Neural Operators for Predicting Time Series

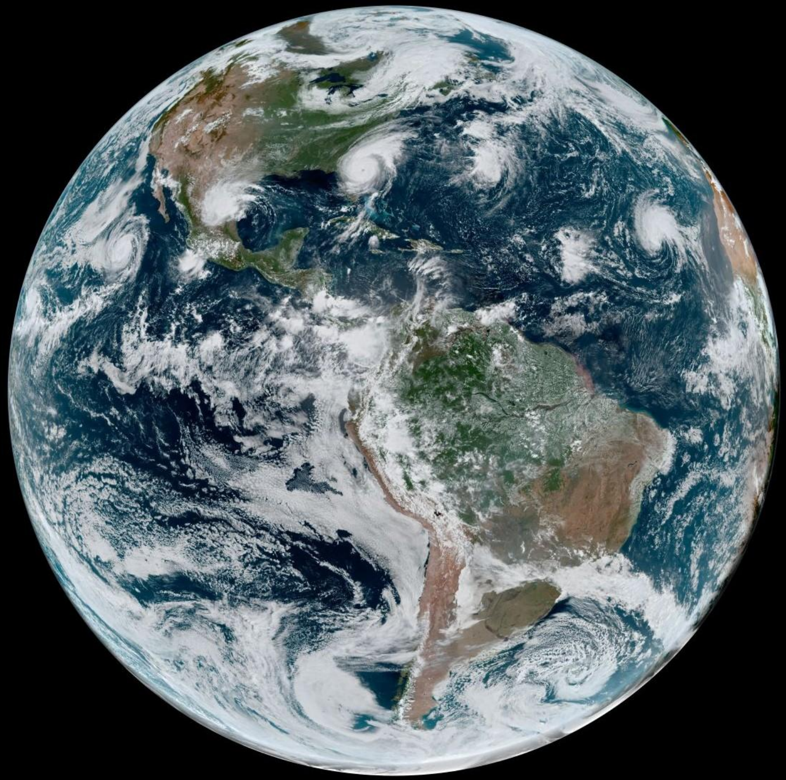
Presenter: Marius Kurz
Oct 13-16, 2025
AMD @ CASTIEL AI Workshop

AMD
together we advance_

Predicting Time Series

A central task in computational science is to predict how a physical system will evolve, or

"Given the current state of the system, what is its state at some later point in time?"



Predicting Time Series

A central task in computational science is to predict how a physical system will evolve, or

"Given the current state of the system, what is its state at some later point in time?"

$$\boxed{U(x, t_0 + \Delta t) = R(U(x, t_0))}$$

Predicting Time Series

A central task in computational science is to predict how a physical system will evolve, or

"Given the current state of the system, what is its state at some later point in time?"

$$U(x, t_0 + \Delta t) = \boxed{R}(U(x, t_0))$$

The "dynamics" of the system

- Possible Difficulties
  - Multi-scale
  - Chaotic
  - Exponential error growth

Predicting Time Series

A central task in computational science is to predict how a physical system will evolve, or

"Given the current state of the system, what is its state at some later point in time?"

$$U(x, t_0 + \Delta t) = \boxed{R}(U(x, t_0))$$

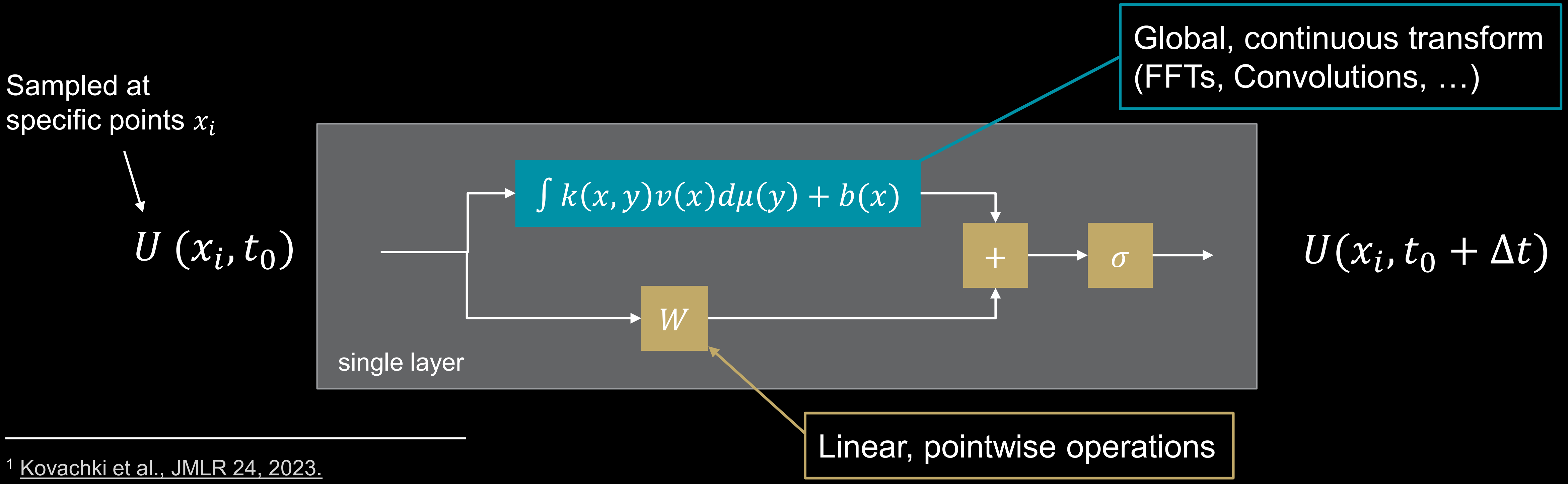
The "dynamics" of the system

- Numerical Simulations
  - Rigorous guarantees
  - Highly accurate
  - (Prohibitively) Expensive

- Artificial Neural Networks
  - Cheaper to evaluate (once trained)
  - Hard to get guarantees
  - Transferability across cases difficult
  - Limited by available training data

Neural Operators – A Brief Introduction

- Artificial Neural Networks typically work on discretized data (images with pixels, tokens in sequence)
- Discretization-dependency can limit the models’ usefulness
- Neural Operators[1]: Learn to map from continuous functions to continuous functions



[1] Kovachki et al., JMLR 24, 2023.

Scientific Datasets and Benchmarks

- Much progress in AI was fueled by established benchmarks and contests within scientific communities
  - Computer Vision: MNIST, CIFAR, ImageNet
  - Natural Language Processing: Open LLM Leaderboard, BookCorpus, WMT 2014 English → German
  - AlphaFold: CASP

Scientific Datasets and Benchmarks

- Much progress in AI was fueled by established benchmarks and contests within scientific communities
  - Computer Vision: MNIST, CIFAR, ImageNet
  - Natural Language Processing: Open LLM Leaderboard, BookCorpus, WMT 2014 English → German
  - AlphaFold: CASP

- Many scientific datasets:
  - are multi-dimensional: 2D, 3D
  - describe transient processes: additional time dimension
  - are large: millions of data points per sample, up to several Terabytes
  - exhibit differently structured input/output data: different simulation methods, applications, …

→ Establishing and sharing scientific datasets can be very difficult!

Two Exemplary Datasets for Scientific Simulations

The Well

- 2D and 3D cases from various fields of computational science
- Reference results for variants of FNO and U-Net models

References:

- Paper: Ohana et al., NeurIPS, 2024.
- Github: https://github.com/PolymathicAI/the_well

PDEBench

- Different 1D, 2D and 3D testcases with focus on PDEs prevalent in fluid dynamics
- Provides baseline results for FNO, U-Net and PINN models

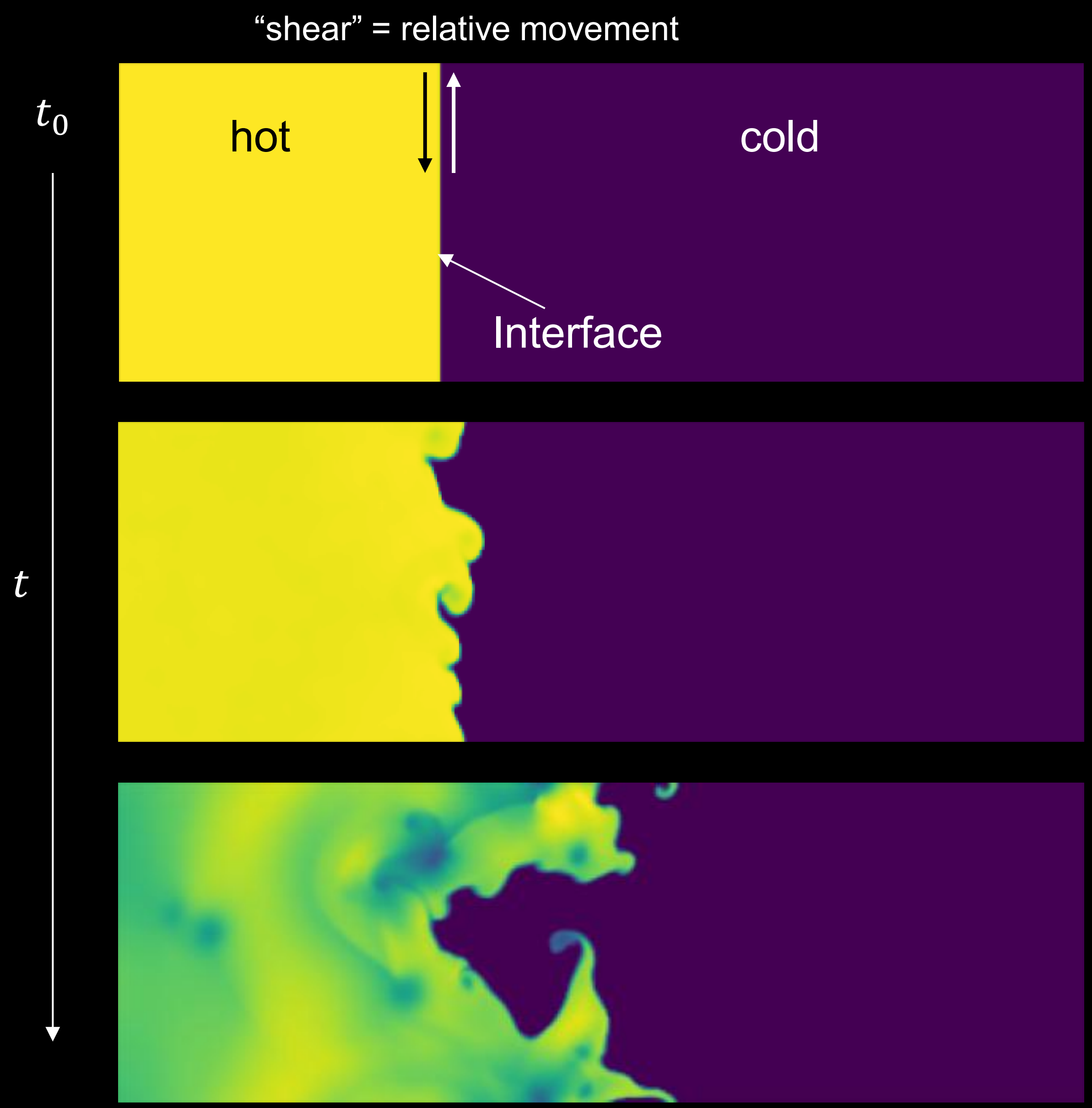
References:

- Paper: Takamoto et al., NeurIPS, 2022.
- Github: https://github.com/pdebench/PDEBench

Let's pick an example!

`turbulent_radiative_layer_2D` from The Well[1] dataset:

