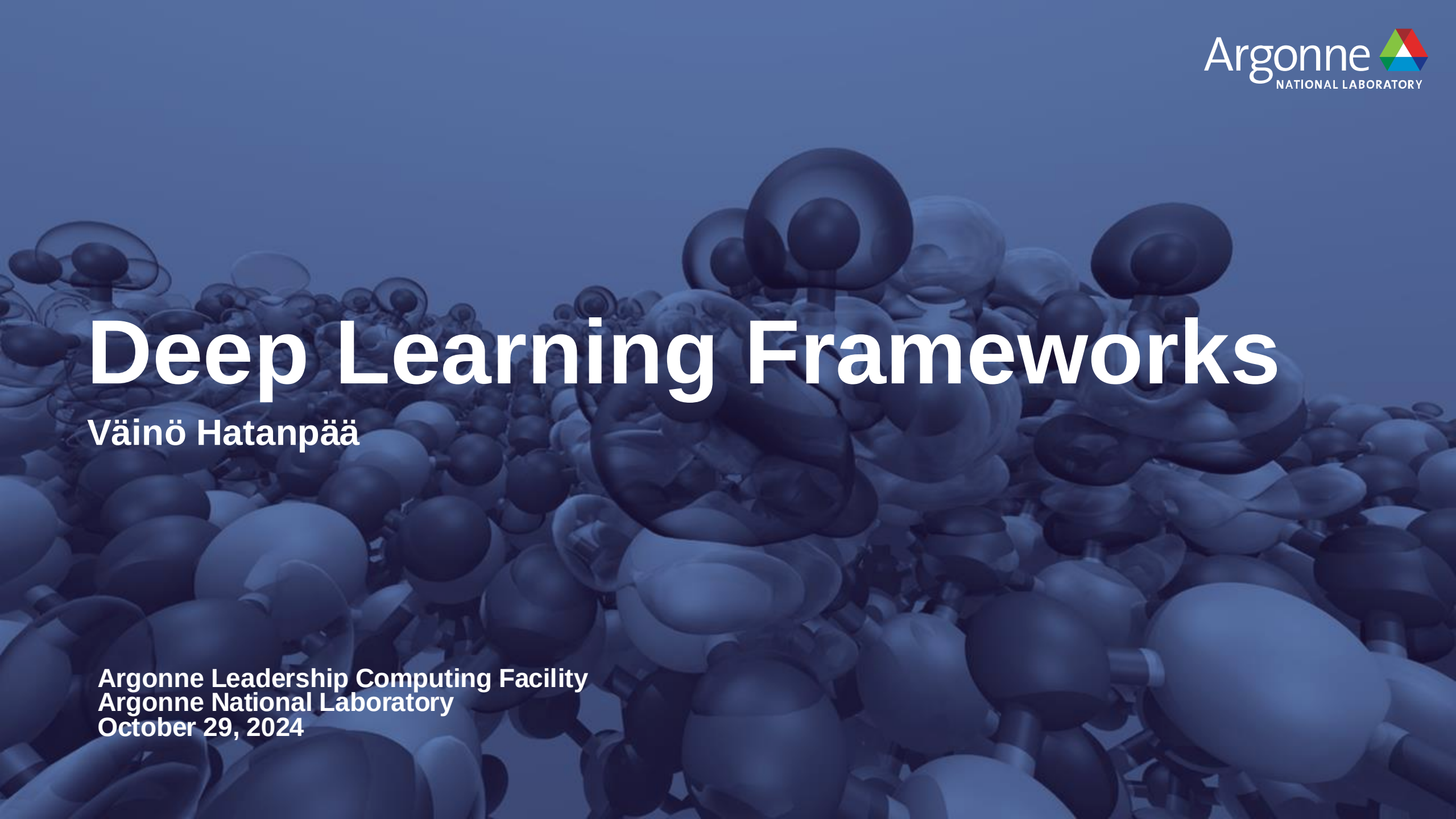

October 29-31, 2024
ALCF Hands-on HPC Workshop



Deep Learning Frameworks

Väinö Hatanpää

Argonne Leadership Computing Facility
Argonne National Laboratory
October 29, 2024

