PyProf: Automating End-to-End PyTorch Profiling♯

https://github.com/dlacceleration/pyprof

Aditya Agrawal†, Marek Kolodziej††

♯ Work done at Nvidia

† Now at Google †† Now at Uber ATG

Acknowledgements

- Michael Carilli
- Alex Settle
- Carl Case
- Natalia Gimelshein
- Bryan Catanzaro
- Jie Jiang
- Andrew Huang
- Sandeep Behera
- Kevin Stephano

other early adopters.

About Us

Aditya is a computer architect and Deep Learning performance engineer. He analyzes and optimizes Deep Learning network performance on a variety of frameworks (PyTorch, TensorFlow etc.) and architectures (GPU, TPU etc.). He was part of the MLPerf team at Nvidia.

Marek is a Tech Lead Manager for GPU Systems on Uber ATG's Autonomy Team. He has a decade of experience as a machine learning engineer, accelerating distributed algorithms on heterogeneous clusters. While at Nvidia, he optimized deep learning framework backends (TF, MXNet, PyTorch) for training and inference on platforms ranging from data center (Tesla) to embedded (Tegra).

Outline

- Motivation & Tool Introduction.
- Basic usage.
- Advanced usage.
- Demo.

Challenges we faced as DL analysts

Start by reading a N page paper. If we are lucky,

- There is a block diagram with layer attributes, tensor shapes and datatype.
- The implementation is the same as the description.
- The network does not use other networks as submodules.

Current profilers e.g. NVprof and NSight Systems provide no information about

- Layer parameters, tensor shapes, data types.
- Call stack i.e. file name, line number.
- Direction e.g. fprop, bprop, loss, optimizer.
- Flops, bytes, tensor core usage per kernel.

What does a DL analyst want?

For any network, quickly obtain a table like this:

Layer	Direction	Call Trace	Op	Parameters	Kernel	Silicon Time	Thread Id	Device Id	Stream Id	Grid Dim	Block Dim	Flops	Bytes	TC
Self Attention	fprop	attn.py: 23, …	Linear	MNK, fp16	volta_s884…	200	23	1	7	x,y,z	x,y,z	1000	500	1
Block_1a	fprop	block.py: 43,	Conv	NCHWKPQRS, fp32	cudnn_…	130	23	1	7	x,y,z	x,y,z	2000	400	1
Hadamard	fprop	net.py: 73, …	mul	T=(128,256), fp16	pointwise…	110	23	1	7	x,y,z	x,y,z	10	2000	0

- Available from NVprof / Nsight Systems.
- PyProf: Intercept PyTorch calls and obtain the call trace, op and parameters.
- PyProf: Calculate direction, flops, bytes and Tensor Core (TC) usage.
- PyProf: User annotation (optional).

