Scaling Deep Learning Applications on Summit

Junqi Yin
Advanced Data and Workflow
Oak Ridge Leadership Computing Facility

ORNL is managed by UT-Battelle LLC for the US Department of Energy
Outline

• DL on Summit overview

• Deployment and distributed DL
  – PyTorch: torch.distributed, Horovod, DDL

• Performance tuning
  – Compute
  – I/O
  – Communication

• Hyperparameter search

• Model inferencing
Summit overview

- 27,648 V100 GPUs
- 3 ExaFlops in FP16 -> Compute
- 7 PB node local NVMe -> I/O
- NVLink2, EDR IB -> Comm.
DL applications on Summit overview

• Full Summit DL
  - (1) “Exascale Deep Learning for Climate Analytics” (arXiv:1810.01993)

<table>
<thead>
<tr>
<th>Application</th>
<th>Network</th>
<th>Sustained Performance (ExaFlops)</th>
<th>Peak Performance (ExaFlops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Climate</td>
<td>DeepLabV3+</td>
<td>0.999</td>
<td>1.13</td>
</tr>
<tr>
<td>(2) Medical</td>
<td>MENNDL</td>
<td>1.3</td>
<td>n/a</td>
</tr>
<tr>
<td>(3) Materials</td>
<td>Tiramisu variant</td>
<td>1.5</td>
<td>2.1</td>
</tr>
</tbody>
</table>
## DL applications on Summit overview

- DL by domains and methods (growing list)

<table>
<thead>
<tr>
<th>Domain\Method</th>
<th>Supervised Learning</th>
<th>Unsupervised Learning</th>
<th>Reinforcement Learning</th>
<th>Hyperparameter Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Random Forest</td>
<td>MLP</td>
<td>CNN</td>
<td>RNN (LSTM)</td>
</tr>
<tr>
<td>Biophysics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chemistry</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Climate</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Computer Science</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fusion</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Life Sciences</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Material Sciences</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Medical Sciences</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Nuclear Physics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Turbulence</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Particle Physics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
DL baselines on Summit: CORAL2 DL Benchmarks

Speedup Over Titan Baseline for CORAL-2 Deep Learning Benchmarks

Strong Scaling of ResNet-50 on Summit

"The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems", SC '18

https://asc.llnl.gov/coral-2-benchmarks/
Outline

- Deployment
- Parallelization
- Hyper-parameter Search
- Performance tuning
- Model inferencing

Diagram with Compute, Data I/O, and Comm. showing dependencies on TensorFlow, cuDNN, ESSL, MAGMA, PyTorch, MPI, NVIDIA, and NCCL.
DL Deployment considerations on Summit

- ML/DL software deployment
- Framework comparisons
  - TensorFlow vs PyTorch
**DL Deployment considerations on Summit**

- **Native vs Container**
  - Impact of loading shared libs
  - Runtime performance

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster loading</td>
<td>Build overhead</td>
</tr>
<tr>
<td>Same runtime performance</td>
<td>Lack of flexibility</td>
</tr>
<tr>
<td>Self-Contain SW</td>
<td></td>
</tr>
</tbody>
</table>
Outline

- Deployment
- Parallelization
- Hyper-parameter Search
- Performance tuning
- Model inferencing
Distributed deep learning: parallel scheme

• Data parallel

P1

P2

P3

Model inconsistency:

Synchronized    Asynchronized

Stale

Review: arXiv:1802.09941

• Hybrid parallel

GPipe: arXiv:1811.06965

Device Mesh: [gpu0, gpu1, ...]

Layout Rule: [N, C, H, W]


Review: arXiv:1802.09941
Difference in scaling up: DL VS simulation

• DL is a global optimization, changing scale (data parallel) -> changing solution space.
  – Hyperparameters often need to be re-tuned at different scale

• Scale in FLOPS ≠ Scale in time-to-solution (accuracy)
  – Tradeoff between samples/s and faster convergence

• High per-node FLOPS makes DL comm- and/or IO- bound at relatively small node count.
  – DL requires optimized comm (mainly all-reduce) and IO pipeline
Distribute your DL codes on Summit

• Typical training flowchart
  – Allreduce of gradients in the backward propagation

```
Input → Cast to fp16 → Forward model → Cast to fp32 → Loss
        Communication
          / scale factor

Allreduce → Weights → Cast to fp16 → Backward model
          → Cast to fp16 → Weight gradients
          → Cast to fp32
```
Distribute your DL codes on Summit