Outline

1. A quick intro to deep learning frameworks: PyTorch, TensorFlow, and Jax
2. Intel developer cloud demo
3. Usage on ALCF systems demo
4. Extending python environments
5. Performance considerations, mixed precision
6. Intro to distributed training

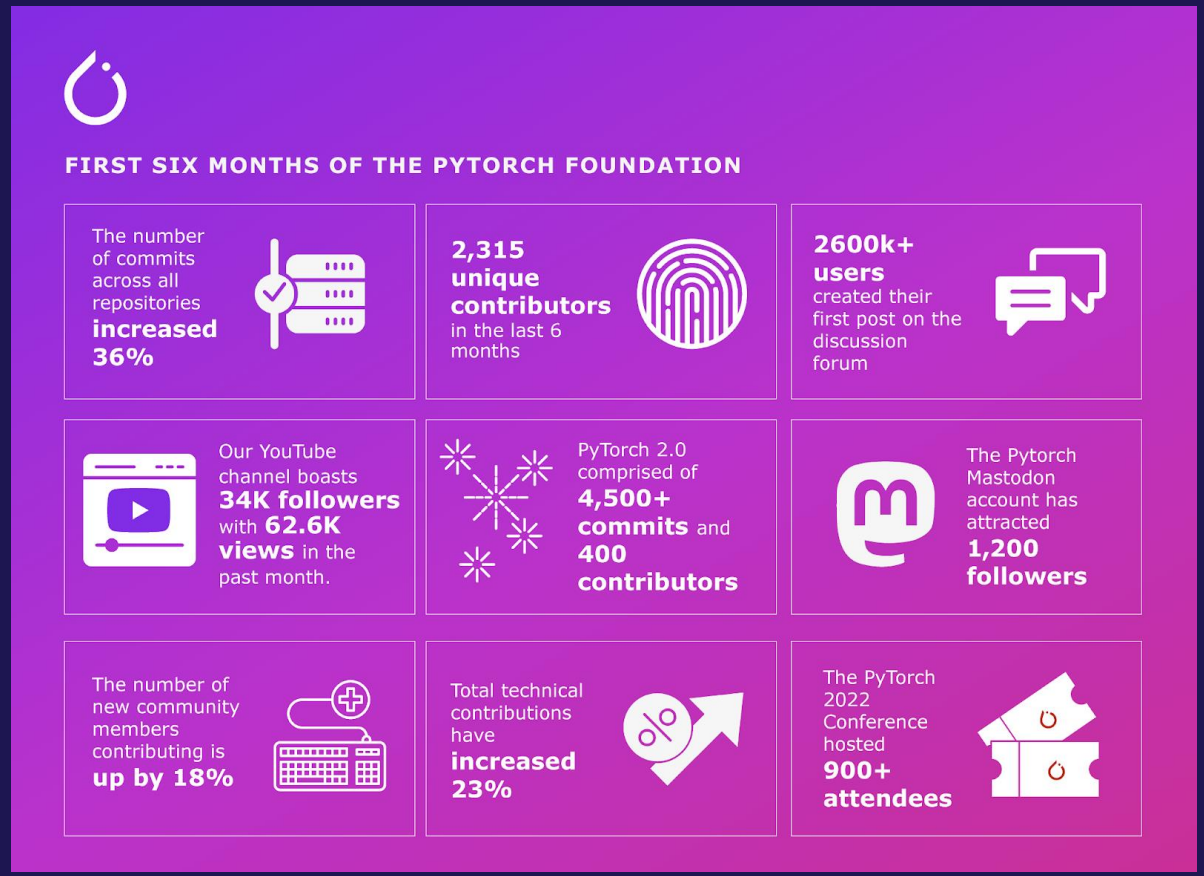
Frameworks

Provide various essential features for deep learning:

