Introduction to High-Performance Machine Learning

Lecture 7 03/07/24

Dr. Kaoutar El Maghraoui

Dr. Parijat Dube

PyTorch Performance – Profiling

A Different Mental Model Required

GPU PERFORMANCE TUNING

CPU

Optimized for single thread performance
- Majority of chip area is control logic & caches

Complex and deep out-of-order pipelines
- Extract instruction level parallelism

The brain
- Job is to keep the accelerator busy

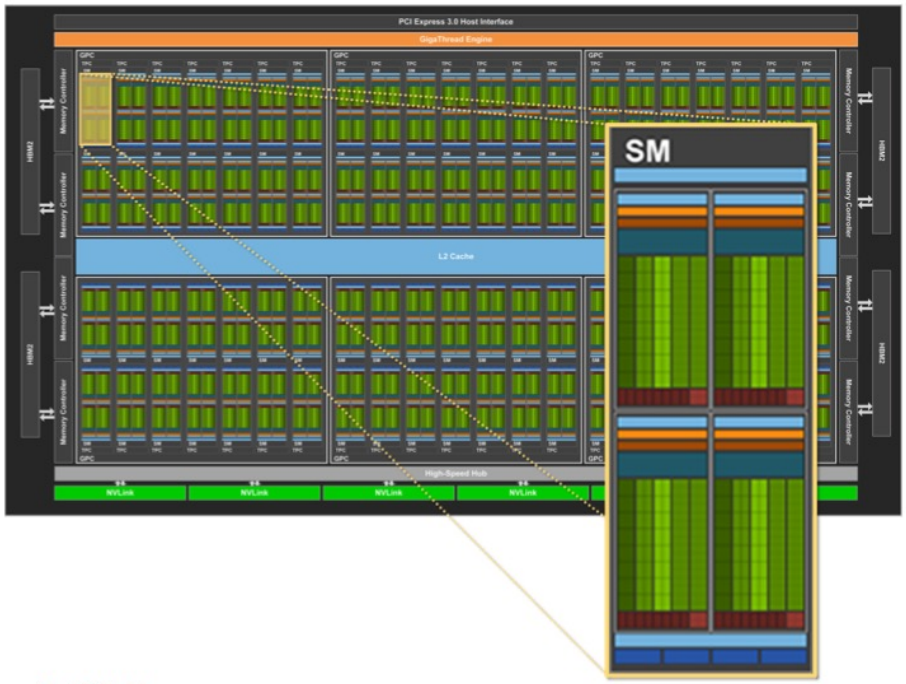
GPU

Optimized for throughput of data-parallel problems
- Majority of chip area is functional units

Simple, relatively slow in-order pipelines
- Achieves much higher total throughput

Accelerator attached via PCIe
- Order of magnitude faster but off to the side

NVIDIA Volta GPU Design



NVIDIA Volta V100 GPU

Composed of Streaming Multiprocessors (SMs)

Volta V100: 80x SMs
Ampere A100: 108 SMs

DGX A100 with 8 GPUs:
864 SMs vs 128 CPU cores

Streaming Multiprocessor Design

Streaming Multiprocessor

64x FP32 units

64x INT, 32x FP64, 32x LD/ST

8x Tensor Cores

5120 (6920 ON A100) FP32 EXECUTION UNITS PER GPU

Common Pitfalls

- Excessive CPU/GPU interactions – e.g., for loop launching GPU operations
  - Dominated by launch overheads
- Short GPU kernel durations – e.g., small inputs
  - Need enough data to feed 10s of thousands of threads
- CPU overheads and I/O bottlenecks are starving the GPU
  - Small operations on the CPU can quickly become dominant
- Framework inefficiencies
  - E.g., unnecessary copies and hidden CPU-side overheads

Visibility is key to understand what is going on and optimize your code

Profiling objectives

- **Measure** and identify critical sections and resource bottlenecks
  - CPU, GPU, Memory, Network, I/O (disk)
- Then**, Tune** the model based on the measured and profiled data

- **What can be profiled?** Almost everything with the right tools…
  - Hardware activity: performance counters
  - Operating system (perf)
  - Memory operations (perf, *valgrind*)
  - Libraries
  - Applications
  - Parallel Distributed Applications (MPI profilers…)

Profiling and Tracing Techniques Review

- **Counting** (Deterministic):
  - Count every time a hardware/software event happens (ex. memory load, function call)
  - Report a table of events count
- **Sampling** (Indeterministic: statistical effect):
  - Interrupt the application at **regular intervals** (sampling frequency) and increment a counter associated with the instruction that was interrupted
  - Compute a histogram associating samples to lines of code
  - Can be used to statistically **infer** the relative time in each part of the code
- **Tracing** (Deterministic):
  - Record every time a hardware/software event happens and also the time at which it happens (timestamp)
  - Report a table of relative time spent in each event

Profiling/Tracing techniques Overhead comparison

Counting:

- Mem. footprint: a counter for each (software/hardware) event
  - **low**

Sampling:

- Mem. footprint: state of the program (instruction counter minimum) at each interval
  - **medium** (depends on sampling frequency and state size)

Tracing:

- Mem. footprint: event type + timestamp at each (software/hardware) event
  - **high**

Identifying Bottlenecks

- You need to identify bottlenecks in the system before optimizing it.
  - Helps focus the optimization efforts on areas with the biggest impact on performance
- Bottlenecks can vary depending on several factors:
  - Size of the dataset
  - Complexity of the model
  - Hardware being used

- Examples:
  - Large dataset -> data loading step could be a bottleneck
  - Model is complex -> training steps could be a bottleneck
  - Code not using GPU -> CPU could be a bottleneck
  - Code using GPU -> bottleneck could be GPU memory or bandwidth between CPU and GPU

How to identify bottlenecks

- Traditional Command Line Tools
  - Helpful in monitoring PyTorch training and identifying bottlenecks
  - Easy to use
  - They can be accessed from any terminal
  - Can monitor various metrics: CPU usage, GPU usage, memory usage, and I/O traffic
- Visualization and tracing tools
  - Examples: Tensorboard, PyTorch Profiler Chrome tracing visualization, weights and biases, Flame Graph visualization, etc.
- Examples:
  - Large dataset -> data loading step could be a bottleneck
  - Model is complex -> training steps could be a bottleneck
  - Code not using GPU -> CPU could be a bottleneck
  - Code using GPU -> bottleneck could be GPU memory or bandwidth between CPU and GPU

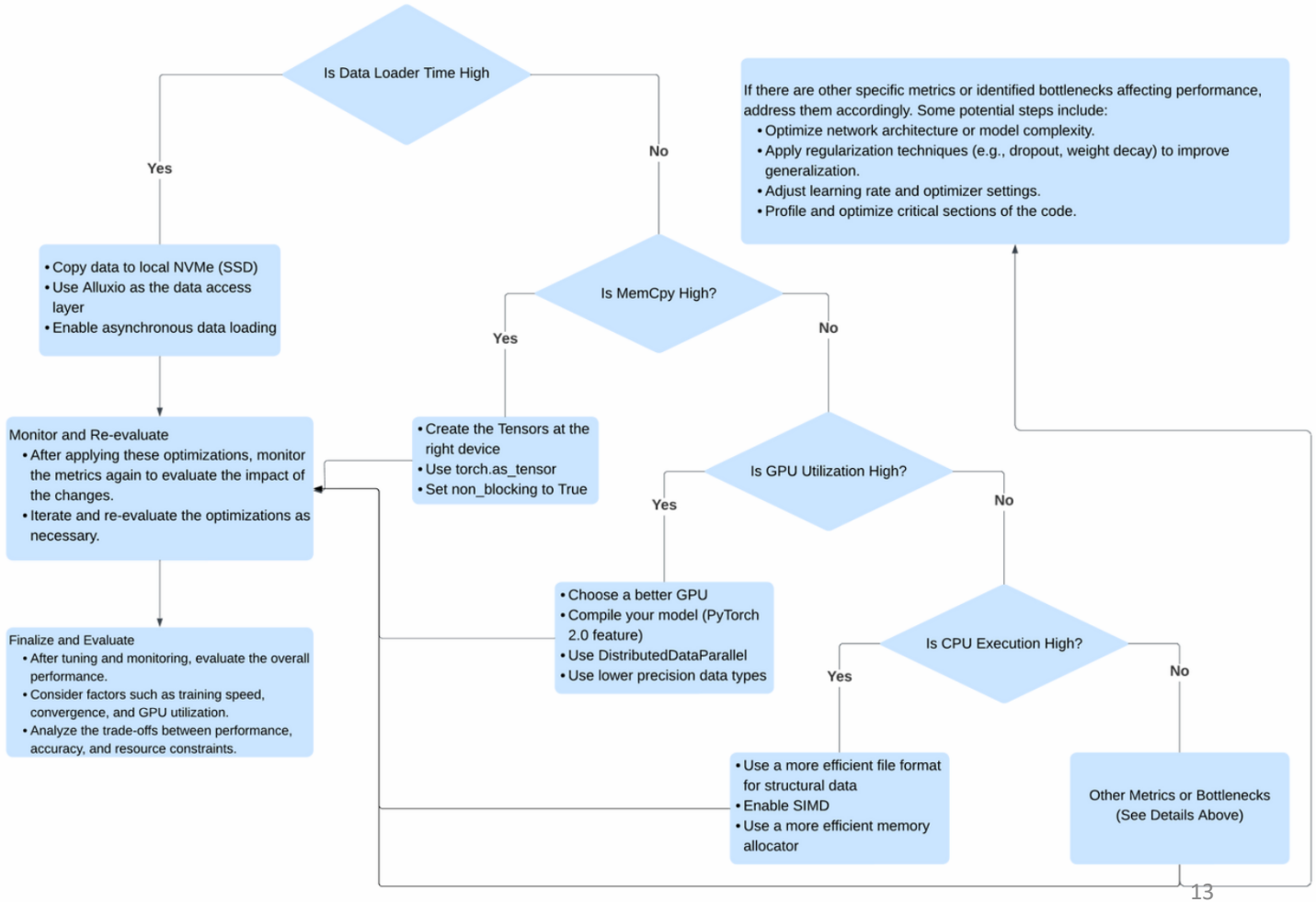
Common Profiling Command Line tools

- Most used command-line tools for monitoring resource usage:

Tool	Description
nvidia-smi	This tool provides information about GPU utilization, memory usage, and other metrics related to the NVIDIA GPU
htop	It is a command-line tool that hierarchically displays system processes and provides insights into CPU and memory usage
iotop	With this tool, you can monitor I/O usage by displaying I/O statistics of processes running on your system
gpustat	It is a Python-based user-friendly command-line tool for monitoring NVIDIA GPU status.
nvtop	Similar to nvidia-smi, nvtop displays real-time GPU usage and other metrics in a user-friendly interface.
py-spy	It is a sampling profiler for Python that helps identify performance bottlenecks in your code.
strace	This tool allows you to trace system calls made by a program, providing insights into its behavior and resource usage.

A General Performance Tuning Guide

A decision tree that shows a structured process for tuning PyTorch training based on key metrics such as GPU utilization, CPU execution, data loader time, and memory copy (memcpy).



Reference: PyTorch Model Training Performance Tuning Ebook by Alluxio

Python profiling tools

- **PyTorch Profiler**
- ***profile:*** pure python module: higher overhead
- ***cProfile*****:** CPython extension modules that traces the execution of Python programs, collecting information on the functions and primitives used:
  - Number of calls
  - Total time (time spent in the function/primitive, excluding nested calls)
  - Cumulative time (time including nested calls)
  - Call graph

  cProfile is the C implementation of the profile interface
- ***pstats***: a module that provides analysis methods for the data collected by the profilers

PyTorch Profiler

- Slides adopted from:

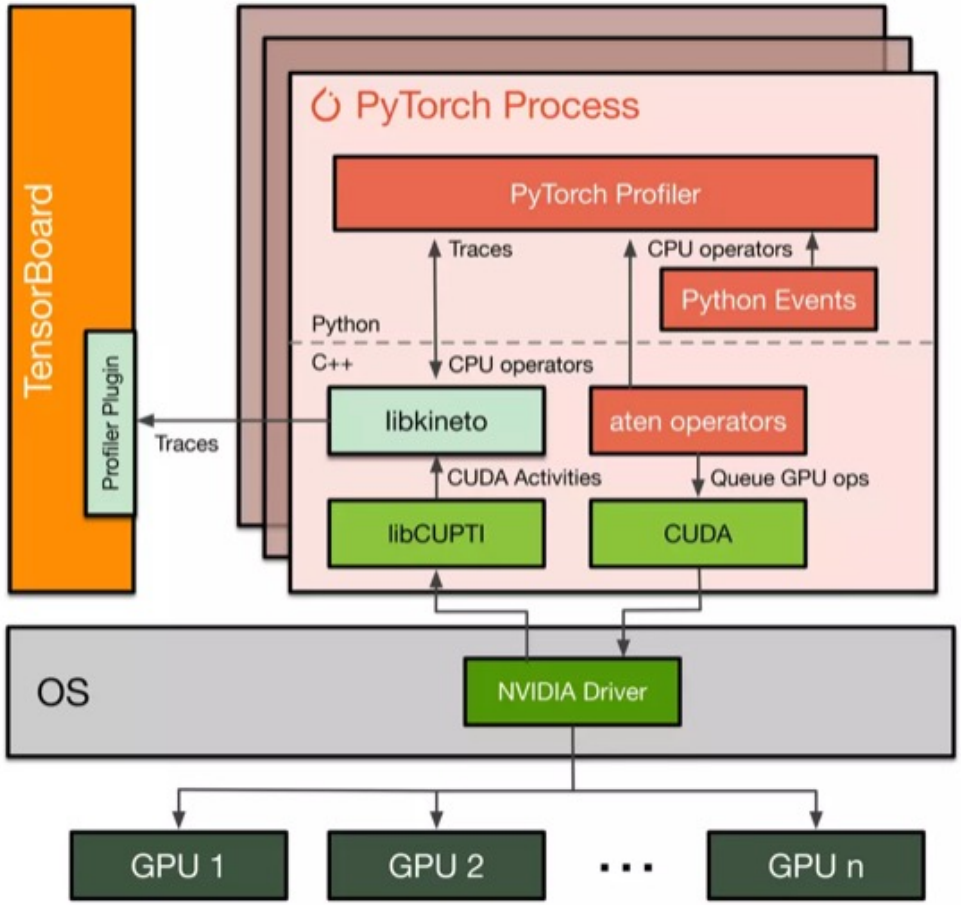
  GEETA CHAUHAN

  PYTORCH PARTNER ENGINEERING

  META AI

PyTorch Profiler

- Contributed by Microsoft & Facebook
  - PyTorch and GPU-level information
  - Automatic bottleneck detection
  - Actionable performance recommendations
  - Data scientist-friendly lifecycle and tools
  - TensorBoard Plugin – Chrome traces visualization
  - Kineto library- built on CUPTI (NVIDIA CUDA Profiling Tools Interface)
  - Easy-to-use Python API
  - VS Code integration



CUPTI: CUDA Profiling Tools Interfaces

ATEN: Short of "A Tensor Library", is a library on top all other Python and C++ interfaces in PyTorch are built

Libkineto: an in-process profiling library integrated with the PyTorch Profile

Profiling API: Basic Usage

```
import torch.profiler

with profiler.profile(..) as prof:
  <code to profile>

# Print results on console
print(prof.key_averages().table(..))
```

https://pytorch.org/tutorials/recipes/recipes/profiler.html

```
import torch
import torchvision.models as models
import torch.profiler as profiler

model = models.resnet18()
inputs = torch.randn(5, 3, 224, 224)

with profiler.profile(record_shapes=True) as prof:
   with profiler.record_function("model_inference"):
      model(inputs)

print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
```

```
# ------------------------  ------------  ----------  ------------  ---------
# Name                      Self CPU total   CPU total  CPU time avg  # Calls
# ------------------------  ------------  ----------  ------------  ---------
# model_inference           3.541ms         69.571ms    69.571ms      1
# conv2d                    69.122us        40.556ms    2.028ms       20
# convolution               79.100us        40.487ms    2.024ms       20
# _convolution              349.533us       40.408ms    2.020ms       20
# mkldnn_convolution        39.822ms        39.988ms    1.999ms       20
# batch_norm                105.559us       15.523ms    776.134us     20
# _batch_norm_impl_index    103.697us       15.417ms    770.856us     20
# native_batch_norm         9.387ms         15.249ms    762.471us     20
# max_pool2d                29.400us        7.200ms     7.200ms       1
# max_pool2d_with_indices   7.154ms         7.170ms     7.170ms       1
# ------------------------  ------------  ----------  ------------  ---------
```

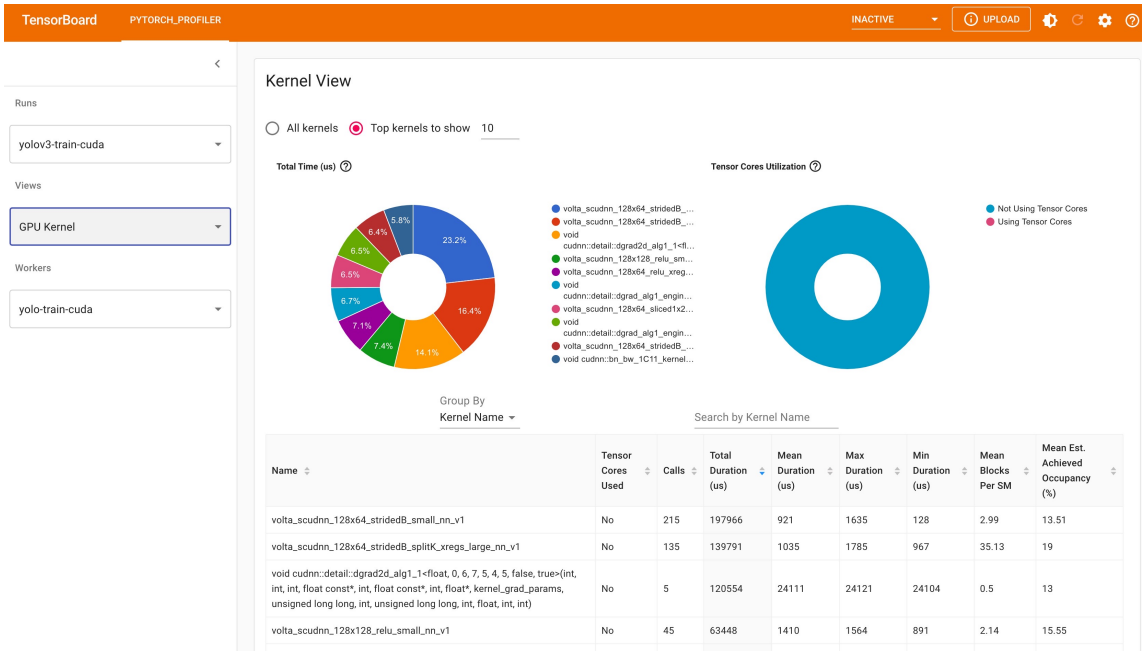
Profiling API: Tensorboard Plugin

```
$PIP INSTALL TORCH-TB-PROFIILER

import torch.profiler

# Export to tensorboard using
# on_trace_handler option
handler=
 tensorboard_trace_handler(..)
with profiler.profile(
 on_trace_ready=handler
) as prof:
 ...
```

```
import torch
import torchvision.models as models
import torch.profiler as profiler

model = models.resnet18()
inputs = torch.randn(5, 3, 224, 224)

with profiler.profile(
  record_shapes=True,
  on_trace_ready=torch.profiler.tensorboard_
trace_handler('results')
) as prof:
   model(inputs)

print(prof.key_averages().table(sort_by=
"cpu_time_total", row_limit=10))
```

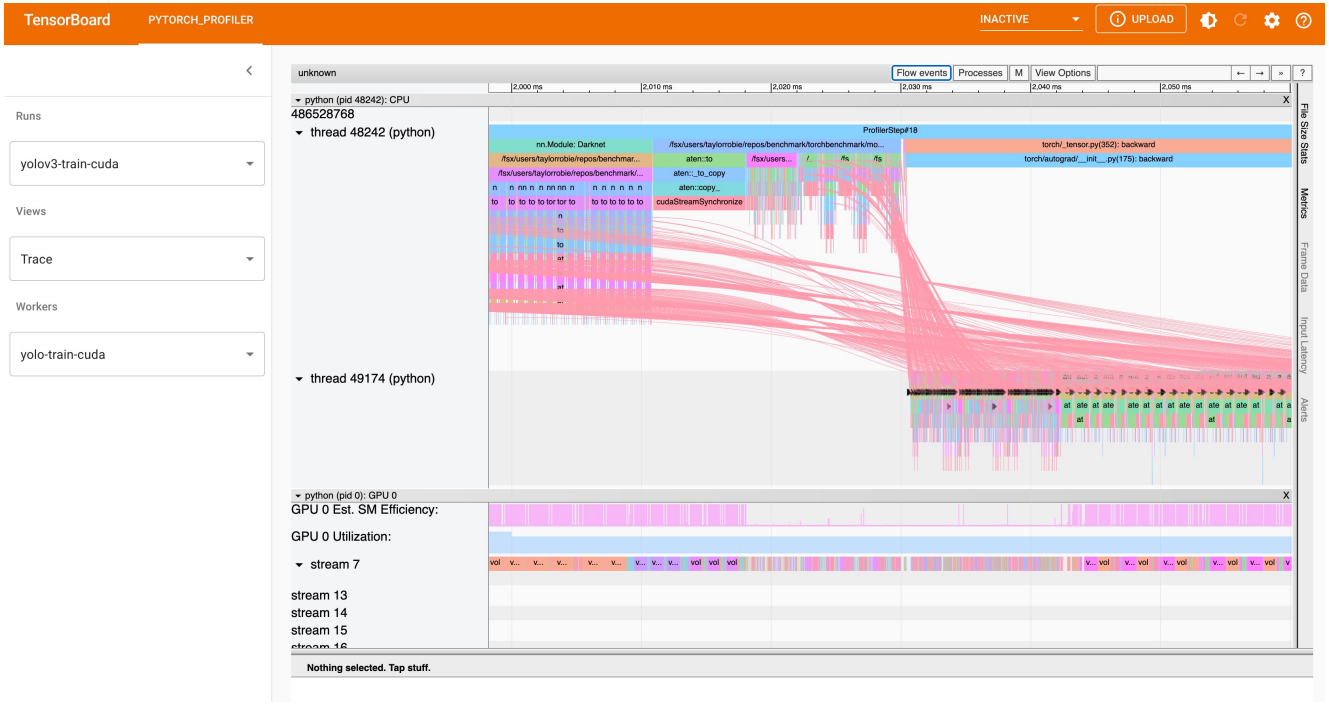
Profiling API: Tensorboard Plugin

```
$PIP INSTALL TORCH-TB-PROFIILER

import torch.profiler

# Export to tensorboard using
# on_trace_handler option
handler=
  tensorboard_trace_handler(..)
with profiler.profile(
  on_trace_ready=handler
) as prof:
  . . .
```



Profiling API: Tensorboard Plugin

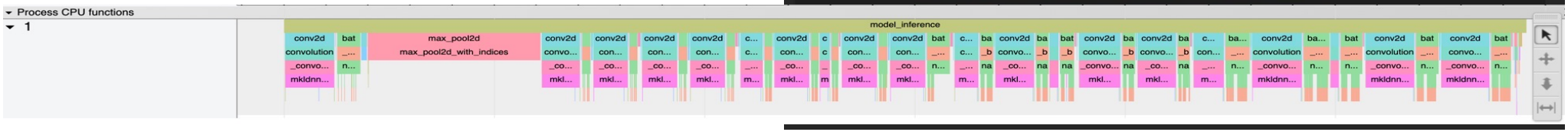
```
$PIP INSTALL TORCH-TB-PROFIILER

import torch.profiler

# Export to tensorboard using
# on_trace_handler option
handler=
  tensorboard_trace_handler(..)
with profiler.profile(
  on_trace_ready=handler
) as prof:
  . . .
```

