High Performance Scalable Deep Learning with the Cray Programming Environments Deep Learning Plugin

Peter Mendygral[1], Nick Hill[1], Krishna Kandalla[1], Diana Moise[2], Jacob Balma[1] and Marcel Schöngens[3]

***Abstract*— Deep Learning (DL) with neural networks is emerging as a critical tool for academia and industry with transformative potential for a wide variety of problems. The amount of computational resources needed to train sufficiently complex networks can limit the use of DL in production, however. High Performance Computing (HPC), in particular efficient scaling to large numbers of nodes, is ideal for addressing this problem. This paper describes the Cray Programming Environments Deep Learning Plugin (Cray PE DL Plugin), a DL framework portable solution for high performance scaling. Performance on Cray platforms and a selection of neural network topology implementations using TensorFlow are described.**

I. INTRODUCTION

Artificial Intelligence (AI) technologies are transforming the way academia and industry are tackling a variety of complex problems. In particular, Deep Learning (DL) with neural networks can be a powerful tool for extracting information from massive datasets through classification, prediction, and regression. DL also has the potential to provide reproducible rigor to analysis tasks that lack objectivity. Training deep neural networks is a computationally intensive task which can take weeks if done on just a single CPU or GPU node. This can be a major hurdle to applying DL in production.

Stochastic gradient descent (SGD) is the optimization technique most commonly used to train deep neural networks. The training process requires a large training dataset labeled with information about each sample that the network is to learn. A step of SGD uses a random subset of that data, called the mini-batch, to compute partial derivatives for each tunable parameter in the network. These derivatives, or gradients, measure the difference between the output of the network and the correct result given by the labels. Each sample in the random subset produces its own gradients. The gradients from each sample are averaged together, and the average is used to update the network parameters for the next SGD step. Modifications to SGD typically involve new optimizers, the method used to update the model given a set of gradients.

SGD can be parallelized by subdividing a sufficiently large mini-batch evenly among a set of processes. Each process computes gradients locally and then communicates them to produce globally averaged gradients. The neural network parameters (model) are then updated with these gradients. This technique is called synchronous data parallel SGD (SSGD).

[1]Cray Inc., Bloomington, USA
[2]Cray Inc., Basel, Switzerland
[3]CSCS, Lugano, Switzerland

It is possible to reduce DL training time to accuracy with SSGD by increasing the global mini-batch size (sum across all processes) and increasing the SGD step size, also referred to as the learning rate. The errors on globally averaged gradients decrease as more samples are used to compute them. Lower errors allow for larger updates to the model at each step potentially leading to faster convergence. There are limitations to how far the global mini-batch size can be increased before slow convergence or lack of convergence is observed [1], [2], [3], [4]. More sophisticated optimizers must then be used to overcome these training difficulties. Adaptive optimizers such as Adam [5] and LARS [6] are proven examples for certain classes of networks.

While many DL frameworks provide functionality for data parallel training, their performance at the needed scales of hundreds of nodes or more is poor. This is mainly attributed to the communication costs associated with computing the globally averaged gradients across a large number of nodes. The Cray PE DL Plugin solves this problem through a combination of unique algorithmic improvements and highly optimized communication based on Message Passing Interface (MPI). While the Cray PE DL Plugin relies on MPI, the communication operations within the Plugin significantly outperform a state-of-the-art distributed DL framework that simply relies on an `MPI_Allreduce()` collective operation to compute the globally averaged gradients. This paper describes the strategies used in the Cray PE DL Plugin and how they result in optimal performance on Cray platforms. We discuss the application of the Cray PE DL Plugin to TensorFlow [7], a popular DL framework, and observed performance improvements on Cray XC platforms with Intel KNL processors and NVIDIA P100 GPUs. We show scaling performance efficiencies up to 91% on 1,024 KNL and GPU nodes running ResNet50, for example. Finally, we describe use cases with Convolution Neural Network (CNN) and Generative Adversarial Network (GAN) examples.

II. PREVIOUS WORK

This section describes some of the relevant state-of-the-art technologies related to distributed Deep Learning frameworks.

A. TensorFlow

TensorFlow [7] is an open source software stack that offers a suite of high performance numerical computation routines. TensorFlow can be easily deployed across many hardware platforms ranging from general purpose CPUs and Graphics Processing Units (GPUs) to highly custom and specialized

Tensor Processing Units (TPUs). Additionally, TensorFlow can also be deployed on clusters comprised of high performance computer servers and interconnects. However, the underlying hardware and communication infrastructure plays a key role in scaling TensorFlow to large number of compute nodes.

B. gRPC

gRPC [8] is an open-source Remote Procedure Call layer that was initially developed by Google. gRPC offers an abstraction layer to develop distributed applications and services and is layered over the HTTP/2 protocol. gRPC also provides a range of features to enable communication, synchronization, and flow-control between clients and servers in a distributed application. gRPC is one of the communication protocols used in Google's TensorFlow framework.