- Automatic differentiation
- GPU offloading and acceleration from python
  - Various efficient implementations included, some JIT compiled
- Essential building blocks for ML operations
- Communication and distributed training libraries integrated to scale out to multiple GPUs/nodes
  - NCCL (Nvidia), oneCCL (Intel), RCCL (AMD)
  - Horovod (TF, PyTorch)
  - DeepSpeed

PyTorch

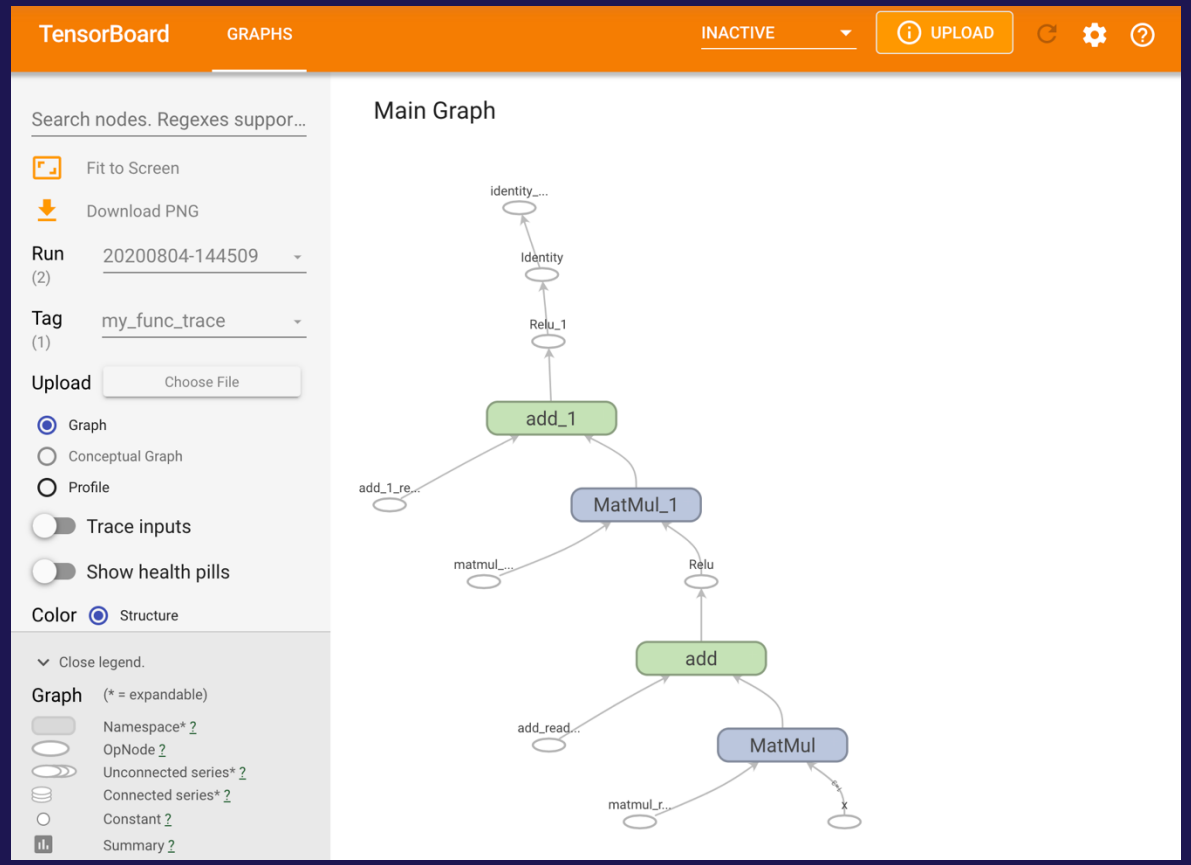
- Based on Torch (2002-2017), a machine learning library based on Lua and C/C++
- In 2017 the development moved to PyTorch, a Python port of the library
- Originally developed by Meta. Since 2023 PyTorch is governed by PyTorch foundation, subsidiary of Linux Foundation
- “NumPy-like” pythonic design
- Dynamic graphs, easy for research and prototyping
  - But still achieves good performance
- “Compile” support since 2.0
  - Compiling the dynamic grap into an efficient one
  - We are interested in how this works for your model
  - 2.0 backward compatible with 1.0 and introduces minimal breaking changes (Unlike TensorFlow)
- Easy to integrate C++ extensions and custom kernels
  - implemented with pybind11
  - https://pytorch.org/tutorials/advanced/cpp_extension.html



https://www.linuxfoundation.org/blog/pytorch-foundation-the-first-six-months

TensorFlow

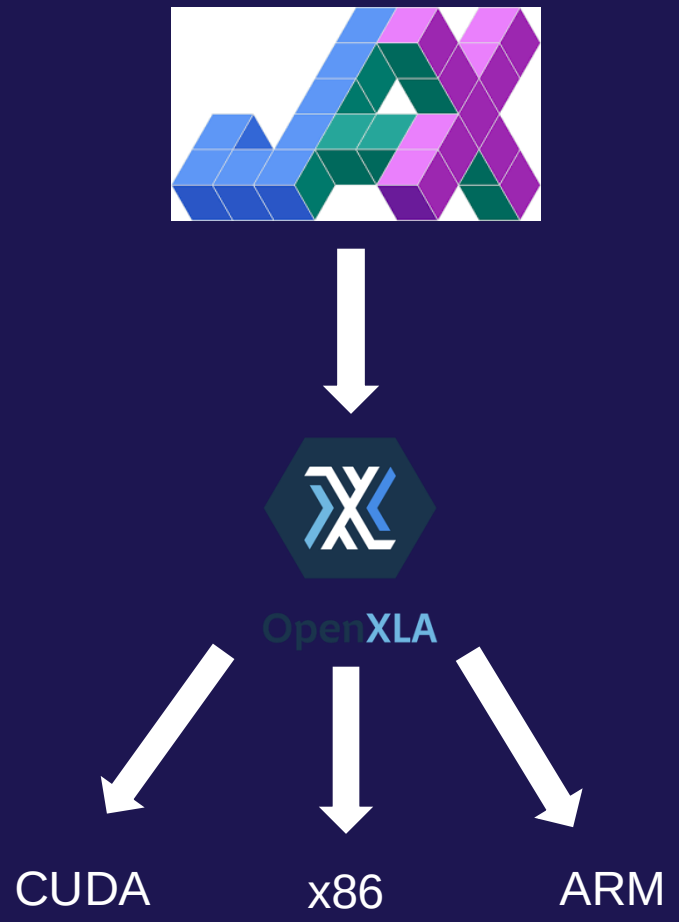
- Released in 2015, developed by Google
- Significant changes from v1 to v2 (2019). Only v2 supported at ALCF
- Keras integration
  - Also in PyTorch but more commonly used with TF
  - Provides higher level API, potentially making usage even easier
- TF has static graphs. Good for production use.
- XLA compilation support
  - Compiles TensorFlow graphs into efficient machine code
  - Potentially harder to debug and profile due to renamed kernels



https://www.tensorflow.org/guide/intro_to_modules

JAX

- Relatively new framework from Google
- Approach driven by purely functional transformations
  - Some gotchas https://jax.readthedocs.io/en/latest/notebooks/Common_Gotchas_in_JAX.html
- JIT compilation with OpenXLA
  - Due to the functional approach, this is just transforming the computational sequence into primitive operations
  - By design supports various hardware platforms due to the compilation to an intermediate representation
- Automatic vectorization of functions
  - vmap: https://jax.readthedocs.io/en/latest/automatic-vectorization.html
  - pmap for multi-process computations https://jax.readthedocs.io/en/latest/multi_process.html