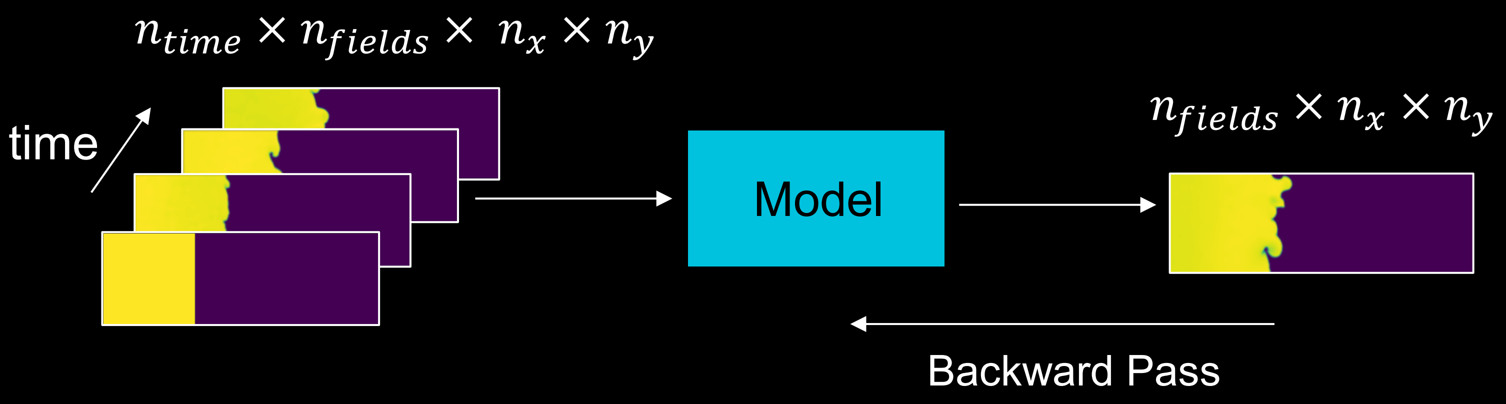
- Shear flow between hot and cold fluid phases
- Compressible Navier-Stokes equations describe temporal evolution
- Radiative cooling modeled through a source term in the energy equation



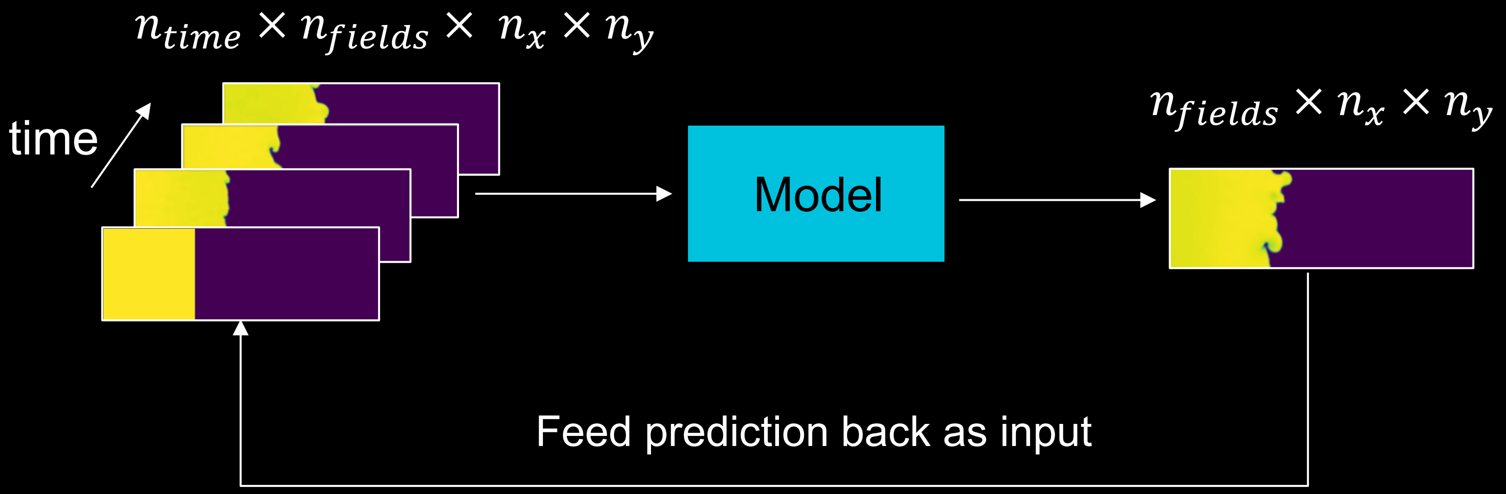
[1] https://github.com/PolymathicAI/the_well

Time Series Prediction with Neural Operators

- Training: Model predicts the next state based on the last n_{time} states, which is a hyperparameter



Time Series Prediction with a Neural Operator



- Inference: Model is evaluated auto-regressively, i.e. predictions are fed back as input

How to use PyTorch packages with ROCm-Support?

- We use the `neuraloperator` package, which implements FNOs in PyTorch.
- Simply install with

```
$> pip install neuralop
```

- And use with

```
import torch
import neuraloperator
```

- Just make sure to have PyTorch with ROCm support installed. That's it…

Key Components of the Training Task

```
model = neuralop.models.FNO(
        n_modes = (16, 16),
        in_channels = args.n_steps_input * n_fields,
        out_channels = n_fields,
        hidden_channels = 128,
        n_layers = 5,
        ).to(device)
```

Model

Data movement from CPU to GPU

```
optimizer = torch.optim.Adam(
            model.parameters(),
            lr = 5e-3
            )
```

Optimizer

```
loss = torch.nn.MSELoss()
```

Loss Function

```
data_loader = torch.utils.data.DataLoader(
              dataset = dataset_train,
              shuffle = True,
              batch_size = args.batchsize,
              )
```

Data Loader

Basic Training Loop

```
for epoch in range(args.num_episodes):
   for batch in tqdm.tqdm(data_loader, unit="batches"):

      # 1. Nullify the gradients for new batch
      optimizer.zero_grad()

      # 2. Move Data to Device
      x = batch["input_fields"].to(device)
      y = batch["output_fields"].to(device)

      # 3. Do some processing
      # ...

      # 4. Forward pass
      y_model = model(x)

      # 5. Backward pass
      output = loss(y_model, y) # Evaluate Loss
      output.backward()         # Compute Gradients
      optimizer.step()          # Optimizer Step
```

Data movement from CPU to GPU

Actual Compute

Run the Training

- Run a few episodes of training on a single GPU

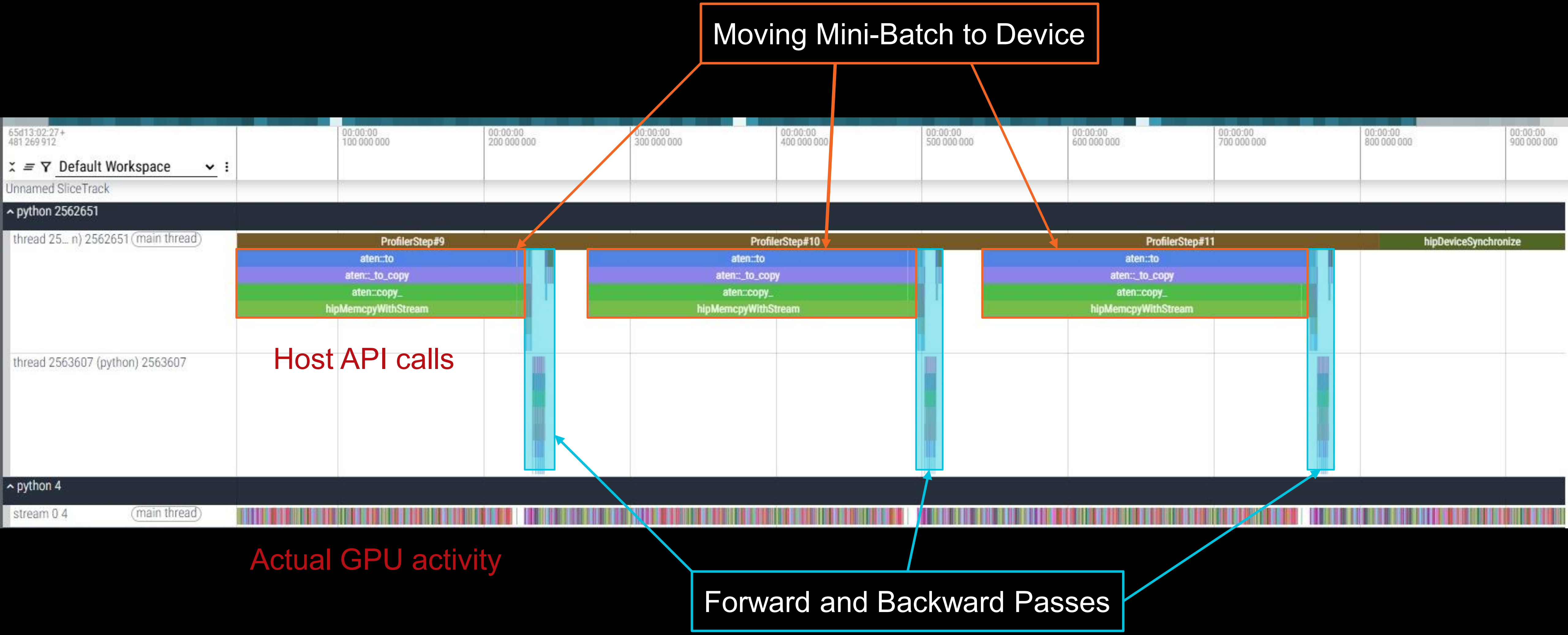
First Touch Penalty

```
$> python3 fno.py --num_episodes=10 --batchsize=16
Running on a GPU!
Found dataset at location ./datasets/turbulent_radiative_layer_2D!
100%|██████████████████████████████| 437/437 [01:25<00:00, 5.12batches/s]
100%|██████████████████████████████| 437/437 [01:18<00:00, 5.58batches/s]
100%|██████████████████████████████| 437/437 [01:08<00:00, 6.36batches/s]
100%|██████████████████████████████| 437/437 [01:08<00:00, 6.35batches/s]
100%|██████████████████████████████| 437/437 [01:12<00:00, 6.07batches/s]
100%|██████████████████████████████| 437/437 [01:05<00:00, 6.68batches/s]
100%|██████████████████████████████| 437/437 [01:08<00:00, 6.34batches/s]
100%|██████████████████████████████| 437/437 [01:08<00:00, 6.40batches/s]
100%|██████████████████████████████| 437/437 [01:10<00:00, 6.23batches/s]
100%|██████████████████████████████| 437/437 [01:08<00:00, 6.39batches/s]
```

