DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models

Bogdan Nicolae*, Jiali Li†, Justin M. Wozniak*, George Bosilca†, Matthieu Dorier*, Franck Cappello*
*Argonne National Laboratory, USA
Email: {bnicolae,woz,mdorier,cappello}@anl.gov
†University of Tennessee, Knoxville, USA
Email: jli111@vols.utk.edu,bosilca@icl.utk.edu

***Abstract*—In the age of big data, deep learning has emerged as a powerful tool to extract insight and exploit its value, both in industry and scientific applications. One common pattern emerging in such applications is frequent checkpointing of the state of the learning model during training, needed in a variety of scenarios: analysis of intermediate states to explain features and correlations with training data, exploration strategies involving alternative models that share a common ancestor, knowledge transfer, resilience, etc. However, with increasing size of the learning models and popularity of distributed data-parallel training approaches, simple checkpointing techniques used so far face several limitations: low serialization performance, blocking I/O, stragglers due to the fact that only a single process is involved in checkpointing. This paper proposes a checkpointing technique specifically designed to address the aforementioned limitations, introducing efficient asynchronous techniques to hide the overhead of serialization and I/O, and distribute the load over all participating processes. Experiments with two deep learning applications (CANDLE and ResNet) on a pre-Exascale HPC platform (Theta) shows significant improvement over state-of-art, both in terms of checkpointing duration and runtime overhead.**

***Index Terms*—checkpointing; deep learning; fine-grain asynchronous I/O; multi-level data persistence**

I. Introduction

Deep learning applications are rapidly gaining traction both in industry and scientific computing. A key driver for this trend has been the unprecedented accumulation of big data, which exposes plentiful learning opportunities thanks to its massive size and variety. Unsurprisingly, there has been significant interest to adopt deep learning at very large scale on supercomputing infrastructures in a wide range of scientific areas: fusion energy science, computational fluid dynamics, lattice quantum chromodynamics, virtual drug response prediction, cancer research, etc.

To keep up with this trend, learning models are becoming increasingly more complex and exhibit deeper structures, prompting the need to employ more scalable training techniques. Such techniques involve the evaluation of several neural network (NN) architectures and their configurations, several potential data representations, and multiple workflows to train, evaluate, and analyze results. In this context, checkpointing is emerging as a key building block.

Traditionally, checkpointing has been used by HPC applications for *defensive* purposes, i.e., to survive failures that happen frequently at large scale using fault tolerance strategies based on *checkpoint-restart*. By capturing the state of learning models at regular intervals, checkpoint-restart is also a viable strategy in the context of deep learning for far more than resilience. Indeed, beyond resilience, checkpoints of learning models are increasingly being used for *productive* purposes: taken at regular intervals during training, they provide rich information about intermediate states. This facilitates further analytics to understand the evolution of training (used NN architecture optimization, configuration and evaluation), to identify correlations between training data and weight updates, and, in general, to explain why a model produces correct results, which is considered one of the grand challenges of deep learning.

Another particularly prominent use case for checkpointing is in the context of *transfer learning*, which involves partial training of a model, capturing its state and retraining/using it in a different context. This can done either to avoid retraining a model from scratch for a similar problem, or to design new search strategies that explore many alternatives in parallel starting from common ancestors. For example, a scientific use case for these capabilities is found in the Cancer Deep Learning Environment (CANDLE) Benchmarks [1], a collection of deep learning -based, cancer-relevant applications. These include the analysis of drug response data, molecular dynamics data, and clinical text data. An ongoing CANDLE study is the analysis of training data, in which partially trained NN models are duplicated and retrained on different data, potentially in a recursive fashion.

Although emerging as a critical building block, checkpointing has seen relatively little attention in the deep learning community. Current state-of-art approaches are rudimentary: they work on single machines and involve single files that emphasize portability over performance and scalability. On the other hand, checkpointing has been studied in-depth in the HPC community, where multi-level techniques that scale on supercomputing infrastructures and leverage heterogeneous storage are common. However, such techniques are not designed to handle the specific I/O patterns and scalability requirements that are involved in the checkpointing of deep learning models, which means they cannot be simply used as a drop-in replacement of the the rudimentary techniques. This problem is further complicated by the increasing size of the models and the complexity of training techniques, which

aim for both horizontal scalability (e.g., synchronized data-parallel training) and vertical scalability (e.g., fine-grain layer-wise parallelism).

This paper aims to address the aforementioned challenges by proposing a novel checkpointing framework for deep learning models that is designed from scratch to take advantage of the I/O patterns and specific properties of synchronous data parallel-training and layer-wise parallelism. In doing so, it acts as a bridge to advanced techniques employed by state-of-art multi-level HPC checkpointing approaches, thereby unlocking the potential to scale on supercomputing infrastructures. We summarize our contributions as follows:

- We introduce a series of design principles that enable efficient fine-grain asynchronous checkpointing of deep learning models. In particular, we emphasize the importance of combining lightweight serialization, sharding and augmentation of the execution graph to asynchronously mask the overhead of capturing weights from tensors without using a separate execution context (Section IV).
- We show how to materialize these design principles in practice as a transparent checkpointing solution on top of the *VeloC* (Very Low Overhead Checkpoint-Restart) runtime, which is a representative multi-level HPC solution. To this end, we introduce an architecture (Section IV-B) and present a reference implementation (Section IV-C).
- We evaluate our approach in a series of experiments conducted on Theta, one of the pre-Exascale systems hosted at Argonne National Laboratory. We use two deep learning applications: one is a popular benchmark used in the machine learning community (ResNet-50), the other is a real-life cancer deep learning research framework (CANDLE). Compared with state-of-art approaches, our proposal shows significantly better scalability and an order of magnitude less checkpointing overhead. (Section V).

II. Related work

Multi-level checkpoint-restart is a popular approach to leverage multiple storage levels in the context of HPC checkpointing. Works representative of this approach include (SCR) [2] and FTI) [3], which introduce support for local storage, partner replication, erasure coding (XOR and Reed-Solomon [4]) and finally external storage (parallel file systems). Recent efforts such as VELOC can take advantage of heterogeneous storage for each level and introduce advanced asynchronous techniques that leverage synergies between the levels [5] and predictions of application behavior to mitigate interference [6].

