AI Frameworks on Aurora and Sunspot

Corey Adams

AI Frameworks

At ALCF, we support the Machine Learning Frameworks on our systems to enable user science applications.

- Tensorflow
  - with Horovod
- Pytorch
  - Scale out with Horovod, DDP, Deepspeed
- JAX
  - Including mpi4jax

What we do

We focus on

- Optimized Installations
- Performance Tuning Suggestions
- Scale out best practices

For Aurora, this is a close collaboration with Intel balancing our user needs with their optimization strategies.

AI Frameworks on Aurora

Aurora's compute nodes have Intel's Data Center Max 1550 GPUs. **CUDA is not supported natively**.

Despite that, the AI frameworks that you are already using are well supported on Sunspot and Aurora.

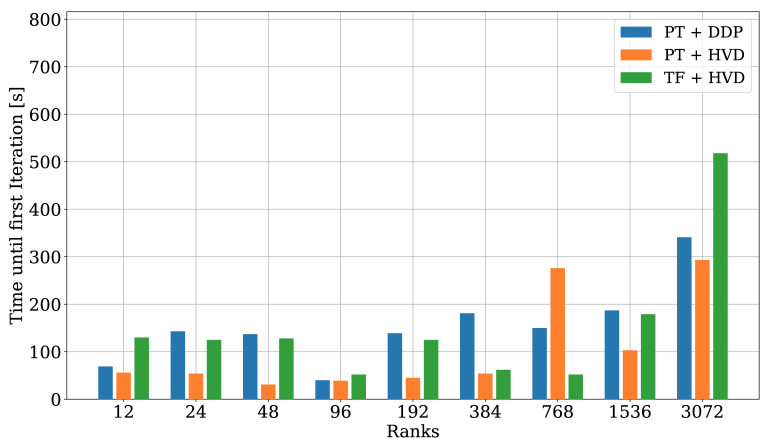
This talk

We'll cover the basics of using these AI softwares on our Intel GPU systems, as well as go through some examples of running AI codes on ALCF systems.

In particular, I'll put some emphasis on the adaptations needed to your code to convert CUDA-based AI codes to Intel-based AI codes.

Built in Frameworks

On Aurora, in early testing of python codes at scale we discovered limitations in the filesystems that was detrimental to start up times.



Startup Times on Aurora increase with node count!

To mitigate this, we now install the core components of the frameworks directly to the node image.

Getting started: Sunspot

On Sunspot:

```
Last login: Sun May 26 01:08:37 2024 from bastion-02.alcf.anl.gov
❯ module avail frameworks

-------------------- /opt/aurora/23.275.1/modulefiles -------------
   frameworks/2023.10.15.001
-------------------- /opt/aurora/23.275.2/modulefiles -------------
   frameworks/2023.12.15.001

❯ module load frameworks/2023.12.15.001

The following have been reloaded with a version change:
  1) oneapi/eng-compiler/2023.12.15.002 => oneapi/release/2023.12.1

❯ which python
/opt/aurora/23.275.2/frameworks/aurora_nre_models_frameworks-2024.0
```

Under the hood

Intel's optimizations for the AI frameworks are built on different libraries than CUDA. Intel's ecosystem (oneAPI) includes many libraries that share functionality with a CUDA equivalent. Of particular relevance:

- oneMKL ⇔ CUDA's math library
- oneDNN ⇔ CUDNN
- oneCCL ⇔ NCCL
- (python) dpctl ⇔ cupy
- (python) daal4py ⇔ RAPIDS (data analytics)

Optimized Performance

From python, many packages require *extensions*:

- Intel Extension for Scikit-learn enables classical machine learning techniques.
- Intel Extension for Tensorflow enables XPU support in Tensorflow.
- Intel Extension for Pytorch brings `torch.xpu` to replace `torch.cuda` in your code, and optimized backend functions.
- Intel Extensions also exist for DeepSpeed, OpenXLA, Triton, Transformers, Horovod

Preview Modulefiles

Putting the learning frameworks in the compute image requires time for validation and deployment, so there is a delay between the "latest" and the "deployed" versions. The **latest** version on Sunspot is currently available on `/soft/`:

```
❯ module use /soft/preview-modulefiles/24.086.0

Due to MODULEPATH changes, the following have been reloaded:
  1) mpich-config/collective-tuning/1024

❯ module avail frameworks

-------------------- /soft/preview-modulefiles/24.086.0 -----------
   frameworks/2024.04.15.001    frameworks/2024.04.15.002 (D)
```

Frameworks Deployment on Aurora

We do not expect to use `/soft/` at scale on Aurora for the frameworks, at least not at first. - If you are happy with the version deployed into the OS, use that. - If you want to test the next drop, use the version on `/soft/` at small scales. - “Small” is likely 128 nodes or less. (Which is actually about as much compute as Polaris!) - If you need the latest and greatest at scale, or need packages not already installed, you will need to use **the tarball ramdisk method**.

