Data Pipelines

Huihuo Zheng
Argonne Leadership Computing Facility

October 8, 2025

huihuo.zheng@anl.gov

Learning goals

Presentation

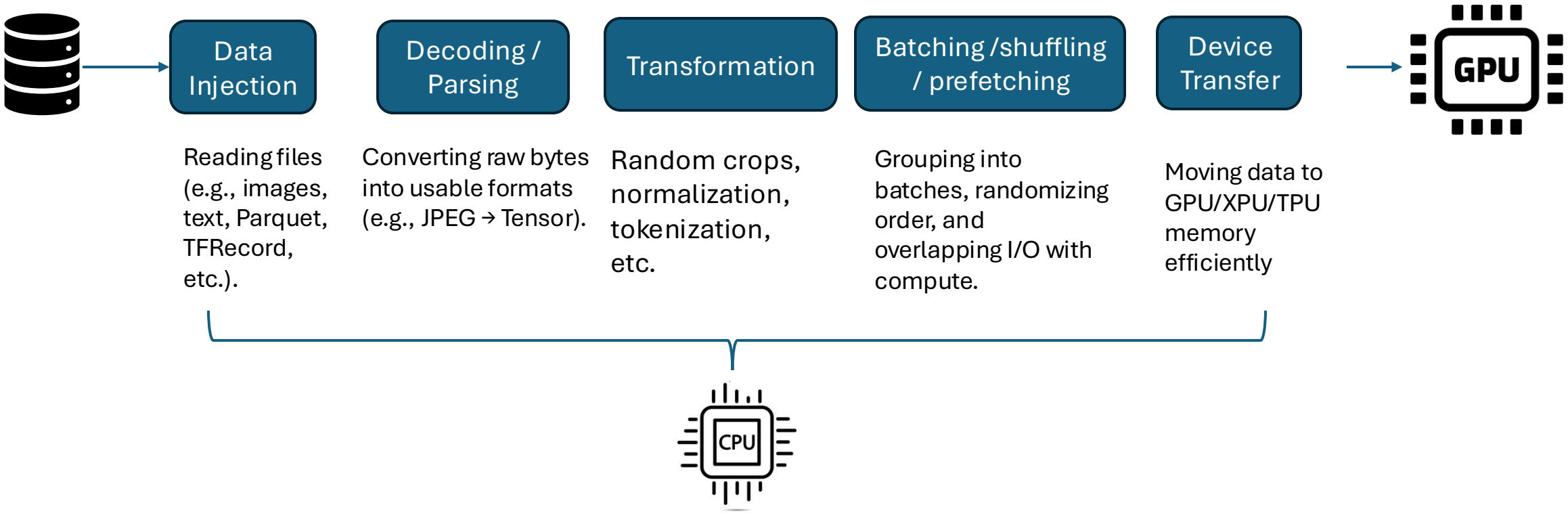
- Understanding data pipelines and know available packages
- Know how to optimize data loading process

Hands on exercise

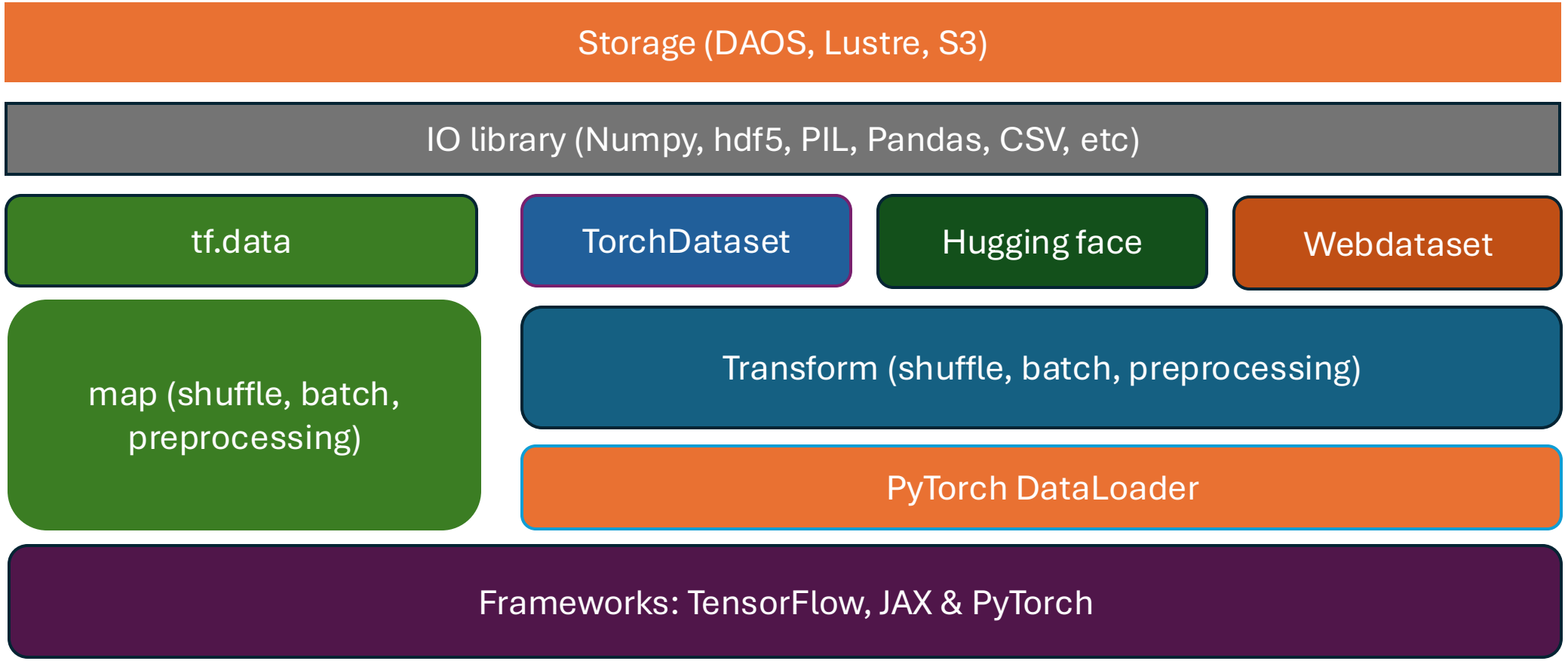
- Using the CPU on a system to build data batches in parallel as ML calculations are performed on the GPU
- Using parallel processes on CPU to speed up the data pipeline process.
- Do all this using the framework's (PyTorch, Tensorflow) data APIs

