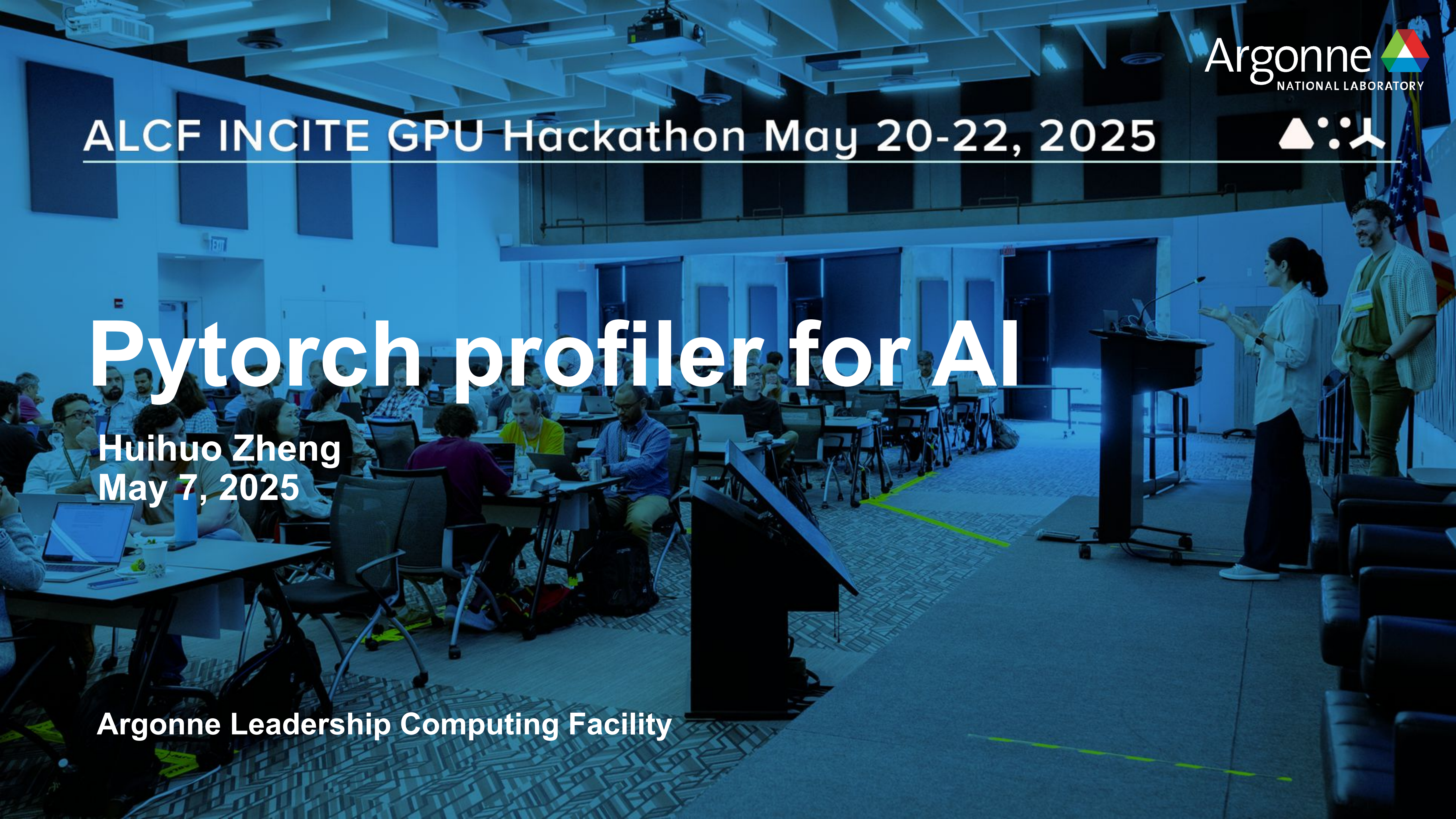


Argonne NATIONAL LABORATORY

ALCF INCITE GPU Hackathon May 20-22, 2025

Pytorch profiler for AI

Huihuo Zheng

May 7, 2025

Argonne Leadership Computing Facility

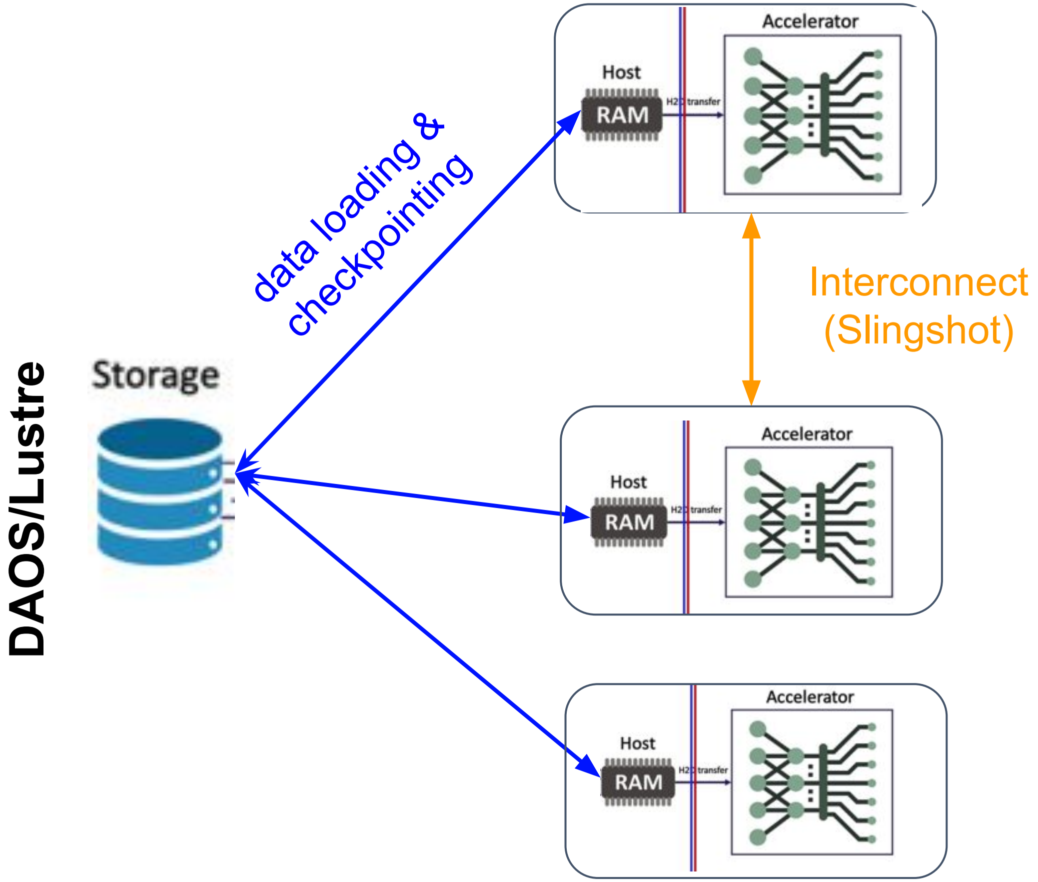
What is profiling and why?

Profiling is the process of analyzing a program's execution to identify performance characteristics and bottlenecks.

- Data loading
- Computation
- Memory access
- Communication.

Option of profilers on Aurora

- TAU
- HPCToolKit
- **Intel VTune/APS/Advisor**
- mpitrace (communication)
- Darshan (I/O)
- **Iprof/unitrace**
- torch.profiler (Comp & comm)
- DFTracer (I/O)



What information can PyTorch profiler provide

- **Operator Execution Time (CPU time & XPU time)**
- **Kernel Execution Details (XPU):** The specific compute kernels launched on the device, including their duration
- ***Operator Input Shapes*:** By setting record_shapes=True
- **Stack Traces and Module Hierarchy:** Enabling with_stack=True allows the profiler to record the Python source code location (file and line number) that invoked each operation.
- **Estimated FLOPs:** For certain common operators like matrix multiplication and 2D convolution, the profiler can estimate the number of floating-point operations (FLOPs) performed if with_flops=True is set.[9] This can help in assessing the computational intensity of different parts of the model.
- **Execution Timeline (Trace View):** Perhaps the most powerful feature for detailed analysis is the ability to export a chronological trace of events.

Example of using PyTorch Profiler on Aurora

```
# loading relevant modules
from torch.profiler import ProfilerActivity, profile, record_function

with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.XPU],
             record_shapes=True,
             profile_memory=True,
             with_stack=True)  as prof:

        with record_function(“data_preprocessing”): #user custom annotation
              …..

        # portion of the code you would like to
        train(model, loader, epochs=args.epochs, steps_per_epoch = args.steps)

# print function statistics
if rank == 0:
        print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=50))

# output timeline trace  (important to have different files for different rank)
os.makedirs(args.trace_dir, exist_ok=True)
prof.export_chrome_trace(f"{args.trace_dir}/torch-trace-{rank}-of-{world_size}.jso
n")
```

CLASS torch.profiler._KinetoProfile(*, *activities=None*, *record_shapes=False*, *profile_memory=False*, *with_stack=False*, *with_flops=False*, *with_modules=False*, *experimental_config=None*, *execution_trace_observer=None*, *acc_events=False*, *custom_trace_id_callback=None*) [SOURCE]

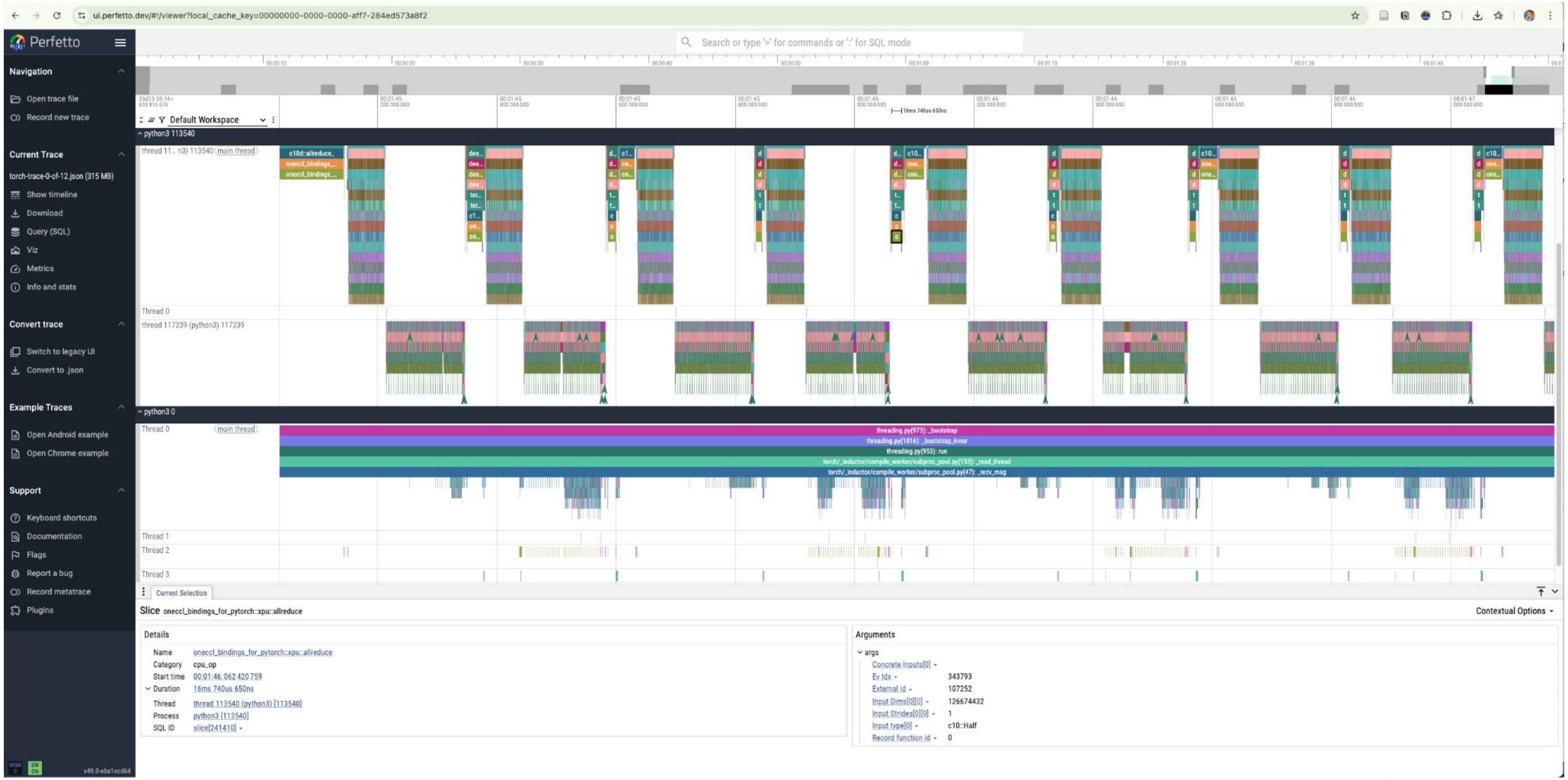
Low-level profiler wrap the autograd profile