Tarball Ramdisk??

To avoid reading from `/soft/`, we have in the past - and likely will in the future - provide tarballs of `oneAPI` and the python ecosystem as deployed into `/tmp/`.

- On compute nodes, `/tmp/` is ramdisk. You can rsync and untar the tarball to `/tmp/` and - while it will decrease available CPU memory - it has a much much better latency and scalability than reading from `/soft`.
- You can also deploy a virtualenv to this space for additional packages.
  - It's also possible to put a virtualenv into `/lus/` or `/home/` - there appears to be a special quirk of `/soft/` that limits that area.
  - But, at the largest, largest scales, expect to need to know how to use the ramdisk for software deployment on the fly!

The Python Packages

There are *many* packages installed into this conda environment.

Try `pip list | grep tensor` for example to see if tensorflow is installed:

```
1 ❯ pip list | grep torch
2 intel-extension-for-pytorch       2.1.30+xpu
3 torch                             2.1.0.post2+cxx11.abi
4 torchvision                       0.16.0.post2+cxx11.abi
```

In general, unless you have a compelling reason not to, it's highly recommended you use this module if your application's python needs are supported.

If you have issues, please contact support@alcf.anl.gov

What to do to get more packages?

If you need to install the latest version of another package, or the dependencies you need are missing, we encourage you to *extend* our python install rather than build your own:

```
# (Set up and activate the conda module!)
export VENV_DIR=/tmp/conda-extension/

# Do this just one tilus/grand/projects/gpu_hack/AwesomeProject/sof
python -m venv --system-site-packages ${VENV_DIR}

# Do this every time you run after `conda activate`:
source ${VENV_DIR}/bin/activate

# Now, pip install whatever you need into your virtual env:
pip install --upgrade pytorch_lightning

# Create an archive of your virtual env:
tar -zcvf virtualenv.tar.gz $VENV_DIR
```

Deploying to ramdisk:

```
# Determine the node count programmatically:
NNODES=`wc -l < $PBS_NODEFILE`

# Untar the package to /tmp once per node!
mpiexec -n ${NNODES} -ppn 1 tar -xzf virtualenv.tar.gz -C /tmp/

export VENV_DIR=/tmp/conda-extension/
```

Which Framework to Use?

The AI frameworks all have some of their own advantages, historically, though in many cases there is convergence.

For example, compilation (seen first in `tensorflow`) and functional transforms (seen first in `JAX`) have both shown up in `pytorch`.

Tensorflow

Many new models are using pytorch or JAX, but if you're using Tensorflow:

- Use mixed precision to get the best performance
- Use tf.function syntax to enable tracing of your code and graph compilation
- (Experimental) Enable XLA in your code by setting `jit_compile=True` in the tf.function calls.
  - XLA is experimental and early development on Intel GPUs. Support is expected to ramp up with improved performance.

Scaling Tensorflow

Historically, at ALCF we have encouraged users to apply `horovod` for scaling their tensorflow models.

The documentation is good and comprehensive for TF+HVD, though moving forward it's unclear if support will continue long term.

The `num_groups=1` argument of `horovod.tensorflow.DistributedOptimizer` or `DistributedGradientTape` is recommended by Intel.

Are you using tensorflow + horovod? Are you planning to continue this long term? We want to know about your use case - please reach out to us!

Adapting your CUDA code to XPU

Tensorflow is among the easiest frameworks to port. Most changes occur only if you're manually setting visible devices:

```
1
2 if self.args.run.compute_mode == ComputeMode.CUDA:
3     gpus = tf.config.list_physical_devices('GPU')
4     tf.config.experimental.set_visible_devices(gpus[hvd.local_rank(
5 elif self.args.run.compute_mode == ComputeMode.XPU:
6     gpus = tf.config.list_physical_devices('XPU')
7     tf.config.experimental.set_visible_devices(gpus[hvd.local_rank(
```

(Code from CosmicTagger)

Pytorch

Pytorch is the most common framework we see on Polaris and Aurora!

