timescales, to build low-dimensional representations of the simulation data to obtain biophysical, biochemical or biological insights, to infer kinetically and energetically coherent conformational substates, and to perform a quantitative comparison with experiments [10]. However, ML approaches require well-designed and often handcrafted features. In contrast, the deep learning approaches learn the lower level representations (or features) from the input data. For example, Convolutional Variational Auto-Encoder (CVAE) can reduce the high-dimensionality protein-folding trajectories and cluster conformation from MD simulations into a small number of conformational states that share similar structural and energy characteristics [4] [11].

a) Dataset: The data set consists of over a million images of size 28×28 where each image represents a contact matrix between C^{α} atoms. The atoms are considered to be in contact if they are separated by less than 8Å. *MDAnalysis* library was used to extract the contact matrices from the MD simulation predicted trajectories. Additional details on the dataset can be found in the literature [4].

b) DL Network: Autoencoders follow the deep learning architecture and they capture key representational information of a data set in a low-dimensional latent space in an unsupervised manner [12]. The variational autoencoders constrain the autoencoder with the requirement of a normal distribution of the latent space [13]. With the convolutional layers, the CVAE learns the convolutional filter maps that can better recognize the local patterns independent of its position and these are better suited for the contact matrix data sets generated from the MD simulations.

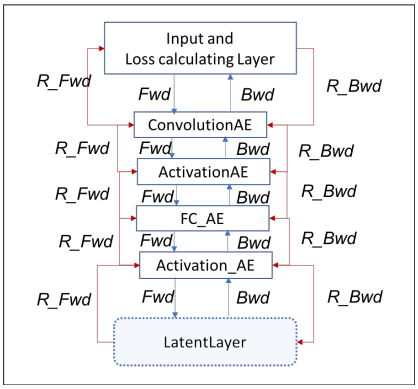

Fig. 3. Implementation of CVAE in DeepEx

To realize CVAE in DEEPEX, we derived an AutoEncoder (AE) version of the layer for all the implemented layers. For example, *convolutionLayer_AE* was derived from*convolution* layer. In addition to generic *forward_execute* and *backward_execute* methods represented as *Fwd* and *Bwd* in Figure 3 in all the layers, the AE version contains *reverse_forward_execute* and *reverse_backward_execute*, which are represented as *R_Fwd* and *R_Bwd*, respectively in Figure 3. Within the *reverse_forward_execute* functionalities like reverse convolution, reverse pooling and so on are implemented at relevant corresponding layers. The *reverse_backward_execute* function implements the back prop-

agation of differentials from the sum of RMSE loss and Kullback-Leibler (KL) divergence loss for the *decoder*. The *encoder* uses *forward_execute* for forward computation and *backward_execute* for back propagation of differentials in the encoder. Thus, the network traversal involves a sequence of *forward_execute* calls from all the layers till the traversal reaches *Latent Layer*. This is followed by a sequence of *reverse_forward_execute* calls from all the layers until the flow reaches the input layer. This is followed by a sequence of *reverse_backward_execute* calls until the traversal reaches *Latent Layer* again, and finally, a sequence of *backward_execute* calls until the traversal reaches *input layer* again. Our implementation of the CVAE network consists of three *Convolution Layer*, a *Fully Connected Layer*, and a *Latent Layer*. The convolution layers have 128 (1 input $\times$ 128 output channels), 16K (128 in $\times$ 128 out), and 16K (128 in $\times$ 128 out) filters of size 3×3 respectively. The fully connected layer has 100352×128 weights. Finally, the latent layer consists of 3 latent variables. In total, the network has 26M trainable weights and biases. During these traversals, the CVAE encodes the input, computes latent values, decodes using latent values, computes errors, propagates differentials, calculates weight gradients and updates weights.