"Throughput"

More details on how to profile AI workloads tomorrow!

How does this look on the system?



Data Movement

- Moving the training data to the GPU can become a major bottleneck, especially if
  - the model is small and cheap in terms of execution time
  - the training samples are large

- Possible Optimizations:
  - Fit the whole training dataset on the GPU (only possible for small datasets)
  - Overlap memory movement with computation (non-blocking communication)

- Easy improvements:
  - Use pinned memory
  - Use multiple CPU threads to prepare data

```
data_loader = torch.utils.data.DataLoader(
     dataset = dataset_train,
     shuffle = True,
     batch_size = args.batchsize,
     pin_memory = True,
     num_workers = 2,
     prefetch_factor = 4
     )
```

[Public]

Run the Training using Pinned Memory

- Pinned Memory and multiprocessing improves throughput by ~15% (7.2 vs. 6.3 batches per second)

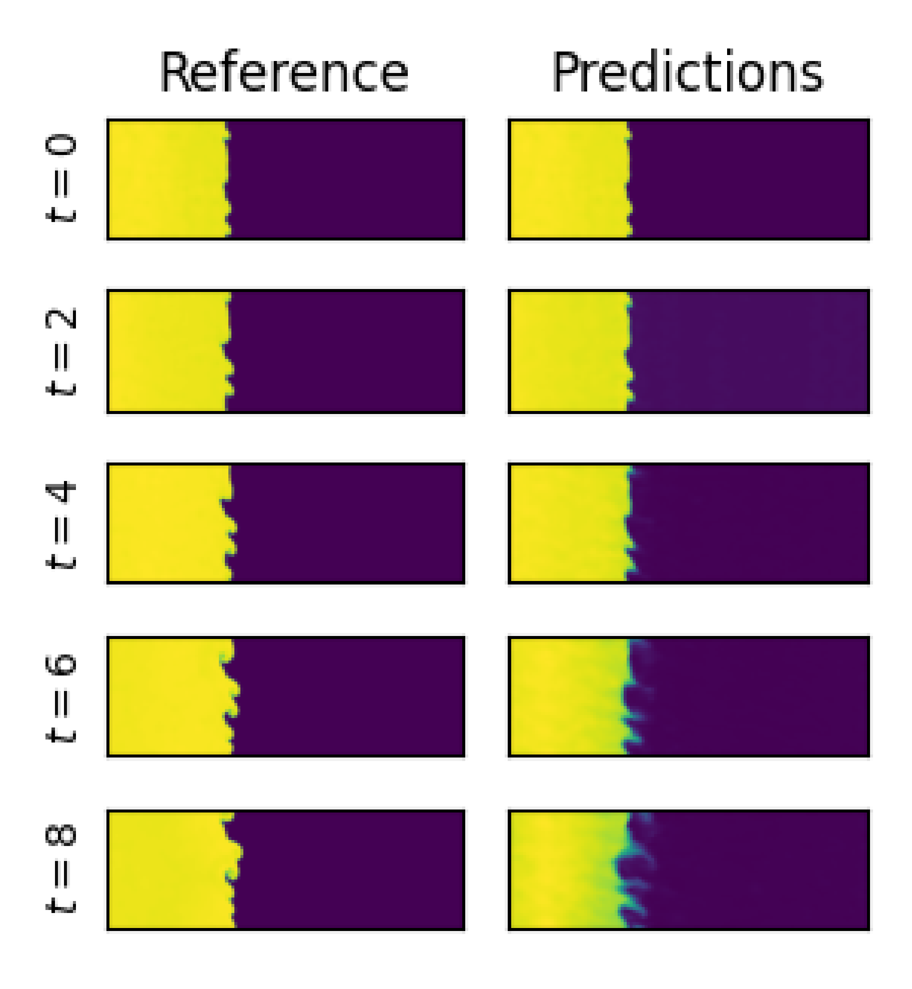
First Touch Penalty

```
$> python3 fno.py --num_episodes=10 --batchsize=16
Running on a GPU!
Found dataset at location ./datasets/turbulent_radiative_layer_2D!
100%|██████████████████████████████████████████| 437/437 [01:11<00:00, 6.10batches/s]
100%|██████████████████████████████████████████| 437/437 [01:01<00:00, 7.08batches/s]
100%|██████████████████████████████████████████| 437/437 [01:00<00:00, 7.20batches/s]
100%|██████████████████████████████████████████| 437/437 [01:00<00:00, 7.21batches/s]
100%|██████████████████████████████████████████| 437/437 [01:00<00:00, 7.21batches/s]
100%|██████████████████████████████████████████| 437/437 [01:00<00:00, 7.22batches/s]
100%|██████████████████████████████████████████| 437/437 [01:00<00:00, 7.23batches/s]