Exploiting local storage as a write cache layer to flush the application data to external storage asynchronously has been proposed before in the context of node-level aggregation of I/O from multiple cores [7], or I/O forwarding [8]. However, such efforts use a single level of caching, placing the emphasis on the aggregation. Other efforts such as [9] focus on smart ordering of asynchronous flushes from memory to local storage, which eliminate the need for blocking writes and are complementary to our approach.

Regarding the issue of I/O and storage for deep learning, both the HPC and deep learning communities have, so far, dedicated most efforts to access large training datasets efficiently [10], [11], [12], [13], while leaving the problem of optimized checkpointing of learning models largely ignored. TensorFlow checkpoints model to files in its SavedModel format,[1] or in HDF5 files through Keras.[2] Pytorch uses Python's Pickle module to serialize its model into files.[3] These file-based methods, while simple and adapted to training on a single machine, are becoming a bottleneck when scaling to a large number of compute nodes.

In ensemble model training [14], [15] and in hyperparameter search workflows [16], [17], we can expect each node to periodically checkpoint the neural network it trains, leading to I/O pressure in the order of typical scientific HPC applications that rely on a file-per-process approach. Hence improving model storage will become more critical. One step in the direction of improving model checkpoint is the CANDLE Model Cache [18], which proposes to use DataSpaces [19] as a distributed, remotely accessible cache instead of the parallel file system. The authors however do not rely on sharding neural network nor on asynchronous I/O to improve performance. Such a caching service is orthogonal to our own work and could be used in conjunction with it.

Compression is another technique that can be used to reduce the overhead of checkpointing by reducing the size of the models. To this end, lossy compression methods are particularly promising and have been designed specifically for deep neural networks. DeepSZ [20] determines appropriate error bounds for each of the neural network layers and assesses the loss of inference accuracy due to compressed layers. Weights quantization [21], [22] is another technique that reduces the precision of network parameters to gain space. Another class of data reduction techniques based on de-duplication of identical content across groups of processes [23] may be promising for deep learning models, especially if relaxed to look for similar instead of identical content. Once again, these data reduction techniques can be used to complement our own work.

To summarize, state-of-art checkpointing used by the deep learning community is rudimentary, while state-of-art checkpointing used by the HPC community is not designed to address the I/O patterns and scalability challenges emerging in modern deep learning applications, which limits their applicability. Our approach aims to fill this gap, taking advantage of such patterns to deliver high performance and scalability. To our best knowledge, *we are the first to explore this problem in-depth*.

III. Background

Deep learning (DL) algorithms are a class of machine learning algorithms that are based on complex neural networks with a large number of layers (hence called deep). They

[1] https://www.tensorflow.org/guide/saved_model
[2] https://www.tensorflow.org/guide/keras/save_and_serialize
[3] https://pytorch.org/tutorials/beginner/saving_loading_models.html

have have been successfully applied in a wide range of tasks: image recognition, machine translation, forecasting [6]. Such algorithms have increasingly gained attention in high performance computing as a complement to simulations (e.g., identify regions of interest, select promising initial conditions, etc.).

DL algorithms primarily use gradient descent to update the weights, an iterative technique that works as follows. First, the forward propagation step, where a training sample is used as the input of first layer of the neural network to compute its output, which is then propagated layer by layer, until a prediction of the result is obtained at the last layer. Then, the difference (gradients) between the predicted and actual result ("ground truth") is used to update the weights layer by layer up to the first layer. This step is called back-propagation. The goal is to converge to a minimum that is representative of all training samples and acts as an interpolation function for the whole problem. An important type of gradient descent is mini-batch gradient descent, where multiple training samples are used in the forward pass and the resulting average gradients is used for back-propagation. This speeds up the training process, both because fewer iterations are needed, and because there are fewer abrupt changes to the descent due to biased samples, which reduces the noise of finding the best direction to take.

Gradient descent is a computationally expensive technique. The explosion of available training data and the need to solve more complex problems have led to the introduction of deeper structures with more layers (e.g., complex residual networks that can be built with 1000+ layers, such as ResNet [24]). Therefore, gradient descent became not only more expensive to run because it needs to process more batches, but also because each batch itself is now more expensive to process. To solve this problem, distributed DL algorithms have been developed, capable of scaling horizontally on multiple compute nodes.

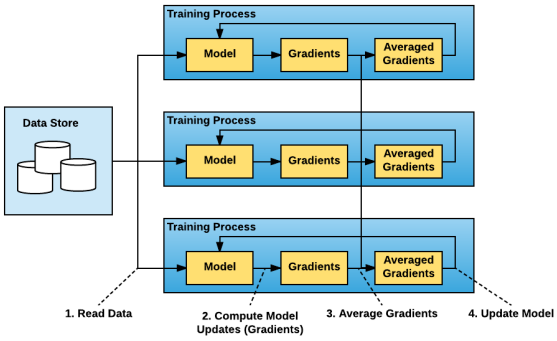


Fig. 1: Synchronous data-parallel training.

The most widely used such technique is *synchronous data-parallel* training. It leverages the idea of creating replicas of the learning model on multiple nodes and training each replica in parallel with a different batch. We denote as *rank* a process responsible for training an individual replica (which is the usual terminology in high performance computing).

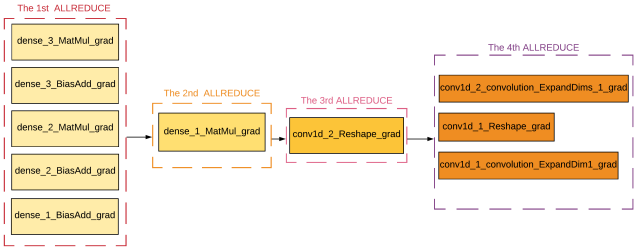


Fig. 2: An example of tensor fusion obtained from the Horovod timeline for the CANDLE-NT3 benchmark.

Forward propagation can be done in an embarrassingly parallel fashion. However, during back-propagation, the weights are not updated with the local gradients, but with global average gradients computed across all ranks using all-reduce operations. This process is illustrated in Figure 1.