C. Image Recognition

We also carried out performance experiments using the well known ImageNet data sets with a ResNet-50 CNN. The ImageNet is an image database used for visual object recognition software research. The database of annotations of third-party image URLs and is freely available. Since 2010, the ImageNet project has been running an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes. The ImageNet Challenge uses a trimmed list of one thousand unambiguous classes. A dramatic 2012 breakthrough in solving the ImageNet Challenge is widely considered to be the beginning of the deep learning revolution of the 2010s [14].

a) Dataset: In our experiments, we use 544,546 images for training and 50,000 images for validation. The data set was downloaded from the ImageNet website and each image was scaled to a size of 224×224. This image set contains color and monochrome images in the JPEG format. While extracting the data from the images, we used all the images with three channels corresponding to RGB values. For monochrome images, a single pixel value was replicated across all the channels.

b) DL Network: The ResNet-50 network discussed in Section IV-A was used for performance evaluation.

D. Summary of Application Datasets

Table I provides a summary of the important aspects of the datasets of the three benchmarks.

We refer to ResNet-50 CNN on the atom microscopy data benchmarks as *AtomAI* benchmarks. CVAE on protein folding data sets will be referred to as *PF* benchmarks. ResNet-50 on ImageNet datasets will be referred to as *ImageNet* benchmarks.

TABLE I
SUMMARY OF BENCHMARK DATASETS

Dataset	Size	Format	Ch	#Images	
				Training	Validation
AtomAI	120×85	PNG	1	656,100	72,900
PF	28×28	ASCII	1	1.1 M	N/A
ImageNet	224×224	JPEG	3	544,546	50,000

V. PERFORMANCE RESULTS

In this section, we evaluate the performance of DEEPEX with the three preceding benchmarks.

A. Parallel Scaling and Per Epoch Speedup

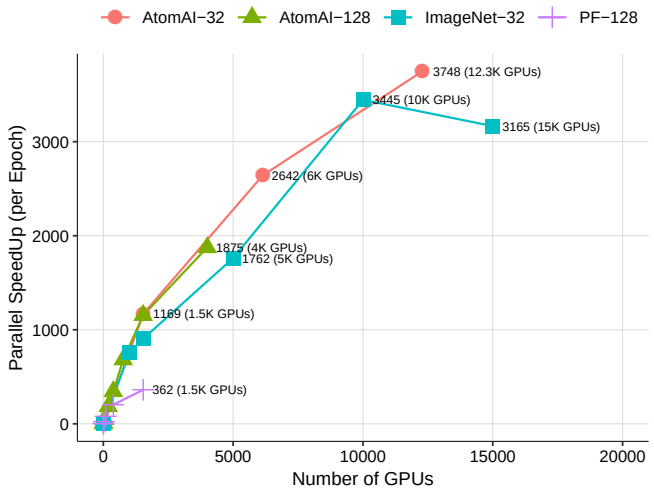

Fig. 4. Parallel SpeedUp per Epoch for different data sets and batch sizes

We calculate the per-epoch speedup (S_p) as

$$S_p = \frac{\tau_{spe}}{\tau_{ppe}}$$

where, τ_{spe} is per-epoch execution time on a single node and τ_{ppe} is parallel per-epoch time. An epoch is defined as one iteration of processing across all available images in the data set.

In Figure 4, we plot the per-epoch speedup for *AtomAI*, *PF* and *ImageNet* benchmarks with different *batch sizes*. The limit on the maximum number of GPUs that can be used with a data set is determined by the *batch size*: every GPU must have at least one complete batch size worth of images to process. Therefore, some data sets are limited to a smaller number of GPUs, while others span very large system size (up to 15,000 GPUs). The *Imagenet-32* benchmark with a *batch size* of 32 was run using up to 15,000 GPUs and *AtomAI-32* with an *batch size* of 32 benchmark was maximum of 12,000 GPUs.

The per epoch speedup observed for the *AtomAI* and *Imagenet* benchmarks that use ResNet-50 network appear to be almost similar. Overall, we see nearly linear speedups across

both benchmarks. We observe a parallel efficiency of around 75% at 1.5K GPUs and over 35% at 10K GPUs. However, there is a slight dip in the speedup *Imagenet* benchmark seen at the 15,000 run. We intend to investigate this high-mark data point later with additional runs to test reproducibility or eliminate the aberration. In contrast to the *AtomAI* and *ImageNet* benchmarks, the *PF* benchmark flattens even at 1,536 GPUs as seen in Figure 4, largely due to *PF*'s tiny image sizes.