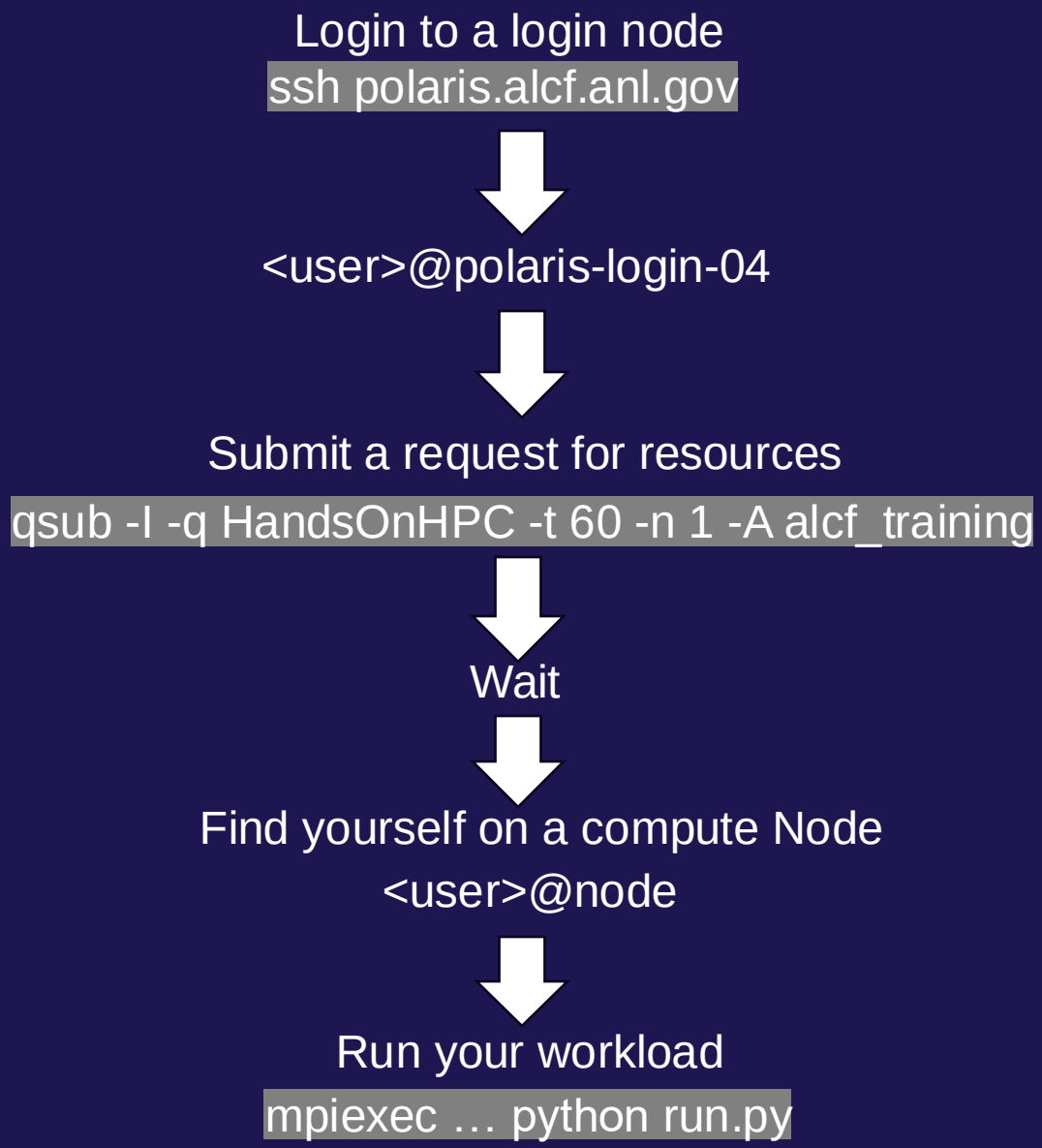
Demo to the intel dev cloud instance

- https://console.cloud.intel.com/docs/guides/get_started.html#launch-instance
- Launch and log in to the console: https://console.cloud.intel.com/
- Learning -> AI with Max Series GPU -> PyTorch on Intel® GPUs -> launch
- The notebook will walk through all the necessary steps to train a model with Intel GPUs
- It is possible to start another notebook from scratch, use the PyTorch 2.5 kernel (as of 10/2024)

Frameworks at ALCF

Reminder: On the ALCF systems you log in into a some kind of a login/entry node and you need to navigate to a compute node to have GPUs!!