DL algorithms take advantage of multi-core and hybrid architectures (e.g., CPUs + GPUs) to parallelize the gradient computation and weight updates. Specifically, once a rank has finished computing the local gradients for a layer, it immediately proceeds to compute the local gradients of the previous layer. At the same time, it waits for all other ranks to finish computing their local gradients for the same layer, then updates the weights based on the average gradients obtained using all-reduce. This is called layer-wise parallelism. An example is depicted in Figure 3 as a DAG (directed acyclic graph): the local gradient of each layer is a dependency for both the previous layer and the rest of the operations (all-reduce and weight updates; for now the reader can ignore shard extraction, which will be explained later). Once the local gradients are computed, both paths in the DAG can be executed in parallel. Over time, several runtimes that implement such ideas have become popular, such as Tensorflow [25], Caffe [26] and Torch [27].

The combination of synchronous data-parallel training and layer-wise parallelism has proven especially popular and many deep learning approaches have introduced support for them: Distributed Tensorflow, Distributed Torch, etc. Some of these runtimes can use MPI as the underlying communication layer that provides an optimized all-reduce implementation, which is a natural fit for supercomputing architectures. A particular implementation, *Horovod* [28], has gained significant traction in production because it can leverage MPI to take advantage of an optimized all-reduce implementation for high-end networking infrastructures, while integrating seamlessly with the Python ecosystem and the high-level machine learning libraries (such as *Keras* [29]) that emphasize ease of use and convenience.

However, reconciling MPI with layer-wise parallelism is non-trivial, because MPI was not designed to support multiple parallel all-reduce operations that need to operate with potentially small data sizes from within the same rank. Therefore, optimizations such as *tensor fusion* have emerged that adopt a producer-consumer model: all-reduce from individual tensors

are collected in a buffer while a separate thread continuously runs MPI all-reduce, combining ("fusing" together) the buffered all-reduce that have accumulated since the last MPI all-reduce call into a single new call to be executed next. An example is depicted in Figure 2.

It must be noted that the combination of layer-wise parallelism and synchronous data-parallel training, although introducing significant complexity, also opens new opportunities, which form the core of this work and will be discussed next. Also, note that checkpointing is a broad primitive in the context of deep learning: it is a basic building block for many productive scenarios (as discussed in Section I), which are in addition to fault tolerance. Therefore, in this paper we assume the need to checkpoint frequently (potentially more often than the optimal checkpoint interval needed to survive failures), which simultaneously satisfies both aspects.

IV. System design

This section introduces the design principles, architecture and implementation of our approach.

A. Design principles

Our proposal is based on the following general design principles:

a) ***Asynchronous multi-level checkpointing****:* We propose a multi-level approach that combines "lightweight" persistence strategies (involving local storage of neighboring nodes to perform replication and erasure coding) with "heavy" persistence strategies (flushing to external storage such as a parallel file system). Using this approach, checkpoints are preserved in a reliable fashion for the duration of a job and beyond. A key goal is to block the training for as little as possible during checkpointing. To this end, we introduce an asynchronous approach that captures a local copy of the learning model, while applying both the lightweight and heavy persistence strategies in the background, while the training continues running. These background operations can run with low priority to avoid negative impact on the application due to interference. A key challenge that differentiates this approach from traditional HPC multi-level checkpointing is the fact that local copies can be expensive. We address this challenge below.

b) ***Hidden complexity of heterogeneous storage****:* Storage is becoming heterogeneous both at node-local (multiple types of volatile and persistent memory, SSDs, etc.) and external level (burst buffers, key-value stores, parallel file systems). Many users are simply unaware of the various types of storage available on the nodes where they need to run data-parallel training. When they are aware of them, most do not fully understand the performance characteristics. Even for the minority of users that are both aware of heterogeneous storage and understand their performance characteristics, leveraging heterogeneous storage is problematic because state-of-art approaches were not designed to take advantage of them: most are limited to single destinations (e.g., a single file on a parallel file system). To address this problem, we propose a transparent solution that automatically detects, mixes and matches heterogeneous storage using vendor-specific APIs when available for optimal performance. This is done in close coordination with asynchronous multi-level checkpointing, introducing awareness of fine-grain I/O operations and optimal flushing strategies based on producer-consumer strategies that rely on performance modeling [5].

c) ***Efficient serialization on local storage****:* Even when advanced asynchronous techniques are employed for multi-level checkpointing, serialization to local storage can still incur significant overhead. This is due to the fact that the models can have deep structures that involve many layers and tensors and it is non-trivial to collect and consolidate the necessary information. Despite this challenge, state-of-art approaches often trade off performance for portability, using self-descriptive formats for model checkpoints (e.g., HDF5) that are expensive to produce. We argue in favor of lightweight serialization approaches that prioritize performance. This is based on the assumption that for the frequent checkpointing scenarios we target in this paper, it is sufficient to capture the weights of the model alone, because the structure of the model changes less frequently and therefore can be captured in a separate checkpoint on a per-need basis. Based on this idea, we make use of a compact binary format that leaves out unnecessary details (e.g. labels of tensors) and minimizes the necessary I/O operations required to assemble a checkpoint.

d) ***Sharding for data-parallel training****:* Synchronous data parallel training approaches use the same gradients to update the weights of each layer. Therefore, at the end of each iteration (when it is safe to checkpoint the model), there will be identical replicas of each layer available on the nodes where the ranks are running. We exploit this property to further reduce the I/O overhead of serialization to local storage as follows: We slice each layer into a number of shards equal to the number of ranks, then each rank writes a different shard to local storage. Since the local storage is not shared, this effectively distributes the I/O workload in a scalable fashion across all ranks. Note that we decided to slice each layer independently instead of grouping all layers together and then slicing the resulting checkpoint. Although the latter may reduce the required I/O operations (i.e., write a single large shard instead of many smaller shards) it is also limiting with respect to further optimizations, which is why we chose the former. Such optimizations will be discussed next.