C. Horovod

Horovod is an open source, distributed deep learning framework for TensorFlow from Uber [9]. Horovod uses MPI to implement a distributed infrastructure for TensorFlow. Internally, the global reduction operation is implemented by using a "ring"-based Allreduce implementation to utilize the communication bandwidth offered by a typical high performance cluster interconnect. Recent implementations of Horovod use NVIDIA's NCCL and NCCL2 [10] communication layers to optimize communication performance on modern systems with multiple GPUs per node.

D. ChainerMN

Chainer is a Python-based deep learning framework developed by Preferred Networks [11]. Chainer was open-sourced in 2015 and continues to be a popular framework for deep learning in the academic and industrial communities. The Chainer framework includes additional packages such as ChainerMN (distributed learning), ChainerRL (reinforcement learning), and ChainerCV (computer vision).

ChainerMN (Chainer Multi-Node) implements distributed training functionality in Chainer. By default, the ChainerMN framework maps one process to a hardware GPU device and relies on MPI to implement inter-process communication and synchronization operations. Additionally, ChainerMN relies on NVIDIA's NCCL [10] to implement intra-node communication between GPU devices.

E. MXNet

Apache MXNet [12] is a deep learning framework designed for efficiency and flexibility. Most common deep learning frameworks either rely on an imperative programming style or a symbolic approach. MXNet allows a practitioner to mix symbolic and imperative programming styles to maximize efficiency and productivity. MXNet is based on a dynamic dependency scheduler that automatically parallelizes both symbolic and imperative operations on the fly. In addition, a graph optimization layer improves the speed and memory efficiency of symbolic execution phases of an application. MXNet is designed to be portable and lightweight. It is also built with a goal of scaling effectively to multiple GPUs and multiple machines.

MXNet implements distributed training features to accelerate training by utilizing multiple compute servers in a system. MXNet fundamentally relies on a Key-Value Store infrastructure to implement multi-device training. Gradients and weights are exchanged between MXNet "worker" and "server" processes. Typically, the KVStore implementation relies on TCP socket for communication operations. [13] describes an implementation of MXNet that uses blocking `MPI_Allreduce()` operations. Additionally, the author also proposes the idea of using multiple helper threads to handle concurrent `MPI_Allreduce()` operations.

In summary, state-of-the-art distributed DL frameworks rely on simple MPI collective operations to compute the average gradients across all compute nodes. The cost of these operations grows with scale and this limits the overall efficiency of a distributed DL framework. In contrast, our proposed implementation relies on the Cray PE DL Plugin to offer highly optimized and scalable communication performance at large scales. Sections IV-A and IV-B describe how users can effectively utilize the Cray PE DL Plugin effectively on Cray systems. Section V describes the performance and scaling characteristics of our proposed solution.

III. Solution Design & Strategy

A. General Design Principles

Many parallelization frameworks for DL, such as gRPC in TensorFlow, consist of two classes of processes. Parameter server (PS) processes gather gradients from worker processes, compute the globally averaged gradients, update the network parameters, and send the new parameter values to workers. Typically the user can select the number of PS processes. A single or limited number of PS processes with a large number of workers will encounter performance issues and limited scaling. This configuration establishes a many-to-few communication pattern, which causes congestion on most networks. It is also difficult for a limited number of PS processes to send out updated parameter values fast enough to keep up with worker demand. Increasing the number of PS processes can reduce the communication and parameter update bottlenecks. Using too many PS processes, however, results in many-to-many communication patterns, which will not scale to large numbers of nodes. Determining the optimal number of PS processes is very cumbersome for users. For gRPC in TensorFlow, users must also provide node names and port numbers, adding to the usability issues.

The Cray PE DL Plugin addresses these scaling performance and usability issues in TensorFlow and other similar DL frameworks. There are no PS processes when using the Cray PE DL Plugin. Every process is a worker, and a custom global reduction operation replaces the gradient aggregation work of PS processes. Every worker then redundantly computes network parameter updates, which is typically a small fraction of the execution time. Algorithm 1 outlines data parallel training with the Cray PE DL

Plugin. The custom reduction is specifically optimized for DL workloads and is ≥35% higher performing than the default `MPI_Iallreduce()` available in Cray MPICH for relevant message sizes and process placements. In addition to improved communication performance at scale, the custom reduction implementation also offers excellent computation/communication overlap. The ability to hide the communication costs of the average gradient computation phase plays a key role in improving the time to accuracy of distributed training. Section V demonstrates the full benefits offered by the Cray PE DL Plugin on Cray systems.

Algorithm 1 Pseudocode for data-parallel training algorithm. The Cray PE DL Plugin is represented by mc, and the gradient aggregation function is $mc.gradients()$.