B. Parallel Runtime Profile

We show the breakdown of parallel runtime for the *ImageNet* benchmark in Figure 5 and Figure 6. Similar breakdown for the *AtomAI-128* benchmark is provided in Figure 7 and Figure 8 and, for the *PF* benchmarks is provided in Figure 9 and Figure 10. The description of corresponding labels in the plots are provided in Table II.

TABLE II
PARALLEL RUNTIME PROFILE LABEL DESCRIPTIONS

Label	Description
IpUpdateTime	Time to read images; first epoch reads and caches the images in memory, so this time is shown amortized across epochs
TotalCompTime	Aggregate computation time (`FwdTime` + `BwdTime` + `WtUpdateTime`)
BwdTime	Time to compute backward propagation
FwdTime	Time to compute forward propagation
WtUpdateTime	Time to update weights
SyncTime	Time to synchronize weights (compute average) across all GPUs at the end of each epoch

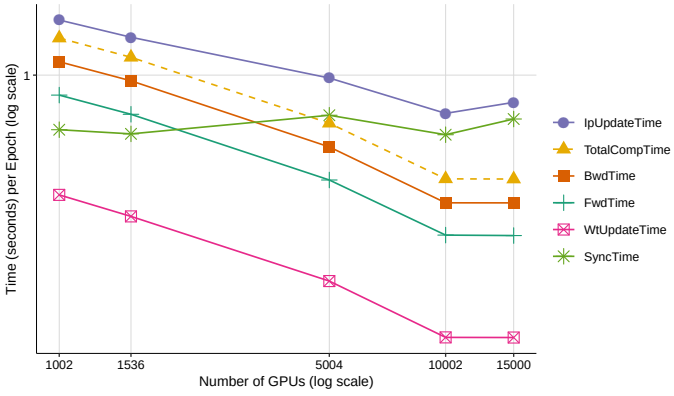

Fig. 5. Scaling trends of learning subtasks illustrated using time-per-epoch required to train ResNet-50 network on ImageNet data set

With an increasing number of GPUs, we observe that the forward propagation runtime, backward propagation runtime, input data update time and the weight update time decrease almost linearly. However, the synchronization time remains nearly unchanged. It is also seen that the insignificant communication cost in the parallel runs with a small number of GPUs becomes a significant runtime determinant on larger scale runs. These observations are consistent across all *ImageNet*, *AtomAI* and *PF* benchmarks. The first set of point in Figure 9 is from a single node run (6 GPUs), hence, the synchronization cost is relatively low.

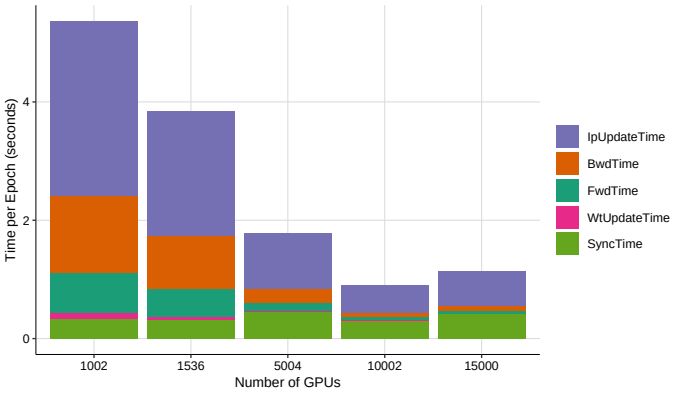

Fig. 6. Contribution of each learning subtask to overall per-epoch runtime cost to train ResNet-50 network on ImageNet data set