e) ***Asynchronous extraction of shards during back-propagation****:* Modern machine learning frameworks are composed of multiple layers of low-level and high-level libraries that offer a trade-off between convenience and simplicity vs. high-performance and fine-tuning. High-level libraries are often implemented in high level languages (e.g. Python) and do not have direct access to the data structures of low-level libraries. Therefore, there are restrictions in terms of when and how it is possible to access such low-level data structures. For example, Tensorflow requires high-level libraries like Keras to create a separate graph execution context in order to extract the value of tensors as high-level Python data structures (e.g.,

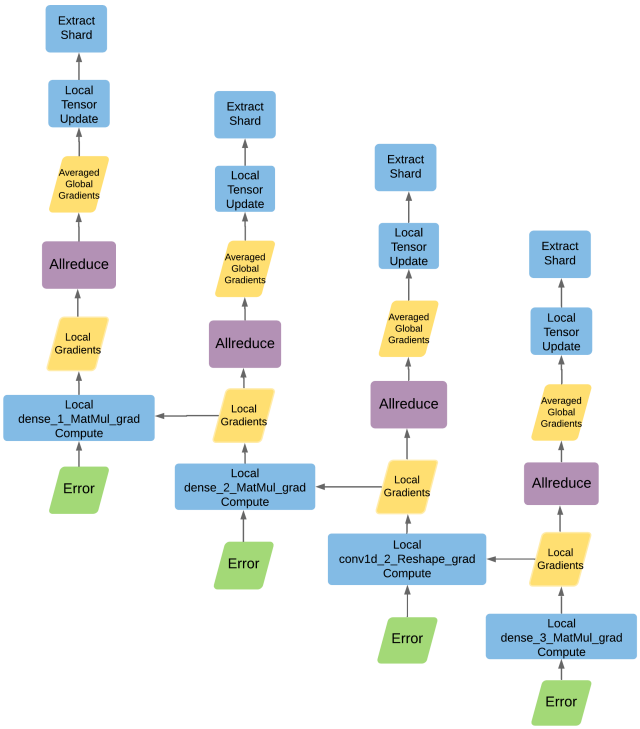


Fig. 3: Example: fine-grain sharding during back-propagation, including the pipeline between the shards.

numpy arrays). This can introduce high overhead by itself, even before being able to perform sharding and lightweight serialization. To address this challenge, we leverage the observation that during back-propagation, weight updates for higher layers run in parallel with the gradient calculation and all-reduce synchronization of the lower layers. Thus, we propose to augment the execution graph by introducing an additional slicing operation immediately after the weight updates in each layer. An example of how this works is depicted in Figure 3, where each computation of the local gradients activates the previous layer, while in parallel advancing towards the weight updates and sharing. Using this approach overlaps the access to the tensors and the slicing with the rest of the operations in the same execution context, which both avoids the need to create a separate execution context and takes advantage of fine-grain asynchronous parallelization opportunities.

B. Architecture

We adopt the design principles introduced in Section IV-A into the architecture depicted in Figure 4.

It consists of three major components: *VELOC*, a low overhead runtime specifically designed for scalable, high-performance asynchronous multi-level checkpointing for HPC applications [5], a checkpointing module responsible to capture tensors to local storage and a bindings library that interfaces the checkpointing module with VELOC. Both the checkpointing module and the bindings library are new components written from scratch and integrated with VELOC.

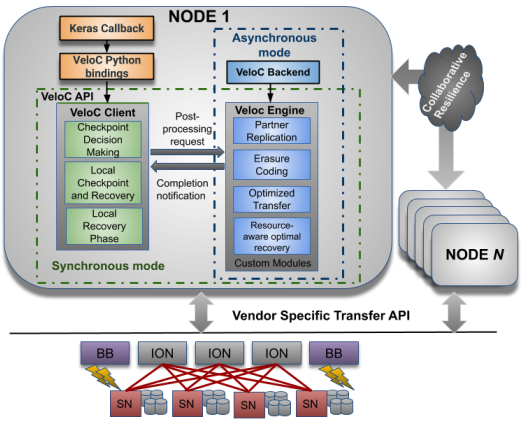


Fig. 4: Architecture of our proposal based on the VELOC (Very Low Overhead Checkpoint-Restart) runtime.

The checkpointing module encapsulates the main contribution of this work and exposes the ability to capture the weights of the learning model by means of a Keras callback, which needs to be added to the list of callbacks supplied to the model.fit method, responsible to run the training process. From the user perspective, this is the only action needed to activate checkpointing support. All optimizations related to efficient serialization of the weights, sharding and augmentation of the execution graph for asynchronous extraction of shards are the responsibility of the checkpointing module.

The library providing Python bindings is a thin intermediate layer specifically optimized to efficiently pass numpy arrays, the main data structures used in Python to represent the content of tensors, to the VELOC *client*, which is responsible for exposing a memory-oriented API to save contiguous regions as a checkpoint into the local storage.

The VELOC *engine* is responsible for running the module pipeline. The default modules perform post-processing on the local checkpoints, which includes both collaborative resilience (e.g. replication and erasure coding using partner compute nodes) and optimized transfer support to heterogeneous external storage (e.g. parallel file systems, burst buffers, key-value stores, etc.) using vendor APIs (where applicable). This is where the assembly of the shards is happening to construct a full checkpoint on external storage.

Two modes of operation are supported by VELOC: *synchronous* and *asynchronous*. In the synchronous mode, the client and the engine are linked together using a trivial control plane in the same front-end library used by the application. Both the local checkpointing and the post-processing are blocking operations. In the asynchronous mode, the engine instance is created only once per node and lives in a separate active back-end. All clients connect to the same active back-end using a configurable control plane that is based on shared-memory or RPC libraries. In this mode, the clients do not need to wait for the engine to finish and can resume the application immediately after the local checkpoints were written and the engine was notified about their existence. For the purpose of

this work, we use the asynchronous mode.

C. Implementation

We implemented the checkpointing module on top of Tensorflow 2.0, which includes an optimized version of Keras tightly integrated with it. In this context, we aim for two design goals. First, we expose an API that is compatible with the existing checkpointing mechanism in Keras, which enables users to perform minimal changes to the code to integrate our approach, therefore aligning to the overall design goal of Keras, i.e. provide ease of use and convenience at high level. Second, we isolate the modifications necessary to augment the execution graph into Keras, which means our approach works out of the box with an existing binary distribution of Tensorflow. This is a very important aspect, because many vendors adapt Tensorflow for their machines by integrating it with custom low-level libraries (e.g. Intel MKL), making it challenging if not impossible to modify, recompile and fine-tune Tensorflow.

To achieve these goals, we adopt the following strategy. In terms of API, we provide a Python class that extends the Keras callback interface and overrides the on_batch_begin and on_batch_end methods. This class can be configured to checkpoint every K iterations, simply ignoring the batch events when it is not time to checkpoint. Otherwise, at the beginning of a batch after which a checkpoint is needed, it sets a boolean tensor to True. This will activate the sharding embedded into the execution graph, which otherwise is inactive. Then, after the batch was completed, it uses the VELOC Python bindings to checkpoint the tensor shards, resets the boolean tensor to False and returns control to the main loop of model.fit. The user simply needs to invoke model.fit with this class added to the list of callbacks.

In order to augment the execution graph, we intercepted the apply_gradients method of the the base optimizer class of Keras (keras.optimizer_v2.OptimizerV2). This method is responsible for building the execution graph for the weight updates, into which we injected additional sharding code that conditionally activates based on the boolean tensor defined in the callback. The sharding itself is implemented in an optimized fashion using Tensorflow's own slice operator. Since tensors can be multi-dimensional and slicing requires a single dimension, we choose the largest dimension. This ensures the best load balancing. Note that all overhead of calculating the largest dimension and adding slice operations is performed only once during initialization when the execution graph is built.

The Python bindings were implemented using ctypes, which facilitates easy integration with C and C++ external libraries. It takes advantage of the fact that numpy arrays are internally represented as contiguous memory regions, which enables it to avoid any extra copies when calling the VELOC API. We integrated the bindings with VELOC v.1.2, which has a modular design that enables the user to configure which plugins to activate that implement multi-level strategies. For the purpose of this work, we activated only the transfer module that is responsible for optimized background flushes to external storage.

V. Evaluation

A. Experimental Setup

For our experiments we use *Theta*, a 11.69 petaflops pre-Exascale Cray XC40 system based on the second-generation KNL Intel Xeon Phi 7230 SKU. The system is equipped with 4392 nodes, each containing a 64 core processor (256 hardware threads) with 16 GB of high-bandwidth in-package memory (MCDRAM, 300-450 GB/s), 192 GB of main memory (DDR4 RAM, 20 GB/s), and a 128 GB SSD (700 MB/s). The interconnect topology is based on Dragonfly with a total bisection bandwidth of 7.2 TB/sec.

For the purpose of this work, we configured KNL to run in caching mode, which means the MCDRAM acts as a cache (implemented in hardware) for the main memory. This is the recommended configuration for deep learning applications, since memory bandwidth has an important impact on performance. The file-system used as local storage is ext4, which is deployed on top of the SSD. The external storage is provided by a Lustre parallel file system deployment (aggregated bandwidth 250 GB/s), which is mounted using POSIX.

In terms of deep learning software, we use *Horovod* v.0.18.1 and *Tensorflow* v.2.0. Note that Tensorflow v.2.0 comes with its own optimized *Keras* library, which we use for our experiments. Furthermore, all these libraries are compiled with optimized support for the KNL architecture by taking advantage of Intel's Math Kernel Library (MKL) and Intel's own Python distribution. Our modifications to Tensorflow are contained within Keras and concern Python code exclusively. Therefore, our approach takes full advantage of the aforementioned optimizations.

B. Methodology

Our work focuses on scenarios where the state of the learning model needs to be checkpointed with high frequency during the training. In this context, the critical state that changes between batch updates are the weights of the layers, which are the focus of our experiments. We assume the rest of the parameters (architecture of the model, training configuration, state of the optimizer) are checkpointed separately as needed.

We compare the following approaches throughout our evaluation.

***Keras-Default**:* This is the default checkpointing available in Keras. Specifically, the user has to register a callback with the model, which is used during the training to signal when a step and/or epoch was completed (which is a safe moment to checkpoint). In the callback, the weights of the model are saved using model.save_weights(ckpt_file), which uses the HDF5 library to serialize the weights in the specified file. Since the weights of all model replicas are synchronized at the end of each batch, only one rank needs to save the weights (we choose by convention rank 0). This operation is

blocking and causes rank 0 to lag behind in the next batch, which ultimately causes and overall performance overhead due to synchronization.

VELOC-Single: A similar approach with Keras−Default, except that it relies on model.get_weights() to obtain the list of all weights as numpy arrays, which are then serialized in bulk on the local storage using the VeloC Python bindings. Again, this is a blocking operation that happens on a single rank. However, unlike the case of Keras−Default, the serialized weights are flushed to external storage (Lustre parallel file system) asynchronously in the background while the training continues.

VELOC-Sharded: This approach adds sharding on top of the VELOC−Single approach. Specifically, each rank obtains the list of weights as numpy arrays and then extracts for each array a slice corresponding to its index. The size of the slice is the total size of the array divided by the number of ranks. Then, each rank independently serializes the slices of all arrays using the VeloC Python bindings. Each rank flushes the serialized slices to external storage in the background.

VELOC-Opt: This is the optimized approach that hides the overhead of extracting and slicing numpy arrays from tensors by embedding these operations directly into the execution graph of Tensorflow. In this case, the only blocking operation on all ranks is the serialization of the slices using the VeloC Python bindings. Same as in the case of VELOC−Sharded, each rank flushes the serialized slices to external storage in the background.

These approaches are compared based on the following metrics.

Blocking Phase: This metric corresponds to the duration of all blocking operations performed during the checkpointing callback that is invoked on batch completion. In the case of Keras−Default, it includes all overheads associated with extracting numpy arrays from tensors, serialization into the HDF5 format and writes to external storage. For VELOC−Single and VELOC−Sharded it includes all overheads associated with extracting numpy arrays from tensors, slicing and serialization using the VeloC Python bindings (writes to external storage are asynchronous). For VELOC−Opt, it includes just the overhead of serialization using the VeloC Python bindings (everything else is performed asynchronously). It is calculated as the average of all checkpoints performed by all ranks. This metric is important because it exposes how much time an individual rank loses on the average if it is involved in checkpointing. It directly impacts scenarios where training is stopped after checkpointing to use the model in a different context.