ResNet50

1	Layer	Direction	Op	Params	Kernel	Sil(ns)	FLOPs	Bytes	TC
2	conv1_x	fprop	conv2d	N=32,C=3,H=224,W=224,K=64,P=112,Q=112,R=7,S=7	cudnn::gemm::computeOffsetsKernel	2176	0	0	-
3	conv1_x	fprop	conv2d	N=32,C=3,H=224,W=224,K=64,P=112,Q=112,R=7,S=7	volta_fp16_scudnn_fp16_128x64_relu_medium	668991	7552892928	61032832	-
4	conv1_x	fprop	__add__	T=[(1,)],fp32,	elementwise_kernel	2848	1	8	-
5	conv1_x	fprop	batch_norm	T=(32,64,112,112),fp16,	batch_norm_collect_statistics_kernel	424863	205520896	205520896	-
6	conv1_x	fprop	batch_norm	T=(32,64,112,112),fp16,	batch_norm_transform_input_kernel	151360	205520896	0	-
7	conv1_x	fprop	relu	T=(32,64,112,112),fp16,	elementwise_kernel	174048	25690112	102760448	-
8	conv1_x	fprop	max_pool2d	T=[(32,64,112,112)],['float16']	max_pool_forward_nchw	189216	0	0	-
9	conv2_x:Bottleneck_1:Conv1	fprop	conv2d	N=32,C=64,H=56,W=56,K=64,P=56,Q=56,R=1,S=1,ph	cudnn::gemm::computeOffsetsKernel	1472	0	0	-
10	conv2_x:Bottleneck_1:Conv1	fprop	conv2d	N=32,C=64,H=56,W=56,K=64,P=56,Q=56,R=1,S=1,ph	volta_fp16_s884cudnn_fp16_256x128_ldg8_rel	85344	822083584	25698304	1
11	conv2_x:Bottleneck_1:BN1	fprop	__add__	T=[(1,)],fp32,	elementwise_kernel	1760	1	8	-
12	conv2_x:Bottleneck_1:BN1	fprop	batch_norm	T=(32,64,56,56),fp16,	batch_norm_collect_statistics_kernel	125984	51380224	51380224	-
13	conv2_x:Bottleneck_1:BN1	fprop	batch_norm	T=(32,64,56,56),fp16,	batch_norm_transform_input_kernel	39328	51380224	0	-
14	conv2_x:Bottleneck_1:ReLU	fprop	relu	T=(32,64,56,56),fp16,	elementwise_kernel	44736	6422528	25690112	-
15	conv2_x:Bottleneck_1:Conv2	fprop	conv2d	N=32,C=64,H=56,W=56,K=64,P=56,Q=56,R=3,S=3,ph	nchwToNhwcKernel	37664	0	0	-
16	conv2_x:Bottleneck_1:Conv2	fprop	conv2d	N=32,C=64,H=56,W=56,K=64,P=56,Q=56,R=3,S=3,ph	nchwToNhwcKernel	2784	0	0	-
17	conv2_x:Bottleneck_1:Conv2	fprop	conv2d	N=32,C=64,H=56,W=56,K=64,P=56,Q=56,R=3,S=3,ph	Volta_hmma_implicit_gemm_fprop_fp32_nhwc_	147967	0	0	-
18	conv2_x:Bottleneck_1:Conv2	fprop	conv2d	N=32,C=64,H=56,W=56,K=64,P=56,Q=56,R=3,S=3,ph	nhwcToNchwKernel	36192	0	0	-
19	conv2_x:Bottleneck_1:BN2	fprop	__add__	T=[(1,)],fp32,	elementwise_kernel	1760	1	8	-
20	conv2_x:Bottleneck_1:BN2	fprop	batch_norm	T=(32,64,56,56),fp16,	batch_norm_collect_statistics_kernel	124384	51380224	51380224	-
21	conv2_x:Bottleneck_1:BN2	fprop	batch_norm	T=(32,64,56,56),fp16,	batch_norm_transform_input_kernel	39136	51380224	0	-
22	conv2_x:Bottleneck_1:ReLU	fprop	relu	T=(32,64,56,56),fp16,	elementwise_kernel	44928	6422528	25690112	-
23	conv2_x:Bottleneck_1:Conv3	fprop	conv2d	N=32,C=64,H=56,W=56,K=256,P=56,Q=56,R=1,S=1,pl	cudnn::gemm::computeOffsetsKernel	1856	0	0	-
24	conv2_x:Bottleneck_1:Conv3	fprop	conv2d	N=32,C=64,H=56,W=56,K=256,P=56,Q=56,R=1,S=1,pl	volta_fp16_s884cudnn_fp16_256x128_ldg8_rel	158655	3288334336	64258048	1
25	conv2_x:Bottleneck_1:BN3	fprop	__add__	T=[(1,)],fp32,	elementwise_kernel	1760	1	8	-
26	conv2_x:Bottleneck_1:BN3	fprop	batch_norm	T=(32,256,56,56),fp16,	batch_norm_collect_statistics_kernel	163328	205520896	205520896	-
27	conv2_x:Bottleneck_1:BN3	fprop	batch_norm	T=(32,256,56,56),fp16,	batch_norm_transform_input_kernel	155488	205520896	0	-
28	conv2_x:Bottleneck_1:Downsample	fprop	conv2d	N=32,C=64,H=56,W=56,K=256,P=56,Q=56,R=1,S=1,pl	cudnn::gemm::computeOffsetsKernel	1472	0	0	-
29	conv2_x:Bottleneck_1:Downsample	fprop	conv2d	N=32,C=64,H=56,W=56,K=256,P=56,Q=56,R=1,S=1,pl	volta_fp16_s884cudnn_fp16_256x128_ldg8_rel	157888	3288334336	64258048	1
30	conv2_x:Bottleneck_1:Downsample	fprop	__add__	T=[(1,)],fp32,	elementwise_kernel	1760	1	8	-
31	conv2_x:Bottleneck_1:Downsample	fprop	batch_norm	T=(32,256,56,56),fp16,	batch_norm_collect_statistics_kernel	163520	205520896	205520896	-
32	conv2_x:Bottleneck_1:Downsample	fprop	batch_norm	T=(32,256,56,56),fp16,	batch_norm_transform_input_kernel	153952	205520896	0	-
33	conv2_x:Bottleneck_1:Residual	fprop	__iadd__	T=[(32,256,56,56),(32,256,56,56)],fp16,	elementwise_kernel	202143	38535168	154140672	-
34	conv2_x:Bottleneck_1:ReLU	fprop	relu	T=(32,256,56,56),fp16,	elementwise_kernel	173536	25690112	102760448	-
35	conv2_x:Bottleneck_2:Conv1	fprop	conv2d	N=32,C=256,H=56,W=56,K=64,P=56,Q=56,R=1,S=1,pl	cudnn::gemm::computeOffsetsKernel	1472	0	0	-