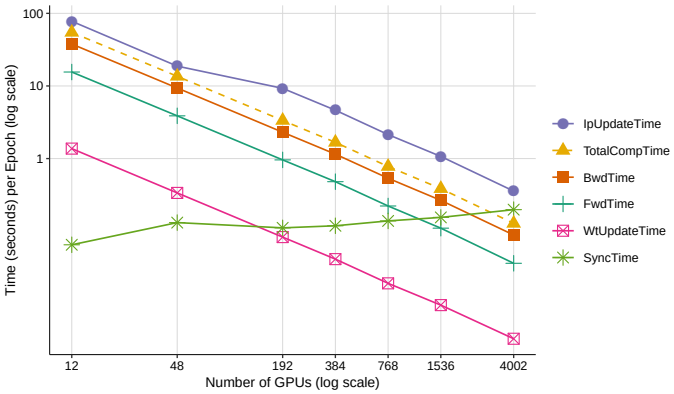

Fig. 7. Scaling trends of learning subtasks illustrated using time-per-epoch required to train ResNet-50 network on AtomAI data set

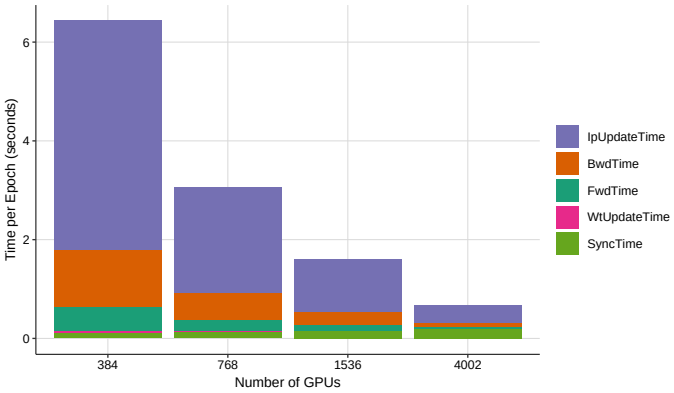

Fig. 8. Contribution of each learning subtask to overall per-epoch runtime cost to train ResNet-50 network on AtomAI data set

The computational load becomes comparable to the communication load at around 5,000 GPUs for the *ImageNet* benchmark, at around 4,000 GPUs for the *AtomAI* benchmark, and at around a few hundred GPUs in the case of the *PF*

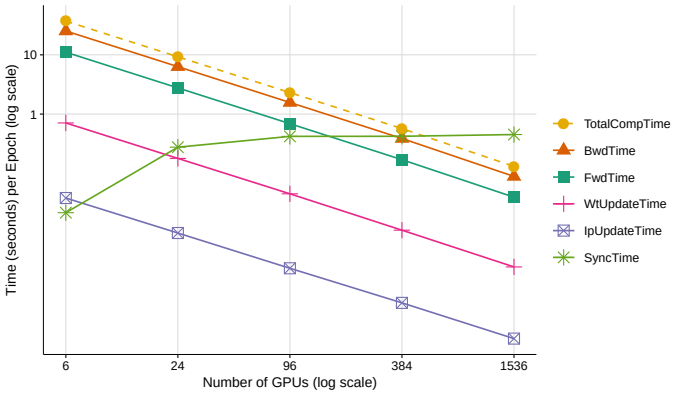

Fig. 9. Scaling trends of learning subtasks illustrated using time-per-epoch required to train CVAE network on Protein Folding data set

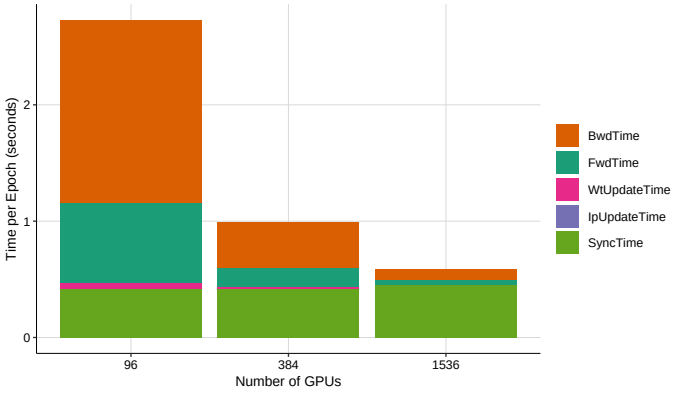

Fig. 10. Contribution of each learning subtask to overall per-epoch runtime cost to train CVAE network on Protein Folding data set

benchmark, as seen in Figure 6, Figure 8 and Figure 10, respectively. Beyond this point, the computational gains are mainly due to a reduction in the time to update the input data. Hence, the *ImageNet* benchmark at larger image dimensions and with three channels shows the best scaling behavior. The *AtomAI* benchmark with a slightly lower image dimension and single channel data sets that are larger in volume also shows good scaling behavior. However, the speedup of the *PF* benchmark with a very small image dimension of 28×28 flattens very quickly because neither computation nor data read time is significant on larger GPU runs.