- **Communication libraries**
  - Low level: NCCL, MPI
  - High level: Horovod, DDL

- **Framework support**
  - PyTorch:
    - `torch.distributed` (NCCL or MPI)
  - TensorFlow:
    - `distributed.Strategy` (NCCL)
  - 3rd Plugin: Horovod, DDL
Horovod: “mini-MPI” for distributed deep learning

- On top of MPI and NCCL
- Plugin support for TensorFlow, PyTorch, MXNet
- Tensor Fusion for better performances
- Key tuning parameters: fusion buffer and cycle time
Distribute your DL codes on Summit: PyTorch

- **torch.distributed with NCCL and MPI backends: Initialization**

```python
world_size = int(os.environ['OMPI_COMM_WORLD_SIZE'])
world_rank = int(os.environ['OMPI_COMM_WORLD_RANK'])
local_rank = int(os.environ['OMPI_COMM_WORLD_LOCAL_RANK'])

if args.backend == 'nccl':
    import subprocess
    get_master = "echo $({} | sort | uniq | grep -v batch | grep -v login | head -1 )".format(os.environ['LSB_DJOB_HOSTFILE'])
    os.environ['MASTER_ADDR'] = str(subprocess.check_output(get_master, shell=True))[2:-3]
    os.environ['MASTER_PORT'] = "23456"
    os.environ['WORLD_SIZE'] = os.environ['OMPI_COMM_WORLD_SIZE']
    os.environ['RANK'] = os.environ['OMPI_COMM_WORLD_RANK']

    dist.init_process_group(args.backend, rank=world_rank, world_size=world_size)
```

https://code.ornl.gov/olcf-analytics/summit/distributed-deep-learning-examples/tree/master/examples/pytorch
Distribute your DL codes on Summit: PyTorch

- torch.distributed with NCCL and MPI backends: Allreduce

```python
def benchmark_step():
    optimizer.zero_grad()
    output = model(data)
    loss = F.cross_entropy(output, target)
    loss.backward()
    if args.backend in ['nccl', 'mpi']:
        average_gradients(model)
    optimizer.step()
```
```python
def average_gradients(model):
    """ Gradient averaging. """
    for param in model.parameters():
        dist.all_reduce(param.grad.data, op=dist.ReduceOp.SUM)
        param.grad.data /= world_size
```
Distribute your DL codes on Summit: PyTorch

• Plugin distribution with Horovod and DDL:

```python
if args.backend in ['horovod','ddl']:
    import horovod.torch as hvd
    hvd.init()

    # Horovod: pin GPU to local rank.
    torch.cuda.set_device(hvd.local_rank())

    # Horovod: wrap optimizer with DistributedOptimizer.
    optimizer = hvd.DistributedOptimizer(optimizer,
                                          named_parameters=model.named_parameters(),
                                          compression=compression)

    # Horovod: broadcast parameters & optimizer state.
    hvd.broadcast_parameters(model.state_dict(), root_rank=0)
    hvd.broadcast_optimizer_state(optimizer, root_rank=0)
```
Distribute your DL codes on Summit: TensorFlow

### Build-in support for Data parallel

<table>
<thead>
<tr>
<th>Framework \ Distribution</th>
<th>Single node</th>
<th>Multi node</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>MirroredStrategy</td>
<td>MultiWorkerMirroredStrategy</td>
</tr>
<tr>
<td>PyTorch</td>
<td>DataParallel</td>
<td>DistributedDataParallel</td>
</tr>
</tbody>
</table>

- Easy of use and framework support
- High performance with NCCL backend
- Less flexible
Distribute your DL codes on Summit: TensorFlow

- Multi Worker Mirrored Strategy: TF_CONFIG setup

```python
get_cnodes = "echo $(cat {} | sort | uniq | grep -v batch | grep -v login)".format(os.environ['LSB_DJOB_HOSTFILE'])
cnodes = subprocess.check_output(get_cnodes, shell=True)
cnodes = str(cnodes)[2:-3].split(' ')
nodes_list = [c + ":2222" for c in cnodes]

# Add a port number
# Get the rank of the compute node that is running on
index = int(os.environ['PMIX_RANK'])

# Set the TF_CONFIG environment variable to configure the cluster setting
os.environ['TF_CONFIG'] = json.dumps({
    'cluster': {'worker': nodes_list },
    'task' : {'type': 'worker', 'index': index}
})
```
Distribute your DL codes on Summit: TensorFlow