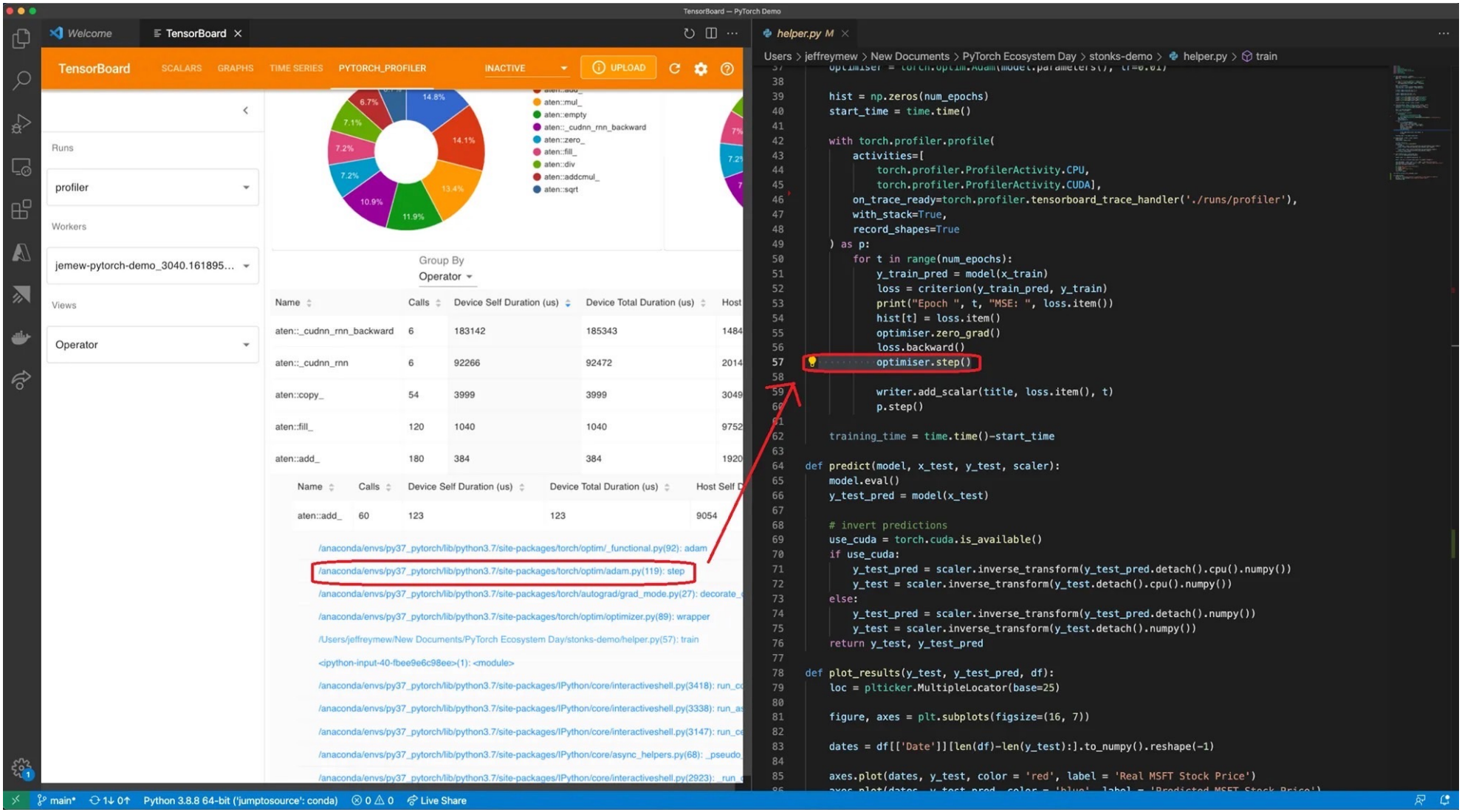


Advanced Options

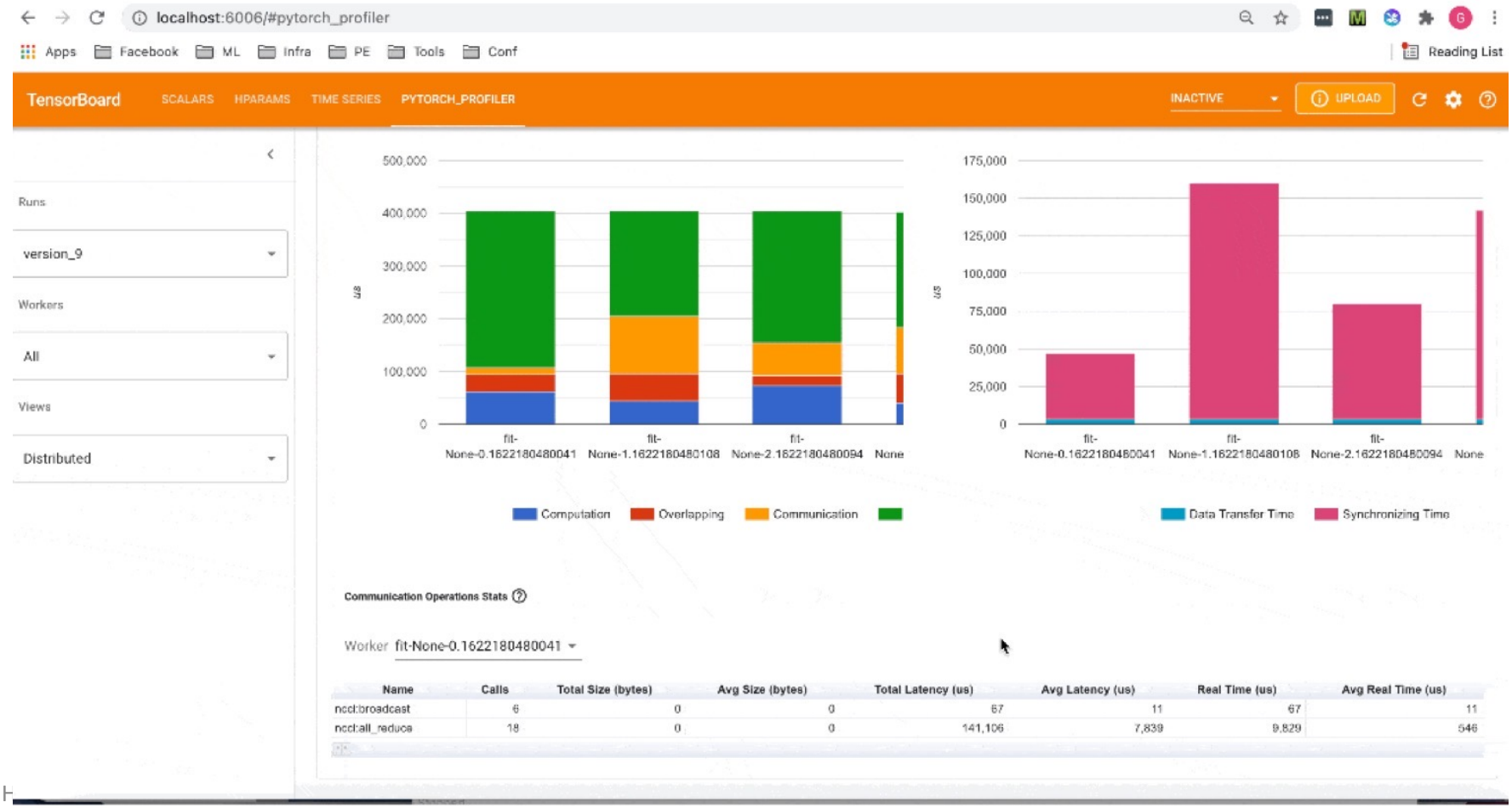
- When to trigger
- How many steps to profile
- Which activities to profile
- Callable handler to save the results
- Extra metadata, eg., shapes, stacks, memory
- Output options: eg.,, Chrome tracing, TensorBoard

```
import torch.profiler as profiler

with torch.profiler.profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA],
    schedule=torch.profiler.schedule(
        wait=2,
        warmup=3,
        active=6),
    on_trace_ready=torch.profiler.tensorboard_trace_handler('./result'),
    record_shapes=True,
) as p:
    for step, data in enumerate(trainloader, 0):

        inputs, labels = data[0].to(device=device), data[1].to(device=device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if step + 1 >= 11:
            break
        p.step()

    p.export_chrome_trace(path)
    print(p.key_averages().table(
            sort_by="self_cuda_time_total", row_limit=-1))
```