Preparation Phase: This metric targets VELOC−Single and VELOC−Sharded. It measures the overhead introduced by extracting numpy arrays from tensors and, in the case of VELOC−Sharded, performing the slicing. It is calculated as the average of all checkpoints performed by all ranks. This metric is important because it emphasizes the overhead of post-processing tensors outside of the execution graph in a blocking fashion, which translates to a direct increase of the duration of the blocking phase for the two approaches.

Runtime Overhead: This metric evaluates the runtime overhead caused by checkpointing for the whole group. In the case of VELOC−Sharded and VELOC−Opt, all ranks are checkpointing and therefore a slowdown is experienced by the whole group during the same iteration. In the case of Keras−Default and VELOC−Single, only rank 0 is checkpointing while the rest move on to the next iteration. Therefore, rank 0 lags behind in the next iteration, causing a slowdown for the whole group there. Therefore, to ensure a fair comparison, we measure this slowdown by calculating the average of all iterations where a checkpoint is taken and the corresponding iterations that are immediately following, from which we subtract the baseline (average duration of iterations without checkpointing). This metric is important because it exposes the end-impact when the training is continued after checkpointing, including the interference caused by asynchronous operations.

C. Applications

We study two representative deep learning applications, each of which can benefit from checkpointing in a variety of scenarios, as outlined in Section I.

1) CANDLE NT3: CANDLE [14] (Cancer Distributed Learning Environment) is a project that aims to combine the power Exascale computing with deep learning to address a series of loosely connected problems in cancer research. Each such problem is driven by a series of benchmarks. One such direction (Pilot 1) aims to predict drug response based on molecular features of tumor cells and drug descriptors. In this context, we study on NT3 [1], which consists of a 1D convolutional network for classifying tissue, expressed as gene sequences, as normal or tumorous. This type of network follows the classic architecture of convolutional models with multiple 1D convolutional layers interleaved with pooling layers followed by final dense layers. The optimizer used by NT3 is SGD (stochastic gradient descent). The training data size for this benchmark is $\approx$ 600 MB, which includes 1120 training samples. We adapted NT3 for data-parallel training by introducing a partitioning scheme that evenly distributes the training data to the ranks and a new distributed optimizer based on Horovod.

2) ResNet-50: is a deep neural network where the layers learn residual functions with reference to the input layers, instead of learning unreferenced functions. This allows ResNet to train extremely deep neural networks with 150+ layers, which was difficult prior to its introduction due to the problem of vanishing gradients [24]. Thanks to this breakthrough, ResNet became a highly popular image classification benchmark especially in a simpler form that uses 50 layers. We study this form, called ResNet-50. A data parallel implementation is shipped together with Horovod as an example [30]. The optimizer used by this implementation is also SGD. As training data, we use the ImageNet dataset [31], which is $\approx$ 200 MB large and includes 100,000 samples. The training set of each worker is randomly sampled from the training data.

D. Results

We focus our study on the weak scalability of data-parallel training for each of the approaches introduced in Section V-B. In this case, the batch size remains constant as the number of ranks increases, which means more training data is processed with each iteration. This is the most popular data-parallel training scenario.

TABLE I: Application parameters

Application	Batch size	Ckpt tensors	Ckpt size
CANDLE-NT3	20	9	600 MB
ResNet-50	32	27	100 MB

We run a single Horovod rank per node. Each rank runs a Tensorflow instance that was configured to use two intra-threads (used to parallelize single operations internally) and 128 inter-threads (used to parallelize independent operations in the graph). These are the optimal settings for the Theta pre-Exascale machine according to previous findings [32]. The application parameters are listed in Table I. We include both the number of tensors holding the weights to be checkpointed and their total size.

TABLE II: Application performance: average duration of iterations

Application	1 node	2 nodes	4 nodes	8 nodes
CANDLE-NT3	2.7s	2.8s	4.2s	5.1s
ResNet-50	4.79s	5.12s	5.29s	5.35s

Table II lists the average duration of the iterations without any checkpointing as the number of nodes increases from one up to eight. This is used as a baseline. Note that there is a significant increase in the average duration as the number of nodes increases, which is explained by increasingly larger synchronization overhead introduced by frequent all-reduce operations.

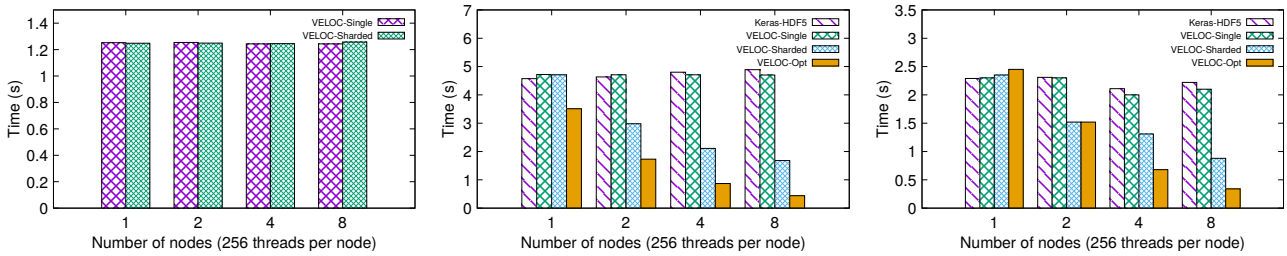
For the purpose of this work, we take a checkpoint every 15 iterations, which amounts to a total of 8 checkpoints for CANDLE−NT3 and 5 checkpoints for ResNet−50. The metrics introduced in Section V-B are averages of these checkpoints.

The results for CANDLE−NT3 are depicted in Figure 5. As can be observed, the preparation phase for VELOC−Single and VELOC−Sharded has a significant overhead, which is almost half of the duration of an iteration when using a single rank. As can be observed, this overhead is close for both approaches. Therefore, we conclude that the dominating factor of the preparation phase is the conversion from tensors to numpy arrays, which requires an invocation of the Tensorflow backend and costly initialization of Python data structures. The additional slicing performed by VELOC−Sharded on the numpy arrays seems to introduce negligible overhead, which can be explained by the fact that the model consists of few tensors of large size, therefore few slicing operations are needed. Furthermore, the preparation phase remains relatively constant regardless of the number of nodes, which is expected given the negligible overhead of slicing (which is the only operation that depends on the number of nodes).