Require: N = total number of epochs
Require: n = total number of training samples
Require: k = number of MPI ranks
Require: b = number of training samples in a batch per process
1: **for** epoch $= 1 \cdots N$ **do**
2: **for** step $= 1 \cdots n/(bk)$ **do**
3: $g_{step} \leftarrow$compute_gradients(local_batch$_{step}$)
4: $G_{step} \leftarrow$mc.gradients(g_{step})
5: $loss_{step} \leftarrow$apply_gradients(G_{step})

No modifications to TensorFlow are required to use the Cray PE DL Plugin for parallelization. The TensorFlow Op feature[1] is used to add the necessary communication steps to the execution graph in an optimal way. Users can start with a serial TensorFlow or other framework client script and add the required calls for initialization, communication, and finalization. For situations where multiple gradient reductions are needed, such as GANs, teams of reduction threads can be used to accelerate each independently with a few simple calls. C/C++ and Python 2/3 interfaces are available with the Cray PE DL Plugin.

B. Supported Platforms

The Cray PE DL Plugin is included with the Urika-XC analytics package available for Cray XC systems. The current Urika-XC 1.1 release supports TensorFlow 1.3 and Intel® CPU and NVIDIA GPU based architectures. The current Urika-XC documentation refers to the Cray PE ML Plugin. This will be updated to Cray PE DL Plugin in future releases.

IV. USAGE

A. Script Porting

The ideal porting process to utilize the Cray PE DL Plugin starts from a serial training script. The steps outlined in this section cover the requirements for the Cray PE DL Plugin and modifications for parallelization in general. The required modifications are:

- Initialize the Cray PE DL Plugin
  - Specifying the number of teams, threads, model size
- Broadcast initial model parameter values
- Use the Cray PE DL Plugin to communicate gradients after gradient computation and before the model update
- Finalize the Cray PE DL Plugin

Modifications that are not required by the Cray PE DL Plugin, yet common for parallelization, are:

- Correct the definition of an epoch for the global mini-batch size (all processes)
- Correct the learning rate
  - Linear or square root scaling rule
  - Add a learning rate decay schedule
- Average performance metrics using Cray PE DL Plugin helper functions
- Only a single rank does any desired prints
- Either a single rank writes checkpoints or each rank writes to a unique location

In the following sections we review the required steps applied to a serial Python training script. This example leaves the original serial code in place and adds in conditions for parallel execution. This is done to allow comparisons to the original code.

1) Initialization: The first step is to initialize the Cray PE DL Plugin. This is done by first importing the module and calling the `init()` routine as follows:

```
# import the Cray PE DL Plugin module
import ml_comm as mc

def main():

  # Build the model
  model=build_model(n_in, n_layer, n_hid, n_out)

  # if the Plugin is enabled initialize it
  if (flags.enable_ml_comm == 1):

      # determine the model size
      totsize = sum([v for v in
              tf.trainable_variables()])

      # initialize the Cray PE DL Plugin
      mc.init(nthread_per_team=2,
      nteams=1,
      msglen=totsize,
      "tensorflow")
      ...
```

Note the call to `init()` includes the additional argument of "*tensorflow*", which indicates that TensorFlow is the training framework. There are two effects from specifying "*tensorflow*". First, calls to the Cray PE DL Plugin operations for broadcasting initial model parameters (`mc.broadcast()`) and gradient aggregation (`mc.gradients()`) switch from expecting NumPy array data buffers to native TensorFlow Tensor buffers. Second, TensorFlow Ops for both broadcast and gradient aggregation get loaded, which allows those calls to be added to the execution graph.

The next initialization step is to configure the single thread team to be aware of the total number of training steps. Users first decide on the number of training epochs and number of samples in a mini-batch per process. The total number of training steps is then given by

[1]Documentation on custom TensorFlow operations is found at https://www.tensorflow.org/extend/adding_an_op.

$$M_{steps} = N_{epochs} \times \frac{n_{train}}{(k_{ranks} \times b_{local})}, \quad (1)$$

where N_{epochs} is the number of training epochs, n_{train} is the number of samples in training dataset, k_{ranks} is the number of workers (MPI ranks or processes), and b_{local} is the local batch size.

The configuration step is shown below, continuing from the previous code block.

```
...
# use the CPE DL Plugin to get our rank and
# the number of processes
myrank     = mc.get_rank()
numworkers = mc.get_nranks()

# num_train_samps is the number of samples in
# our training data set
max_steps = int(math.ceil(flags.train_epochs *
          (num_train_samps) /
          (numworkers * batch_size)))

# configure the single thread team and have rank 0
# print out communication performance metrics
# every 100 steps
mc.config_team(0, 0, max_steps, max_steps, 1, 100)
...
```

2) Broadcast initial model parameters: With Tensorflow, it is ideal to use a SessionRunHook to manage the broadcast and any desired averaging of metrics. Session hooks can be passed to convenience classes, such as Estimator, to be run at specific points in session management. A typical SessionRunHook for broadcasting initial model parameters is defined as follows:

```
class BcastTensors(tf.train.SessionRunHook):

  def __init__(self):
    self.bcast = None

  def begin(self):
    if not self.bcast:
      new_vars   =
          mc.broadcast(tf.trainable_variables(),0)
      self.bcast =
          tf.group(*[tf.assign(v,new_vars[k])
            for k,v in
            enumerate(tf.trainable_variables())])

  def after_create_session(self, session, coord):
     session.run(self.bcast)
```

