PYTORCH ECOSYSTEM DAY

TRTorch: A Compiler for TorchScript Targeting NVIDIA GPUs with TensorRT

Naren Dasan NVIDIA

Deployment, Inference, TorchScript, TorchScript Backends, JIT, Post Training Quantization

A1

Merging TensorRT Inference Optimization with PyTorch Usability

We present TRTorch, a compiler for PyTorch and TorchScript targeting NVIDIA GPUs, which combines the usability of PyTorch with the performance of TensorRT allowing users to optimize easily inference workloads on NVIDIA GPUs. For experimentation and the development of machine learning models, few tools are as approachable as PyTorch. However, some of the features that make PyTorch great for development make it hard to deploy. With TorchScript, PyTorch now has solid tooling for addressing some of these problems. TorchScript removes the dependency on Python and produces portable, self contained, static representations of code and weights. In addition to portability, users also look to optimize performance in deployment. On NVIDIA GPUs, TensorRT, NVIDIA's deep learning optimizer, provides the capability to maximize performance of workloads by tuning the execution of models for specific target hardware. TRTorch merges the benefits of TorchScript and TensorRT to simplify conducting optimization including post training quantization by leveraging common PyTorch tooling. It can be used directly from PyTorch as a TorchScript Backend, via CLI or embedded (C++/Python) in an application to easily increase inference performance.

TRTorch QuickStart

From Freshly Trained Model to Fully Optimized Inference Workload in Ten Lines.

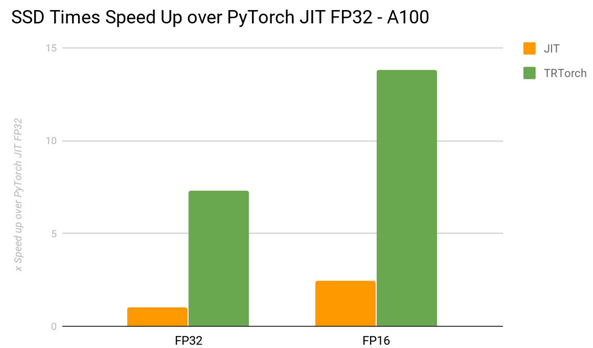
TRTorch accepts and produces TorchScript programs. Compiled programs contain an embedded TensorRT engine to execute the computation but remain fully valid TorchScript programs. Therefore, programs from TRTorch behave almost identically to normal PyTorch or TorchScript. This means that users can continue to use the same execution, (de)serialization, and composition workflows they are already familiar with from using PyTorch.

```
❯ python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> model = torch.hub.load('NVIDIA/…:torchhub', 'nvidia_ssd', pretrained=True)
>>> model = model.eval()
>>> ts_model = torch.jit.script(model)
>>> torch.jit.save(ts_model, "torchscript_ssd.ts")

❯ trtorchc -p float16 torchscript_ssd.ts trt_torchscript_ssd.ts "(32,3,300,300)"

❯ python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import trtorch
>>> ts_model = torch.jit.load("trt_torchscript_ssd.ts")
>>> ts_model(torch.randn((32, 3, 300, 300)).to("cuda").half())
```

Start to end example of exporting, compiling and running a TRTorch compiled TorchScript program

Performance

Potential Performance Gains over Deploying Standard TorchScript



Input Size 32x3x300x300 - DGX A100 - TensorRT 7.1 - cuDNN 8.0 - CUDA 11.0 - PyTorch 1.6.0 (NGC w/ CUDA 11)

Due to the use of TensorRT and extremely targeted optimization (in this case targeting the NVIDIA A100 GPU), significant performance gains can be seen by using TRTorch as part of the model deployment pipeline. Using standard execution of TorchScript on the PyTorch JIT Interpreter in FP32 as a baseline, we have measured up to 7x performance improvement with TRTorch / TensorRT even without moving to FP16. After moving to FP16 we see a 14x improvement over JIT FP32 (JIT FP16 sees a 2x improvement over FP32).

Using TRTorch Directly From PyTorch

TRTorch integrates TensorRT into PyTorch as a TorchScript Backend

```
import torch
import trtorch

compilation_spec = {
  "forward": trtorch.TensorRTCompileSpec({
    "input_shapes": [(1, 3, 224, 224)],
    "op_precision": torch.half,
  })
}

trt_module = torch._C._jit_to_backend("tensorrt", ts_module._c, compilation_spec)
trt_module(torch.randn((1, 3, 224, 224)).to("cuda").half())
trt_module.save("trt_ts_module.ts")
```

Example of using the direct TensorRT TorchScript Backend Integration

TRTorch implements the TorchScript Backend API to allow users to directly use TensorRT from PyTorch. This allows users to do complex workflows like partial model freezing for tasks like accelerated fine tuning. TRTorch operations and APIs self register in the PyTorch runtime when users import the trtorch Python package.

Leveraging PyTorch for Easy Post Training Quantization

Reusing DataLoaders to implement TensorRT INT8 Calibrators in one line

```
testing_dataset = torchvision.datasets.CIFAR10(root='./data',
                    train=False,
                    download=True)

testing_dataloader = torch.utils.data.DataLoader(testing_dataset,
                     batch_size=1,
                     shuffle=False)

calibrator = trtorch.ptq.DataLoaderCalibrator(testing_dataloader,
              cache_file='./calibration.cache',
              use_cache=False,
              algo_type=trtorch.ptq.CalibrationAlgo.ENTROPY_CALIBRATION_2,
              device=torch.device('cuda:0'))

compile_spec = {
  "input_shapes": [[1, 3, 32, 32]],
  "op_precision": torch.int8,
  "calibrator": calibrator,
}

trt_mod = trtorch.compile(ts_model, compile_spec)
```

Utilizing a PyTorch Dataset and DataLoader to implement a TensorRT INT8 Calibrator to use with TRTorch

A core tentpole of TensorRT's optimization strategy is utilizing reduced operating precisions. Training is primarily conducted in FP32 in order to achieve better convergence, but for inference, lower precisions like FP16 and INT8 utilize less computational resources without loss in accuracy. They also can leverage specialized hardware on NVIDIA GPUs (Tensor Cores) to further accelerate inference.

While converting models to run in FP16 is reasonably straight forward, the process to convert models to run in INT8, called quantization, is more involved. It is possible to train models in a "quantization aware" way but for models not trained with this process, TensorRT provides post training quantization (PTQ) facilities. Using a subset of data from the deployment distribution and running through the model in FP32, TensorRT measures activation spaces in the model to develop a quantization scheme to map the model execution to an INT8 dynamic range.

The difficulty in using this feature comes from providing TensorRT with data for calibration. PyTorch has great infrastructure for formatting, preprocessing and batching data, primarily for training. TRTorch provides calibrator factories based on PyTorch DataLoaders in order to make PTQ much more accessible, allowing users to reuse systems they probably already have for training in order to easily take advantage of the great performance INT8 can provide.

References

Dasan N., Gottbrath C., and Park J. (2020). "PyTorch-TensorRT: Accelerating Inference in PyTorch with TensorRT.". In: GTC 2020.

Krizhevsky, A., & Hinton, G. (2009). "Learning multiple layers of features from tiny images."

Take a photo to learn more: