Accelerate Training, Inference, and ML Applications on GPUs

Nathan Luehr, Maggie Zhang, Josh Romero, Pooya Davoodi, Davide Onofrio
Agenda

- Automatic Mixed Precision Training
- DALI: Fast Data Pipelines for Deep Learning
- Distributed Training with Horovod
- Accelerating Inference using TensorRT
- NVIDIA Deep Learning Profiler
Here’s what we’re running on today

VM Description

AWS EC2 VM Instance

Instance Name: g4dn.2xlarge

GPU: NVIDIA T4 Tensor Core GPUs

CPU: 8 vCPUs

Memory: 32 GiB

Disk Space 130 GB

A Special thanks to Amazon for reserving the VMs we used in this session
Here’s what we’re running on today

Setting up JupyterLab Session

1. Get IP address and credentials from instructor
2. Connect to machine via ssh: ssh nvidia@<IP ADDRESS>
3. **Startup NVIDIA 19.10 TF container:**
   
   nvidia-docker run --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 --rm -it -p 8080:8080 -v $PWD/data:/data -v $PWD:/workspace nvcr.io/nvidia/tensorflow:19.10-py3

4. **Start up JupyterLab:**
   
   jupyter lab --ip=0.0.0.0 --port=8080 --allow-root --no-browser --NotebookApp.token='' --NotebookApp.password=''

5. In your web browser, navigate to http://<IP ADDRESS>:8080
Automatic Mixed Precision Training
Motivation

- Reduced precision (16-bit floating point) for *speed or scale*
- Full precision (32-bit floating point) to *maintain task-specific accuracy*
- By using *multiple* precisions, we avoid a pure tradeoff of speed and accuracy
  - 1.5 to 3x speedup compared to FP32 training
  - with no loss in accuracy or changes to hyper-parameters
Strategy

► Use Tensor Cores to accelerate convolutions and matrix multiplications

► Store most activations in FP16
  ○ Enables larger models and/or larger batch sizes
  ○ Double effective bandwidth compared to FP32

► Use FP32 for likely to overflow ops (e.g., sums, reductions, exp)

► Update model parameters in FP32 to avoid truncation

► Use loss scaling to maintain gradients in the FP16 representable range
Using Automatic Mixed Precision
Graph Rewrite (TensorFlow 1.14.0+)

► Enabled with one line in your current FP32 training script.

```python
opt = tf.train.MomentumOptimizer(learning_rate, momentum)
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
update_op = opt.minimize(loss)
```

► Requires optimizer from `tf.train` or `tf.keras.optimizers`

► In TensorFlow 2.x `@tf.function` or Keras graph mode is required.

► Docs: tensorflow.org/api_docs/python/tf/train/experimental/enable_mixed_precision_graph_rewrite

Automatic Loss Scaling

- Scaling the loss, $L(x)$, by $s$ uniformly increases gradient values by a factor of $s$
  \[
  \frac{\partial}{\partial x_i} s \cdot L(x) = s \cdot \frac{\partial}{\partial x_i} L(x)
  \]
- Unscale weight gradients (in FP32) for weight update
- Reduce $s$ when gradients contain NaN, boost $s$ after a few thousand iterations without seeing a NaN.
Graph Optimization
Initial FP32 Graph
Graph Conversion
Choosing what to cast

<table>
<thead>
<tr>
<th>Always Cast: whitelist</th>
<th>Ops highly accelerated by float16. These always justify performance costs of casting inputs. Examples: MatMul and Conv2d.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maybe Cast: graylist</td>
<td>Ops available for float16 execution but not accelerated sufficiently to justify casting overhead on their own. Examples: Add and ReLu.</td>
</tr>
<tr>
<td>Never Cast: blacklist</td>
<td>Ops requiring float32 evaluation in order to maintain numerical stability. Examples: Exp and Sum.</td>
</tr>
<tr>
<td>Everything Else:</td>
<td>Ops lacking float16 implementations or operating on non-floating point inputs.</td>
</tr>
</tbody>
</table>
Tweaking AMP

Specific lists of ops on the whitelist, graylist, and blacklist:

tensorflow/core/grappler/optimizers/auto_mixed_precision_lists.h

You can modify the lists at runtime as follows.

```
# Comma separated list of subtractions/additionals to lists
export TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_BLACKLIST_REMOVE=Sum
export TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_GRAYLIST_ADD=Sum,YourCustomOp
```
AMP Optimizer Logging

How do I verify AMP is working?