This SessionRunHook broadcasts the model parameters from rank 0 to all other MPI ranks and then assigns the new values. The actual operation is performed after the session is created. This SessionRunHook must be provided to the Estimator instance as follows:

```
...
train_hooks = None

# if the Cray PE DL Plugin is enabled add the hook
if (mlcomm == 1):
    train_hooks = [BcastTensors(),
        AverageTrainMetrics(loss,metrics,
            log_freq,batch_size)]

cnf = tf.estimator.EstimatorSpec(
    mode=mode,
    predictions=predictions,
    loss=loss,
    train_op=train_op,
    training_hooks=train_hooks,
    eval_metric_ops=metrics)

classifier = tf.estimator.Estimator(model_fn=cnf)
classifier.train(input_fn=input_fn_train,
    steps=tsteps, max_steps=flags.max_train_steps)
...
```

Another SessionRunHook can be defined for averaging performance, training loss and accuracy if the user desires. In this SessioRunHook, `mc.average()` is used to average a NumPy array of values across all processes. A hook called `AverageTrainMetrics()` that performs these tasks is used in the code block above:

```
class AverageTrainMetrics(tf.train.SessionRunHook):

  def __init__(self,loss,accuracy,log_freq,batch_size):
    self.log_freq = log_freq
    self.batch_size = batch_size * mc.get_nranks()
    self.both = [loss,accuracy[0]]
    self.samples_per_sec = 0.
    self.step = 0
    self.start_time = None

  def begin(self):
    self.step = 0
    self.start_time = time.time()

  def before_run(self, run_context):
    self.step += 1
    return tf.train.SessionRunArgs(self.both)

  def after_run(self, run_context, run_values):
    if self.step % self.log_freq == 0:

      current_time = time.time()
      duration = current_time - self.start_time
      self.start_time = current_time

      loss_value =
          np.asarray([run_values.results[0]],
              dtype='float32')
      acc_value =
          np.asarray([run_values.results[1]],
              dtype='float32')
      samples_per_sec =
          self.log_freq * self.batch_size / duration
      mc.average(loss_value)
      mc.average(acc_value)

      format_str = ('%s: step %d, loss = %.3f,
          acc = %.3f, (%.1f examples/sec; %.3f '
          'sec/batch)')
      if (mc.get_rank() == 0):
          print (format_str %
              (datetime.now(), self.step, loss_value,
              acc_value, examples_per_sec,
              sec_per_batch))
```

3) Gradient aggregation: The communication and performance intensive operation that is highly optimized in the Cray PE DL Plugin is gradient aggregation. This is placed between gradient computation and model update as follows:

```
def train(model_function, train_samp, eval_samp,
        batch_size)

    ....
    # often a serial code will use the
    # optimizer minimize() method
    if (mlcomm != 1):

      minimize_op =
          optimizer.minimize(loss, global_step)

    else:
      # for the Cray PE DL Plugin
      # we need to split out the minimize call below
      # so we can communicate/average gradients
      grads_and_vars = optimizer.compute_gradients(loss)

      grads     =
          mc.gradients(
```

```
            [gv[0] for gv in grads_and_vars], 0)
    gs_and_vs =
        [(g,v) for (_,v),
            g in zip(grads_and_vars, grads)]

    minimize_op = optimizer.apply_gradients(gs_and_vs,
      global_step)
...
```

It is common for a serial training script to use the `minimize()` method of an optimizer. This method computes gradients and updates the model with those gradients. The global reduction of local gradients must be done between those steps for data parallel training, however. In the code block above, `minimize()` is split into `compute_gradients()` and `apply_gradients()` so that the `mc.average()` call can be added. This operation is added to the execution graph and will have direct access to gradient Tensors, located in CPU or GPU memory, without additional buffering.

4) Finalization: The final required step for porting a serial training script is to finalize the Cray PE DL Plugin, similar to finalizing MPI. This call should be added after training is complete and the session is closed. Often this is at the end of the `main()` function.

```
def main():

    ...
    # training is complete and we're ready to exit
    mc.finalize()
```

Cray provides several examples with the Cray PE DL Plugin package for users to reference. MNIST and tf_cnn_benchmarks examples are provided with the latest 1.0.1 release available with Urika-XC 1.1. The tf_cnn_benchmarks example is commonly used to benchmark the performance across a set of standard CNNs. Performance results with the version provided with the Cray PE DL Plugin are given in §V.

B. Execution and Tuning

Below, we outline a few guiding principles for achieving the best performance on Cray XC systems with Intel® CPU or NVIDIA GPU architectures and provide example commands for execution.