The total byte size of the ResNet-50 weights exchanged across the network is approximately 89 MB, while that of the CVAE is approximately 100 MB. From the breakdown of time per epoch, it is observed that the synchronization cost from the collective operation of averaging across all GPUs is not affected significantly as the number of GPUs is increased from 1,000 GPUs to as much as 15,000 GPUs. From this observation, we infer that the communication is not a bottleneck, and the execution does not appear to be bandwidth-constrained. This shows the potential to sustain bigger image sizes and/or larger volumes of images. In particular, in emerging scientific data sets, the image sizes are expected to be significantly larger. For example, with *AtomAI*, it is possible to encounter images of sizes 512×512 or even larger.

C. Parallel Learning Speedup

Although we observe a very good linear per-epoch *computational* speedup on over 15,000 GPUs, the actual amount of *effective gain* in training large deep learning networks may be different. This is because the parallel training process, through which the per-epoch computation time is diminished, may be countered by an increase in the number of epochs taken by the parallel training task to converge to a similar solution as a serial task (or a smaller parallel task). In our experiments, for a fair comparison, we do not enforce any decay in the learning rate. We did this to ensure that a large number of epochs are not additionally penalized with lower learning rates due to decay.

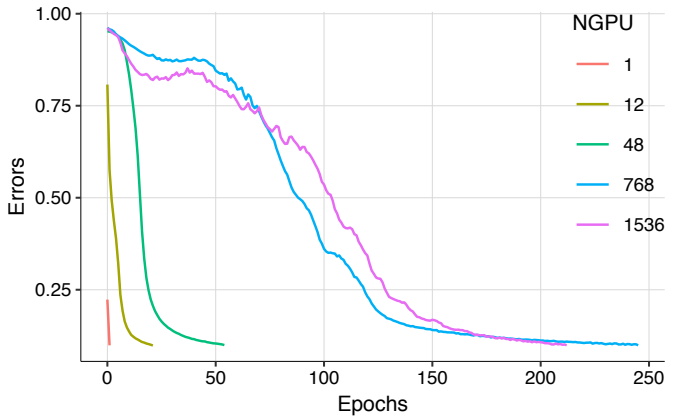

Fig. 11. Convergence for AtomAI data set with ResNet-50 network with a batch size of 128

a) Convergence: For classification problems using CNN, the error rate (ϵ) is calculated as $\epsilon = I_e/I_t$, where I_e is the number of erroneously predicted images and I_t is the total number of images processed.

Convergence in the *AtomAI* benchmark using ResNet-50 is observed when the error rate ϵ drops beneath a certain error threshold (`0.1`). The error rate reduction for the *AtomAI* benchmark for varying number of GPUs from 1 to 1500 can be observed in Figure 11.

For the *PF* benchmark, the Root Mean Square Error (RMSE) plotted are calculated using decoder-generated output (using latent variables) and its corresponding input. The RMSE decreases when CVAE learns the latent variables well and is able to decode images very similar to the inputs. Hence, convergence in the *PF* benchmark is achieved when RMSE drops to a certain predetermined threshold. The RMSE reduction can be seen in Figure 12 for training CVAE for varying number of GPUs from 1 to 1,536.

b) Learning Speedup: While the per-epoch speedup is a measure of the parallel computation, another useful application-level metric measures the gain in overall time for training the deep learning network in parallel. To obtain a perspective on the realistic gains from parallel execution, we