What is Data Pipeline

A **data pipeline** is the sequence of operations that takes raw data (from disk, remote storage, or memory) and transforms it into **batches of tensors** ready to feed into a training loop.



Data Pipeline Software Stack



🧠 TensorFlow: tf.data API

python

```python
import tensorflow as tf

ds = (tf.data.TFRecordDataset(filenames)
      .map(parse_fn, num_parallel_calls=tf.data.AUTOTUNE)
      .shuffle(10000)
      .batch(128)
      .prefetch(tf.data.AUTOTUNE))
```

⚠️ **Limitations / Trends:**
Graph-based execution (in TF 2.x eager mode improved it, but still verbose).
Declining adoption in research and open-source ML: PyTorch's DataLoader and Hugging Face datasets dominate.

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

PyTorch: torch.utils.data and Ecosystem

Core abstractions:

- Dataset: defines how to access samples.
- DataLoader: handles batching, multiprocessing, shuffling, pinning, etc.

python Copy

```python
from torch.utils.data import DataLoader, Dataset

class MyDataset(Dataset):
    def __getitem__(self, idx):
        # load one sample
        return x, y
    def __len__(self):
        return len(files)

loader = DataLoader(MyDataset(), batch_size=128,
                    num_workers=8, pin_memory=True, prefetch_factor=2)
```

Customizable

https://docs.pytorch.org/tutorials/beginner/basics/data_tutorial.html

Hugging Face Datasets

python

```python
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding
from torch.utils.data import DataLoader

tok = AutoTokenizer.from_pretrained("gpt2")

ds = load_dataset("c4", "en", streaming=True, split="train")
ds = ds.shuffle(buffer_size=10000, seed=42)

def tokenize(batch):
    return tok(batch["text"], truncation=True, padding=False)

ds = ds.map(tokenize, batched=True, remove_columns=["text"])
ds = ds.with_format("torch", columns=["input_ids", "attention_mask"])

collate = DataCollatorWithPadding(tok, return_tensors="pt")

loader = DataLoader(ds, batch_size=128, collate_fn=collate, num_workers=4)

for epoch in range(3):
    ds.set_epoch(epoch)     # reshuffle buffer per epoch
    for batch in loader:
```

```
pip install datasets
```

I/O: for massive text, consider streaming=True;
for repeated epochs over fixed data, prefer non-streaming + caching.

Can be used in both TensorFlow and PyTorch

https://huggingface.co/docs/datasets/en/quickstart

WebDataset

WebDataset (often imported as webdataset or wds) is a **sharded TAR-based dataset format + PyTorch loader** designed for **streaming, large-scale training**.

python Copy

```python
import torch
import webdataset as wds
from torchvision import transforms

urls = "s3://bucket/ds/shard-{000000..000999}.tar"   # or "https://host/shard-{...}.

transform = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224)])

dataset = (
    wds.WebDataset(urls, shardshuffle=1000)     # shuffle shards each epoch
      .decode("pil")                             # decode images
      .to_tuple("jpg", "txt")                    # map files per sample to a tuple
      .map_tuple(transform, lambda x: x)         # transform image, keep text
      .shuffle(10000)                            # sample-level buffer shuffle
      .batched(128)                              # make batches inside the dataset
)

loader = torch.utils.data.DataLoader(dataset, batch_size=None, num_workers=8)

for images, captions in loader:
    # training step
    pass
```