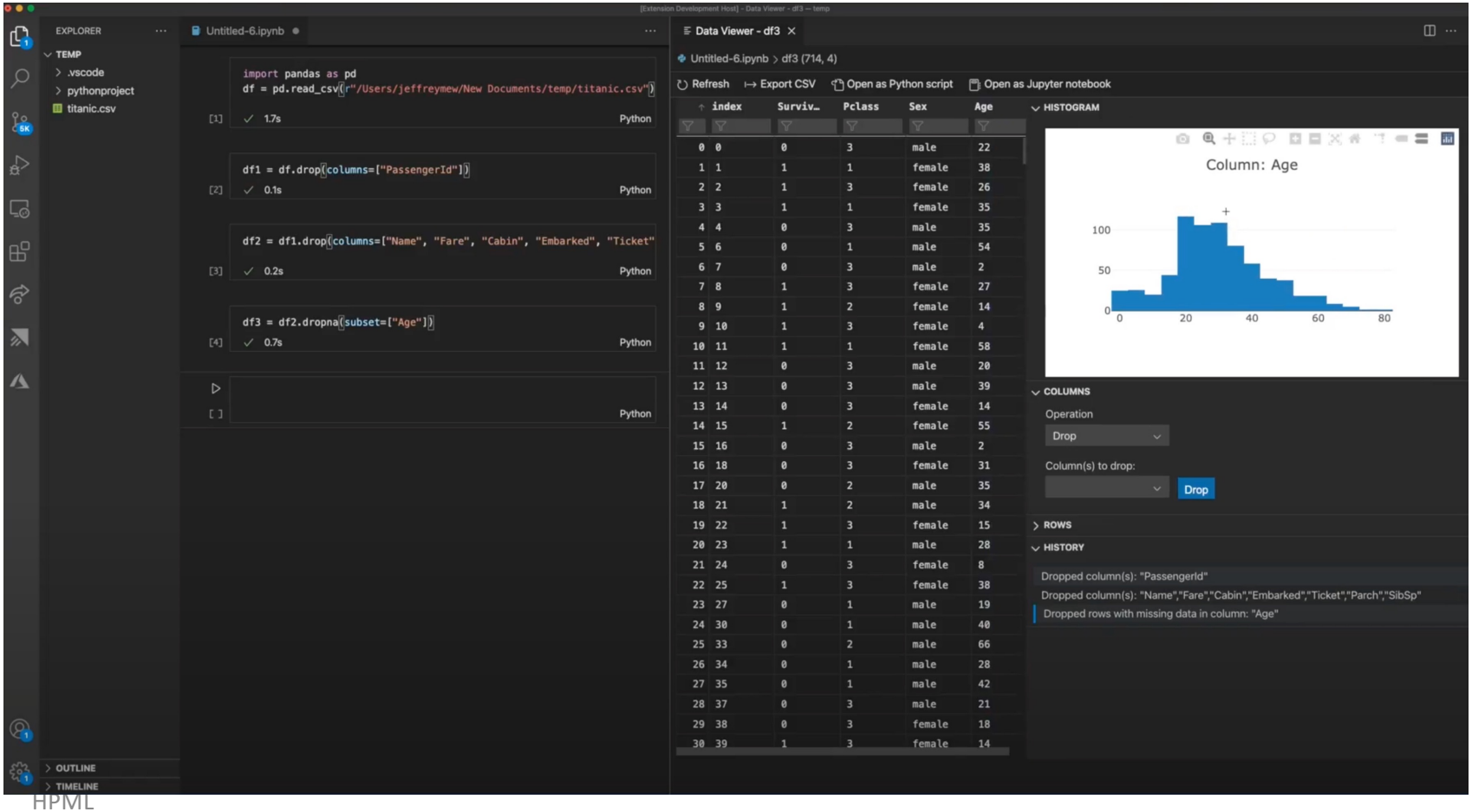



TensorBoard — PyTorch Demo
Welcome
TensorBoard
TensorBoard
SCALARS
GRAPHS
TIME SERIES
PYTORCH_PROFILER
INACTIVE
UPLOAD
Runs
profiler
Workers
jemew-pytorch-demo_3040.161895...
Views
Operator
14.8%
14.1%
13.4%
11.9%
10.9%
7.2%
7.2%
7.1%
6.7%
aten::mul_
aten::empty
aten::_cudnn_rnn_backward
aten::zero_
aten::fill_
aten::div
aten::addcmul_
aten::sqrt
Group By
Operator
Name
Calls
Device Self Duration (us)
Device Total Duration (us)
aten::_cudnn_rnn_backward 6 183142 185343
aten::_cudnn_rnn 6 92266 92472
aten::copy_ 54 3999 3999
aten::fill_ 120 1040 1040
aten::add_ 180 384 384
aten::add_ 60 123 123
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/torch/optim/_functional.py(92): adam
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/torch/optim/adam.py(119): step
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/torch/autograd/grad_mode.py(27): decorate_
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/torch/optim/optimizer.py(89): wrapper
/Users/jeffreymew/New Documents/PyTorch Ecosystem Day/stonks-demo/helper.py(57): train
<ipython-input-40-fbee9e6c98ee>(1): <module>
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/interactiveshell.py(3418): run_c
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/interactiveshell.py(3338): run_as
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/interactiveshell.py(3147): run_ce
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/async_helpers.py(68): _pseudo_
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/interactiveshell.py(2923): run_
helper.py M
Users > jeffreymew > New Documents > PyTorch Ecosystem Day > stonks-demo > helper.py > train
optimiser = torch.optim.Adam(model.parameters(), lr=0.01)
hist = np.zeros(num_epochs)
start_time = time.time()
with torch.profiler.profile(
activities=[
torch.profiler.ProfilerActivity.CPU,
torch.profiler.ProfilerActivity.CUDA],
on_trace_ready=torch.profiler.tensorboard_trace_handler('./runs/profiler'),
with_stack=True,
record_shapes=True
) as p:
for t in range(num_epochs):
y_train_pred = model(x_train)
loss = criterion(y_train_pred, y_train)
print("Epoch ", t, "MSE: ", loss.item())
hist[t] = loss.item()
optimiser.zero_grad()
loss.backward()
optimiser.step()
writer.add_scalar(title, loss.item(), t)
p.step()
training_time = time.time()-start_time
def predict(model, x_test, y_test, scaler):
model.eval()
y_test_pred = model(x_test)
# invert predictions
use_cuda = torch.cuda.is_available()
if use_cuda:
y_test_pred = scaler.inverse_transform(y_test_pred.detach().cpu().numpy())
y_test = scaler.inverse_transform(y_test.detach().cpu().numpy())
else:
y_test_pred = scaler.inverse_transform(y_test_pred.detach().numpy())
y_test = scaler.inverse_transform(y_test.detach().numpy())
return y_test, y_test_pred
def plot_results(y_test, y_test_pred, df):
loc = plticker.MultipleLocator(base=25)
figure, axes = plt.subplots(figsize=(16, 7))
dates = df[['Date']][len(df)-len(y_test):].to_numpy().reshape(-1)
axes.plot(dates, y_test, color = 'red', label = 'Real MSFT Stock Price')
main*
Python 3.8.8 64-bit ('jumptosource': conda)
Live Share