In general, the best performance is achieved with one MPI rank per node. The Cray PE DL Plugin should be configured to use 2-4 communication threads. In some cases with GPU nodes, performance can be improved using up to 8 threads. For training with MKL and MKL-DNN, it is important to not set `OMP_NUM_THREADS` too high, lest cores become oversubscribed. For example, if there are 36 physical cores on a node, optimal performance is achieved with `OMP_NUM_THREADS=34` while leaving two cores/threads for communication with the Cray PE DL Plugin. Additionally, with TensorFlow and the tf_cnn_benchmarks example, `num_intra_threads` should be set to match the value of `OMP_NUM_THREADS`, and `num_inter_threads` can typically be set to 1-3 threads depending on the number of HyperThreads available per core. For KNL CPUs, it is typically best to leave one HyperThread free on each core. The `KMP_BLOCKTIME` environment variable may yield slightly improved performance if set to 0 or 30.

For GPU nodes, the number of CUDA streams used to buffer data to the host can be modified via the `ML_COMM_NUM_CUDA_STREAMS` environment variable, and the number of copies each of those streams performs is changed with the `ML_COMM_CPY_PER_CUDA_STREAM` environment variable. The default settings, 2 and 8, respectively, have empirically been found to be best for nearly all situations.

Using these rules and assuming a Python 2 environment with TensorFlow 1.3, the 1.0.1 release of the Cray PE DL Plugin and its included tf_cnn_benchmarks example, a user can execute the example in one of the following manners:

```
# Load the Cray PE DL Plugin for Python 2
module load craype-ml-plugin-py2

# Execute tf_cnn_benchmarks for GPU architecture
# with Slurm
srun --ntasks=4 --ntasks-per-node=1 --cpu_bind=none   \
      python tf_cnn_benchmarks.py                     \
      --variable_update=ps_ml_comm                    \
      --number_ml_comm_threads=4 --model=inception3   \
      --train_dir=/loc/to/store/chkpt

# The same example as above targeting a PBS/Moab/Torque
# environment
aprun -n4 -N1 -cc none -b python tf_cnn_benchmarks.py \
      --variable_update=ps_ml_comm                    \
      --number_ml_comm_threads=4 --model=inception3   \
          --train_dir=/loc/to/store/chkpt

# We modify the first example while targeting a dual
# socket 36 core Broadwell node
srun --ntasks=4 --ntasks-per-node=1 --cpu_bind=none   \
     python tf_cnn_benchmarks.py                      \
     --variable_update=ps_ml_comm                     \
     --number_ml_comm_threads=2 --model=inception3    \
     --num_intra_threads=34 --num_inter_threads=1     \
     --train_dir=/loc/to/store/chkpt                  \
     --mkl=True --local_parameter_device=cpu          \
     --device=cpu --data_format=NHWC
```

On the other hand, if the user desires to use the TensorFlow 1.3 version included in Urika-XC 1.1, they need to call the `run_training` program packaged with Urika-XC to execute the tf_cnn_benchmarks example. In this case, the above examples targeting a GPU architecture become the following:

```
# Load Urika-XC 1.1 module
module load analytics

# Execute the run_training program targeting 2 nodes
# and 1 process per node
run_training -n 2 --ppn 1                             \
    --cudnn-libs /path/to/cudnn-8.0-v51/cuda/lib64    \
    --no-node-list "python tf_cnn_benchmarks.py
    --variable_update=ps_ml_comm
    --number_ml_comm_threads=4 --model=inception3
    --train_dir=/loc/to/store/chkpt"
```

V. PERFORMANCE

The number of model parameters in a neural network defines the volume of data that must be communicated, as the number of gradients to reduce equals the number of parameters. Consequently, with data parallel SGD, the message size does not vary with the number of processes, and communication cost will increase with job size according to the nature of the allreduce algorithm and the algorithm's

ability to efficiently communicate. The effect of this communication cost on overall execution time is coupled with the computational cost of the neural network.

That computational cost is determined by the design of the neural network and the number of samples in a mini-batch, ie: mini-batch size (MBS). For a smaller MBS, the computational cost will be less than for a larger MBS. As more processes are provided for training, the extra processes allow more training data to be analyzed, but analyzing that larger amount of data does not increase the computational cost, presenting a weak scaling scenario. Therefore, as training scales, the cost of communication increases and that of computation remains static. If that cost is not minimized, execution time naturally increases and throughput suffers.

To measure the ability of the Cray PE DL Plugin to manage that increasing cost, we use TensorFlow and the [14] script included with the Plugin to measure throughput and efficiency at scale. Figure 1 shows the performance of two popular CNNs, Inception v3 [15] and ResNet50 [16], on Piz Daint (NVIDIA P100 nodes) at CSCS and Cori (Intel KNL) at NERSC. Both networks have a comparable number of parameters (∼25 million) and therefore allreduce message sizes. The aggregate throughput performance, measured by samples per second, increases almost ideally with the number of nodes for both networks. Relative to single node execution without any communication present, the Cray PE DL Plugin measures 91% efficient at 512 nodes with Inception v3 and 89% efficient at 1,024 nodes with ResNet50. The performance of gRPC is also included for Inception v3 in Figure 1. At 128 nodes the Cray PE DL Plugin is nearly 1.8X higher performance than gRPC.