Preparing the datasets

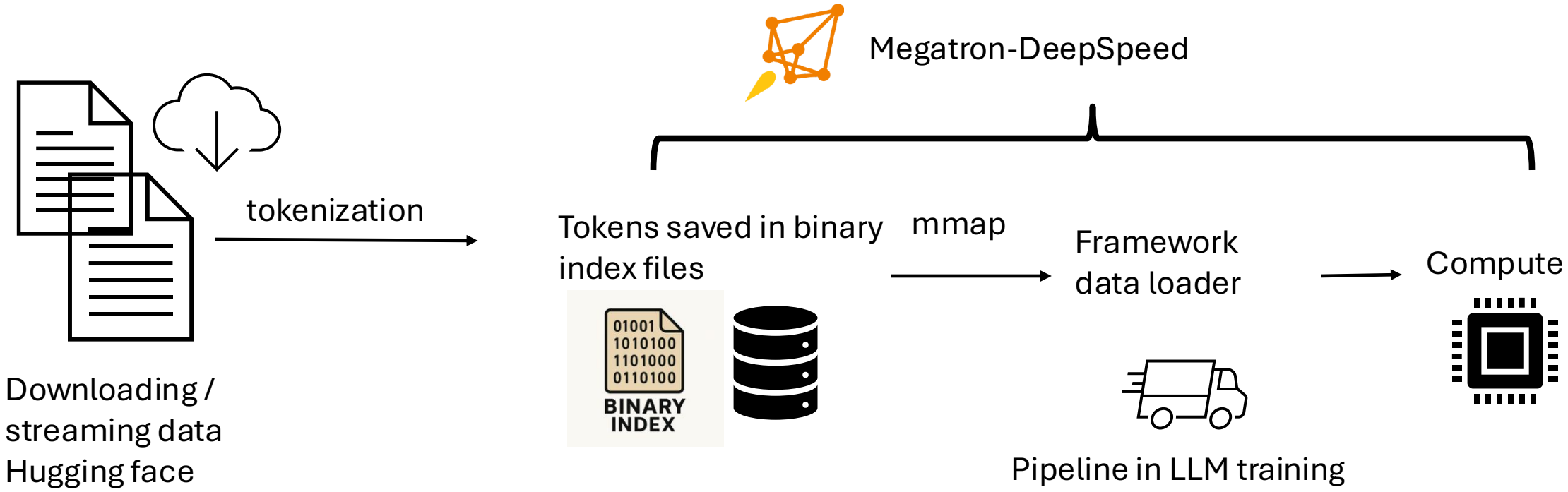
Writer (make shards)

python

```python
import webdataset as wds

with wds.TarWriter("shard-000000.tar") as sink:
    for i, (img_bytes, caption) in enumerate(data_iter):
        sample = {
            "__key__": f"{i:09d}",
            "jpg": img_bytes,          # bytes of an encoded image
            "txt": caption.encode(),   # bytes of caption
        }
        sink.write(sample)
```

https://github.com/webdataset/webdataset

BlendCorpus for Large Language Models Training



BlendCorpus

https://github.com/zhenghh04/blendcorpus

Data Loading Optimization

I/O Patterns: HPC vs AI

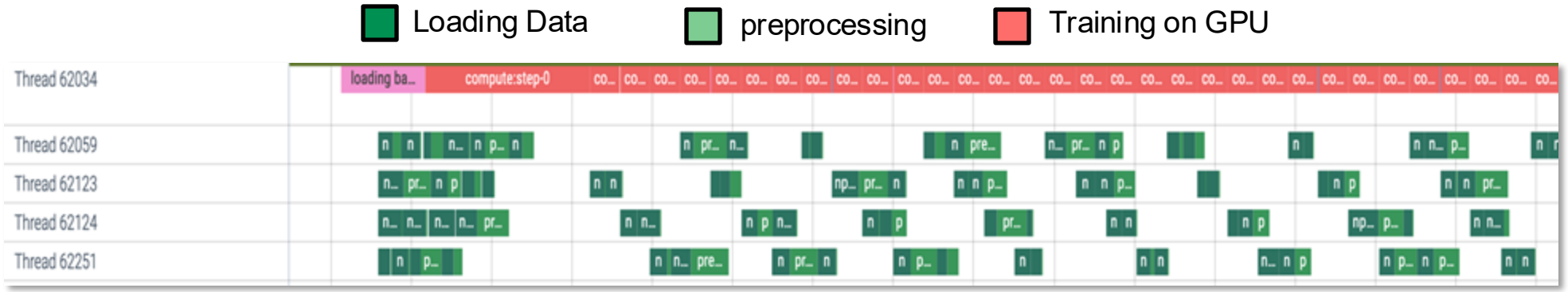
HPC workloads

- Write intensive (checkpointing)
- Bulk parallel I/O
- Multi-dimensional array (binary, HDF5, NetCDF)
- Single thread & sync. I/O
- Global file system + SSD (but SSD are rarely been used)

AI workloads

- Read or write intensive
- Small and sparse I/O & metadata intensive random access
- Complex data format (json, text, key-value)
- Multithreaded async. I/O
- Storage hierarchy + caching & staging
- Variety of needs for different stages: preprocessing, data loading, checkpointing, inferencing