Figure 5b depicts the duration of the blocking phase for each of the approaches. For a single node, VELOC−Single and VELOC−Sharded are identical, because no slicing is possible. Interesting to note though is how close these approaches are to Keras−Default, which flushes the checkpoint directly to external storage in HDF5 format. This can be explained by the long preparation phase, which negates the benefits of fast writes to local storage, therefore negating the benefits of multi-level asynchronous flushing. This effect is clearly visible when comparing with VELOC−Opt, which does not have a preparation phase and therefore only needs to block while writing to local storage. As the number of nodes increases, the results begin to show a different trend. Keras−Default is exhibiting an increasingly higher overhead, as the flushing to external storage shares the network bandwidth with other ranks that moved on to the next iteration. This effect is not visible for VELOC−Single, as it uses local storage during the blocking phase. As expected, VELOC−Sharded becomes increasingly faster with increasing number of nodes, because each rank needs to write an increasingly smaller amount of data to local storage. The same trend is visible for VELOC−Opt, but at much faster rate: for 8 nodes, it becomes 3.8x faster than VELOC−Sharded, 10.6x faster than VELOC−Single and 11.1x faster than Keras−Default.

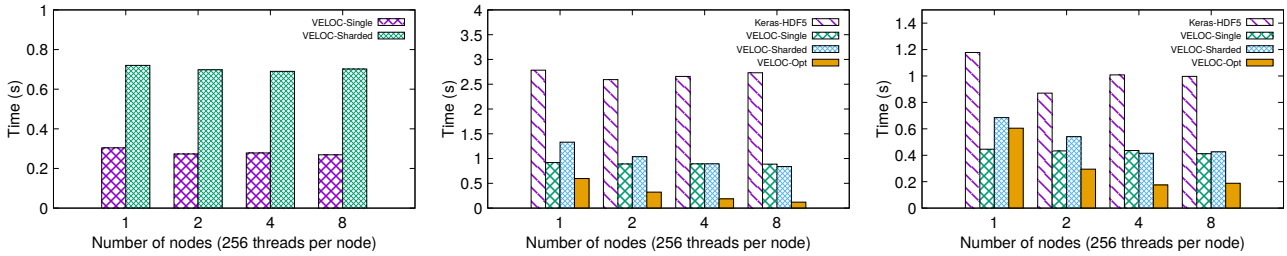
A comparison of the runtime overhead (Figure 5c) reveals similar overall trends for an increasing number of nodes but with notable differences. In the case of Keras−Default, the increasingly higher blocking phase does not cause a higher runtime overhead, which can be explained by the fact that more nodes have higher synchronization overhead, which masks some of the lag of the checkpointing rank. This effect is visible for VELOC−Single as well and even more pronounced due to the fact that its blocking phase is relatively constant. For VELOC−Sharded, the runtime overhead looks very similar to the blocking phase, which is due to the fact that all ranks are checkpointing and therefore there are no laggers. This is true for VELOC−Opt as well, but in this case, the extra operations running in the execution graph during checkpointing to avoid the preparation phase lead to a slightly higher runtime overhead when compared with the corresponding blocking phase. Even with this extra overhead, for 8 nodes, VELOC−Opt is 2.5x faster than VELOC−Sharded, 6.2x faster than VELOC−Single and 6.5x faster than Keras−Default.

The results for Resnet−50 are depicted in Figure 6. Just like in the case of CANDLE−NT3, the preparation phase for VELOC−Single and VELOC−Sharded (Figure 6a) shows significant overhead for both approaches. However, the overhead for VELOC−Sharded is much higher because there are many tensors of small size, which means many slicing operations need to be performed on fewer bytes, therefore introducing an non-negligible overhead on top of the conversion from tensors to numpy arrays. This also has an impact on the scalability of the preparation phase, with slicing becoming slightly cheaper as the number of nodes increases (which is expected given that it involves fewer bytes with increasing number of nodes). On the other hand, the preparation phase



(a) Preparation phase (lower is better). (b) Blocking phase (lower is better). (c) Runtime overhead (lower is better).

Fig. 5: CANDLE-NT3: Checkpointing scalability



(a) Preparation phase (lower is better). (b) Blocking phase (lower is better). (c) Runtime overhead (lower is better).

Fig. 6: ResNet-50: Checkpointing scalability

for VELOC−Single remains relatively constant just like in the case of CANDLE−NT3.

The blocking phase, which is depicted in Figure 6b, exhibits significant differences between Keras−Default and the rest of the approaches, even for a single node. In this case, VELOC−Opt is 2.2x faster than VELOC−Sharded, 1.5x faster than VELOC−Single and 4.7x faster than Keras−Default. This can be explained by the fact that Keras−Default writes to the parallel file system using many small I/O operations, which is sub-optimal. On the other hand, the rest of the approaches write to local storage, which can handle smaller I/O operations better. As the number of nodes increases, the blocking phase for Keras−Default and VELOC−Single remains constant, as expected. In the case of VELOC−Sharded, the blocking phase slowly decreases up to the point where it is close to the preparation phase, which already happens at 4 nodes and shows limited scalability potential. The opposite is true for VELOC−Opt: not only does its blocking phase start lower than the rest of the approaches, but the gap is also increasing with the number of nodes, hinting at much better scalability. For 8 nodes, its blocking phase is 7x faster than VELOC−Sharded, 7.4x faster than VELOC−Single, 22x faster than Keras−Default.

The runtime overhead (depicted in Figure 6c), follows a similar pattern with the blocking phase. Specifically, the high blocking overhead of Keras−Default is reflected in the runtime overhead as well, leading to a situation where the other approaches are two times faster for most configurations. For a single node, just like in the case of CANDLE−NT3, VELOC−Opt has higher overhead than VELOC−Single due to the extra operations running in the execution graph. With increasing number of nodes, Keras−Default experiences slightly lower runtime overhead due to higher all-reduce synchronization overhead, while the overhead of VELOC−Single remains relatively constant. Interesting to note is that VELOC−Single is close to VELOC−Sharded, which emphasizes the poor performance of sharding for many tensors of small sizes. Both VELOC−Sharded and VELOC−Opt experience a visible reduction in runtime overhead. However, this reduction is much sharper for VELOC−Opt, which is 5.15x faster than Keras−Default, 2.2x faster than VELOC−Single and 2.3x faster than VELOC−Sharded for 8 nodes.