- Multi Worker Mirrored Strategy: build-in support for tf.estimator

```python
communication = tf.distribute.experimental.CollectiveCommunication.NCCL
distribution_strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy(
    communication=communication)
run_config = tf.estimator.RunConfig(
    train_distribute=distribution_strategy,
    session_config=session_config,
    save_checkpoints_secs=60*60*24,
    save_checkpoints_steps=None)
```

https://code.ornl.gov/olcf-analytics/summit/distributed-deep-learning-examples/tree/master/examples/tensorflow
Always checkpointing

- It is relatively straightforward and cheap to checkpoint in DL.
- for data parallel, it is essentially the same as for single GPU.

```python
# Save checkpoint
if hvd.rank() == 0:
    state = {
        'model': model.state_dict(),
        'optimizer': optimizer.state_dict(),
    }
torch.save(state, filepath)

# Load checkpoint
if hvd.rank() == 0:
    checkpoint = torch.load(filepath)
    model.load_state_dict(checkpoint['model'])
    optimizer.load_state_dict(checkpoint['optimizer'])

# Horovod: broadcast parameters & optimizer state.
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
hvd.broadcast_optimizer_state(optimizer, root_rank=0)
```
Outline

- Deployment
- Parallelization
- Performance tuning
- Hyper-parameter Search
- Model inferencing
Scaling strategies

1. Compute:
   - Tune on single node with synthetic data

2. I/O
   - Tune on NVMe with input pipelining

3. Communication
   - Tune at scale with comm. libs
Scaling considerations: Compute

• Use Tensor Cores (2 ~ 4x) 15 TFLOPS(FP32) vs 120 TFLOPS(FP16)
  – PyTorch: NVIDIA Apex plugin https://github.com/NVIDIA/apex
    
    # Added after model and optimizer construction
    model, optimizer = amp.initialize(model, optimizer, flags...)
    # loss.backward() changed to:
    with amp.scale_loss(loss, optimizer) as scaled_loss:
      scaled_loss.backward()

  – TensorFlow:
    
    #Enable TF-AMP graph rewrite:
    os.environ['TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE'] = "1"
    #Enable Automated Mixed Precision:
    os.environ['TF_ENABLE_AUTO_MIXED_PRECISION'] = '1'

  – Verify on Tensorcore:
    
    #Turn off Tensorcore:
    os.environ['TF_DISABLE_CUDNN_TENSOR_OP_MATH'] = "0"
    #nvprof: tensor_precision_fu_utilization to show TC utilization
Scaling considerations: Compute

- Use XLA (~ 1.5x for ResNet50)

  ```
  #Enable XLA:
  session_config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.ON_1
  ```

- Tune cuDNN algorithms (e.g. 7 implementations for conv)

  ```
  #TensorFlow
  os.environ['TF_CUDNN_USE_AUTOTUNE'] = '1'
  #PyTorch
  torch.backends.cudnn.benchmark = True
  ```

- Know your kernels (optimal scheduling policy)
Scaling considerations: Compute

• Benchmark kernels
  – CNN:
    kernel size, # of kernels, etc.
  – RNN:
    batch size, timesteps, etc.
Scaling considerations: I/O

- Use node local NVMe

<table>
<thead>
<tr>
<th>Device \ Bandwidth</th>
<th>Full system run</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUs processing</td>
<td>$3 \times 224 \times 224 \times 4 \times 1200 \times 6 \times 4608 / 10^{12}$ ~ 20 TB/s</td>
</tr>
<tr>
<td>GPFS reading</td>
<td>2.5 TB/s</td>
</tr>
<tr>
<td>NVMe reading</td>
<td>6 GB/s $\times$ 4608 ~ 27 TB/s</td>
</tr>
</tbody>
</table>

- Use LMDB input format

---

Uncompressed ImageNet data, batch=256 per GPU

I/O read throughput for 6 ranks on a Summit node

<table>
<thead>
<tr>
<th>SMT</th>
<th>TFRecord</th>
<th>HDF5</th>
<th>LMDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image1.png" alt="Graph" /></td>
<td><img src="image2.png" alt="Graph" /></td>
<td><img src="image3.png" alt="Graph" /></td>
</tr>
</tbody>
</table>

SMT: Simultaneous Multithreading

GPU processing speed is $3 \times 224 \times 224 \times 4 \times 1200 \times 6 \times 4608 / 10^{12}$ ~ 20 TB/s.