Data Loading Optimization



3D-Unet Data loading Tracing

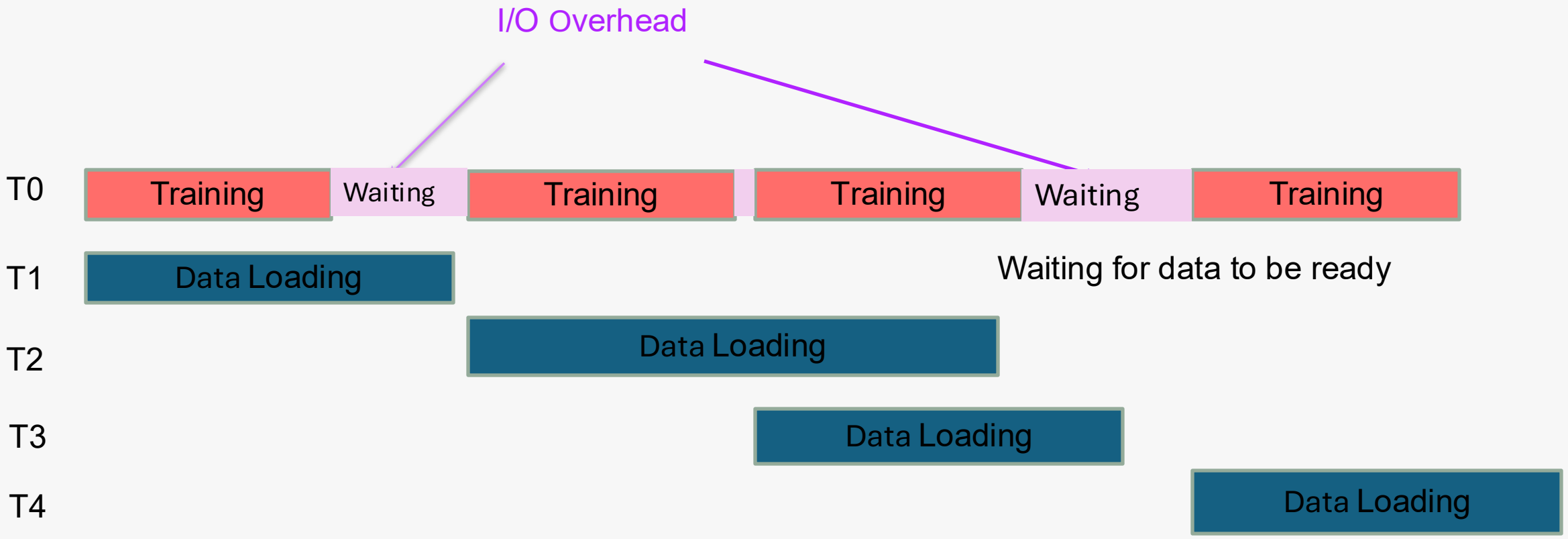
python Copy

```python
from torch.utils.data import DataLoader, Dataset

class MyDataset(Dataset):
    def __getitem__(self, idx):
        # load one sample
        return x, y
    def __len__(self):
        return len(files)

loader = DataLoader(MyDataset(), batch_size=128,
                    num_workers=8, pin_memory=True, prefetch_factor=2)
```

Controlling the number of workers

Data Loading Optimization

$$Throughput = \frac{num\_samples}{Time\ per\ epoch}$$



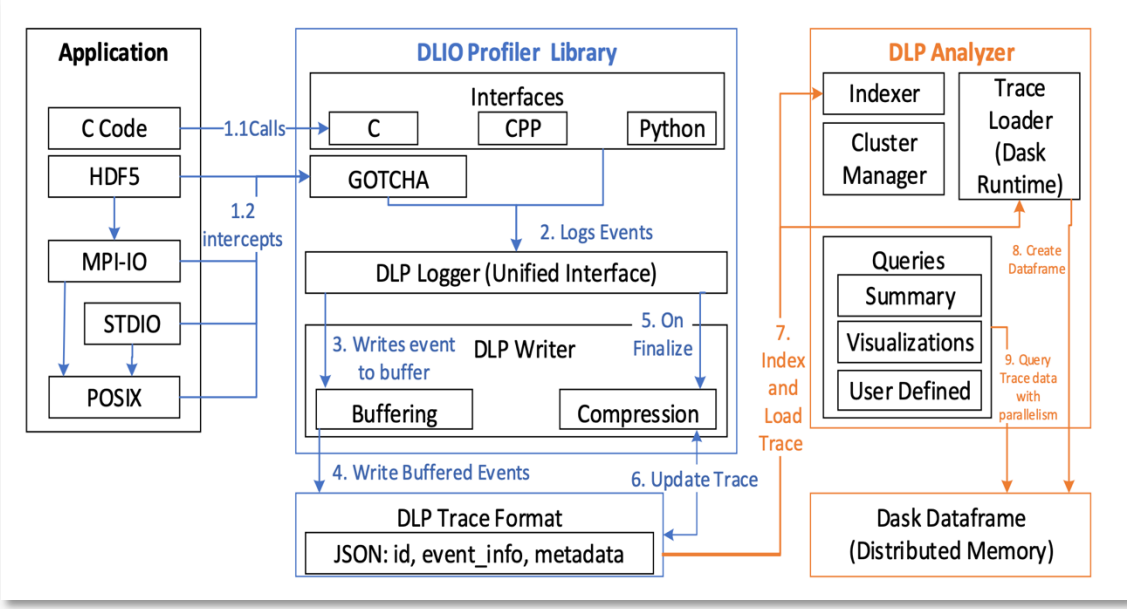
I/O Profiler and Tracer for AI: DFTracer