Overall, we conclude that the combination of our proposed techniques give VELOC−Opt a large performance and scalability advantage over the other approaches, both for the blocking phase and the runtime overhead.

VI. Conclusions

This paper introduced an approach specifically optimized for frequent checkpointing of deep learning models subject to synchronous data-parallel training and optimized to take advantage of layer-wise parallelism. Despite the fact that frequent checkpointing is an increasingly important building block in a broad range of deep learning scenarios, state-of-art checkpointing approaches are rudimentary and lack high-performance and scalability considerations.

To address this gap, we contributed with several novel ideas, including lightweight serialization, sharding and augmentation of the execution graph to asynchronously mask the overhead of capturing weights from tensors without using a separate execution context. These ideas facilitate efficient serialization into contiguous byte arrays, which be used by

multi-level checkpointing approaches to persist the state of learning models in a resilient fashion. The combination of these techniques has shown major improvements for real-life deep learning applications, both in terms of reducing the blocking overhead (at least 10x) and runtime overhead (at least 5x) when compared with state-of-art. For users, this has an important impact because it carries benefits regardless whether the training needs to continue after taking a checkpoint or not.

Encouraged by these promising results, in future work we plan to explore more trade-offs that emerge in the context of synchronous data parallel training and layer-wise parallelism. One promising direction is gaining direct access to the memory regions used by the tensors, which enables zero-copy on one hand, but introduces the need to maintain consistency of checkpoints by ensuring tensors are not changed while being checkpointed. In this regard our previous work [9] introduces several ideas we can start from.

Acknowledgments

This research was funded by Argonne National Laboratory, under Contract LDRD-1007397. It used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] "CANDLE Benchmarks," https://github.com/ECP-CANDLE/Benchmarks.

[2] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC '10: The 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, USA, 2010, pp. 1:1–1:11.

[3] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *SC '11: The 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, USA, 2011, pp. 32:1–32:32.

[4] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[5] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards high performance adaptive asynchronous checkpointing at large scale," in *IPDPS'19: The 2019 IEEE International Parallel and Distributed Processing Symposium*, Rio de Janeiro, Brazil, 2019, pp. 911–920.

[6] S.-M. Tseng, B. Nicolae, G. Bosilca, E. Jeannot, and F. Cappello, "Towards portable online prediction of network utilization using MPI-level monitoring," in *EuroPar'19 : 25th International European Conference on Parallel and Distributed Systems*, Goettingen, Germany, 2019, pp. 1–14.

[7] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *CLUSTER '12 - Proceedings of the 2012 IEEE International Conference on Cluster Computing*, Beijing, China, 2012, pp. 155–163.

[8] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole, "Optimizing I/O forwarding techniques for extreme-scale event tracing," *Cluster Computing*, vol. 17, no. 1, pp. 1–18, Mar. 2014.

[9] B. Nicolae and F. Cappello, "AI-Ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing," in *HPDC '13: 22th International ACM Symposium on High-Performance Parallel and Distributed Computing*, New York, USA, 2013, pp. 155–166.

[10] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient user-level storage disaggregation for deep learning," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.

[11] S. Pumma, M. Si, W.-c. Feng, and P. Balaji, "Parallel I/O optimizations for scalable deep learning," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 720–729.

[12] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "FanStore: Enabling efficient and scalable I/O for distributed deep learning," *arXiv preprint arXiv:1809.10799*, 2018.

[13] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-aware I/O pipelining for large-scale deep learning on hpc systems," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 145–156.

[14] J. Wozniak, R. Jain, P. Balaprakash *et al.*, "CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, no. 19, 2018.

[15] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan *et al.*, "Population based training of neural networks," *arXiv preprint arXiv:1711.09846*, 2017.

[16] P. Balaprakash, M. Salim, T. Uram, V. Vishwanath, and S. Wild, "DeepHyper: Asynchronous hyperparameter search for deep neural networks," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2018, pp. 42–51.

[17] G. K. P. Kaul and D. Golovin, "Hyperparameter tuning in cloud machine learning engine using bayesian optimization," 2017.

[18] J. M. Wozniak, P. E. Davis, T. Shu, J. Ozik, N. Collier, M. Parashar, I. Foster, T. Brettin, and R. Stevens, "Scaling deep learning for cancer with advanced workflow storage integration," in *Proceedings of Machine Learning in High Performance Computing Environments (MLHPC)*, 2018.

[19] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[20] S. Jin, S. Di, X. Liang, J. Tian, D. Tao, and F. Cappello, "DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression," 2019.

[21] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[22] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[23] B. Nicolae, "Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal," in *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, Boston, USA, 2013, pp. 1–10.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR'16: 2016 IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, USA, 2016, pp. 770–778.

[25] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[26] Y. Jia, E. Shelhamer, J. Donahue *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *ICM'14: The 22Nd ACM International Conference on Multimedia*, Orlando, USA, 2014, pp. 675–678.

[27] "Torch: A scientific computing framework for luajit," http://torch.ch/.

[28] A. Sergeev and M. D. Balso, "Meet Horovod: Uber's open source distributed deep learning framework for tensorflow," https://eng.uber.com/horovod.

[29] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[30] "Horovod repository," https://github.com/horovod.

[31] J. Deng, W. Dong, R. Socher *et al.*, "ImageNet: A large-scale hierarchical image database," in *CVPR'09: Conference on Computer Vision and Pattern Recognition*, Miami, USA, 2009, pp. 248–255.

[32] J. Li, B. Nicolae, J. Wozniak, and G. Bosilca, "Understanding scalability and fine-grain parallelism of synchronous data parallel training," in *5th Workshop on Machine Learning in HPC Environments (in conjunction with SC19)*, 2019.