Salient Features

- Network: Analyze any network e.g. Torchvision, MLPerf, BERT, GPT, Waveglow, Tacotron2.
- Fast: Analyze any network in 10 min e.g. entire Transformer inference with ~ 200, 000 kernels.
- Coverage: Supports lot of layers e.g. Conv, GEMM, Pointwise, Reduction, Loss, Optimizer etc.
- Low effort: About 5 lines of instrumentation.
- Plug & Play: No changes to PyTorch.

Testimonials

“For me PyProf was the fastest way to analyze the **sequence of GPU kernels** that get executed during **beam search**. The availability of **high level information**, such as **GEMM dimensions** made it much easier to understand what was going on.”

-- User 1

“I used PyProf to profile **GPT2** in a single GPU system. It took **less than 10 minutes** to set up and provided deep insights such as **what layers are launched**, what are the compute bottlenecks and most importantly **program trace** of the specific performance-limiting kernel. Thanks, PyProf team!”

-- User 2

Testimonials

“Deep learning models like **Transformer** language translation or **BERT** language models can have on the order of **800 to 1000 kernels** in a training step. While there is a repetitive pattern to the kernels, you likely have to be an expert in cuDNN, cuBLAS, and PyTorch kernel naming conventions to decipher the difference in kernels over such a large pool of kernels. PyProf, out-of-the-box, allows the model writer to see kernel times in the context of their model giving them better **instant feedback** on hot spots in their model that they otherwise might ignore given the high bar of analysis effort. Having done the analysis with and without PyProf, I saw my **time commitment shrink** from a day or more to more like an hour or two!”

-- User 3

“PyProf **reduced the time** it takes to analyze our neural network workloads by an **order of magnitude**. Its **modular software design** has allowed us to integrate it to our workflow, as a result it had a positive impact for **several teams** at once.”