GPFS reading speed is 2.5 TB/s.

NVMe reading speed is 6 GB/s $\times$ 4608 ~ 27 TB/s.
Scaling considerations: I/O

- CPU affinity settings (for pre-processing OPs)
  - Default binding:
  - Correct binding (numactl + OMP_PLACE):

```
# for SMT4
case($(($PMIX_RANK%6))) in
  [0])
    export PAMI_IBV_DEVICE_NAME=mlx5_0:1
    export OMP_PLACES={0:28}
    numactl --physcpubind=0-27 --membind=0 $APP
  ;;
  [1])
    export PAMI_IBV_DEVICE_NAME=mlx5_1:1
    export OMP_PLACES={28:28}
    numactl --physcpubind=28-55 --membind=0 $APP
...
```

- Use pre-processing pipeline https://www.tensorflow.org/guide/data_performance: staging, prefetch, parallel_interleave, etc
Scaling considerations: Communication

- Use Horovod with NCCL backend

**TensorFlow**

![Graph showing images/sec vs. # of compute nodes for TensorFlow using Horovod and Multi-worker Mirrored Strategy.](image)

**PyTorch**

![Graph showing images/s vs. # of Summit nodes for PyTorch using different distributed strategies.](image)
Scaling considerations: Communication

• Tune Horovod parameters
  – Key knobs: HOROVOD_CYCLE_TIME, HOROVOD_FUSION_THRESHOLD

```bash
#Horovod autotuner
export HOROVOD_AUTOTUNE=1
export HOROVOD_HIERARCHICAL_ALLGATHER=0
export HOROVOD_HIERARCHICAL_ALLREDUCE=0
export NCCL_DEBUG_SUBSYS=COLL
```
Scaling considerations: Communication

• Add Bit-Allreduce to Horovod  [https://arxiv.org/abs/1909.11150]

Original coordination strategy:

Improved coordination strategy:
Scaling considerations: Communication


- Bit-Allreduce and Grouping generates overall 8x improvement in parallel efficiency
Scaling considerations: putting it together

- ImageNet training

TF_CNN_Benchmark on Summit: ResNet50
batch-size = 256 per GPU

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Images/second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5320</td>
</tr>
<tr>
<td>8</td>
<td>41746</td>
</tr>
<tr>
<td>64</td>
<td>321492</td>
</tr>
<tr>
<td>128</td>
<td>631664</td>
</tr>
<tr>
<td>256</td>
<td>1227241</td>
</tr>
<tr>
<td>512</td>
<td>2408388</td>
</tr>
<tr>
<td>1024</td>
<td>4731623</td>
</tr>
</tbody>
</table>

87%
Scaling strategies: convergence considerations

- Large batch training: Optimal batch size \(\sim\) gradient noise scale (arXiv: 1812.06162)


- Scaling in time-to-solution is more challenging

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Mini-batch size</th>
<th>Top-1 Val accuracy</th>
<th>Training time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>12288</td>
<td>0.750</td>
<td>27</td>
</tr>
<tr>
<td>32</td>
<td>12288</td>
<td>0.766</td>
<td>17</td>
</tr>
<tr>
<td>64</td>
<td>15360</td>
<td>0.763</td>
<td>12</td>
</tr>
</tbody>
</table>

(b) Empirical Accuracy (ResNet-50, figure adapted from [Goyal et al. 2017], lower is better)
Outline

Deployment → Parallelization

Hyper-parameter Search → Performance tuning

Model inferencing
Hyperparameter search with Ray Tune


**Setup on Summit**

- **Redis Cluster**
- **Trail Runner/Executor**

**Scripts to start/stop Ray cluster**

```bash
nodes=$(<${LSB_DJOB_HOSTFILE} | sort | uniq | grep -v login | grep -v batch)
head=${nodes[0]}

ssh $head ray start --head --no-ui --redis-port=6379 --temp-dir=$tmpdir --num-cpus=42 --num-gpus=6

for worker in ${nodes[@]}; do
  ssh $worker ray start --redis-address="$head:6379" --temp-dir=$tmpdir --num-cpus=42 --num-gpus=6 &
  if [ $? -eq 0 ]; then
    echo "Ray worker started on $worker"
  fi
done
wait
```
Hyperparameter search with Ray Tune