Running auto_mixed_precision graph optimizer
No whitelist ops found, nothing to do
Running auto_mixed_precision graph optimizer
Converted 824/3507 nodes to float16 precision using 1 cast(s) to float16
(excluding Const and Variable casts)

What EXACTLY did AMP do to my model?

# Save before/after snapshots of optimized graphs
export TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_LOG_PATH="/my/log/path"

# Enable VERY verbose logging of all decisions made by AMP optimizer
export TF_CPP_VMODULE="auto_mixed_precision=2"
ResNet50 v1.5
Training on a single V100

Training script resnet_v1_5.py from github.com/NVIDIA/DeepLearningExamples

```python
256 tf.identity(learning_rate, name='learning_rate_ref')
257 tf.summary.scalar('learning_rate', learning_rate)
258
259 opt = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                        momentum=params['momentum'])
260
261 opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
```
ResNet50 v1.5
Training on a single V100

<table>
<thead>
<tr>
<th>Float32 BS=128</th>
<th>Enable AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.1x</td>
</tr>
</tbody>
</table>

Images / Second

Bar chart showing a 2.1x improvement in images per second when enabling AMP compared to Float32 BS=128.
ResNet50 v1.5
Training on a single V100
ResNet50 v1.5
Training on a single V100

Images / Second

- Float32 BS=128
- Enable AMP: 2.1x
- Enable XLA: 2.8x
- Boost BS to 256: 3.3x
Mixed Precision Results
V100 Training Speedups

Keras Mixed Precision Policies
Coming in TensorFlow 2.1

- Enabled by calling `set_policy()` before constructing model.

```python
policy = tf.keras.mixed_precision.experimental.Policy('mixed_float16')
tf.keras.mixed_precision.experimental.set_policy(policy)
```

- All model layers should inherit from `tf.keras.layers.Layer`
- Works with Eager execution
- Data types are user visible
- Allows explicit control with `tf.cast`
What if AMP hurts accuracy?
Are you using loss scaling?

 › Make sure the optimizer is from `tf.keras.optimizers` or `tf.train`.

 › Make sure the model uses either `optimizer.minimize()` or both of `optimizer.compute_gradients()` and `optimizer.apply_gradients()`.

 › Specifically, models calling `tf.gradients()` directly will not enable loss scaling.

 › Still having problems?
   - Let us know on the DevTalk Forum
   - or file a bug on developer.nvidia.com
Why doesn’t AMP speed up my model?

- Are you using TensorFlow 1.14+?
- Is the training script IO or CPU bound.
  - To test, run the preprocessing pipeline with a trivial one-layer DL model.
  - Optimize the preprocessing pipeline there until it is significantly faster than your original model training script.
  - Note: DALI can significantly boost preprocessing performance for image data
- Are Tensor Core shape constraints being satisfied?
  - Note: DLProf can verify whether Tensor Cores are being used
  - Choose minibatch to be a multiple of 8
  - Choose layer dimensions and channel counts to be multiples of 8
  - Pad vocabulary and sequence lengths to a multiple of 8
  - Reference: Tensor Core Performance: The Ultimate Guide
DALI: FAST DATA PIPELINES FOR DEEP LEARNING
NVIDIA DATA LOADING LIBRARY (DALI)

Outline

- Overview
- Installation
- Pipeline and augmentation
- Hands on
### PROBLEM: THE CPU BOTTLENECK

#### 3 main issues

<table>
<thead>
<tr>
<th>Falling CPU:GPU Ratio</th>
<th>Complexity of I/O Pipeline</th>
<th>Many frameworks: Lots of Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multi-GPU, dense systems are more common today (DGX-1V, DGX-2)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGX-1V: 40 cores / 8, 5 cores / GPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGX-2: 48 cores / 16, 3 cores / GPU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Falling CPU:GPU Ratio**