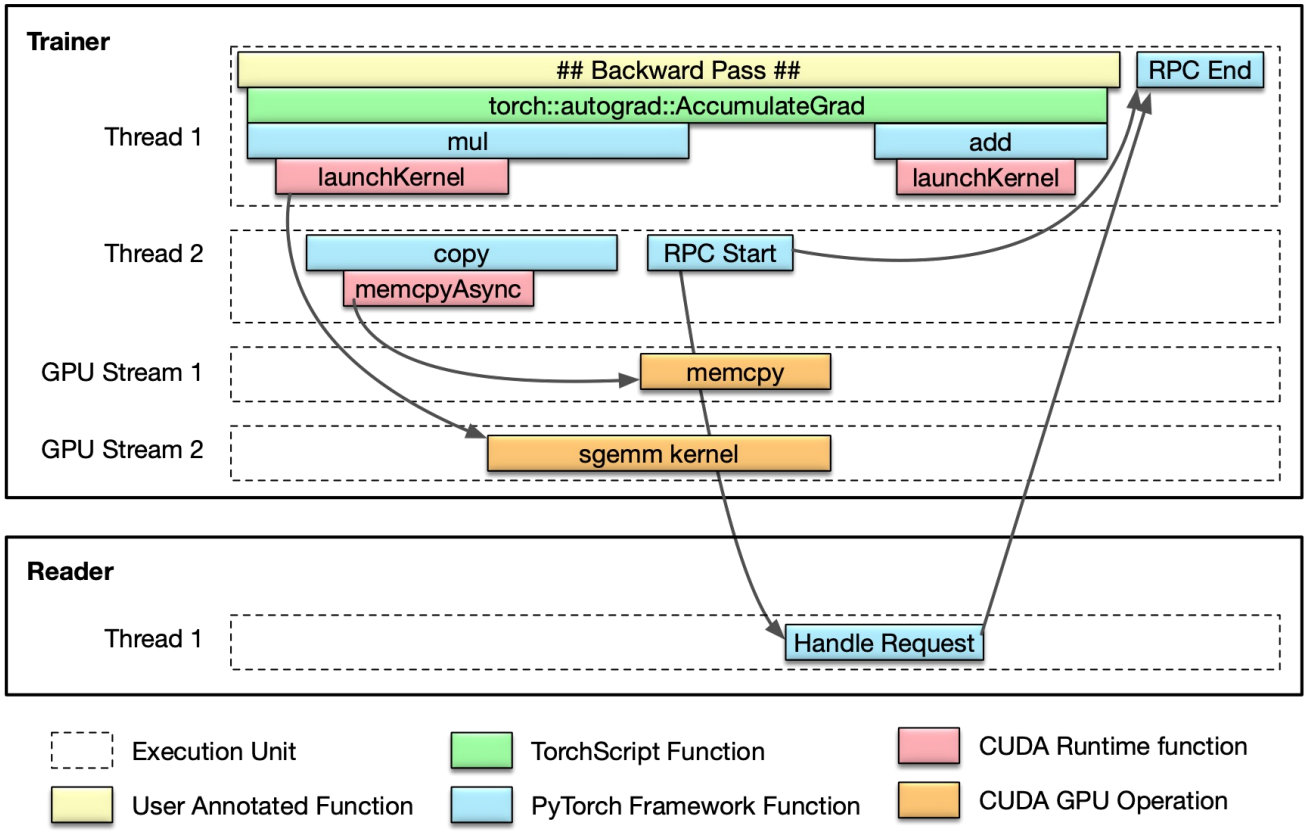
Distributed Training View



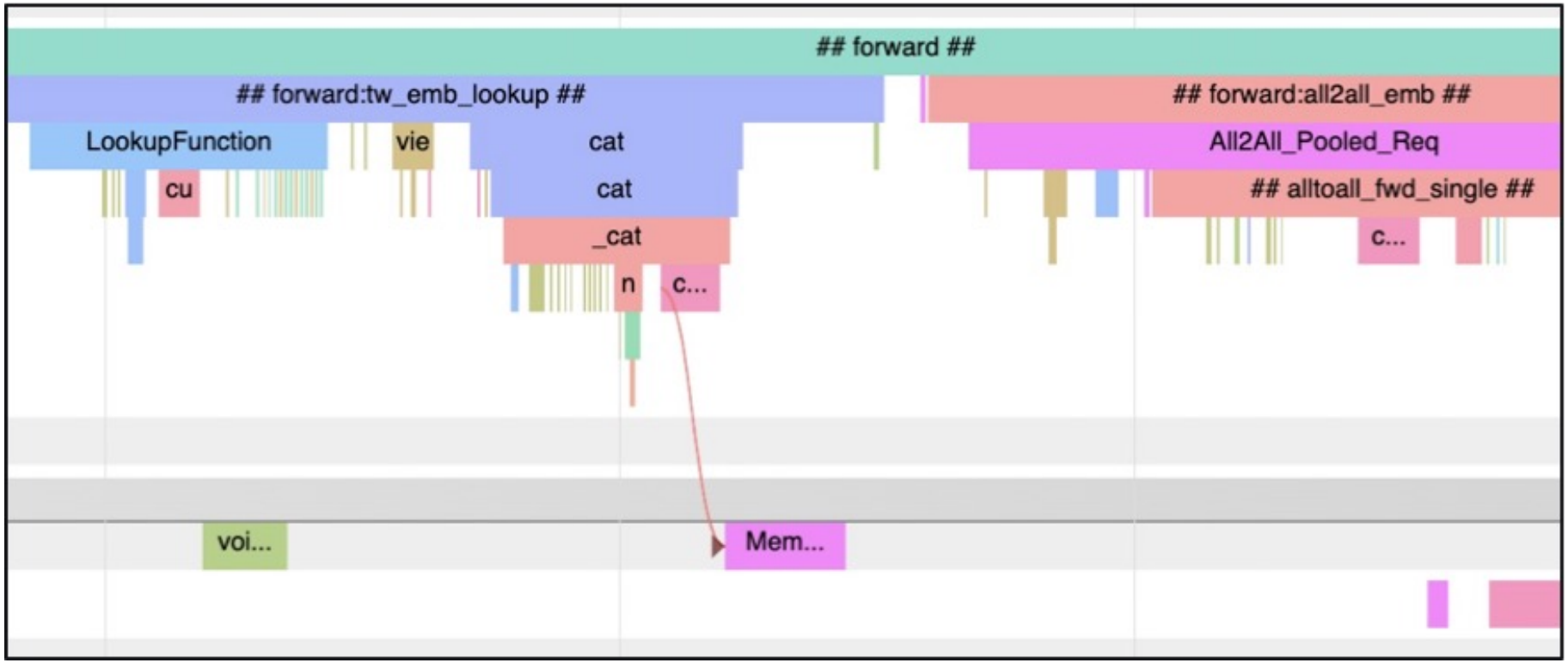
Vscode Integration: Data Wrangler



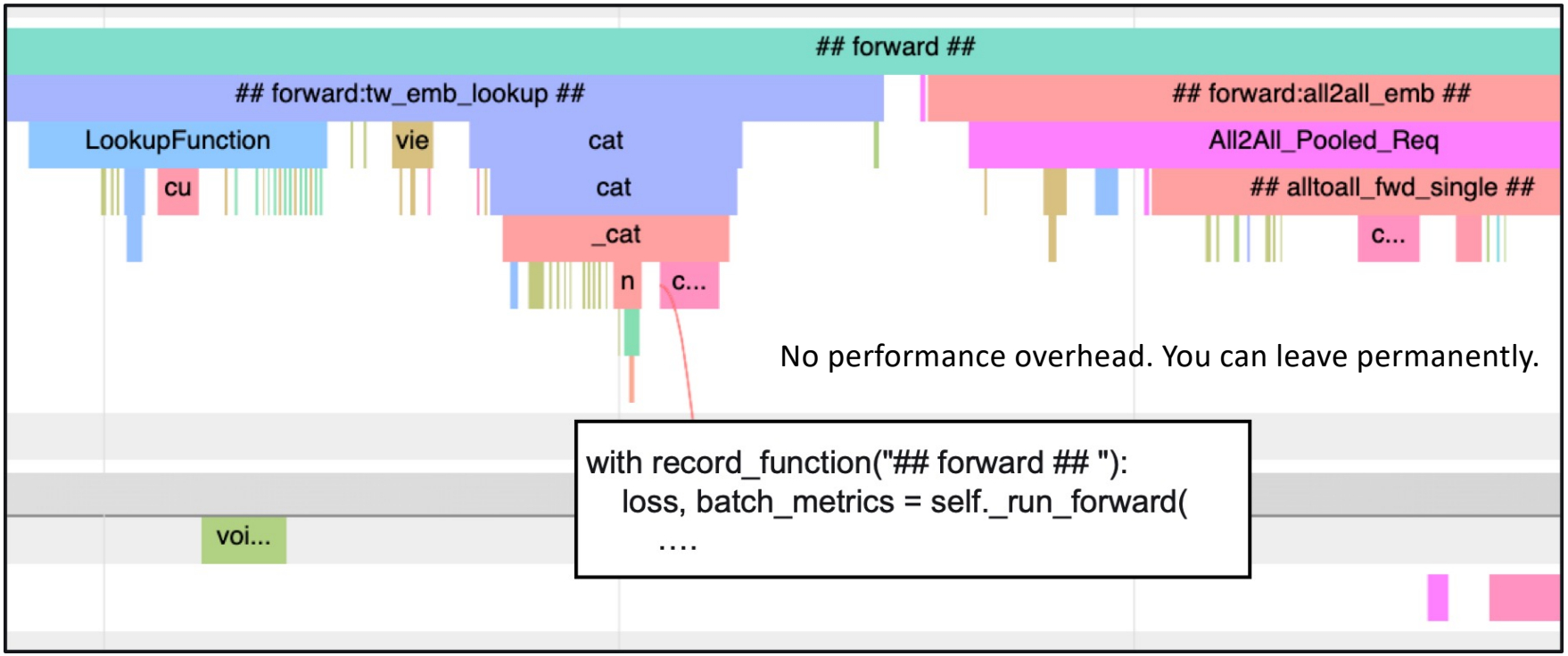
Timeline tracing: CPU and GPU Activities



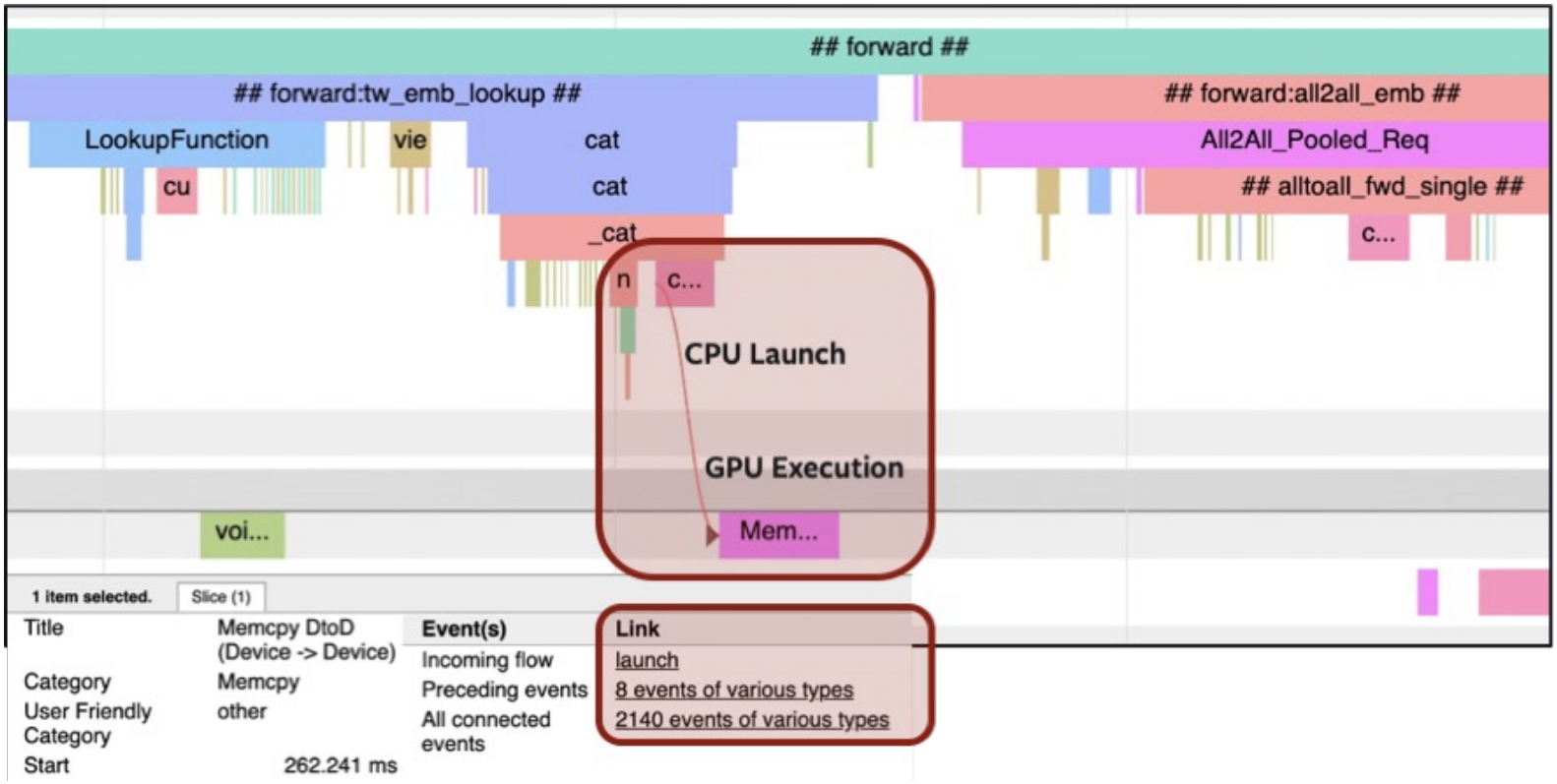
Timeline Tracing:
Chrome Trace Viewer- CPU and GPU timelines



Timeline Tracing:



You can see how CPU and GPU ops are connected



Timeline Tracing:
Inspect stats to individual activities

Nvidia-smi shows 86% utilization

But.. only a fraction of Streaming Multiprocessor are actually used by these kernels!

Timeline Tracing:
Inspect stats to individual activities

Much better utilization after increasing input sizes

Trace Analysis Examples with PyTorch Profiler

Thanks to Facebook Engineers for examples:
Lei Tian, Natalia Gimelshein, Lingyi Liu, Feng Shi & Zhicheng Yan

Anti-pattern: Long GPU idle time

Issue:

1. Large periods of GPU inactivity
2. Trace does not show why

Solution:

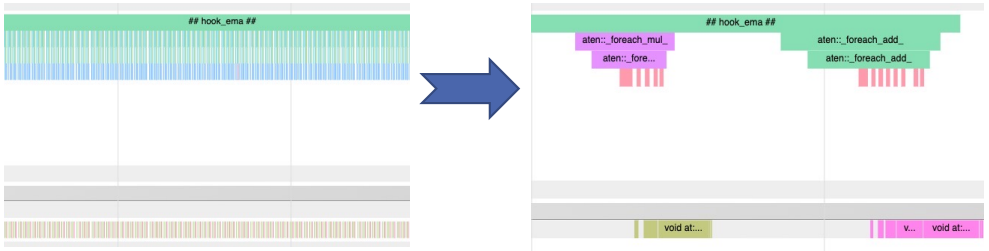
1. Use record_function to reveal bottlenecks on CPU
2. Parallelize CPU operations
3. Overlap CPU and GPU operations

```
temp = ""
num_substr = len(emb[k])

with record_function("## join_string {} ##".format(num_substr)):
    temp = ",".join(str(x) for x in emb[k]) # string concatenation

with record_function("## append_record_in_else ##"):
    records.append(f"{input_df.id[i + k]}\t{temp}\n") # list append
```

Anti-pattern: Excessive CPU/GPU Interactions

First issue:

- Exponential moving avg hook function has a loop – CPU bottleneck
- Can rewrite using torch._foreach ops – loop now on GPU



BEFORE

```
def on_step(self, task) -> None:
   ...
   with torch.no_grad():
      it = model_state_iterator(task.base_model)
      # iterate on every name & param
      for name, param in it:
         s = self.state.ema_model_state
         s[name] = self.decay * s[name] +
            (1 – self.decay) *
            param.to(device= self.device)
```

AFTER

```
def on_step(self, task) -> None:
   ...
   with torch.no_grad():
      torch._foreach_mul_(
         self.ema_model_state_list, self.decay)
      torch._foreach_add_(
         self.ema_model_state_list,
         self.param_list,
         alpha=(1 – self.decay))
```

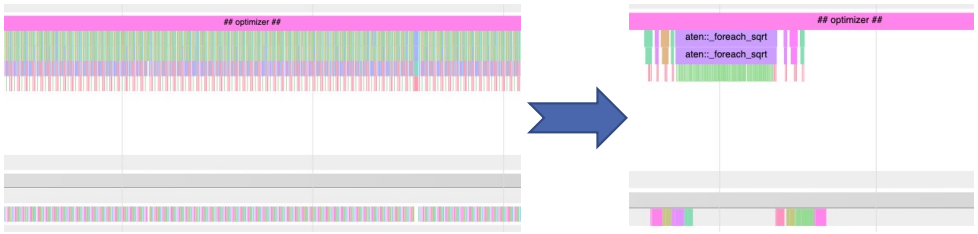
EMA HOOK 100X FASTER

ITERATION TIME: 860MS -> 770MS

Anti-pattern: Long GPU idle time

Second issue:

- Optimizer step uses a naïve implementation of RMSProp
- PyTorch provides an optimized multi-tensor version – using torch._foreach
- Switch to optimized version!




```
BEFORE
def prepare(self, param_groups):
   self.optimizer = RMSpropTFV2Optimizer(
      param_groups,
      …


AFTER
import torch.optim._multi_tensor as optim_mt
def prepare(self, param_groups):
   self.optimizer = optim_mt.RMSprop(
      param_groups,
      …
```

