presentation

September 11, 2024

[1]:
```python
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

1 Debugging and Optimization of PyTorch Models - PyTorch Profiler

1.0.1 Collin Wilson - University of Guelph

2 Outline

- Why you should profile and optimize PyTorch code
- When should you profile/optimize?
- PyTorch Profiler
  - Basic profiling
  - Generating and analysing traces
  - Memory profiling
- Holistic trace analysis
- Q&A

3 Why you should profile and optimize PyTorch code

- ## 3.1 Profiling allows you to understand exactly how your model is running:
  - how data is flowing between main memory and the GPU
  - identify performace bottlenecks
  - identify bugs - bad data flows, OOM errors etc.
- ## 3.2 Increase efficiency
  - greater research throughput
  - make full use of allocated resources

4 When should you profile/optimize?

-

4.1 There are many reasons to profile early:

  -

4.1.1 Profiling can help with debugging - memory issues, bad gradient flows etc

  -

4.1.2 Profiling can help with estimating required resources

  -

4.1.3 Identifying bottlenecks outside of your training loop

-

4.2 Once you have a working training loop, you can start thinking about optimization

  -

4.2.1 Full run efficiency

  -

4.2.2 Preparation for hyperparameter tuning

-

4.3 Are you going to be modifying model operations?

5 PyTorch Profiler

-

5.0.1 Built-in module of Pytorch

-

5.0.2 Can profile run times, memory and generate detailed traces of PyTorch code

  -

5.0.3 see layer-by-layer breakdown of model performance

-

5.0.4 Both CPU and GPU profiling

-

5.0.5 Code modification required - wrap code with a context manager

-

5.0.6 Works for NVIDIA and AMD GPUS

[2]:
```
import torch
from torchvision.datasets import ImageNet
from torchvision.models import resnet50
from torchvision import transforms
from torch.profiler import profile, record_function, ProfilerActivity, schedule

model = resnet50()
inputs = torch.randn(1, 3, 224, 224)
with profile(activities=[ProfilerActivity.CPU], record_shapes=True) as prof:
    with record_function("model_inference"):
        model(inputs)
```

6 Pytorch Profiler - simple example

[3]:
```
print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
```

```
---------------------------------  ------------  ------------  ------------
------------  ------------  ------------
                             Name    Self CPU %      Self CPU   CPU total %
CPU total  CPU time avg    # of Calls
---------------------------------  ------------  ------------  ------------
------------  ------------  ------------
                  model_inference        13.21%       7.756ms       100.00%
58.730ms      58.730ms             1
                     aten::conv2d         1.13%     663.999us        46.36%
27.227ms     513.724us            53
                aten::convolution         2.11%       1.238ms        45.23%
26.563ms     501.196us            53
               aten::_convolution         1.66%     977.708us        43.12%
25.325ms     477.829us            53
                aten::thnn_conv2d         0.69%     404.246us        40.85%
23.993ms     452.707us            53
       aten::_slow_conv2d_forward        39.11%      22.968ms        40.17%
23.589ms     445.080us            53
                 aten::batch_norm         0.54%     315.043us        17.48%
10.266ms     193.690us            53
     aten::_batch_norm_impl_index         0.23%     134.992us        16.94%
9.951ms     187.746us            53
```

```
          aten::native_batch_norm        16.00%       9.400ms        16.69%
9.801ms     184.925us            53
        aten::adaptive_avg_pool2d         0.71%     414.834us         7.90%
4.637ms       4.637ms             1
---------------------------------  ------------  ------------  ------------
------------  ------------  ------------
Self CPU time total: 58.730ms
```

7 Pytorch Profiler - simple example

[4]:
```python
print(prof.key_averages(group_by_input_shape=True).
  ↪table(sort_by="cpu_time_total", row_limit=10))
```