-- User 4

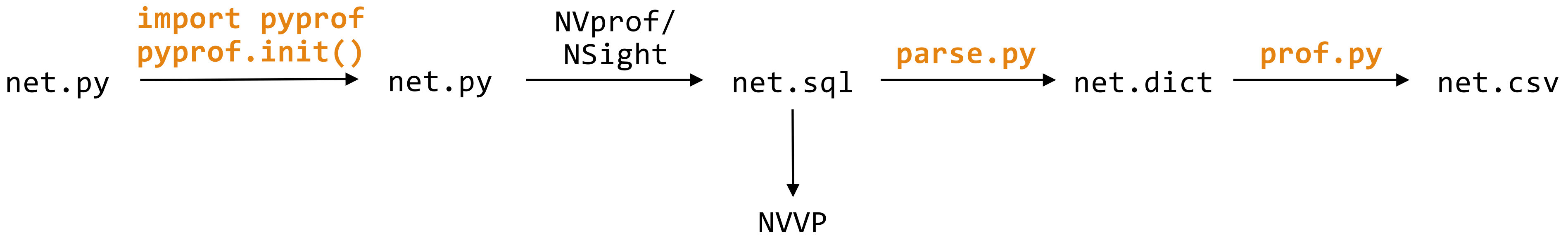
Basic Usage

Basic Usage

- Take any off the shelf PyTorch network.
- Add ~ 5 lines of instrumentation code.
- Run NVprof/Nsight Systems to generate a SQL database.
- Extract layer name, call trace, direction, operator, kernel name, tensor dims & type, silicon time etc.
- Use the operator, tensor dimensions and type to calculate flops and bytes per kernel.

PyProf: Components and Flow

- `import pyprof`: Intercept all PyTorch, custom functions and modules.
- Run NVprof/NSight Systems to obtain a SQL database.
- `parse.py`: Extract information from the SQL database.
- `prof.py`: Use this information to calculate flops and bytes.



Code Instrumentation

```
# examples/simple.py

import torch
import torch.cuda.profiler as profiler                    # Import CUDA profiler
import pyprof                                             # Import pyprof
pyprof.init()                                             # Initialize pyprof

with torch.autograd.profiler.emit_nvtx():                 # Enable PyTorch NVTX
    for epoch in range(100):
        for iteration in range(100):
            if (epoch == 0 and iteration == 20):
                profiler.start()                          # Start profiler (optional)

            ...

            if (epoch == 0 and iteration == 25):
                profiler.stop()                           # Stop profiler (optional)
```

NVprof

```
# If you did not use profiler start/stop
$ nvprof
    -f                              # Overwrite existing file
    -o net.sql                      # Create net.sql
    python net.py


# If you used profiler start/stop
$ nvprof
    -f
    -o net.sql
    --profile-from-start off        # Profiling start/stop inside net.py
    python net.py
```

NSight Systems

```
$ nsys profile
    -f true                                 # Overwrite existing files
    -o net                                  # Create net.qdrep (used by Nsys)
    -c cudaProfilerApi                      # Control profile start/stop, like NVprof
    -s none                                 # Don't sample CPU (otherwise very slow)
    --stop-on-range-end true
    --export sqlite                         # Export net.sql (similar to NVprof)
    python net.py
```

Parse SQL DB

```
$ parse/parse.py net.sql > net.dict
```

For each GPU kernel extract

Tool	Value	Example
NVProf / NSight	Kernel Name	elementwise_kernel
	Duration	44736 ns
	Grid and block dimensions	(160,1,1) (128,1,1)
	Thread Id, Device Id, Stream Id	23, 0, 7
+ PyProf	Call stack	resnet.py:210, resnet.py:168
	Layer name	Conv2_x:Bottleneck_1:ReLU
	Operator	ReLU
	Tensor Shapes	[32, 64, 56, 56]
	Datatype	fp16

Get Flops, Bytes & TC Usage

```
$ prof/prof.py --csv net.dict                      # CSV output
$ prof/prof.py net.dict                            # Space separated output
$ prof/prof.py –w 150 net.dict                     # Columnated output with width 150
$ prof/prof.py –c op,kernel,sil net.dict           # Space separated output with 3 cols
```

In addition to the previous information, for every GPU kernel obtain

- Direction (fprop, bprop).
- Flops and bytes.
- Tensor Core Usage.

Advanced Usage

(Optional)

Advanced Usage

- Layer annotation.
- Custom functions and modules.
- Extensibility.