Diagnose the form, reveal the root, tune the whole.
诊其形，明其本，调其全。

Existing tools are not friendly to AI: darshan, recorder, perf, ioprof

We need an AI I/O profiler / tracer

- Easy to use
- Works with Python applications (multiple threads)
- Able to profile at different levels
- Easy to integrate with other tracing tools
- Easy to visualize the tracing (*Perfetto*)
- Small overhead



https://github.com/LLNL/dftracer

H. Devarajan *et al.*, "DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows," *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA, 2024, pp. 1-24,

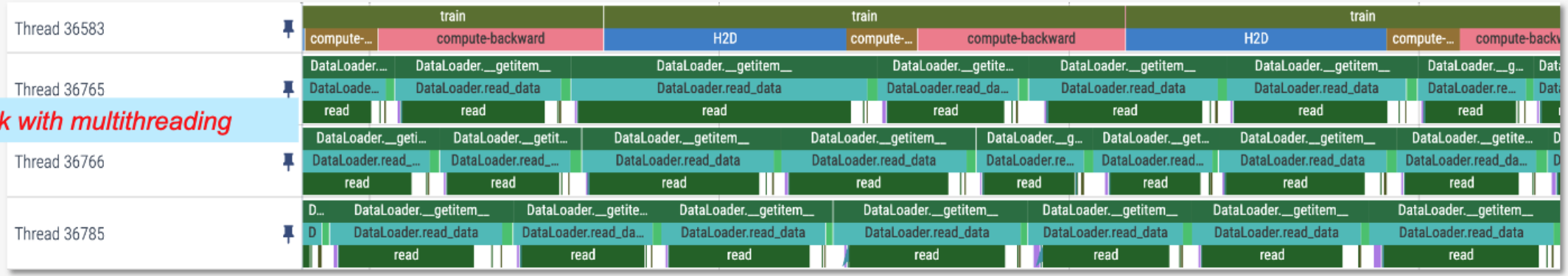
I/O Profiler and Tracer for AI: DFTracer

Easy to use

```python
from dlio_profiler.logger import fn_interceptor as Profile
dlp_data = Profile("DataLoader")
class DataLoader(datasets.ImageFolder):
    @dlp_data.log
    def preprocess(self, sample, target):
        if self.transform is not None:
            sample = self.transform(sample)
        if self.target_transform is not None:
            target = self.target_transform(target)
            return sample, target
    @dlp_data.log
    def read_data(self, index):
        path, target = self.samples[index]
        return self.loader(path), target
    @dlp_data.log
    def __getitem__(self, index):
        sample, target = self.read_data(index)
        sample, target = self.preprocess(sample, target)
        return sample, target
```

Name	Wall duration (ms)
	400698.238
DataLoader.__getitem__	142903.235
DataLoader.read_data	134743.702
read	110562.695
DataLoader.preprocess	7762.795
close	2356.922
open64	2335.804
__fxstat64	22.914
lseek64	10.171