- Via the batch job system on Polaris/Aurora/Sunspot (and most if not all others)
- https://docs.alcf.anl.gov/running-jobs/job-and-queue-scheduling/
- Sometimes a GPU is a requirement to build certain packages



Frameworks at ALCF

Polaris

- all three frameworks in the Conda module:
  module use /soft/modulefiles
  module load conda
  conda activate
- https://docs.alcf.anl.gov/polaris/data-science-workflows/frameworks/pytorch/
- The module is automatically setting various environment variables: https://docs.alcf.anl.gov/polaris/data-science-workflows/frameworks/pytorch/#multi-gpu-multi-node-scale-up
  - If you do a manual install without the module you need to set them yourself

Frameworks at ALCF

Aurora/Sunspot

- frameworks installed in frameworks module
  `module load frameworks`
  - Activates the conda environment automatically
- Before torch 2.5 is fully adapted, Intel extension for PyTorch is required (Currently PyTorch 2.3 on Aurora)
  ```
  import torch
  import intel_extension_for_pytorch as ipex
  ```
- Unfortunately no Jax on Aurora as of 10/2024
  - Intel has support for OpenXLA: https://github.com/intel/intel-extension-for-openxla
- Multiple environment variables for the distributed comms are automatically set by the frameworks module
  - Best practices constantly evolving so not documented here

Polaris demo here

- JupyterHub: https://docs.alcf.anl.gov/services/jupyter-hub/
  - Login: https://jupyter.alcf.anl.gov/
- Getting an interactive session on the reservation:
  qsub -I -l select=1 -l filesystems=home:eagle -l walltime=1:00:00 -q HandsOnHPC -A alcf_training
- To load the modules on Polaris:
  module use /soft/modulefiles
  module load conda
  conda activate
- Bash job submission:
  qsub submission_script.sh
- For Aurora/Sunspot this would be fairly similar but the queue names and available storage systems can differ slightly

“Advanced” topics, performance considerations

- Extending Python environments with your own packages
- Mixed precision
- Distributed training for multi-GPU/multi-node training
- And much more in the docs:
  - https://docs.alcf.anl.gov/polaris/data-science-workflows/frameworks/pytorch/#deepspeed

Extending Python installs

Not every package is installed in the framework modules, you likely need to extend the environments to fit your needs

- You can extend the conda environment with a python virtual environment
  - python -m venv --system-site-packages /path/to/virtualenv
    source /path/to/virtualenv/bin/activate
- Install packages to another location with pip
  - Not recommended
  - You could use '--user' when installing packages from pip or building from source
  - This defaults to the home directory, which is not ideal location due to performance and potential for conflicts and easy to mess it up. If different clusters share the home folder you can end up with binaries compiled to a wrong architecture/compiler/setup
  - You can set PYTHONUSERPATH environment variable to control where the '--user'–installed packages end up
- You can just install everything from scratch yourself
  - Start with conda and get going
  - Can be tricky to get everything right, don't start with this one. Cuda version of PyTorch does not work on Aurora and so on.