```
---------------------------------  ------------  ------------  ------------
------------  ------------  ------------
--------------------------------------------------------------------------------
                             Name    Self CPU %      Self CPU   CPU total %
CPU total  CPU time avg    # of Calls
Input Shapes
---------------------------------  ------------  ------------  ------------
------------  ------------  ------------
--------------------------------------------------------------------------------
                  model_inference        13.21%       7.756ms       100.00%
58.730ms      58.730ms             1
[]
                     aten::conv2d         1.05%     615.876us        13.05%
7.665ms       7.665ms             1                            [[1, 3, 224,
224], [64, 3, 7, 7], [], [], [], [], []]
                aten::convolution         1.86%       1.094ms        12.00%
7.049ms       7.049ms             1                    [[1, 3, 224, 224], [64,
3, 7, 7], [], [], [], [], [], [], []]
               aten::_convolution         1.51%     885.707us        10.14%
5.955ms       5.955ms             1     [[1, 3, 224, 224], [64, 3, 7, 7], [],
[], [], [], [], [], [], [], [], [], []]
                aten::thnn_conv2d         0.63%     369.959us         8.04%
4.719ms       4.719ms             1                                [[1, 3, 224,
224], [64, 3, 7, 7], [], [], [], []]
        aten::adaptive_avg_pool2d         0.71%     414.834us         7.90%
4.637ms       4.637ms             1
[[1, 2048, 7, 7], []]
       aten::_slow_conv2d_forward         6.51%       3.825ms         7.41%
4.349ms       4.349ms             1                                [[1, 3, 224,
224], [64, 3, 7, 7], [], [], [], []]
                       aten::mean         2.13%       1.248ms         7.19%
4.222ms       4.222ms             1
[[1, 2048, 7, 7], [], [], []]
```

```
                    aten::conv2d         0.01%       8.376us         5.99%
3.520ms       1.173ms             3                          [[1, 64, 56,
56], [64, 64, 3, 3], [], [], [], [], []]
               aten::convolution         0.02%      11.208us         5.98%
3.511ms       1.170ms             3                   [[1, 64, 56, 56], [64,
64, 3, 3], [], [], [], [], [], [], []]
--------------------------------  ------------  ------------  ------------
------------  ------------  ------------
--------------------------------------------------------------------------------
Self CPU time total: 58.730ms
```

8 Basic profiling

8.1 Resnet50 on ImageNet

- Validation with pretrained weights
- batch size 1024 on A100 4g 20GB MIG (>16GB CUDA memory usage)

```
import os
import argparse
import torch
from tqdm import tqdm
from torchvision.datasets import ImageNet
from torchvision.models import resnet50
from torchvision import transforms
from torch.profiler import profile, record_function, ProfilerActivity, schedule

def main():

    # Command line arguments
    parser = argparse.ArgumentParser(
                    prog='ResNet50 ImageNet',
                    description='Trains ResNet50 on the ImageNet ILSVRC2012 dataset.',
                    epilog='Text at the bottom of help')
    parser.add_argument('batch_size', type=int, default=64,
                        help='Batch size used for inference.')
    parser.add_argument('--nWorkers', type=int, default=1,
                        help='number of cores to use in data loading')
    parser.add_argument('--compile', action='store_true',
                        help='Compile the model before evaluation.')

    args = parser.parse_args()

    # Load data
    slurm_tmpidir = os.environ['SLURM_TMPDIR']
    image_net_path = os.path.join(slurm_tmpidir, 'imagenet')
```

```python
mean = (0.485, 0.456, 0.406)  # standard data transformations
std = (0.229, 0.224, 0.225)
val_transform = transforms.Compose(
            [
                transforms.Resize(256),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize(mean, std),
            ]
        )

imagenet_val_data = ImageNet(image_net_path, split='val', transform=val_transform)
val_dataloader = torch.utils.data.DataLoader(imagenet_val_data,
                                             batch_size=args.batch_size,
                                             shuffle=True,
                                             num_workers=args.nWorkers)

# Initialize model
model = resnet50(weights="IMAGENET1K_V2")
if args.compile:
    model = torch.compile(model) # optionally compile model

model.eval().cuda() # move model to GPU

# Make profiler schedule
my_schedule = schedule(
skip_first=10,
wait=5,
warmup=5,
active=5,
repeat=1)

correct = 0
total = 0
with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
             profile_memory=True,
             record_shapes=True,
             with_stack=True,
             schedule=my_schedule) as prof:

    # Validation training loop
    with torch.no_grad():
        for x, y in tqdm(val_dataloader):
            with record_function('model_inference'):
                y_pred = model(x.cuda())
            correct += (y_pred.argmax(axis=1) == y.cuda()).sum().item()
            total += len(y)
            prof.step()
```