OPTIMIZER 12X FASTER

ITERATION TIME: 770MS -> 600MS

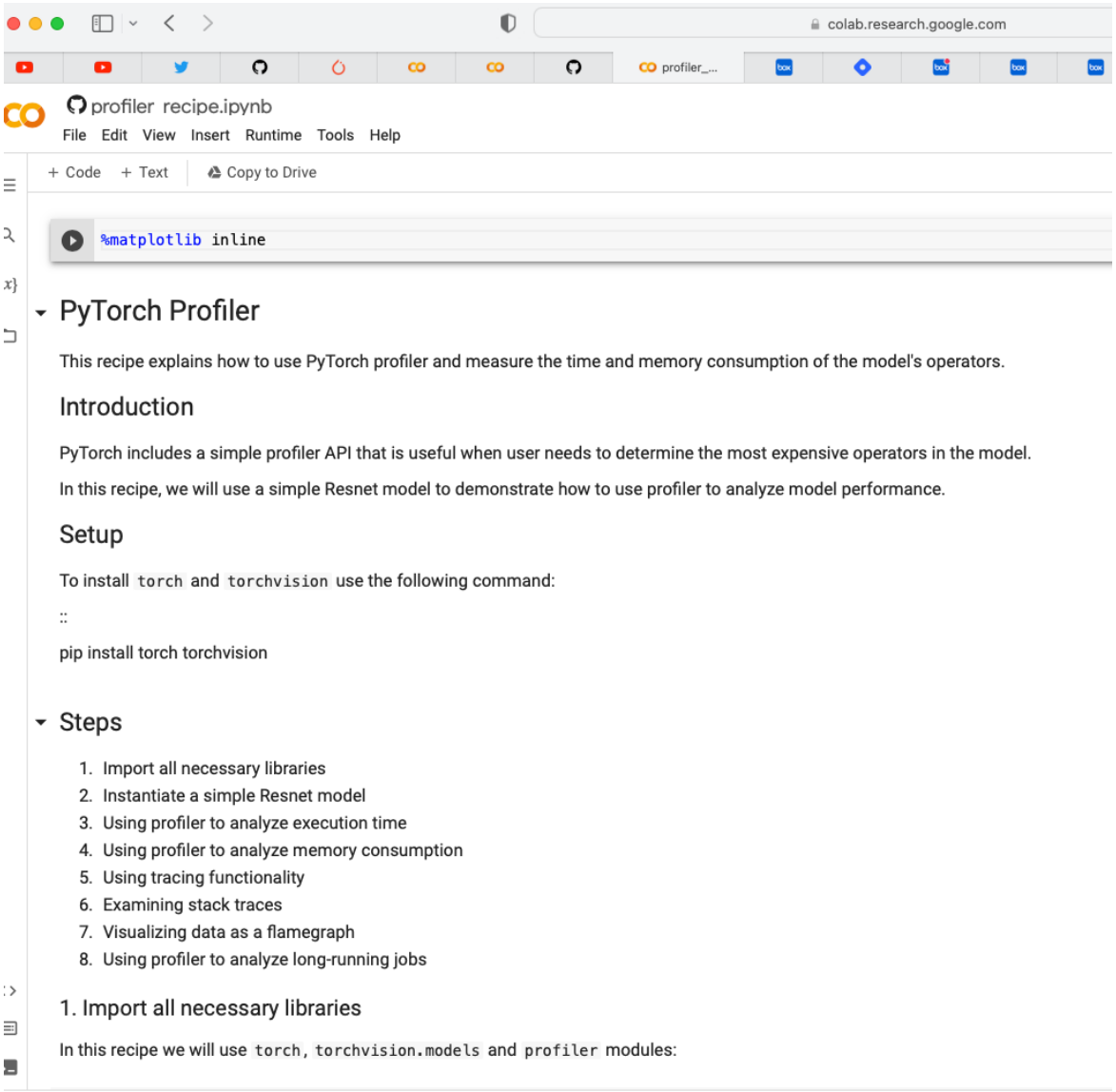
BERT Performance Optimization Case Study

- From 2.4 req/s to 1,400+ req/s
- CPU Inference
  - torch.set_num_threads(1)
  - Inel IPEX
  - Quantization
- GPU Inference on 1 T4 GPU
  - model.half()
  - DistiBERT
  - Increase batch size
  - Do not overpad
  - Faster Transformer

	Throughput	P99
BERT unoptimized bs=1	70.67 seq/s	20.44ms
BERT model.half() bs=8	359 seq/s	23.58ms
DistilBERT model.half() bs=16	689 seq/s	22.8ms
BERT Faster Transformer	885 seq/s	19.83ms
DistilBERT no padding model.half() bs=32	1423 seq/s	19.7ms

Example from PyTorch Documentation

https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html



Another Example from PyTorch Documentation with Tensorboard integration

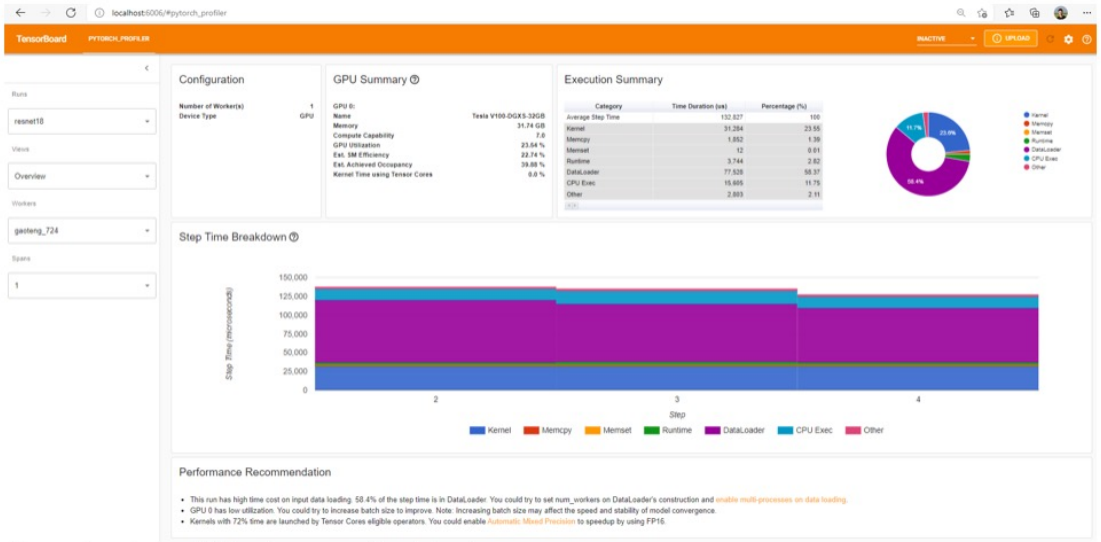
https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html

Tutorials > PyTorch Profiler With TensorBoard

```
http://localhost:6006/#pytorch_profiler
```

You could see Profiler plugin page as shown below.

- Overview



The overview shows a high-level summary of model performance.

The "GPU Summary" panel shows the GPU configuration, GPU usage and Tensor Cores usage. In this example, the GPU Utilization is low. The details of these metrics are here.

The "Step Time Breakdown" shows distribution of time spent in each step over different categories of execution. In this example, you can see the `DataLoader` overhead is significant.

The bottom "Performance Recommendation" uses the profiling data to automatically highlight likely bottlenecks, and gives you

CUDA Profiling and Debugging

CUDA-MEMCHECK

- Memory debugging tool
  - No recompilation necessary

```
$ cuda-memcheck ./exe
```

- Can detect the following errors
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- For line numbers use the following compiler flags:

```
-Xcompiler-rdynamic-lineinfo
```

NVPROF

- Command Line Profiler
  - Compute time in each kernel
  - Compute memory transfer time
  - Collect metrics and events
  - Support complex process hierarchy's
  - Collect profiles for NVIDIA Visual Profiler
  - No need to recompile

https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

CUDA-GDB (like gdb)

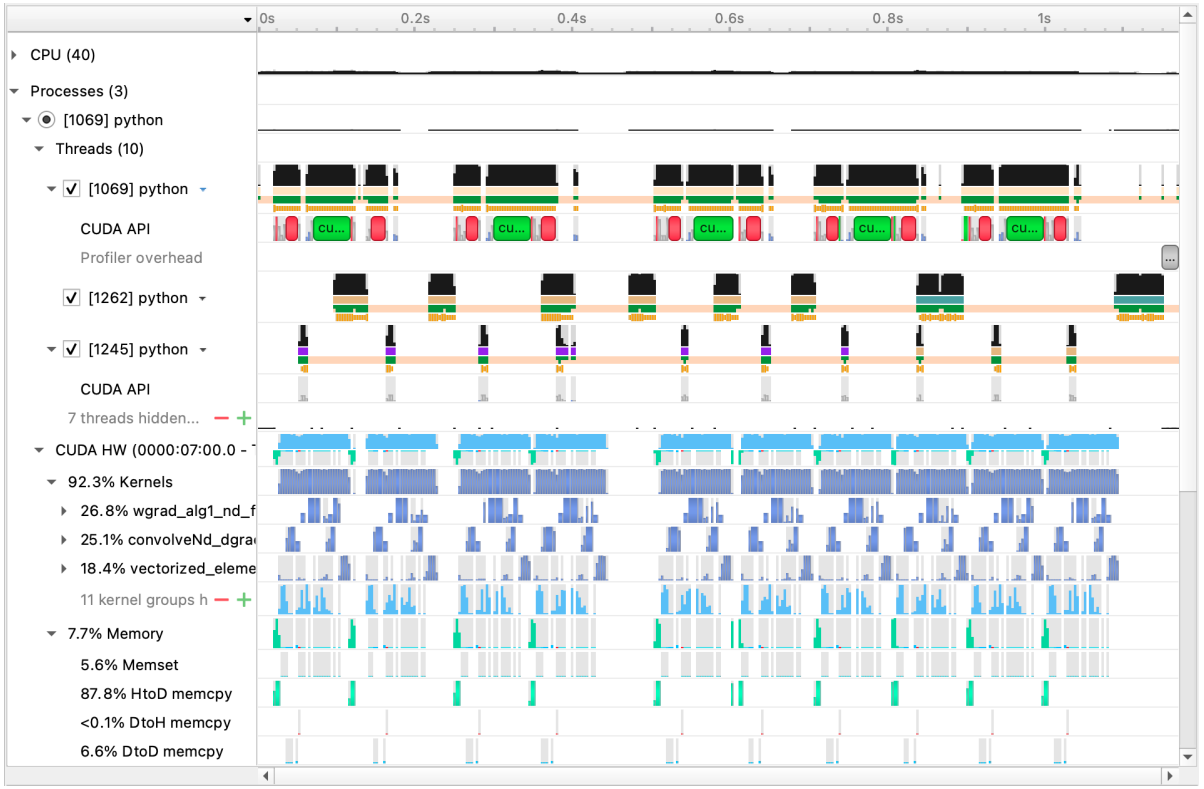
- *cuda-gdb* is an extension of GDB
  - Provides seamless debugging of CUDA and CPU code

```
$ cuda-gdb --args ./exe
```

  - Step-by-step execution, Breakpoints, etc.
- Works on Linux and Macintosh
  - For a Windows debugger use NSIGHT Visual Studio Edition
- http://docs.nvidia.com/cuda/cuda-gdb

Profiling Code

Using NVIDIA Nsight Systems



Using a profiler is an essential step in optimizing any code

Nsight Systems timeline provides a high-level view of your workload and helps you identify bottlenecks:

- I/O, data input pipeline
- Compute
- Scheduling (e.g. unexpected synchronization)

To generate a profile:

```
nsys profile –o myprofile python train.py
```

```
nsys profile –o myprofile -t cuda,nvtx python train.py
```

Profiling Code

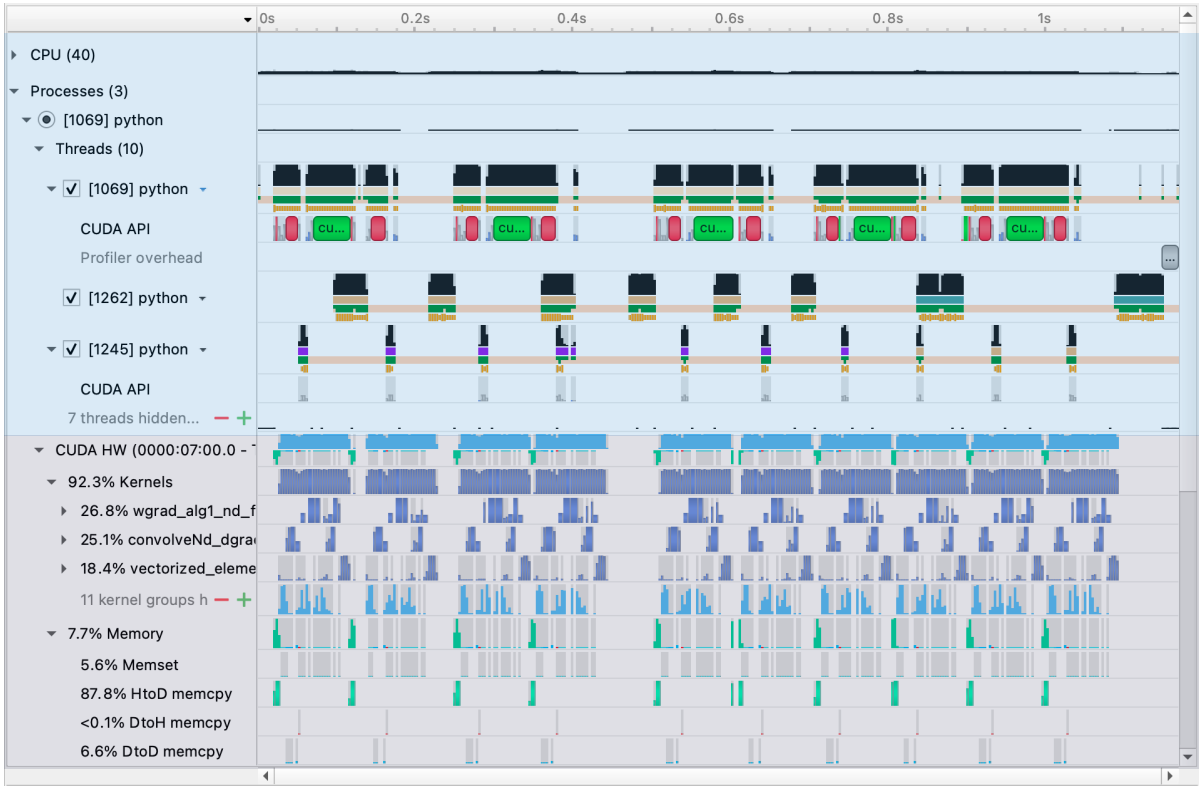
Using NVIDIA Nsight Systems

On the right is a sample of the Nsight Systems GUI view of a profile

Nsight Systems can show information about:

- CUDA API calls on CPU threads
- OS calls (if OSRT profiling is enabled)
- GPU kernels and memcopies running on the GPU

Provides tons of information, but can benefit from adding additional context



Profiling Code

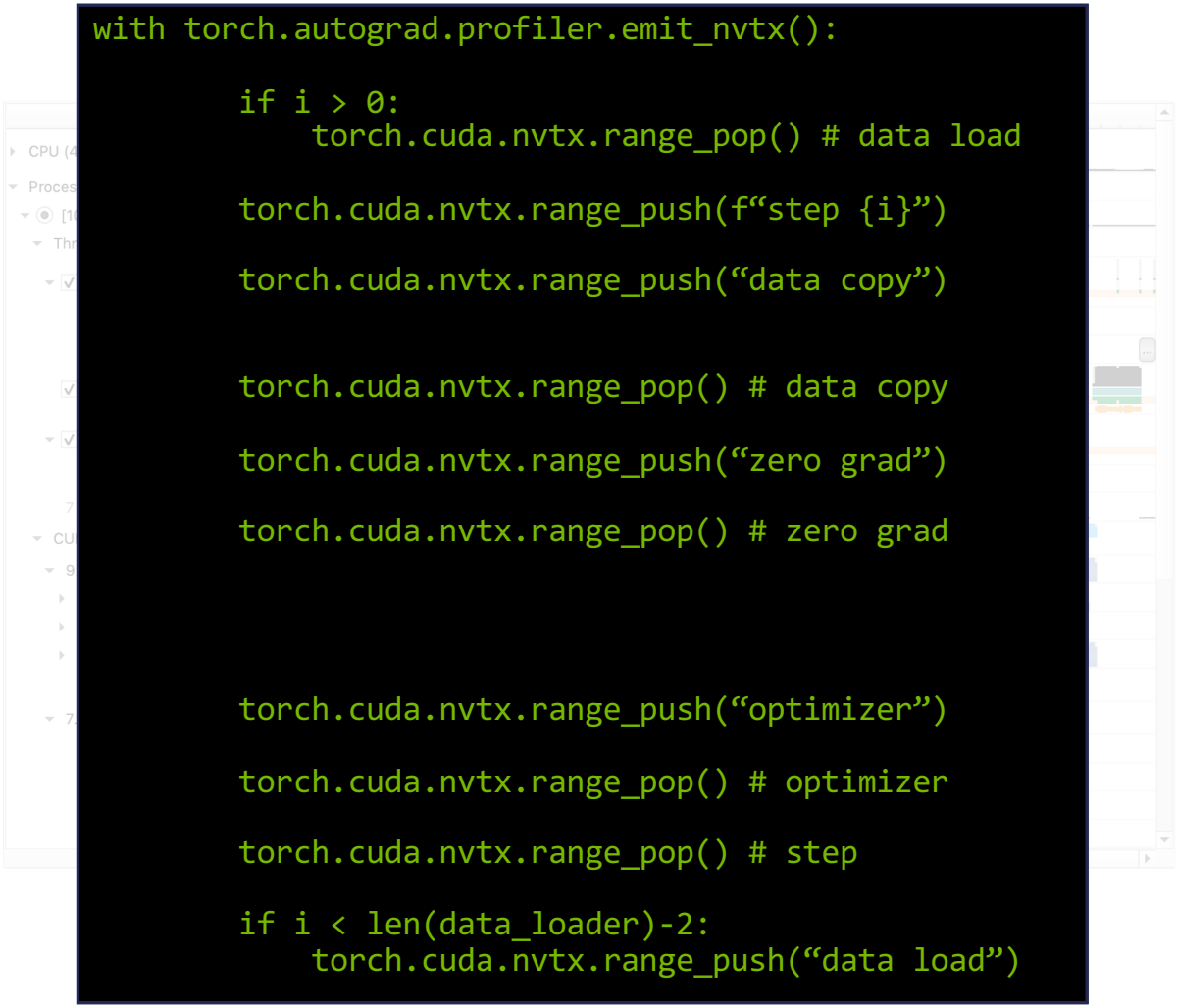
Adding NVTX Annotations

Can use NVTX ranges to annotate profiles:

```
torch.cuda.nvtx.range_push(“foo”)
torch.cuda.nvtx.range_pop()
torch.autograd.profiler.emit_nvtx()
```

NVTX ranges can be used to annotate:

- large general code regions (training step/epoch, file I/O, etc)
- targeted code locations suspected of leading to GPU idle time



```
with torch.autograd.profiler.emit_nvtx():

        if i > 0:
            torch.cuda.nvtx.range_pop() # data load

        torch.cuda.nvtx.range_push(f“step {i}”)

        torch.cuda.nvtx.range_push(“data copy”)


        torch.cuda.nvtx.range_pop() # data copy

        torch.cuda.nvtx.range_push(“zero grad”)

        torch.cuda.nvtx.range_pop() # zero grad




        torch.cuda.nvtx.range_push(“optimizer”)

        torch.cuda.nvtx.range_pop() # optimizer

        torch.cuda.nvtx.range_pop() # step

        if i < len(data_loader)-2:
            torch.cuda.nvtx.range_push(“data load”)
```

Profiling Code

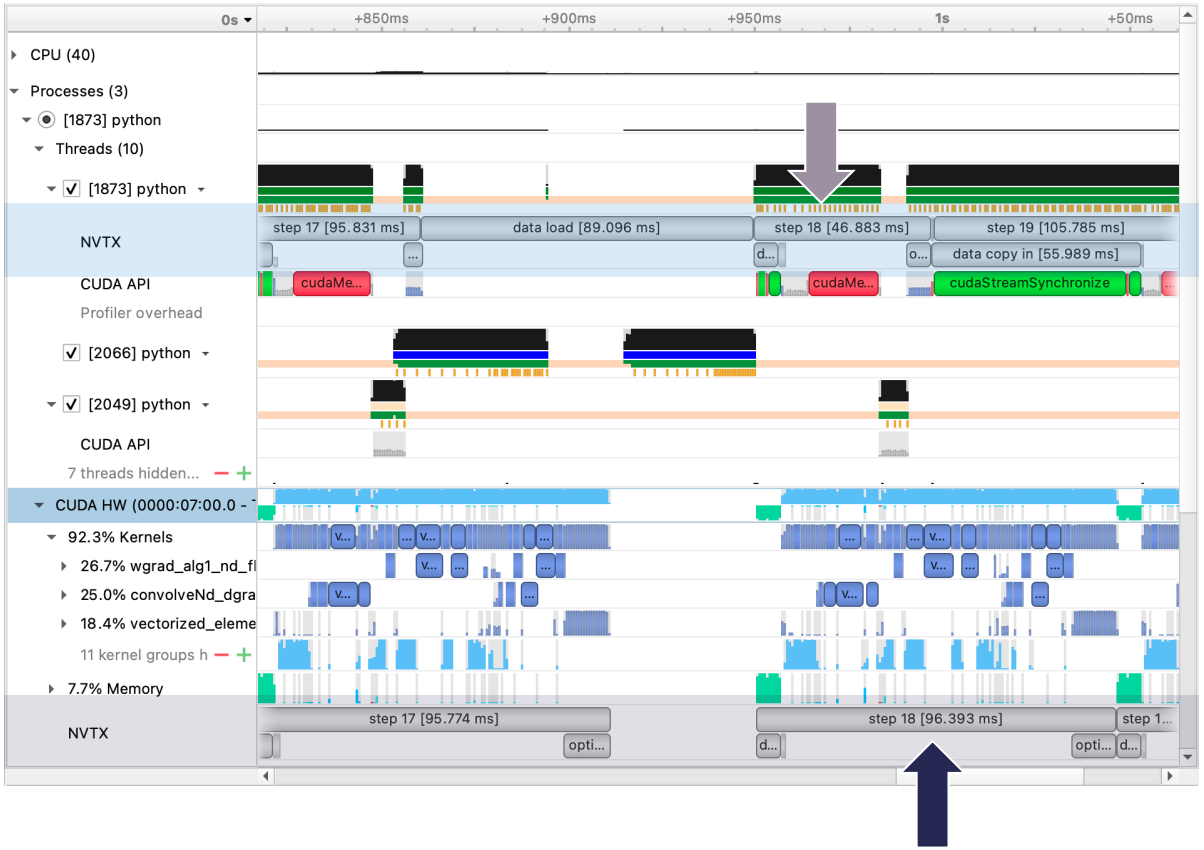
Profile Example with NVTX

Here is what the profile looks like with NVTX ranges added.

With these ranges annotated, there is a lot more context, enabling you to better focus your optimization efforts.

Ranges under CPU threads span CPU activity only (API calls/kernel launches, Python work)

Ranges under GPU span actual kernel and memcopy durations



Profiling Code

Profile Example with NVTX

With profiling in place, you can start addressing observed issues and track performance improvements.

Resources

- NVIDIA NSight Systems user guide (link)
- NVTX Documentation (link)
- Video tutorial: https://www.youtube.com/watch?v=kKANP0kL_hk
- Github with examples: https://github.com/NVIDIA/nsight-training/tree/master
- https://github.com/openhackathons-org/AI-Profiler/tree/main

Other Profiling Tools

Using *profile/cProfile*

- From your program:
  - Example profiling a regular expression

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

- To profile a script:

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

- By default a summary is provided, using *pstats*
- By specifying an *output_file* the profile can be processed afterwards
- https://docs.python.org/3/library/profile.html

Profiling a PyTorch neural network

- Consider this NN example: a two-layers network with ReLU activation

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

loss_fn = torch.nn.MSELoss(size_average=False)
learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)

    loss = loss_fn(y_pred, y)
    print(t, loss.data[0])

    model.zero_grad()

    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
def main():
    x, y, model, loss_fn = setup(N, D_in, H, D_out)
    learn(x, y, model, loss_fn)

if __name__ == "__main__":
    main()
```

Profiling a PyTorch neural network

- Profile (partial) collected with:

```
python -m cProfile -s time nn.py
```

- Output:

```
326044 function calls (313968 primitive calls) in 1.073 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      500    0.249    0.000    0.249    0.000 {method 'run_backward' of 'torch._C._EngineBase' objects}
     1000    0.189    0.000    0.189    0.000 {built-in method torch._C.addmm}
    27/26    0.081    0.003    0.084    0.003 {built-in method _imp.create_dynamic}
      261    0.054    0.000    0.057    0.000 <frozen importlib._bootstrap_external>:830(get_data)
     1386    0.037    0.000    0.037    0.000 {built-in method posix.stat}
        3    0.026    0.009    0.063    0.021 utils.py:61(parse_header)
      261    0.025    0.000    0.025    0.000 {built-in method marshal.loads}
12000/5000   0.024    0.000    0.039    0.000 module.py:513(named_parameters)
     2000    0.023    0.000    0.023    0.000 {method 'mul' of 'torch._C.FloatTensorBase' objects}
  801/798    0.021    0.000    0.033    0.000 {built-in method builtins.__build_class__}
        1    0.021    0.021    1.073    1.073 nn.py:1(<module>)
```

Snakeviz

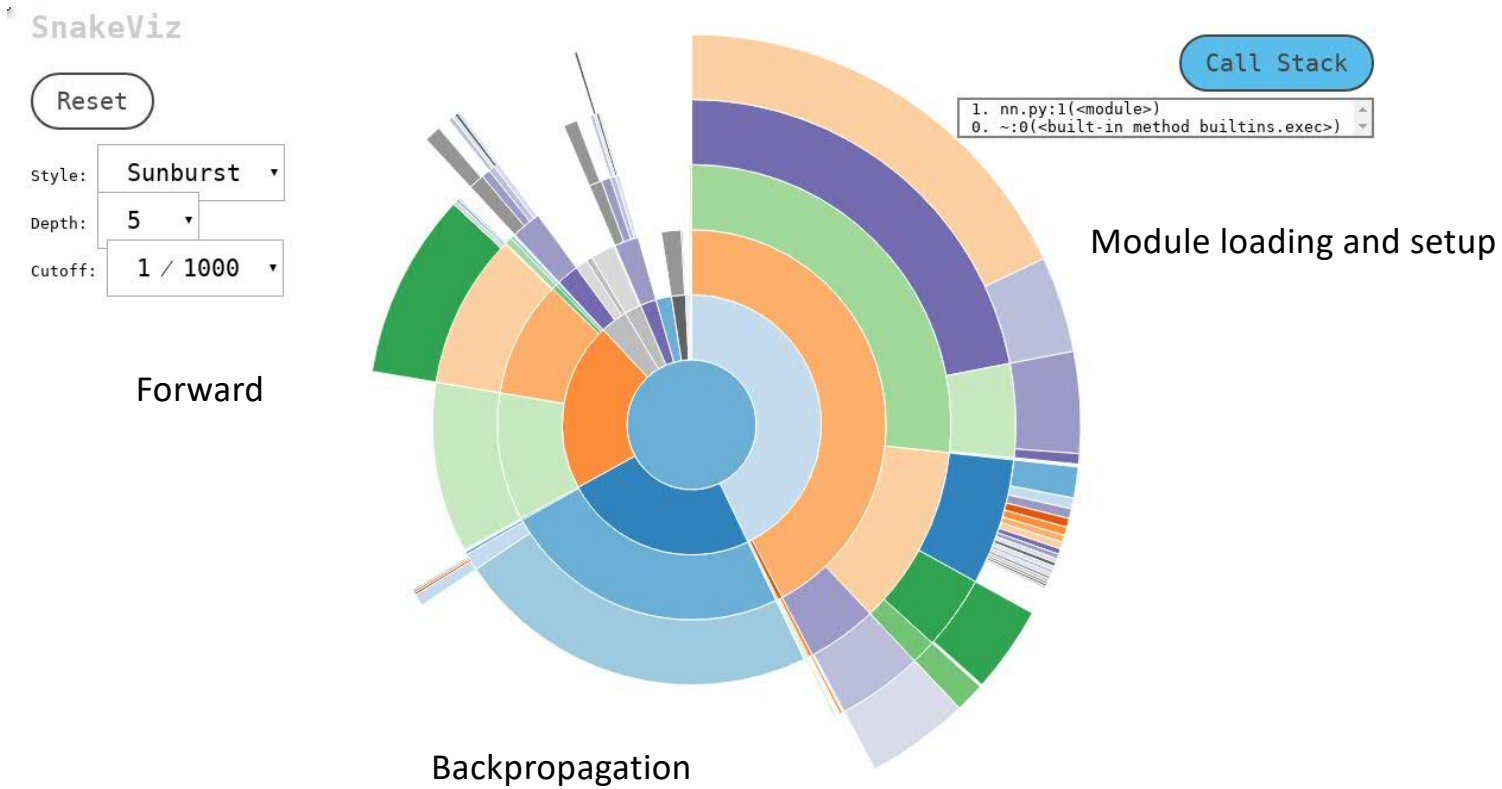
- Graphical tool to visualize content of profile created with cProfile
- Need to use the profile output file on your laptop
- Usage:

```
$ snakeviz nn.profile --server
snakeviz web server started on 127.0.0.1:8080; enter Ctrl-C to exit
http://127.0.0.1:8080/snakeviz/%2Fcygdrive%2Fc%2FUsers%2FAlessandroMORARI%2FBox+Sync%2Fwork%2Fcygwin%2FAlessandroMORARI%2Fnn.profile
```

- Then open the browser and connect to URL provided….

- https://jiffyclub.github.io/snakeviz/

SnakeViz and sunburst plots

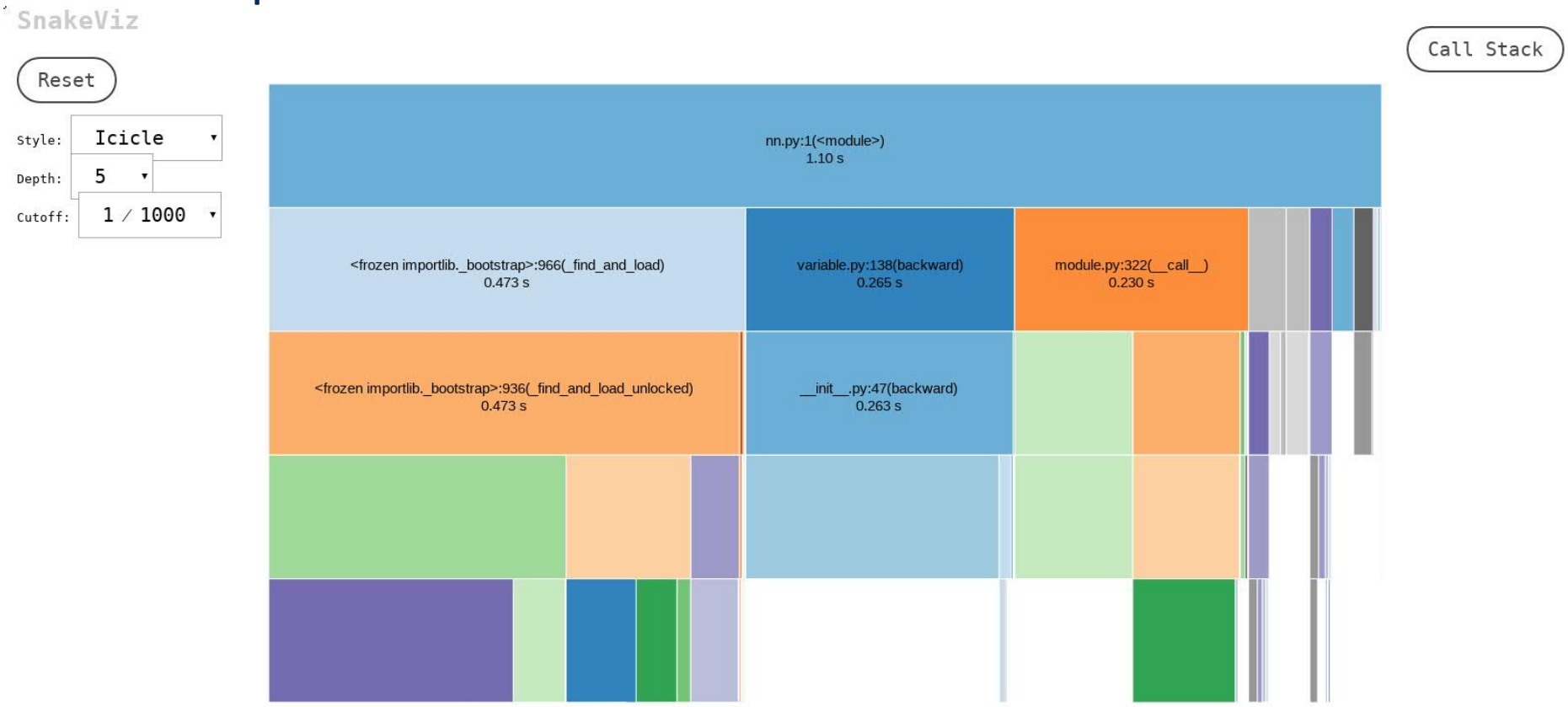


Forward

Module loading and setup

Backpropagation

Icicle plots



Profiling memory usage

- Important to determine whether:
  - Allocations can be avoided in the critical paths (e.g. by reusing allocated memory)
  - The overall memory usage is too high and limits the problem size that can be solved
- https://pypi.python.org/pypi/memory_profiler
- Usage:
  - `python -m memory_profiler nn.py`

Example

- In the following example, we create a simple function `my_func` that allocates lists `a`, `b` and then deletes `b`:

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

```
$ python -m memory_profiler example.py
```

Example

- Execute the code passing the option `-m memory_profiler` to the python interpreter to load the memory_profiler module and print to stdout the line-by-line analysis. If the file name was example.py

```
$ python -m memory_profiler example.py
```

- this would result in:

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     3   38.816 MiB   38.816 MiB           1   @profile
     4                                         def my_func():
     5   46.492 MiB    7.676 MiB           1       a = [1] * (10 ** 6)
     6  199.117 MiB  152.625 MiB           1       b = [2] * (2 * 10 ** 7)
     7   46.629 MiB -152.488 MiB           1       del b
     8   46.629 MiB    0.000 MiB           1       return a
```

1st column: line number of the code that has been profiled

2nd column: memory usage of the Python interpreter after that line has been executed

3rd column: difference in memory of the current line with respect to the last one

Another Example: Output of *memory_profiler*

```
Filename: nn.py

Line #    Mem usage    Increment   Line Contents
================================================
    20   85.031 MiB   85.031 MiB   @profile
    21                             def learn(x, y, model, loss_fn):
    22   85.031 MiB    0.000 MiB       learning_rate = 1e-4
    23   91.242 MiB    0.000 MiB       for t in range(500):
    24   91.242 MiB    2.184 MiB           y_pred = model(x)
    25
    26   91.242 MiB    0.047 MiB           loss = loss_fn(y_pred, y)
    27   91.242 MiB    0.164 MiB           print(t, loss.data[0])
    28
    29   91.242 MiB    0.000 MiB           model.zero_grad()
    30
    31   91.242 MiB    3.242 MiB           loss.backward()
    32
    33   91.242 MiB    0.000 MiB           for param in model.parameters():
    34   91.242 MiB    0.574 MiB               param.data -= learning_rate * param.grad.data
```

PyTorch Performance - Benchmarking

Profiling vs. benchmarking

- In benchmarking we're interested in assessing the **absolute speed** of a piece of code
- Profiling results are not reliable for absolute values
  - Profiling introduces **overhead**
  - C++ and Python sections of the application are affected by profiling in a different way, depending on the profiling tools being used
- When benchmarking, variability has to be taken into account:
  - Dependencies on the input
  - Dependencies on temporary conditions
    - Always collect stats on multiple executions

Benchmarking: the *timeit* module

- The *timeit* module deals with many of the requirements of benchmarking
- Execute the code in a loop, and take the best of multiple runs
- Using from the command line
  - example (timing a matrix multiply in numpy, 5 runs of 20 iterations each):

```
% python3 -m timeit -v -n 20 -r 5 -s "import numpy; x=numpy.random.rand(1000, 1000)" "x=x.dot(x)"

raw times: 210 msec, 217 msec, 184 msec, 199 msec, 211 msec

20 loops, best of 5: 9.19 msec per loop
```

The *timeit* module

- From Python code:

```
import timeit

t = timeit.Timer("x = x.dot(x)",
                 setup = "import numpy; x = numpy.random.rand(1000, 1000)").repeat(number=20, repeat=10)

print("raw times: {0}\nbest of 5: {1:.0f} ms".format(" ".join([ "%.3f" % x for x in t ]), min(t)/20*1000))
```

  - Output:

```
% python3 test_timit.py
  raw times: 0.209 0.228 0.183 0.170 0.179 0.174 0.211 0.191 0.170 0.169
  best of 10: 8 ms
```