- Pytorch is more "numpy-like" than tensorflow (though honestly we're now all using the phrase "pytorch-like").
- Since Pytorch 2.0, graph compilation is also available in pytorch and you should try to use it if you can.
  - Torch.compile is experimentally supported and functional in the 2024.1 frameworks drop, `frameworks/2024.04.15.002`. Please test it but expect you may see *decreased* performance in some models at this time.
- The pytorch "ecosystem" is broad and we haven't installed every package under the sun. Please take advantage of the `virtualenv` interface to get what you need and reach out to `support@alcf.anl.gov` for help.

Adapting your CUDA code to XPU

You must manually import Intel's extension to pytorch:

```
1 import torch
2 try:
3     # I put it in a try/except to make portable code w/ CUDA ...
4     import intel_extension_for_pytorch as ipex
5 except:
6     pass
```

Access the XPU device instead of CUDA:

```
1 device = ipex.xpu.device("xpu:0")
2 # Or, with ipex imported:
3 device = torch.device("xpu:0")
```

Adapting your CUDA code to XPU

Optimal performance on XPU is often found with the channels-last format:

```
1
2 if self.args.data.data_format == DataFormatKind.channels_last:
3     if self.args.run.compute_mode == ComputeMode.XPU:
4         self._raw_net = self._raw_net.to("xpu").to(memory_format=tc
```

Apply to the inputs and labels too:

```
1 minibatch_data["image"] == minibatch_data['image'].to(memory_format
2 minibatch_data["label"] == minibatch_data['label'].to(memory_format
```

Scaling Pytorch

- Pytorch can be scaled with horovod or the built in DDP method. Both are supported on Aurora and Sunspot.
- Differences in concurrency features on CUDA vs. Intel GPUs leads to different optimizations for horovod and DDP.
  - Use `num_groups=1` for horovod as well.

On XPU, the required backend for DDP is "CCL":

```
1 if self.args.run.compute_mode == ComputeMode.XPU:
2     import oneccl_bindings_for_pytorch
3     backend = 'ccl'
```

DeepSpeed

Users pursuing very large models should consider tools such as DeepSpeed and it's derivatives:

- ZeRO Offloading can save memory by partitioning the optimizer state, gradients, and model weights, across a distributed run.
- Pipeline parallelism can support other use cases that don't fit on a single device.
- DeepSpeed techniques have trickled into other areas (pytorch's Fully Sharded Data Parallel, Megatron)
- Intel's extension for DeepSpeed is essential for good performance and DeepSpeed on XPU is still in development.

Pytorch Lightning

Pytorch Lightning is a very user friendly package for your pytorch models. It *does* scale out on Sunspot and Aurora (and it's pretty easy) but you have to use the right plugins. You also must use ALCF's branch of Pytorch Lightning. Be sure to select the XPU Accelerator Option:

```
# Select the accelerator:
if args.run.compute_mode == ComputeMode.XPU:
    from lightning.fabric.accelerators import XPUAccelerator
    accelerator = XPUAccelerator()
elif args.run.compute_mode == ComputeMode.CUDA:
    from lightning.fabric.accelerators import CUDAAccelerator
    accelerator = CUDAAccelerator()
else:
    from lightning.fabric.accelerators import CPUAccelerator
    accelerator = CPUAccelerator()
```

Scaling Lightning Code

```
 1 import torch
 2 import pytorch_lightning as pl
 3
 4 from lightning_fabric.plugins.environments import MPIEnvironment
 5 environment = MPIEnvironment()
 6
 7 from pytorch_lightning.strategies import DDPStrategy
 8 strategy = DDPStrategy(
 9     cluster_environment = environment,
10 )
11
12 trainer = pl.Trainer(
13     strategy = strategy,
14 )
```

JAX

- JAX is the successor of `autograd` and built with `XLA` as a backend for `numpy`.
  - JAX implements `numpy`'s interface - even on the GPU!
  - JAX has a performant backend for nearly every operation.
  - JAX is also purely functional, and python performance issues can be significantly removed with `jit`
  - JAX has the most flexible and well documented `autograd`, which can be composed with other transformations (like `vmap`).

JAX on Sunspot/Aurora

JAX has early support on Sunspot and Aurora through the Intel Extension for OpenXLA. All JAX modules have been built and installed by LCF staff (me).