While these throughput numbers illustrate the ability of the Cray PE DL Plugin to efficiently scale to large node counts, they do not address how to get a model to learn at scale. A model's ability to learn and ultimately converge to a reported evaluation accuracy depends on the model, the input data, and the optimizer in use. In §VI-A, several of these dependencies are discussed in context of training the ResNet50 model with the ImageNet dataset [17].

VI. SAMPLE USE CASES

A. ResNet50 Convolutional Neural Network (CNN)

The deep residual network framework ResNet is able to train to very low error rates on the ILSVRC 2015 challenge [17]. For example, the ResNet50 model achieves a top-1 error rate of 22.85% [16]. This model is 50 layers deep and has a total of 25,559,081 tunable parameters (97.5 MB single precision). Recent work with this model shows that it is possible to scale the global mini-batch size to very large values while maintaining high accuracy [18], [19], [6]. The ability to train with very large global mini-batch sizes enables efficient scaling on supercomputers because each process can compute gradients with enough local samples to keep high computational efficiency and communication frequency low. For example, [19] trained ResNet50 to 74.0% accuracy in 15 minutes on 1,024 NVIDIA P100 GPUs each processing a local mini-batch size of 32. This run had 80% scaling efficiency relative to a single node with 8 GPUs using the ChainerMN framework. This task requires over 25 hours to complete on a single node with 8 NVIDIA P100 GPUs.

We adapt the official TensorFlow version of the ResNet50 model[2] for use with the Cray PE DL Plugin to allow users to study the implementation and techniques for convergence at scale. The code modifications are similar to those described in §IV-A. This script uses momentum SGD as the base optimizer, which is not able to overcome training difficulties with large global mini-batches. A more sophisticated optimizer is needed. A future release of the Cray PE DL Plugin will include the modified training script, sample run scripts, and notes on performance tuning.

1) Optimizer: To overcome large global mini-batch training difficulties we implemented a custom optimizer based on LARS [6]. This optimizer uses momentum SGD as its base but includes an adaptive step that scales the local layer learning rate based on the ratios of the L^2 norm of local layer weights and gradients. This prevents weight updates from being too large and potentially unstable if the global learning rate is high. At the same time, layers that can benefit from a large global learning rate for faster convergence are allowed to. A learning rate warm-up is still needed, however, to improve final accuracy [6]. We use a 12 epoch warm-up where the global learning rate is linearly ramped from 0.1 to 40. For the remainder of training a `poly2` learning rate decay is used as in [6].

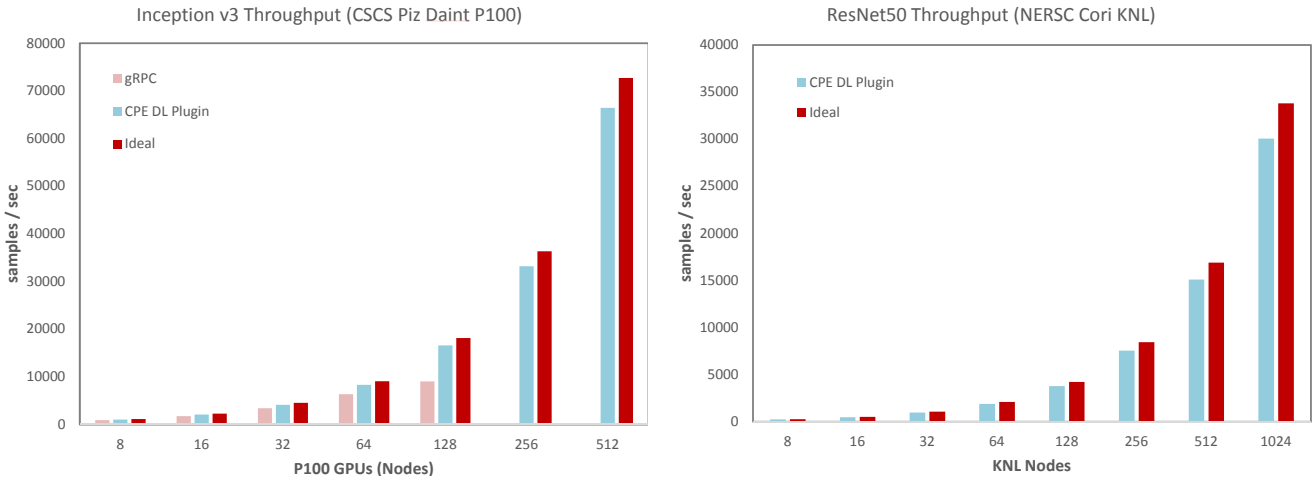
2) Preliminary Results and Tuning: The training techniques used by [6] resulted in 72.5% accuracy after 100 epochs of training. Our preliminary results with a similar setup is 70% accuracy, however. We believe this is due to differing aspects of the `poly2` learning rate decay schedule in our setup relative to [6]. Notably the authors do not give a minimum learning rate. If the learning rate is allowed to drop to zero, as implied by their description, we observe no improvement in accuracy during the last few epochs. The version of this training script that will be provided in future releases of the Cray PE DL Plugin will have this issue resolved.