```
    # print profiles
    print("GPU profiles:")
    print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10))
    print(prof.key_averages(group_by_input_shape=True).table(sort_by="cuda_time_total",
                                                             row_limit=10))

    print("\nCPU profile:")
    print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
    print("\nCPU memory profile:")
    print(prof.key_averages().table(sort_by="self_cpu_memory_usage", row_limit=10))

    compiled = '_compiled' if args.compile else ''
    prof.export_chrome_trace(
        f'/home/c7wilson/project/pytorch_profiler_talk/traces/'
        + f'basic_inference_batch_{args.batch_size}_nworkers_{args.nWorkers}{compiled}_trace.j
    prof.export_memory_timeline(f'/home/c7wilson/project/pytorch_profiler_talk/traces/'
        + f'basic_inference_batch_{args.batch_size}_nworkers_{args.nWorkers}{compiled}_memory'
        + '.html',
        device='cuda:0')


    print(correct / total)

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

9 Basic profiling - schedule

- For long running jobs, the profiles/traces become very large
- First iteration profiles after initializing the profile may be skewed - profiler needs a warmup
- "Profiler skips the first `skip_first` steps, then wait for `wait` steps, then do the warmup for the next `warmup` steps, then do the active recording for the next `active` steps and then repeat the cycle starting with `wait` steps. The optional number of cycles is specified with the `repeat` parameter, the zero value means that the cycles will continue until the profiling is finished." `torch.profiler.schedule` docs

```
my_schedule = schedule(
    skip_first=10,
    wait=5,
    warmup=5,
    active=5,
    repeat=1)
```

10 Basic profiling - `record_function`

- Using `record_function` in a context manager, we can label different parts of the code:

```python
with torch.no_grad():
    for x, y in tqdm(val_dataloader):
        with record_function('model_inference'):
            y_pred = model(x.cuda())
        correct += (y_pred.argmax(axis=1) == y.cuda()).sum().item()
        total += len(y)
        prof.step()    # advances the profiler

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

```
---------------------------------  ------------  ------------  ------------  ------------  ---
                             Name     Self CUDA   Self CUDA %    CUDA total  CUDA time avg   C
---------------------------------  ------------  ------------  ------------  ------------  ---
                  model_inference        3.553s        50.02%        3.553s     710.665ms
                    ProfilerStep*       0.000us         0.00%        3.549s     709.756ms
                  model_inference       0.000us         0.00%        3.549s     709.734ms
                aten::convolution       0.000us         0.00%        1.288s       4.861ms    2
               aten::_convolution       0.000us         0.00%        1.288s       4.861ms    2
          aten::cudnn_convolution        1.288s        18.14%        1.288s       4.861ms    2
                     aten::conv2d       0.000us         0.00%        1.190s       4.490ms    2
         aten::cudnn_batch_norm       735.817ms        10.36%     735.817ms       2.777ms    2
void cudnn::bn_fw_inf_1C11_kernel     735.817ms        10.36%     735.817ms       2.777ms
     aten::_batch_norm_impl_index       0.000us         0.00%     731.277ms       2.760ms    2
---------------------------------  ------------  ------------  ------------  ------------  ---
Self CPU time total: 16.965s
Self CUDA time total: 7.103s
```

11 Generating and analysing chrome traces

- The tabular summaries can tell us some things, but are quite basic
- Pytorch profiler can be used to generate traces in a `.json` format readable by Chrome at `chrome://tracing`

```python
prof.export_chrome_trace('/path/to/traces/trace.json')
```

12 Generating and analysing chrome traces - Case Study 1

12.1 GPU Idle time

12.1.1 Solution:

Request more cores and use parallelism in your `Dataloader`:

```python
val_dataloader = torch.utils.data.DataLoader(imagenet_val_data,
                                             batch_size=args.batch_size,
                                             shuffle=True,
                                             num_workers=args.nWorkers) # <----
```

13 Generating and analysing chrome traces - Case Study 2

13.1 Strange idle patterns

While investigating the effect of `num_workers` on data loading, we caught a strange pause in training.

14 Generating and analysing chrome traces - Case Study 3

14.1 Easy optimization with `torch.compile`

- Compilation with `torch.compile` is an extremely simple optimization

```
model = resnet50()
optimized_model = torch.compile(model)
```

- Can compile single models, user defined functions, even the optimizer (beta):

```
opt = torch.optim.Adam(model.parameters(), lr=0.01)