Parameters

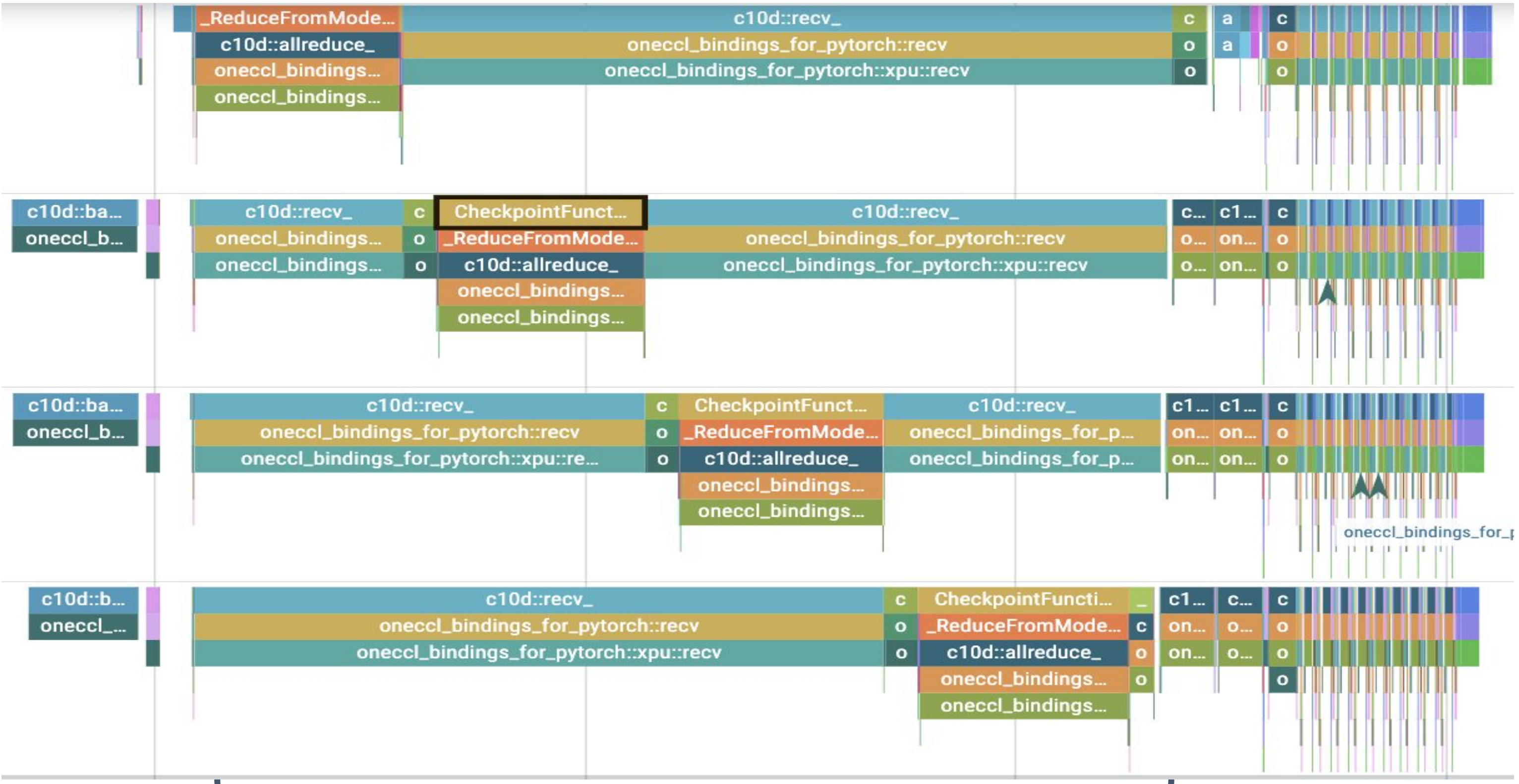
- **activities** (*iterable*) – list of activity groups (CPU, CUDA) to use in profiling, supported values: `torch.profiler.ProfilerActivity.CPU`, `torch.profiler.ProfilerActivity.CUDA`, `torch.profiler.ProfilerActivity.XPU`. Default value: ProfilerActivity.CPU and (when available) ProfilerActivity.CUDA or (when available) ProfilerActivity.XPU.
- **record_shapes** (*bool*) – save information about operator’s input shapes.
- **profile_memory** (*bool*) – track tensor memory allocation/deallocation (see `export_memory_timeline` for more details).
- **with_stack** (*bool*) – record source information (file and line number) for the ops.
- **with_flops** (*bool*) – use formula to estimate the FLOPS of specific operators (matrix multiplication and 2D convolution).
- **with_modules** (*bool*) – record module hierarchy (including function names) corresponding to the callstack of the op. e.g. If module A’s forward call’s module B’s forward which contains an aten::add op, then aten::add’s module hierarchy is A.B Note that this support exist, at the moment, only for TorchScript models and not eager mode models.
- **experimental_config** (*_ExperimentalConfig*) – A set of experimental options used by profiler libraries like Kineto. Note, backward compatibility is not guaranteed.
- **execution_trace_observer** (*ExecutionTraceObserver*) – A PyTorch Execution Trace Observer object. PyTorch Execution Traces offer a graph based representation of AI/ML workloads and enable replay benchmarks, simulators, and emulators. When this argument is included the observer start() and stop() will be called for the same time window as PyTorch profiler.
- **acc_events** (*bool*) – Enable the accumulation of FunctionEvents across multiple profiling cycles