Our implementation sustains 148 samples/sec on a single XC50 P100 node on Piz Daint at CSCS. Initial results scaling to 1,024 nodes show poor performance at 35 samples/sec/node with training data striped across 16 OSTs on the Sonexion 3000 lustre filesystem. Staging the training dataset on 2 DataWarp nodes instead, performance increases to 111 samples/sec/node sustained (115 samples/sec/node peak), which is 78% scaling efficiency. Average read bandwidth is therefore a performance bottleneck for these runs.

Our ResNet50 implementation uses the Dataset API in Tensorflow, which has the ability to buffer training data asynchronously from computation. Assuming perfect overlap of reads behind computation, the minimum global read bandwidth requirement of a DL training process can be estimated from

[2]Unmodified code available at https://github.com/tensorflow/models/tree/master/official

Fig. 1. Throughput performance scaling is shown for the Inception v3 (left) and ResNet50 (right) models on Cray XC. For ResNet50, data is from NERSC Cori KNL nodes with a mini-batch size of 32. For Inception v3, data is from CSCS Piz Daint P100 nodes with a mini-batch size of 64.



$$BW_{min}(MB/s) \approx \frac{k \times b \times S(MB)}{t(s)}, \tag{2}$$

where k is the number of processes, b is the local mini-batch size, S is the size of a training sample, and t is the step time measured from one node. For these runs, $k = 1,024$, $b = 32$, $S = 0.14$ MB, and $t = 0.3$ s, and $BW_{min} = 14.9$ GB/s. Each OST is capable of delivering ≈ 2.5 GB/s. 16 OSTs should provide enough read bandwidth for these runs. There are several potential explanations for the poor performance from lustre, however. Most importantly it is a heavily used, shared resource, and these runs cannot be guarantee dedicated sustained bandwidth. Each DataWarp node is capable of a 6.5 GB/s read bandwidth, and DataWarp has much different usage model than lustre. These runs had approximately dedicated access to each of the DataWarp nodes. We expect even better performance will be achieved using an additional DataWarp node to satisfy the global read bandwidth requirement and from further tuning of the training script.

B. Generative Adversarial Networks (GANs) for HEP simulations

The recent and remarkable success of Deep Learning in commercial applications has stimulated various attempts of applying DL in scientific disciplines. High Energy Physics (HEP) is one of the sciences that attempt to evaluate the potential of DL methods for enabling scientific discovery. Generative Adversarial Networks (GANs) are among the most promising and interesting neural network architectures developed over the past years. Introduced by Goodfellow in [20], GANs have rapidly become of great interest to scientific communities due to their capability of generating new, plausible images, based on an unsupervised training set. In HEP, researchers have been experimenting with GANs for generating images belonging to particularly complicated distributions [21], [22].

The work in [23] aims at utilizing GANs as a valid, fast alternative to standard Monte Carlo approaches, as part of the GEANTV project [24]. In this context, a 3D convolutional GAN (3DGAN) is implemented and trained with the goal of developing a tool for calorimeter simulation, that can generate particle energy showers in 25× less time than traditional simulators. The GAN is trained on simulation data consisting of $200,000$ particles (electrons and photons), represented as energy showers captured by the cells of a calorimeter. The energy showers can be viewed as 3D images, where each cell translates into a pixel in an image and the amount of deposited energy in that cell, into the pixel grey-scale intensity.

1) Distributed training with the Cray PE DL Plugin: The 3DGAN is defined and trained using Keras [25] and its functional API. To scale the training via the Cray PE DL Plugin, we integrate the modifications described in §IV-A into the TensorFlow backend in the Keras source code. By doing so, we enable users to train their models across multiple machines, while preserving the ease-of-use of the Keras API. The only changes to the user script are the calls to the Keras backend to *initialize* the Cray PE DL Plugin before starting the training, and to *finalize* at the end. All the other calls to the Cray PE DL Plugin API (gradient aggregation at each SGD step) are transparent to the user.

An interesting feature of Keras (with a direct application to GANs) is that the API allows users to define multiple models and to train them using the same TensorFlow graph and session. The support for multiple communication teams in the Cray PE DL Plugin maps smoothly to this multiple models feature: we enhance the backend API to allow users to configure each model to train using a dedicated team; the additional call only requires the users to set the team number to use for training a model, right after the model has been compiled. This also allows users to mix synchronous and asynchronous training for the models, without any additional effort on the user side.

2) Scaling and Physics validation: Scaling the training of the GAN via data parallelism allows to reduce the training time, by reducing the number of training epochs. On a Cray-XC50 (1 P100 GPU per node), the 3DGAN trains for 50 epochs, taking 24h on a single GPU. With the Cray PE DL Plugin and the integration with Keras, we are able to scale the training process, on an internal system, consisting of 16 XC50 nodes (16 GPUs). On 16 nodes, the GAN trains for 3 epochs without any loss in model quality: the generated samples comply with the physics validation described in the next paragraph.