Layer Annotation

```python
# examples/user_annotation/resnet.py
# Use the "layer:" prefix

class Bottleneck(nn.Module):
    def forward(self, x):
        nvtx.range_push("layer:Bottleneck_{}".format(self.id))    # NVTX push marker.

        nvtx.range_push("layer:Conv1")                            # Nested NVTX push/pop markers.
        out = self.conv1(x)
        nvtx.range_pop()

        nvtx.range_push("layer:BN1")                              # Use the "layer:" prefix.
        out = self.bn1(out)
        nvtx.range_pop()

        nvtx.range_push("layer:ReLU")
        out = self.relu(out)
        nvtx.range_pop()
        ...
        nvtx.range_pop()                                          # NVTX pop marker.
        return out
```

Custom Function

```
# examples/custom_func_module/custom_function.py

import torch
import pyprof
pyprof.init()

class Foo(torch.autograd.Function):
    @staticmethod
    def forward(ctx, in1, in2):
        out = in1 + in2                      # This could be a custom C++ function.
        return out
    @staticmethod
    def backward(ctx, grad):
        in1_grad, in2_grad = grad, grad      # This could be a custom C++ function.
        return in1_grad, in2_grad

# Hook the forward and backward functions to pyprof.
pyprof.wrap(Foo, 'forward')
pyprof.wrap(Foo, 'backward')
```

Custom Module

```python
# examples/custom_func_module/custom_module.py

import torch
import pyprof
pyprof.init()

class Foo(torch.nn.Module):
    def __init__(self, size):
        super(Foo, self).__init__()
        self.n = torch.nn.Parameter(torch.ones(size))
        self.m = torch.nn.Parameter(torch.ones(size))

    def forward(self, input):
        return self.n*input + self.m        # This could be a custom C++ function.

# Hook the forward function to pyprof.
pyprof.wrap(Foo, 'forward')
```

Extensibility

- For custom functions and modules, users can add flops and bytes calculation.
- Python code is easy to extend – no need to recompile, no need to change the PyTorch backend and resolve merge conflicts on every version upgrade.

Actionable Items

- NvProf / Nsight Systems tell us what the hotspots are, but not if we can act on them.
- If a kernel runs close to max perf based on FLOPs and bytes (and maximum FLOPs and bandwidth of the GPU), then there's no point in optimizing it even if it's a hotspot.
- If the ideal timing based on FLOPs and bytes (max(compute_time, bandwidth_time)) is much shorter than the silicon time, there's scope for improvement.
- Tensor Core usage (conv): for Volta, convolutions should have the input channel count (C) and the output channel count (K) divisible by 8, in order to use tensor cores. For Turing, it's optimal for C and K to be divisible by 16.
- Tensor core usage (GEMM): M, N and K divisible by 8 (Volta) or 16 (Turing)

  (https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html)

Summary

- We presented PyProf, a tool which automates end-to-end kernel-level neural network analysis for PyTorch.
- Adding ~5 lines of code generates layer type, dimensions, data type, direction, layer parameters, CUDA launch information, kernel duration, FLOPs and bandwidth.
- The tool is really easy to use and extend.
- From weeks to minutes to actionable insights.

Layer	Direction	Call Trace	Op	Parameters	Kernel	Silicon Time	Thread Id	Device Id	Stream Id	Grid Dim	Block Dim	Flops	Bytes	TC
Self Attention	fprop	attn.py: 23, …	Linear	MNK, fp16	volta_s884…	200	23	1	7	x,y,z	x,y,z	1000	500	1
Block_1a	fprop	block.py: 43,	Conv	NCHWKPQRS, fp32	cudnn_…	130	23	1	7	x,y,z	x,y,z	2000	400	1
Hadamard	fprop	net.py: 73, …	mul	T=(128,256), fp16	pointwise…	110	23	1	7	x,y,z	x,y,z	10	2000	0

Repository Note

- The original code for PyProf used to be in Apex: https://github.com/NVIDIA/apex/tree/master/apex/pyprof
- We created a new repo to rapidly iterate over new features (e.g. Nsight Systems support). The latest code can be found at: https://github.com/dlacceleration/pyprof
- NVIDIA is planning to create a new home for PyProf. Our repo will point to it once it goes live.

Contact: Questions & Contributions

GitHub: https://github.com/dlacceleration/pyprof

Aditya: aditya.iitb@gmail.com, https://github.com/adityaiitb

Marek: mkolod@gmail.com, https://github.com/mkolod