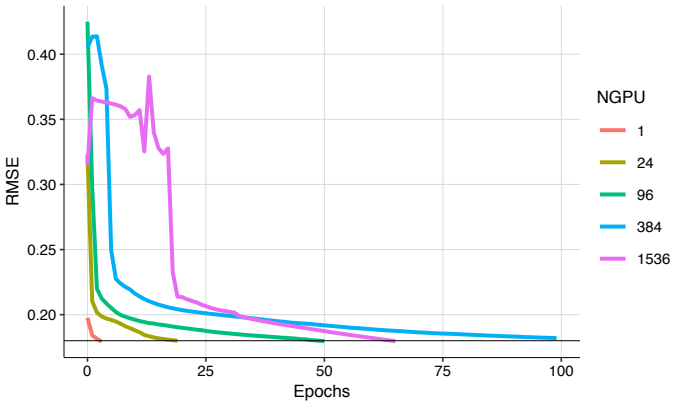

Fig. 12. Convergence for Protein Folding data set (with and without synchronization) with CVAE network with a batch size of 128

need to take convergence into consideration. We define the *learning speedup* of parallel deep learning (S_L) as

$$S_L = \frac{(\tau_{spe} \times N_s)}{(\tau_{ppe} \times N_p)}$$

where N_s is the number of epochs needed for a single node or a baseline training to converge and N_p is the number of epochs needed for the parallel training to converge.

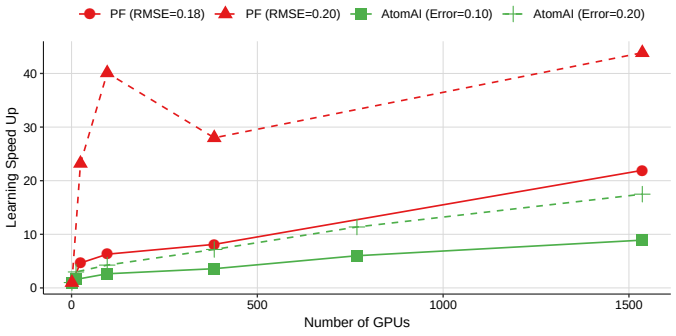

Fig. 13. Learning speedup for the AtomAI and Protein folding benchmarks, with each benchmark using a batch size of 128

The *learning speedup* S_L for the *AtomAI* and *PF* benchmarks is shown in Figure 13. With over 1500 GPUs, the *learning speedup* of 40 and 20 is observed in training *PF* benchmark for error rate cut-off values of 20% and 10%, respectively. Similarly, a *learning speedup* of 20 and 10 are observed in training *AtomAI* benchmark for error rate cut-off values of 20% and 10%, respectively. We observe that the non-linear *learning speedup* of the *PF* benchmark seems better than that of the *AtomAI* benchmark, which is in contrast with the per-epoch speedup presented in Figure 4.

We kept the learning rate fixed in these performance runs for fair runtime comparison. However, we are aware that faster convergence may be obtained by dynamically tuning the learning rates [15] [16]. We are also aware that by increasing the synchronization frequency, a faster convergence and hence better learning speedups may be obtained. However, due to diminished parallelism and increased communication, the faster convergence comes at the cost of degraded per-epoch parallel speedup. Determination of an optimal synchronization frequency per epoch to attain better learning speedups without compromising per-epoch parallel speedup is a part of our future work.

D. Validation Results

In this paper, the convergence plots for training or validation results for *ImageNet* benchmark are not obtained. This is because *ImageNet* was executed only for one hundred epochs, although several thousands of epochs are needed for convergence. Due to our focus on runtime performance, and to minimize the consumption of our limited wall-time allocation on the leadership-class HPC system, we turned off *parallel* validation and also limited the number of epochs to execute. Using the synchronized weights at the end of every epoch, we performed *sequential* validation on a single parallel task (rank 0). In Figure 14, we plot the training and validation curves for the *AtomAI* benchmark on 1,536 GPUs. We observe that the validation results closely follow the training results.