JAX is still very early in development for XPU. Most operations are functional, but performance can be subpar. `mpi4jax` is supported but not concurrently with `jax.device_put` (there is a bug in upstream JAX, already fixed, but hasn't made it to XPU yet.)

```
1 ❯ module use /soft/modulefiles
2 ❯ module avail jax
3
4 ---------------------- /soft/modulefiles ------------------------
5    jax/0.4.4    jax/0.4.20    jax/0.4.24    jax/0.4.25 (D)
```

Scaling JAX on Sunspot

JAX can scale out with several methods:

- `pmap` and `pjit` (which is just regular `jit` with extra args) are parallelization routines to parallelize your function. They do things "for you" but you still have to know what changes they will make.
- `mpi4jax` is the simplest scale out method: implement `jit`able mpi collectives within JAX, supported on Polaris.
- JAX's newer `shard map` technique is more complicated but offers more fine granularity for parallelization. It's also a little harder to use.

All of these are experimental on XPU and may not work yet!

Best Practices for running on Polaris

Cosmic Tagger

We'll use an example application to demonstrate some key differences on Polaris in terms of performance.

CosmicTagger is a high resolution computer vision application for semantic segmentation of neutrino images.

It was an acceptance test for Polaris and is an acceptance test for Aurora, and available in 3 frameworks.

Initial Setup

You can follow along with this demonstration if you like, or return later and run these examples.

Download the code for CosmicTagger:

```
1 git clone https://github.com/coreyjadams/CosmicTagger.git
2 cd CosmicTagger
3 git checkout v2.1
```

Run from a Compute Node

Determine the node count and number of ranks:

```
#!/bin/bash -l

# What's the cosmic tagger work directory?
WORK_DIR=/home/cadams/Polaris/CosmicTagger
cd ${WORK_DIR}

# MPI and OpenMP settings
NNODES=`wc -l < $PBS_NODEFILE`
NRANKS_PER_NODE=1

let NRANKS=${NNODES}*${NRANKS_PER_NODE}
```

Software setup

Set the batch size per GPU and activate conda:

```
1 LOCAL_BATCH_SIZE=1
2
3 # Set up software deps:
4 module use /soft/preview-modulefiles/24.086.0
5 module load frameworks/2024.04.15.002
6 source /home/cadams/frameworks-2024.1-extension/bin/activate
```

Python call

Run configuration and call:

```
1
2  python ${WORK_DIR}/bin/exec.py \
3  --config-name a21 \
4  framework=torch \
5  data.data_format=channels_last \
6  run.compute_mode=XPU \
7  data=synthetic \
8  run.id=polaris_${LOCAL_BATCH_SIZE}-ranks${NRANKS}-nodes${NNODES} \
9  run.distributed=True \
10 run.minibatch_size=${LOCAL_BATCH_SIZE} \
11 run.iterations=100
```

The full Script

Here's a script that runs CosmicTagger in Pytorch (from an interactive node)

```
#!/bin/bash -l

# What's the cosmic tagger work directory?
WORK_DIR=/home/cadams/Polaris/CosmicTagger
cd ${WORK_DIR}

# MPI and OpenMP settings
NNODES=`wc -l < $PBS_NODEFILE`
NRANKS_PER_NODE=1

let NRANKS=${NNODES}*${NRANKS_PER_NODE}

LOCAL_BATCH_SIZE=1

# Set up software deps:

# Set up software deps:
module use /soft/preview-modulefiles/24.086.0
```

Performance - Pytorch, single GPU

Performance in Img/s (synthetic data)

Minibatch Size	**1**	**2**	**4**	**8**	**16**
tf32	15.1	22.8	30.2	32.0	31.0
bf16	14.7	24.7	38.6	44.3	32.8
fp16	14.9	23.0	30.1	31.9	31.0

Performance Tuning

For some applications, better performance is seen with `unset IPEX_XPU_ONEDNN_LAYOUT_OPT`.

Minibatch Size	1	2	4	8	16
tf32	16.3	25.2	31.5	33.2	31.7
bf16	15.6	26.9	40.5	45.9	33.4
fp16	16.5	24.9	31.7	33.2	31.8

This is yielding 5 to 10% improvement for CosmicTagger. But, you may not see benefit from this change, not all apps do.

Comparison with A100

For certain configurations, PVC (single-tile) **without** `torch.compile` competes A100 **with** `torch.compile`:

Minibatch Size	PVC-1	A100-1	PVC-8	A100-8
compiled tf32	16.3	22.9	33.2	33.1

In general, for applications we are testing at Argonne, performance of a single PVC tile is on par or better than an A100.