Validating GANs is typically achieved by applying the trained generator (producing images) and comparing the generated samples to real ones. For HEP simulations, validation looks at different metrics that are important features of a particle detector (such as a calorimeter). One of these metrics, the energy shower shape, looks at energy profiles along the three calorimeter axes; these profiles are relevant as they allow to determine the type of the particle that generated them. Thus, we validate the models trained on 2 to 16 GPUs according to this metric, showing that the generated samples are comparable to state-of-the-art Monte Carlo simulation software.

VII. CONCLUSION

The Cray PE DL Plugin is a powerful new tool for Cray users, allowing them to scale DL training to large numbers of nodes with exceptional efficiency. With its global reduction that all workers participate in, it has no need for parameter servers and consequently solves the problems of network congestion that can result from the use of parameter servers via their many-to-few reduction, as well as the tuning problem of finding the best performing ratio of parameter servers to workers.

It simplifies the parallelization steps for DL frameworks, like TensorFlow, and automatically optimizes communication via the global reduction specifically written for the goal of scaling DL workloads. Users only need to augment their existing serial training scripts with steps to initialize the Cray PE DL Plugin, broadcast initial parameters to all workers, reduce the training gradients, and finalize the Plugin. As scaling up DL training typically requires modification to learning rates and other components of serial training models, we have shown how to implement the Cray PE DL Plugin and parallelize DL frameworks by training the ResNet50 model at 1,024 nodes and using GANs as an accurate, fast alternative to HEP Monte Carlo simulations. In the process of scaling the ResNet50 and Inception v3 models, we also find efficient scaling on the order of 90% up to 1,024 nodes.

With its ease of use and scaling efficiency, the Cray PE DL Plugin provides users the means to greatly reduce training time for neural networks and the ability to more rapidly explore new ideas with deep learning.

ACKNOWLEDGMENT

We would like to thank the Swiss National Supercomputing Centre (CSCS) for access to their computing resources.

REFERENCES

[1] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer, "FireCaffe: near-linear acceleration of deep neural network training on compute clusters," *ArXiv e-prints*, Oct. 2015.
[2] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed Deep Learning Using Synchronous Stochastic Gradient Descent," *ArXiv e-prints*, Feb. 2016.
[3] X. Pan, J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting Distributed Synchronous SGD," *ArXiv e-prints*, Feb. 2017.
[4] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu, "Asynchronous Stochastic Gradient Descent with Delay Compensation," *ArXiv e-prints*, Sep. 2016.
[5] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *ArXiv e-prints*, Dec. 2014.
[6] Y. You, I. Gitman, and B. Ginsburg, "Scaling SGD batch size to 32k for imagenet training," *CoRR*, vol. abs/1708.03888, 2017. [Online]. Available: http://arxiv.org/abs/1708.03888
[7] https://www.tensorflow.org, TensorFlow.
[8] https://grpc.io/, gRPC: A high performance, open-source universal RPC framework.
[9] https://eng.uber.com/horovod/, Horovod.
[10] https://developer.nvidia.com/nccl, NVIDIA Collective Communication Library (NCCL).
[11] https://www.preferred-networks.jp/en, Preferred Networks.
[12] https://mxnet.incubator.apache.org/, MXNet: A Scalable Deep Learning Framework.
[13] A. Mamidala, "Efficient Embedding of MPI Collectives in MXNET DAGs for scaling Deep Learning," *arXiv preprint arXiv:1802.06949*, 2017.
[14] https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks, tf_cnn_benchmarks: High performance benchmarks.
[15] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. [Online]. Available: http://arxiv.org/abs/1512.00567
[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385
[17] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
[18] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017. [Online]. Available: http://arxiv.org/abs/1706.02677
[19] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes," *CoRR*, vol. abs/1711.04325, 2017. [Online]. Available: http://arxiv.org/abs/1711.04325
[20] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *ArXiv e-prints*, Jun. 2014.
[21] L. de Oliveira, M. Paganini, and B. Nachman, "Learning particle physics by example: Location-aware generative adversarial networks for physics synthesis," *arXiv preprint arXiv:1701.05927*, 2017.
[22] M. Paganini, L. de Oliveira, and B. Nachman, "Calogan: Simulating 3d high energy particle showers in multi-layer electromagnetic calorimeters with generative adversarial networks," *arXiv preprint arXiv:1705.02355*, 2017.
[23] G. Khattak, P. M. Lorenzo, S. Vallecorsa, and S. Sharan, "Volumetric Generative Adversarial Networks," in *Poster presented at: The International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado, USA, 2017, pp. 15–17. [Online]. Available: http://sc17.supercomputing.org/SC17Archive/tech{_}poster/tech{_}poster{_}pages/post159.html
[24] G. A. et al, "Geantv: from cpu to accelerators," *J. Phys.: Conf. Ser. 762 012019*.
[25] F. Chollet *et al.*, "Keras," https://github.com/keras-team/keras, 2015.