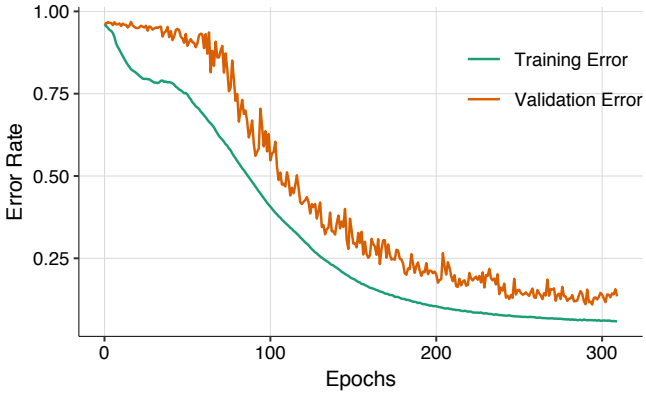

Fig. 14. Training and validation errors for AtomAI data set with ResNet-50 network with a batch size of 128 on 1536 GPUs

VI. Related Work on Scaling Deep Learning

Many recent studies have been conducted to scale deep learning tools to a large number of accelerated processors. In one of the early scaling efforts, ImageNet was trained by Goyal et al with the ResNet-50 network using 256 GPUs in an hour of wall clock time with the Caffe2 software [16]. This was performed on a cluster of servers, each server containing 8 NVIDIA Tesla P100 GPUs interconnected with NVIDIA NVLink. The servers were connected with Mellanox ConnectX-4 50Gbit Ethernet network cards and Wedge100 Ethernet switches. ImageNet has been later trained by Jia et al using 2048 GPUs with Tensorflow and ResNet-50 network [17]. Each of their computing nodes contained 8 NVIDIA Tesla P40 GPUs interconnected with PCIe. Nodes are connected using Mellanox ConnectX-4 100Gbit Ethernet network cards and RoCEv2 (RDMA over Converged Ethernet). More

recently, Mikami et al used up to 3264 GPUs to scale ImageNet classification with ResNet-50 networks using Neural Network Libraries (NNL) [18], [19]. Each of their compute nodes has 4 NVIDIA Tesla V100 GPUs linked with NVLink2. The computing nodes are connected using 2 InfiniBand EDR interconnects. In one of the notable scaling efforts not using the GPUs, a large climate dataset was trained by Kruth et al with Intel Caffe using 9600 Intel Xeon-Phi 7250 co-processors [20]. The computing nodes are interconnected by the Cray Aries interconnect utilizing the dragonfly topology. In another recent work, Ying et al used 1024 Google Tensor Processing Unit (TPU) v3 accelerators to train ImageNet classification with ResNet-50 networks using TensorFlow.

VII. Discussion and Future Work

In this paper, we report the current status of our ongoing effort to realize a scalable deep-learning library specifically for native execution on next generation exascale systems. This work highlights the possible role of deep learning in scientific computing and the need for native execution to support a backward compatible, efficient and scalable deep learning library. A new deep learning library called DeepEx has been developed with features including low software footprint, customization for native execution on large HPC systems with heterogeneous architectures, and efficient execution on the largest and fastest high performance computing facilities. Three benchmarks were introduced including two new scientific computing benchmark applications on which DeepEx was applied. We executed DeepEx on benchmark applications on up to 15,000 GPUs (NVIDIA Volta V100) of the Summit supercomputing systems and presented a detailed performance analysis using the three applications. Excellent scaling is observed in terms of parallel scaling and GPU utilization (see Figure 4). The results represent one of the largest scale executions of deep learning reported in the literature.

Here, we focused on attaining the best parallel scaling by minimizing the communication cost and maximizing the parallelism in computation, while still being able to converge. As a result, we see perfect linear per-epoch speedups with convergence on up to 1.5K GPUs. While we know that with an increase in the number of synchronizations per epoch, the rate of convergence significantly increases, we are yet to determine the optimal synchronization frequency per epoch to achieve the best combination of per-epoch scaling and overall learning convergence. With a single synchronization per epoch, we obtain the best per-epoch speedup, although the learning speedup is effected. We are currently exploring learning rate optimization methods such as layer-wise adaptive rate scaling [15] to attain faster convergence without significantly increased synchronization.