@torch.compile(fullgraph=False)
def fn():
    opt.step()
```

15 Memory profiling

- Requires `profile_memory` and `with_stack` to be enabled in your profiler:

```
with profile(activities=[ProfilerActivity.CUDA],
                     profile_memory=True,
                     with_stack=True,
                     schedule=my_schedule) as prof:
```

- Pytorch profiler provides tools for exporting memory plots:

```
# exports an html file containg a .png of memory usage
prof.export_memory_timeline('path/to/figure.html', device='cuda:0')
```

16 Memory profiling

- Requires `profile_memory` and `with_stack` to be enabled in your profiler:

```
with profile(activities=[ProfilerActivity.CUDA],
                     profile_memory=True,
                     with_stack=True,
                     schedule=my_schedule) as prof:
```

- Can also specify a .json extension to export a JSON file, which you can parse and plot yourself

```
prof.export_memory_timeline('path/to/memory_trace.json', device='cuda:0')
```

17 Memory profiling - training on ImageNet

- Traced a resnet50 training loop with batch size 128, using SGD

18 Memory profiling - training on ImageNet

- Traced a resnet50 training loop with batch size 128, using SGD

18.0.1 Solution:

optimizer.zero_grad()

19 Holistic trace analysis

- ### 19.0.1 Performance and visualization library for PyTorch

- ### 19.0.2 Uses traces from PyTorch profiler to provide more detailed analysis

- ### 19.0.3 HTA provides several features, in this talk we'll focus on three

  1. Temporal Breakdown
  2. Idle Time Breakdown
  3. Trace Diff

20 Holistic trace analysis

- Performance and visualization library for PyTorch

[5]:
```
from hta.trace_analysis import TraceAnalysis
analyzer = TraceAnalysis(trace_dir='traces/hta/')
```

```
2024-09-11 11:21:24,741 - hta - trace.py:L389 - INFO - traces/hta/
2024-09-11 11:21:24,742 - hta - trace_file.py:L94 - INFO - Rank to trace file
map:
{2: 'traces/hta/ng30403.narval.calcul.quebec_1532931.1726011714978383209.pt.trac
e.json', 0: 'traces/hta/ng30403.narval.calcul.quebec_1525846.1726011534744368289
.pt.trace.json', 1: 'traces/hta/ng30403.narval.calcul.quebec_1531011.17260116482
24348245.pt.trace.json'}
2024-09-11 11:21:24,742 - hta - trace.py:L535 - INFO - ranks=[0, 1, 2]
2024-09-11 11:21:24,919 - hta - trace.py:L118 - INFO - Parsed traces/hta/ng30403
.narval.calcul.quebec_1532931.1726011714978383209.pt.trace.json time = 0.16
seconds
2024-09-11 11:21:24,932 - hta - trace.py:L118 - INFO - Parsed traces/hta/ng30403
```

```
.narval.calcul.quebec_1531011.1726011648224348245.pt.trace.json time = 0.17
seconds
2024-09-11 11:21:24,935 - hta - trace.py:L118 - INFO - Parsed traces/hta/ng30403
.narval.calcul.quebec_1525846.1726011534744368289.pt.trace.json time = 0.17
seconds
```

21 Holistic trace analysis - gathering traces

- To export usable traces for HTA, need to use the Tesnorboard Trace handler:

```
trace_handler = tensorboard_trace_handler(dir_name='/path/to/traces/',
                                          use_gzip=True)
with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
             profile_memory=True,
             record_shapes=True,
             with_stack=True,
             schedule=my_schedule,
             on_trace_ready=trace_handler) as prof:
```

- If you are not running a distributed job, you will have to mannually edit the trace.json files to include `"distributedInfo": {"rank": 0}`, using a different rank for each file (or a separate directory)

22 Holistic trace analysis - temporal breakdown

[6]:
```
time_spent_df = analyzer.get_temporal_breakdown(visualize=True)
time_spent_df   # Rank 0: serial, Rank 1: 6 workers, Rank 2: 6 workers, compiled
```

[6]:
```
   rank  idle_time(us)  compute_time(us)  non_compute_time(us)  \
0     0       10337590           3149452                355723
1     1        1328410           3149392                414529
2     2        1528898           1952855                443521