Work with multithreading

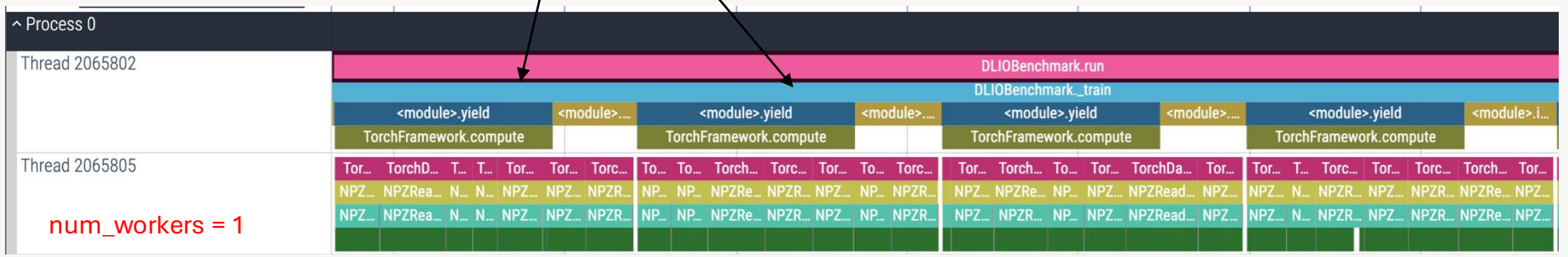


https://github.com/LLNL/dftracer

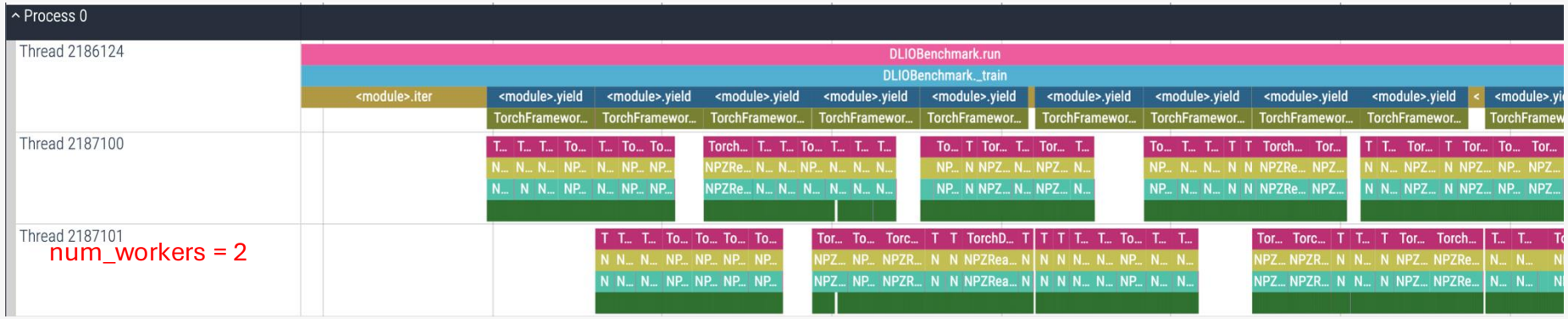
Multithreading improves the I/O throughput

Waiting for data

Sample size: 100–150 MB (NPZ)



With num_workers, IO / compute = 1.3, 30% of the time, GPU is waiting for data

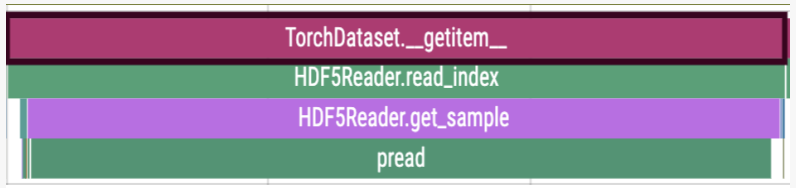


With num_workers = 2, the data loading is completely hidden behind compute

Try different data formats: HDF5 vs NPZ

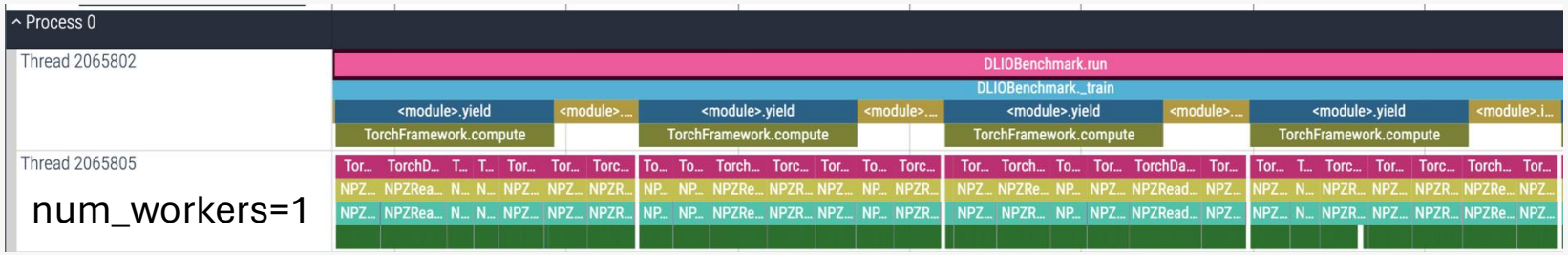
NPZ is inefficient -> changing to HDF5

Single NPZ file is loaded in chunks of 256 kB, which is very inefficient

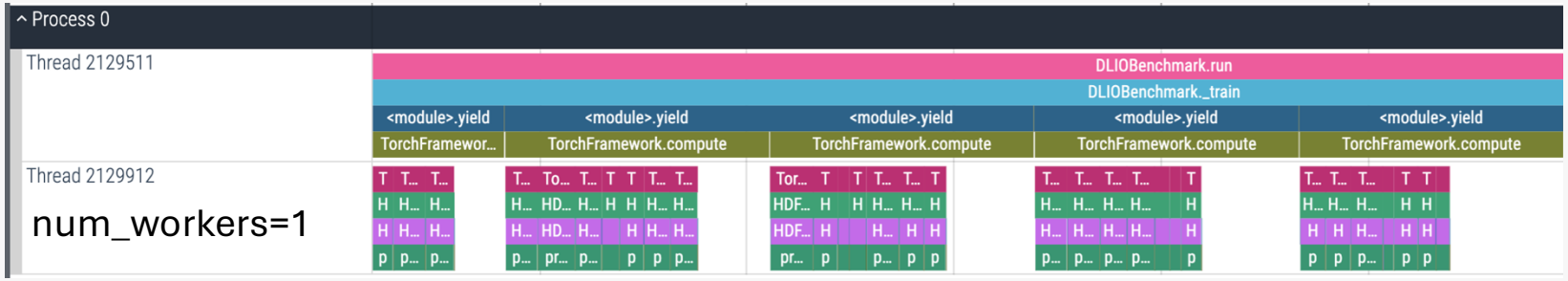


HDF5 is loaded in one chunk (~150MB)