In our future work, we intend to investigate new performance optimization techniques (such as intra-network synchronization, exploitation of tensor cores, and large image size customization) to improve the parallel deep learning training process. Towards this end, we expect to evolve the algorithmic implementations accordingly and incorporate into the DeepEx library. This paper is, therefore, a first step in the direction of optimized native execution of deep learning codes on leadership-class HPC systems.

References

[1] T. Lin, S. U. Stich, M. Jaggi, Don't use large mini-batches, use local sgd, arXiv preprint arXiv:1808.07217.

[2] C. T. Koch, Using dynamically scattered electrons for three-dimensional potential reconstruction, Acta Crystallographica Section A 65 (5) (2009) 364–370.

[3] J. M. Cowley, A. F. Moodie, The scattering of electrons by atoms and crystals. i. a new theoretical approach, Acta Crystallographica 10 (10) (1957) 609–619.

[4] D. Bhowmik, S. Gao, M. T. Young, A. Ramanathan, Deep clustering of protein folding simulations, BMC Bioinformatics 19 (18) (2018) 484.

[5] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, CoRR abs/1512.03385.

[6] S. A. Adcock, J. A. McCammon, Molecular dynamics: survey of methods for simulating the activity of proteins, Chemical Reviews 106 (5) (2006) 1589–1615.

[7] D. Bhowmik, P. Ganesh, B. G. Sumpter, M. Goswami, Dynamical disparity between hydration shell water and rna in a hydrated rna system, Phys. Rev. E 98 (2018) 062407.

[8] G. K. Dhindsa, D. Bhowmik, M. Goswami, H. ONeill, E. Mamontov, B. G. Sumpter, L. Hong, P. Ganesh, X.-q. Chu, Enhanced dynamics of hydrated trna on nanodiamond surfaces: A combined neutron scattering and md simulation study, The Journal of Physical Chemistry B 120 (38) (2016) 10059–10068.

[9] V. E. Lynch, J. M. Borreguero, D. Bhowmik, P. Ganesh, B. G. Sumpter, T. E. Proffen, M. Goswami, An automated analysis workflow for optimization of force-field parameters using neutron scattering data, Journal of Computational Physics 340 (2017) 128 – 137.

[10] A. Ramanathan, A. Savol, V. Burger, S. Quinn, P. K. Agarwal, C. Chennubhotla, Statistical inference for big data problems in molecular biophysics, in: Neural Information Processing Systems Workshop on Big Learning, 2012.

[11] R. Romero, A. Ramanathan, T. Yuen, D. Bhowmik, M. Mathew, L. B. Munshi, S. Javaid, M. Bloch, D. Lizneva, A. Rahimova, A. Khan, C. Taneja, S.-M. Kim, L. Sun, M. I. New, S. Haider, M. Zaidi, Mechanism of glucocerebrosidase activation and dysfunction in gaucher disease unraveled by molecular dynamics and deep learning 116 (11) (2019) 5086–5095.

[12] P. Baldi, Autoencoders, unsupervised learning, and deep architectures, in: Proceedings of ICML workshop on unsupervised and transfer learning, 2012, pp. 37–49.

[13] C. Doersch, Tutorial on variational autoencoders, arXiv preprint arXiv:1606.05908.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: CVPR09, 2009.

[15] Y. You, Z. Zhang, C. Hsieh, J. Demmel, 100-epoch imagenet training with alexnet in 24 minutes, CoRR abs/1709.05011.

[16] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch SGD: training imagenet in 1 hour, CoRR abs/1706.02677.

[17] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, X. Chu, Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes, CoRR abs/1807.11205.

[18] H. Mikami, H. Suganuma, P. U.-Chupala, Y. Tanaka, Y. Kageyama, Imagenet/resnet-50 training in 224 seconds, CoRR abs/1811.05233.

[19] Sony, Neural Network Libraries (2018).
URL `https://nnabla.org/`

[20] T. Kurth, J. Zhang, N. Satish, E. Racah, I. Mitliagkas, M. M. A. Patwary, T. Malas, N. Sundaram, W. Bhimji, M. Smorkalov, et al., Deep learning at 15pf: supervised and semi-supervised classification for scientific data, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2017, p. 7.