• Using ray.tune.Trainable class

class Cifar10Model(Trainable):
    def _setup(self, config):
        model = self._build_model(depth=self.config['depth'])
        opt = tf.keras.optimizers.Adam(lr=self.config['lr'],
                                        decay=self.config['decay'])

    def _train(self):
        self.model.fit_generator(generator=gen,
                                  steps_per_epoch=self.config['batch_size'],
                                  epochs=self.config['epochs'])

• Run experiments

    ray.init(redis_address=vars.redis_address)
    pbt = PopulationBasedTraining(perturbation_interval=10,...)
    run_experiments({'pbt_cifar10': train_spec}, scheduler=pbt)

https://code.ornl.gov/olcf-analytics/summit/distributed-deep-learning-examples/tree/master/examples/ray
Hyperparameter search with Ray Tune

Run experiments

```python
train_spec = {
    "run": Cifar10Model,
    "resources_per_trial": {
        "cpu": 42,
        "gpu": 6
    },
    "stop": {
        "mean_accuracy": 0.90,
        "training_iteration": 50
    },
    "config": {
        "epochs": 10,
        "batch_size": 64*6,
        "lr": grid_search([10**-3, 10**-4]),
        "decay": sample_from(lamda spec: spec.config.lr / 10.0),
        "depth": grid_search([20,32,44,50])
    }
}
```

Main tuning parameters
Hyperparameter search with Ray Tune

• Vis with TensorBoard
  – TensorFlow
  – PyTorch

• Population based training example
Outline

- Deployment
- Parallelization
- Hyper-parameter Search
- Performance tuning
- Model inferencing

Diagram with Compute, Data I/O, and Communication nodes connected to various technologies like TensorFlow, cuDNN, PyTorch, ESSL, MAGMA, MPI, NCCL, HOROVD, and others.
Use TensorFlow C++ binding for Inferencing

• Train offline with Python interface
• Save the model checkpoint
  – Optimize the model
    tensorflow.python.tools.optimize_for_inference
• Deploy the model to simulation code with C++ interface
• Example use case: surrogate modeling

Binary files contain weights, gradients, and other variables
Protocol buffer file saves the TensorFlow graph
Use TensorFlow C++ binding for Inferencing

// Create TF session
tensorflow::Session sess; tensorflow::SessionOptions options
options.config.mutable_gpu_options()->set_visible_device_list(std::to_string(local_rank));
tensorflow::NewSession(options, sess)

// Load TF graph
tensorflow::MetaGraphDef graph_def;
ReadBinaryProto(tensorflow::Env::Default(), graph_file, &graph_def)
sess->Create(graph_def.graph_def())

// Load TF checkpoint
tensorflow::Tensor checkpointPathTensor(tensorflow::DT_STRING, tensorflow::TensorShape());
checkpointPathTensor.scalar<std::string>() = checkpoint_file;
std::vector<std::pair<std::string, tensorflow::Tensor>> feed_dict = {
    {graph_def.saver_def().filename_tensor_name(), checkpointPathTensor}};
sess->Run(feed_dict, {}, {graph_def.saver_def().restore_op_name()}, nullptr);
Compile with TensorFlow C++ binding

module load ibm-wml-ce gcc/7.4.0

tf_include=$(python -c "import tensorflow as tf; import sys; print (tf.sysconfig.get_include())")
tf_lib=$(python -c "import tensorflow as tf; import sys; print (tf.sysconfig.get_lib())")

export CPATH=$tf_include:$CPATH
export LIBRARY_PATH=$tf_lib:$LIBRARY_PATH

mpic++ inference.cpp -ltensorflow_cc -ltensorflow_framework -std=gnu++11
Conclusion

• Summit is ideal for deep learning applications.
• ImageNet training with ResNet50 can achieve 87% scaling efficiency up to 1024 nodes on Summit.
• Convergence issue of large-batch training may require hybrid-parallel to utilize full Summit parallelism.

For more information:
https://code.ornl.gov/olcf-analytics/summit/distributed-deep-learning-examples

Other resources

- TensorFlow distributed training: https://www.tensorflow.org/guide/distributed_training
- PyTorch data parallel: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html
- Horovod examples: https://github.com/horovod/horovod/tree/master/examples