https://docs.pytorch.org/docs/stable/profiler.html

Timeline trace visualization: https://ui.perfetto.dev/



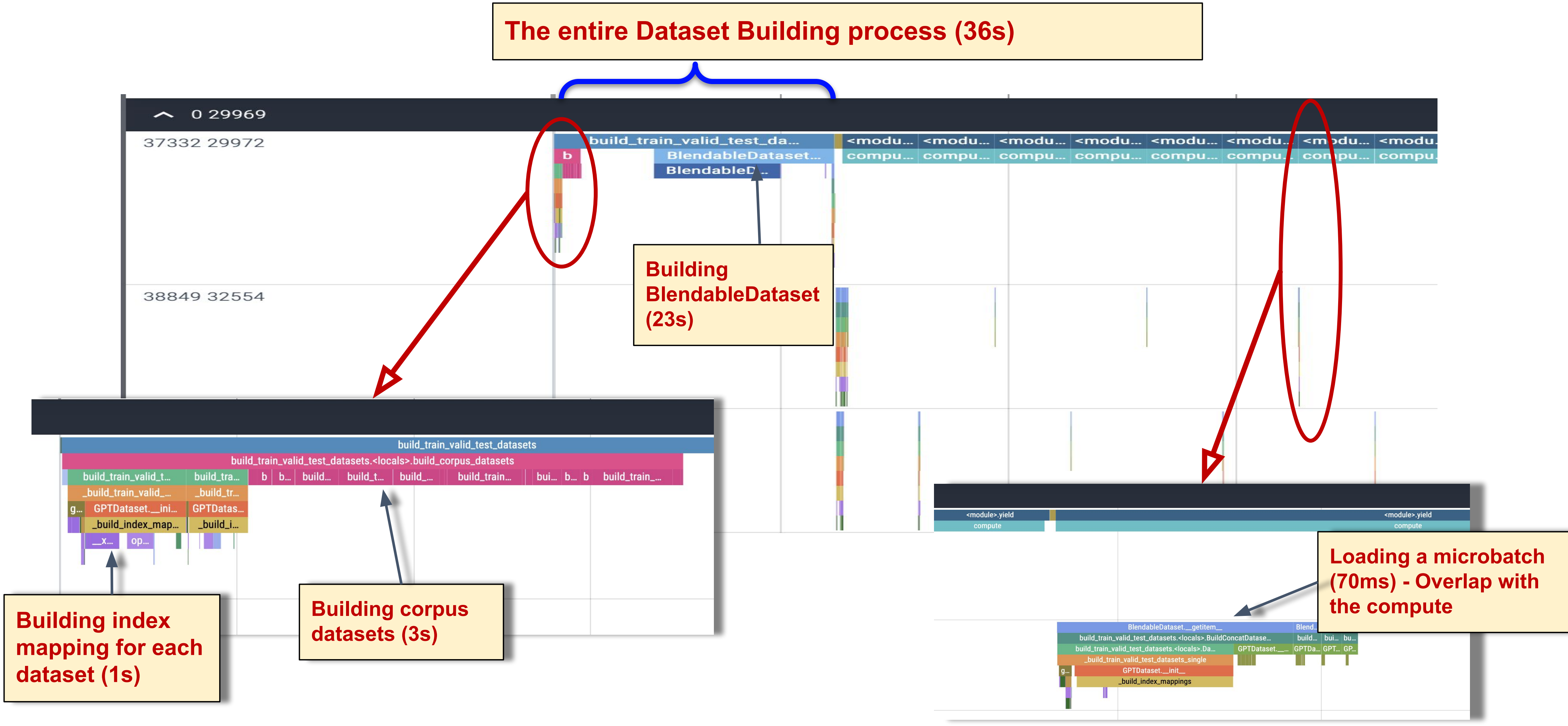
Case 1: First step large overhead in pipeline parallelism



First step cost is primarily due to consecutive initializing GPU kernels from one stage to another

Argonne NATIONAL LABORATORY

Case 2: I/O in Megatron-DeepSpeed



Existing issues for torch.profiler on Aurora

When profiling on Intel XPU backends, enabling ProfilerActivity.XPU can occasionally lead to hangs in multi-process or multi-node runs. To avoid this issue:

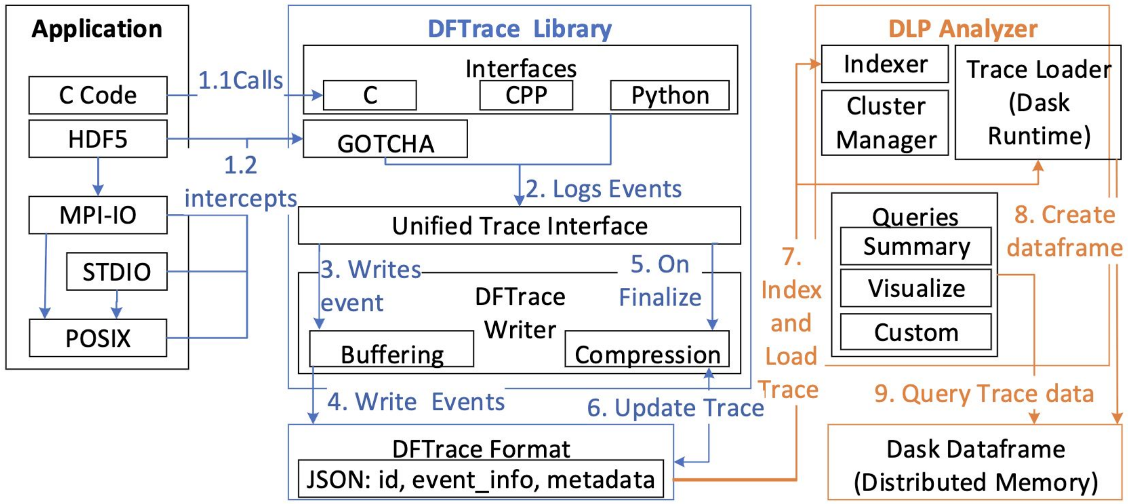
- **Single-Node Profiling:** Use activities=[ProfilerActivity.CPU, ProfilerActivity.XPU] and collect a full trace on **one** node (or a single process per node) to capture XPU kernel timings safely without multi-node synchronization issues .

- **Multi-Node Profiling:** For distributed workloads spanning multiple nodes, omit ProfilerActivity.XPU and rely solely on CPU events (ProfilerActivity.CPU) to prevent hanging barriers in the Kineto profiler’s XPU hooks .