100%|██████████████████████████████████████████| 437/437 [01:00<00:00, 7.23batches/s]
100%|██████████████████████████████████████████| 437/437 [01:00<00:00, 7.23batches/s]
100%|██████████████████████████████████████████| 437/437 [01:00<00:00, 7.23batches/s]
```

~15% higher throughput

Evaluating the Trained Model

- First 2-3 predictions are close to the ground-truth
- Starts diverging after few timesteps (exponential error growth)
- Predicting several timesteps successively was not part of the training
- See the original paper[1] to find the datasets and models that perform better!



[1] https://openreview.net/pdf?id=7UunRhzX4H

Key Takeaways

- Simulation datasets are oftentimes very large and non-trivial to handle
- Neural Operators promise to generalize better across resolutions than e.g. ResNet, U-Net, …
- AI models oftentimes lack long-term stability in auto-regressive tasks due to exponential error growth
- Data movement might be the bottleneck of training, especially if models are small and data samples large

  → More details on profiling tomorrow!

Try it yourself!

- Get on a node via SLURM and setup your environment:

```
module load rocm
module load pytorch
python3 -m venv $HOME/venv-pt
source $HOME/venv-pt/bin/activate
cd HPCTrainingExamples/MLExamples/Neural_Operators
pip3 install -r requirements.txt
```

Make sure ROCm is loaded for the PyTorch module to become visible!

- Run a few episodes of training with

```
python3 fno.py --num_episodes=3 --batchsize=4
```

- Next, feel free to:
  - test the model on another dataset
  - replace the Neural Operator by your favorite ML model (e.g. a ResNet)
  - follow the GitHub Readme to reproduce the shown profile data (much more information on profiling tomorrow!)
  - or do whatever you find interesting!

Disclaimer

AMD