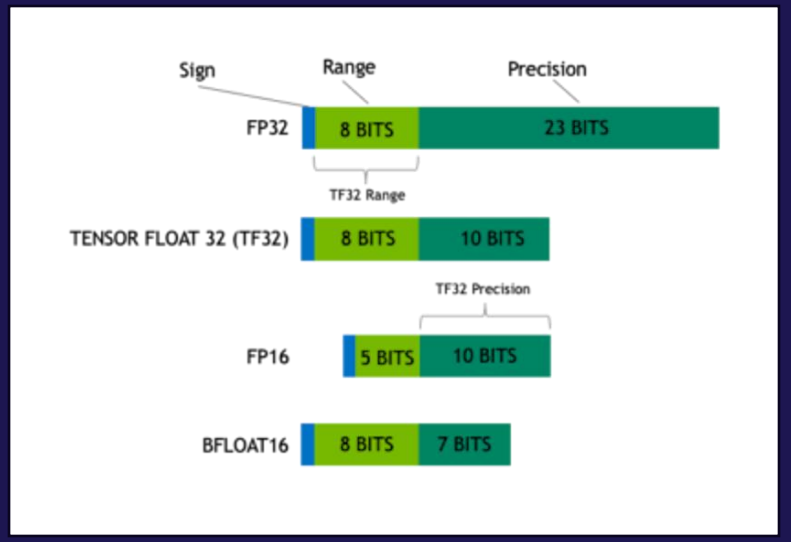
Mixed precision

- Modern GPUs have significantly better BF16/FP16/TF32 throughput compared to traditional single precision floating point (FP32)
  - Not all parts of deep learning require the full float precision to get good results -> significant potential for speedup with negligible loss of predictive performance
- FP16 might not have enough range for loss/gradients
  - requires loss scaling
  - largely superseded by BF16 in deep learning applications as BF16 does not require loss scaling
- Pure FP16/BF16 rarely used in practice as certain operations require still better to be stored in FP32
  - Mixed precision required
  - Gradient accumulation
  - Holding a set of master weights

Specifications

	A100 80GB PCIe	A1
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	

https://www.nvidia.com/en-us/data-center/a100/



https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/

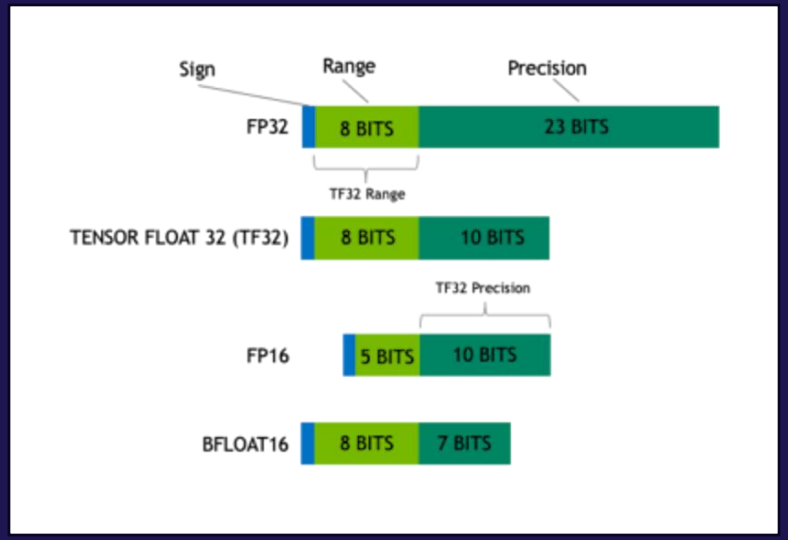
Mixed precision

- PyTorch,TensorFlow can take care of things automatically:
  - PT: https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html with torch.autocast(device_type=device, dtype=torch.float16)
  - TF: https://www.tensorflow.org/guide/mixed_precision
    from tensorflow.keras import mixed_precision
    policy = mixed_precision.Policy('mixed_float16')
    mixed_precision.set_global_policy(policy)
- Frameworks like Megatron-DeepSpeed
  - Adjust Deepspeed config.json
    - https://www.deepspeed.ai/docs/config-json/
  - Enable in Megatron with --bf16 flag