NPZ

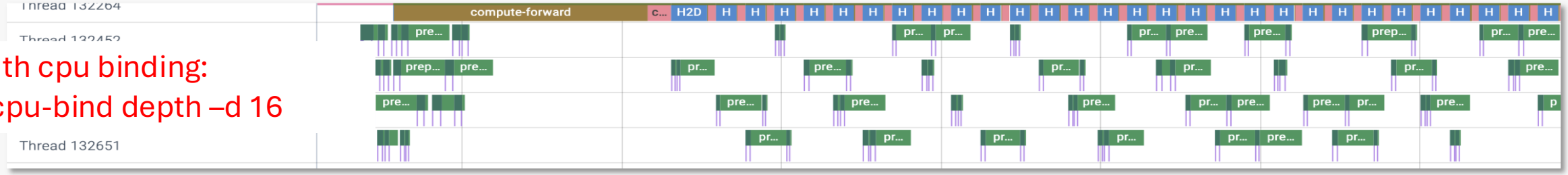


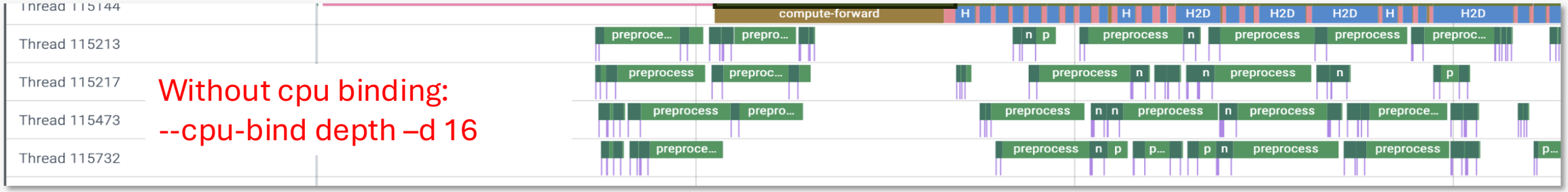
HDF5



CPU Binding

With cpu binding:
--cpu-bind depth –d 16



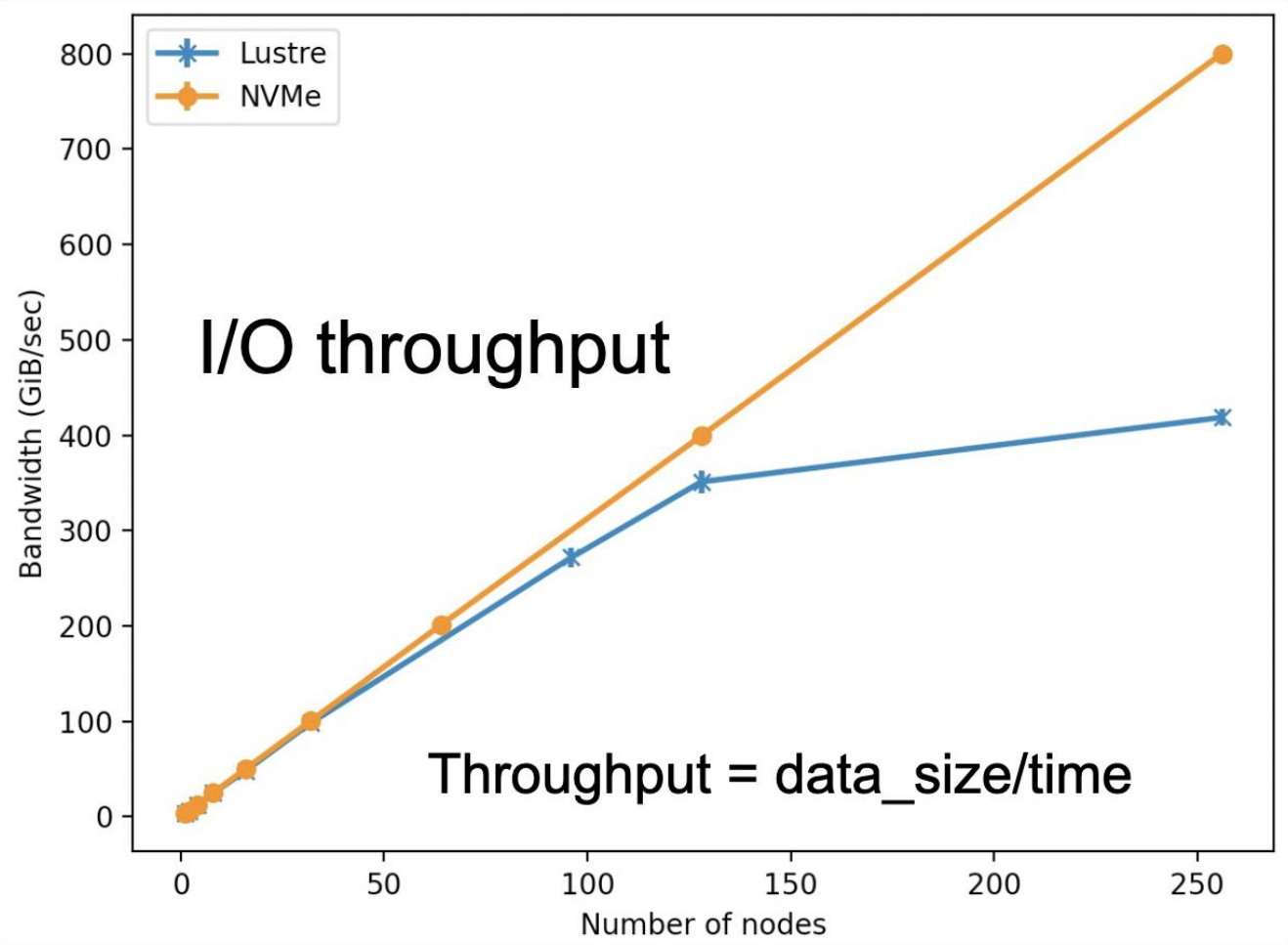


Without cpu binding:
--cpu-bind depth –d 16

CPU binding is crucial for making sure there are enough cores for I/O threads. Otherwise, all the threads will be put on the same CPU core.

	w/ --cc depth –d	w/o --cc depth –d
I/O	6s	30s
Preprocess	19s	89s

Caching to Node Local Storage

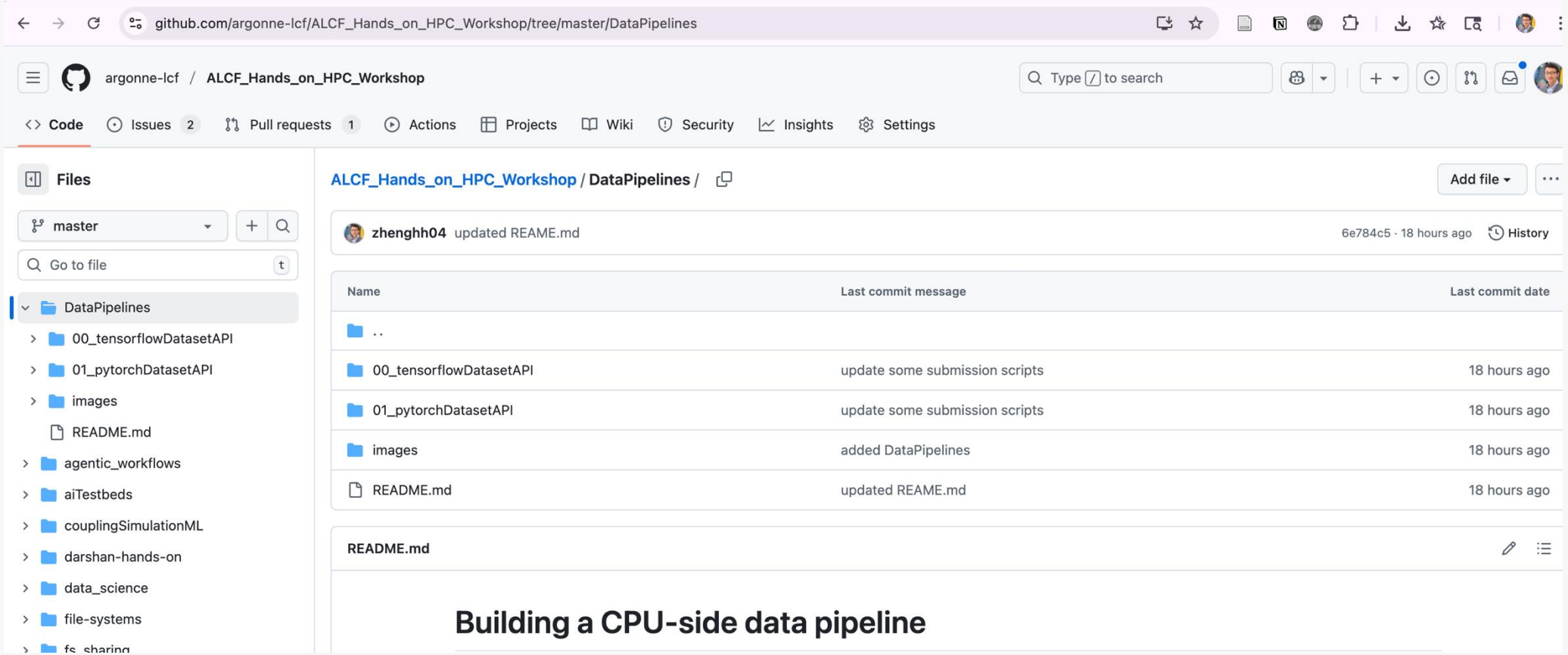


I/O throughput at different scale on Polaris for UNet3D model

Experiences

- Organize data in an optimal way (avoid large number of small files; use binary file formats instead of text files)
- Use multithreading to load data (increase num_workers)
- Set cpu-bind to make sure each MPI task has multiple cpu cores for data loading and preprocessing
- Cach data to NVMe storage if I/O is not hidden behind compute after all optimization

Hands on examples



https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop/tree/master/DataPipelines

Acknowledgements

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.