- ./test_dtensor_1d.py works fine for multiple node with ProfilerActivity.XPU
- ./test_miniGPT.py hangs on 2+ nodes

DFTracer - A multilevel I/O profiler with application context

- Works with Python applications (multiple threads)
- Able to profile I/O in the context of application functions
- Able combine with results from other tools
- With small overhead



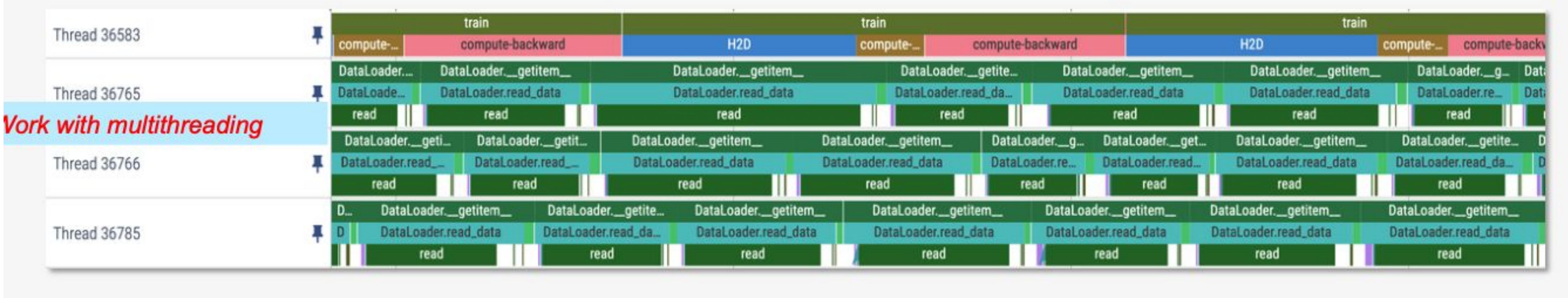
https://github.com/LLNL/dftracer.git

DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows.
Devarajan, H., L. Pottier, K. Velusamy, H. Zheng, I. Yildirim, O. Kogiou, W. Yu, A. Kougakas, X.-H. Sun, J. S. Yeom, and K. Mohror, SC24: International Conference for High Performance Computing, Networking, Storage and Analysis