Scaling up the model

If your code is configured for scale up, it can be easy:

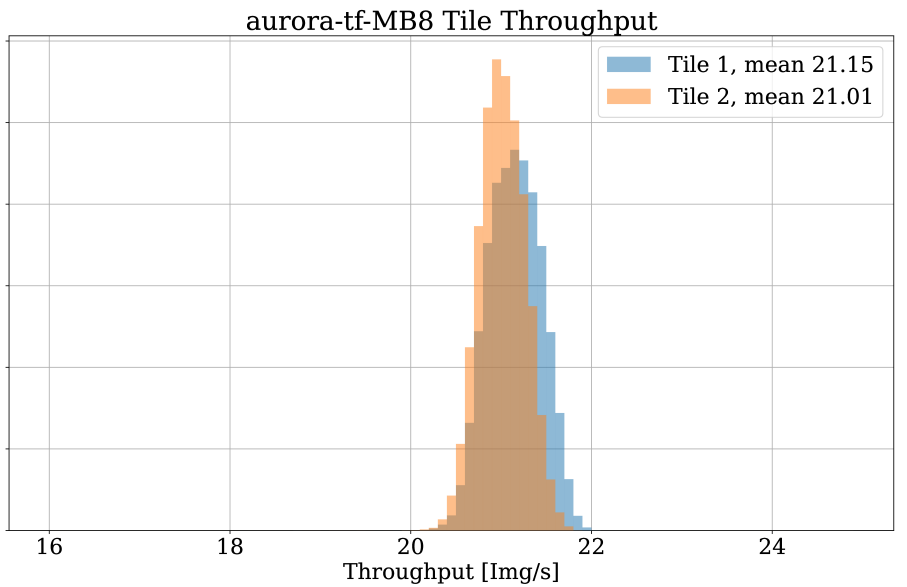
```
mpiexec -n ${NRANKS} -ppn ${NRANKS_PER_NODE} \
python ${WORK_DIR}/bin/exec.py \
--config-name a21 \
framework=torch \
data.data_format=channels_last \
run.compute_mode=GPU \
data=synthetic \
run.id=polaris_${LOCAL_BATCH_SIZE}-ranks${NRANKS}-nodes${NNODES} \
run.distributed=True \
run.minibatch_size=${LOCAL_BATCH_SIZE} \
run.iterations=100
```

On PVC, optimal performance is seen with 2 ranks per GPU (One / tile) so `NRANKS_PER_NODE` is typically **12**.

Controlling GPU Placement

Similar to the environment variable `CUDA_VISIBLE_DEVICES`, Intel provides a variable to control GPU device placement: `ZE_AFFINITY_MASK`. Settings are typically `{GPU}.{TILE}`, so available GPUS are 0.0, 0.1, 1.0, 1.1, 2.0, 2.1, 3.0, 3.1, 4.0, 4.1, 5.0, 5.1.

Tile 0 is typically *slightly* faster than tile 1, as measured in March with CosmicTagger on 768 GPUs:



CPU Affinity:

The CPU binding can have a large impact on performance with scale out of models. In particular, `numa` has been observed to degrade performance on Sunspot. For CosmicTagger, running in the configuration shown above, the rank-local throughput was measured on the latest drop as:

- no binding: 37.3 Img/s
- depth: 37.2 Img/s
- numa: 33.4 Img/s
- dedicated: 35.7 Img/s

Here, "dedicated" means this:

```
1 export CPU_AFFINITY="list:0-7,104-111:8-15,112-119:16-23,120-127:24-31,128-135:32-39,136-143:40-47,1
2
3 # MPI call:
4 mpiexec --cpubind=$CPU_AFFINITY -n ....
```

This binding is suggested by Intel. More experimentation is needed to determine final, optimal settings but we recommend you try a few at the 1 or 2 node scale.

Large Scale Jobs

When you scale out beyond 16 nodes, your jobs will require several environment variables to be set:

```
1 # This is a fix for running over 16 nodes:
2 export FI_CXI_DEFAULT_CQ_SIZE=131072
3 export FI_CXI_OVFLOW_BUF_SIZE=8388608
4 export FI_CXI_CQ_FILL_PERCENT=20
```

These fix several hangs and crashes due to communication message buffer sizes. There is no detriment to setting these in your scripts below 16 nodes, so feel free to include them by default.

Questions?