   kernel_time(us)  idle_time_pctg  compute_time_pctg  non_compute_time_pctg
0         13842765           74.68              22.75                   2.57
1          4892331           27.15              64.37                   8.47
2          3925274           38.95              49.75                  11.30
```

23 Holistic trace analysis - idle time breakdown

-

23.0.1 Host wait: is the idle duration on the GPU due to the CPU not enqueuing kernels fast enough to keep the GPU busy.

  - examine what cpu processes are causing the slowdown
  - increasing the batch size

-

23.0.2 Kernel wait: constitutes the short overhead to launch consecutive kernels on the GPU

  - use CUDA Graph optimizations.

-

23.0.3 Other wait: idle that could not be attributed due to insufficient information.

[7]:
```
idle_time_df = analyzer.get_idle_time_breakdown(ranks=[0,1,2])
idle_time_df
```

```
2024-09-11 11:26:06,295 - hta - breakdown_analysis.py:L433 - INFO - Processing
stream 7

2024-09-11 11:26:06,349 - hta - breakdown_analysis.py:L433 - INFO - Processing
stream 7

2024-09-11 11:26:06,376 - hta - breakdown_analysis.py:L433 - INFO - Processing
stream 7
```

[7]:
```
(   rank stream idle_category   idle_time  idle_time_ratio
 0     0      7     host_wait  10335607.0              1.0
 1     0      7   kernel_wait      1983.0              0.0
 2     0      7         other         0.0              0.0
 0     1      7     host_wait   1326416.0              1.0
 1     1      7   kernel_wait      1994.0              0.0
 2     1      7         other         0.0              0.0
 0     2      7     host_wait   1527584.0              1.0
 1     2      7   kernel_wait      1314.0              0.0
 2     2      7         other         0.0              0.0,
 None)
```

24 Holistic trace analysis - trace diff

[8]:
```
from hta.trace_diff import TraceDiff

compare_traces_output = TraceDiff.compare_traces(control='traces/hta/control/',␣
 ↪# control: parallel
                                                  test='traces/hta/test/')       ␣
 ↪# test: parallel, compiled

df = compare_traces_output.sort_values(by="diff_counts", ascending=False).
 ↪head(10)
TraceDiff.visualize_counts_diff(df)
```

```
2024-09-11 11:27:18,366 - hta - trace.py:L389 - INFO - traces/hta/control/
2024-09-11 11:27:18,368 - hta - trace_file.py:L94 - INFO - Rank to trace file
map:
{1: 'traces/hta/control/ng30403.narval.calcul.quebec_1531011.1726011648224348245
.pt.trace.json'}
2024-09-11 11:27:18,368 - hta - trace.py:L535 - INFO - ranks=[1]
2024-09-11 11:27:18,526 - hta - trace.py:L118 - INFO - Parsed traces/hta/control
/ng30403.narval.calcul.quebec_1531011.1726011648224348245.pt.trace.json time =
0.16 seconds
2024-09-11 11:27:18,789 - hta - trace.py:L389 - INFO - traces/hta/test/
2024-09-11 11:27:18,790 - hta - trace_file.py:L94 - INFO - Rank to trace file
map:
{2: 'traces/hta/test/ng30403.narval.calcul.quebec_1532931.1726011714978383209.pt
.trace.json'}
2024-09-11 11:27:18,791 - hta - trace.py:L535 - INFO - ranks=[2]
2024-09-11 11:27:18,839 - hta - trace.py:L118 - INFO - Parsed traces/hta/test/ng
30403.narval.calcul.quebec_1532931.1726011714978383209.pt.trace.json time = 0.05
seconds
2024-09-11 11:27:19,079 - hta - trace_diff.py:L301 - INFO - comparing traces:
Control and Test
```

25 Holistic trace analysis - other features

-
  #### 25.0.1 Kernel Breakdown
-
  #### 25.0.2 Kernel Duration Distribution
-
  #### 25.0.3 Communication Computation Overlap.
-
  #### 25.0.4 CUDA Kernel Launch Statistics
-
  #### 25.0.5 Augmented Counters (Memory copy bandwidth, Queue length)
-
  #### 25.0.6 Frequent CUDA Kernel Patterns
-

25.0.7 CUPTI Counter Analysis

-

25.0.8 Lightweight Critical Path Analysis

Q&A

Please feel free to reach out to me at collin.wilson@sharcnet.ca

Example code and this presentation can be found on GitHub.

Also see the official documentation for PyTorch profiler and Holistic Trace Analysis