Multilevel profiling feature in dftracer

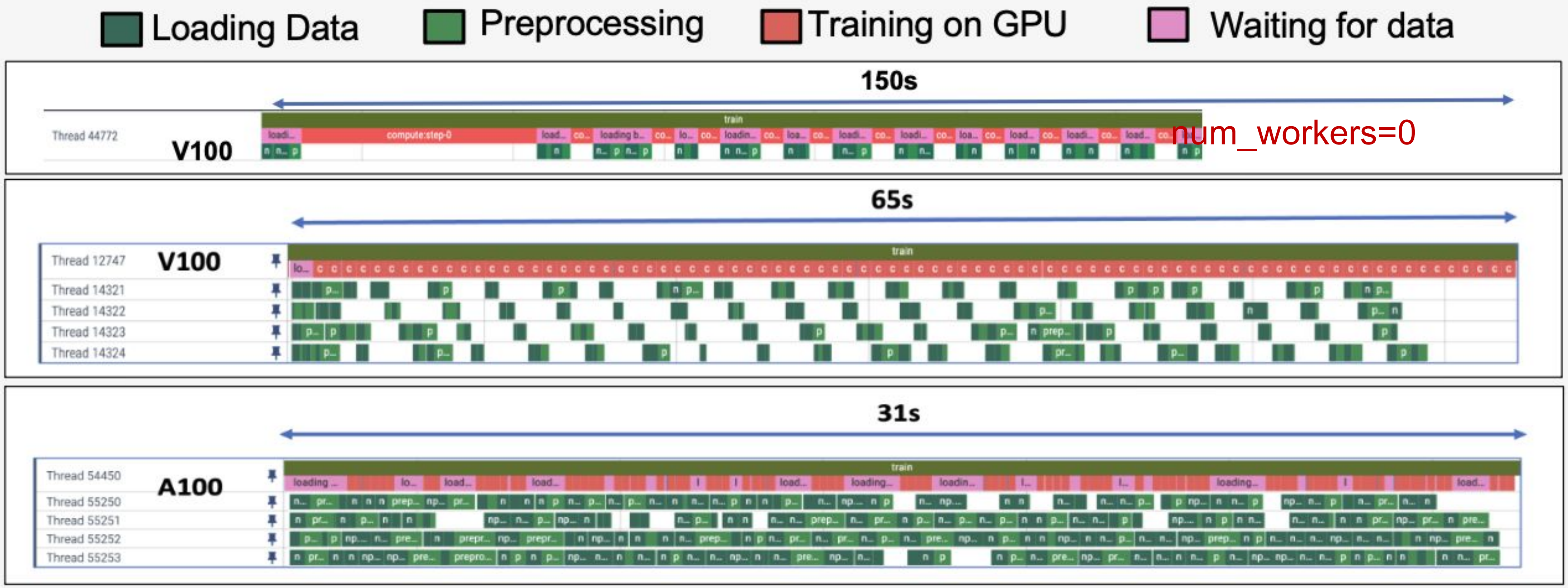
```python
from dftracer.logger import dft_fn as Profile
dlp_data = Profile("DataLoader")
class DataLoader(datasets.ImageFolder):
    @dlp_data.log
    def preprocess(self, sample, target):
        if self.transform is not None:
        Easy to use
        sample = self.transform(sample)
        if self.target_transform 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)
        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



Argonne NATIONAL LABORATORY

I/O bottleneck for UNet3D shown through dftracer



Timeline trace for training the UNet3D workload on a single GPU on JLSE@ALCF.

Performance issue revealed through profiling

PyTorch

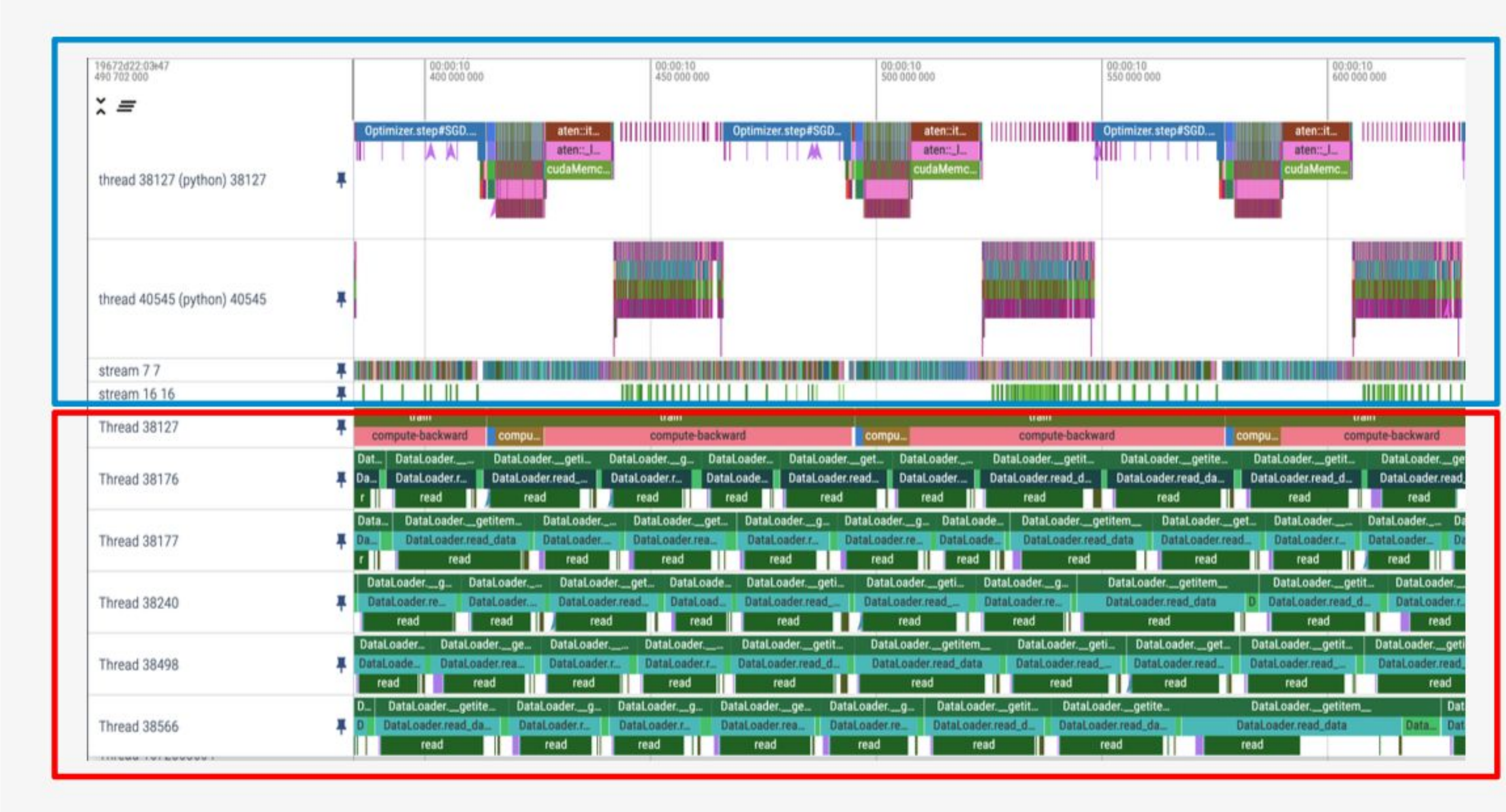
Name	Wall duration (ms)
	20049.987
TorchDataset.__getitem__	5042.087
HDF5Reader.read_index	5030.107
HDF5Reader.get_sample	4202.46
pread	4066.861

TensorFlow

Name	Wall duration (ms)
	17441.186
HDF5Reader.get_sample	7735.878
HDF5Reader.open	3764.007
pread	3134.925
HDF5Reader.close	2466.615

Performance comparison of Torch DataLoader and tf.data for HDF5 datasets. TensorFlow data loader performs worse for HDF5 because of thread locking issue

Combining data loading trace with compute trace



torch.profiler trace

dftracer trace

Argonne NATIONAL LABORATORY

Current existing issues for dftracer on Aurora

⚠️ **Aurora-specific workaround:** The POSIX-level I/O interceptor in DFTracer can hang on Aurora's file system. If you encounter hangs during low-level POSIX tracing, disable it by setting:

```
export DFTRACER_DISABLE_IO=1
```

This turns off the problematic POSIX I/O hooks while still allowing higher-level function and MPI-IO tracing to proceed normally.

Merging multiple timeline trace

Install utils

```
pip install git+https://github.com/zhenghh04/pyutils
```

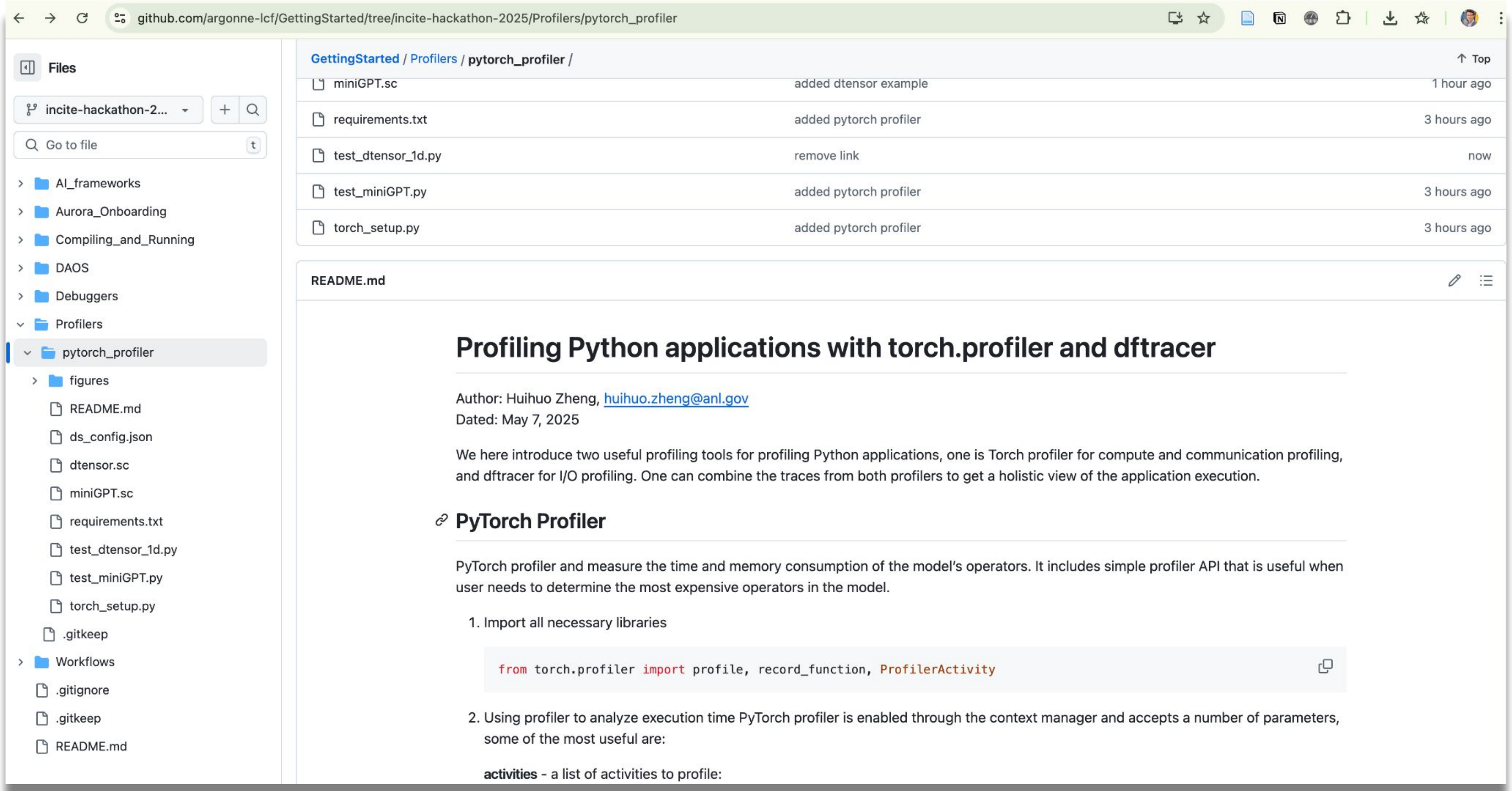
Merging traces from different ranks

```
merge_trace --inputs ./torch-trace-0-of-24.json ./torch-trace-12-of-24.json –output torch-trace-combine.json
```

Merging traces from different profilers (alignment might be needed)

merge_trace –inputs ./torch-trace-0-of-24.json ./trace-0-of-24.pfw –output trace-0-of-24-combine.json

Hands on examples



git clone -b incite-hackathon-2025 https://github.com/argonne-lcf/GettingStarted

Acknowledgments

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.