Specifications

	A100 80GB PCIe	A1(
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	

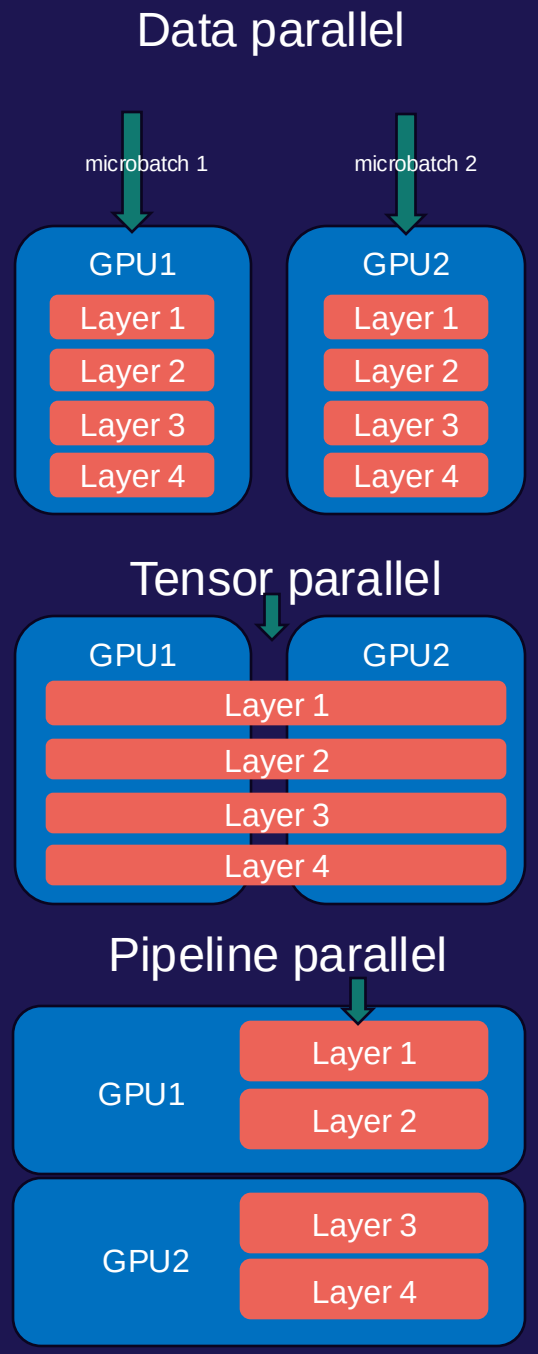
https://www.nvidia.com/en-us/data-center/a100/



https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/

Quick intro to distributed training

- More GPU == more better?
  - Yes, but splitting a problem across multiple devices can require additional attention
  - The communication groups and such need to be prepared appropriately
- Data Parallelism
  - Having multiple copies of the same model operate on different data samples
  - Only communication strictly required is synchronizing the gradient updates before applying them
- Model parallelism
  - Splitting up a single model instance across multiple devices
  - Tensor parallelism, Pipeline parallelism etc.
  - Details not covered in this talk
- When doing parallelism, your batch size grows with more GPUs
  - global_batch_size=micro_batch_size * gradient_accumulation_steps*data_parallel_degree/model_parallel_degree
  - “GBS=MBS*GAS*DP/MP”
  - For example, 2 nodes of 4 GPUs, a microbatch of 2 with gradient accumulation of 2, no model parallelism: GBS of 16==2*4*2*2
- Running at scale, you might also need to care about various things such as I/O, discussed in separate sessions



Distributed setup at ALCF

- Setting up the communication world and ranks is required
  - MPI is a good backed for discovery (on the right).
  - You could also script your own approach based on hostfiles generated by the scheduler
- In the script or in the code define the main process hostname and communication port. For example these env variables are automatically accessed by torch distributed initialization:
  export MASTER_ADDR=$(hostname)
  export MASTER_PORT=29401
- You could also use MPI and sockets library to figure the master address and port and broadcast them
- In the code, set the device and initialize torch distributed.
- alternative is to use torchrun:
  https://pytorch.org/docs/stable/elastic/run.html
  (Also needs some information like above)

```
from mpi4py import MPI
rank = int(MPI.COMM_WORLD.Get_rank())
world_size = int(MPI.COMM_WORLD.Get_size())
device_count = torch.cuda.device_count()
local_rank = rank % device_count
torch.cuda.set_device(local_rank)

torch.distributed.init_process_group(
    backend="nccl",
    world_size=world_size,
    rank=rank,
)
```

For more information

- ALCF docs: https://docs.alcf.anl.gov/
- Documentation for various frameworks
- support@alcf.anl.gov
- Ask at the workshop


Thank you!