- **DGX-1V:** 40 cores / 8, 5 cores / GPU
- **DGX-2:** 48 cores / 16, 3 cores / GPU

**Complexity of I/O Pipeline**

- **MXNet**
- **ImageRecordIter**
- **ImageIO**
- **TensorFlow**
- **PyTorch**

**Many frameworks: Lots of Effort**

- **Python**
- **Manual graph construction**
- **Dataset**

---

NVIDIA DATA LOADING LIBRARY (DALI)
Fast, Portable, Flexible Data Loading and Augmentation Library

Full input pipeline acceleration including data loading and augmentation

Drop-in integration with DL frameworks

Portable workflows through multiple input formats

Flexible through configurable graphs and custom operators

GPU and CPU data pipelines

Currently supports:
- ResNet50 (Image Classification), SSD (Object Detection)
- Data Formats - JPEG, LMDB, RecordIO, TFRecord, COCO, H.264, HEVC
- Python APIs to define, build & run an input pipeline

Monthly releases | Over 1500 GitHub stars | Top 50 ML Projects (out of 22,000 in 2018)
DALI 0.13 PERFORMANCE

Training ResNet50 on TensorFlow

TensorFlow 19.09 NGC Container, batch size: 256 for DGX-2 and DGX-1, batch size: 192 for AWSp3.16x large
DALI: AN OPTIMIZED GRAPH EXECUTOR

Framework Pre-processing – Without DALI

**BEFORE**

Load → Decode → Images → Resize → Augment → Labels → Training

Native pipeline runs on CPU only

Framework Pre-processing – With DALI & nvJPEG

**AFTER**

Load → Decode → Images → Resize → Augment → Labels → Training

DALI pipeline runs on CPU and GPU
SET YOUR DATA FREE

Reuse the same data across different frameworks

TFRecord (TensorFlow)
RecordIO (MXNet)
List of JPEGs (PyTorch, others)
LMDB (Caffe, Caffe2)
Video

Use any file format in any framework with DALI's universal dataloader
USING DALI
Simple Python Interface to Implement Data Processing in a Few Steps

- Many examples & tutorials to help you get started
  - Multi-GPU training
  - How to read data in various frameworks
  - How to create custom operators
  - Pipeline for object detection
  - Image Classification (RN50) for Pytorch, MXNet, TensorFlow
  - Video pipeline
  - more to come...

**INSTALLATION**

- DALI is preinstalled in the **NVIDIA GPU Cloud containers**

- Install prebuilt DALI packages
  - Prerequisites: Linux x64, **NVIDIA Driver** supporting **CUDA 9.0** or later, **TensorFlow 1.7** or later.
  - Install DALI
  - Install DALI TensorFlow Plugin

**INSTALLATION**

https://docs.nvidia.com/deeplearning/sdk/dali-developer-guide/docs/installation.html
- Define the pipeline
- Build the pipeline
- Run the pipeline
**Pipeline**

**Define the Pipeline**

Instantiate operators

```python
def __init__(self, batch_size, num_threads, device_id):
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)
    self.input = ops.FileReader(file_root = image_dir)
    self.decode = ops.ImageDecoder(device = "mixed", output_type = types.RGB)
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

Define graph in imperative way

```python
def define_graph(self):
    jpegs, labels = self.input()
    images = self.decode(jpegs)
    images = self.resize(images)
    return (images, labels)
```

Build it

```python
pipe.build()
```

Run it

```python
images, labels = pipe.run()
```
**PIPELINE**

*Define the Pipeline*

**Instantiate operators**

```python
def __init__(self, batch_size, num_threads, device_id):
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)
    self.input = ops.FileReader(file_root = image_dir)
    self.decode = ops.ImageDecoder(device = "mixed", output_type = types.RGB)
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

**Define graph in imperative way**

```python
def define_graph(self):
    jpegs, labels = self.input()
    images = self.decode(jpegs)
    images = self.resize(images)
    return (images, labels)
```

**Build it**

```python
pipe.build()
```

**Run it**

```python
images, labels = pipe.run()
```
Instantiante operators

```python
def __init__(self, batch_size, num_threads, device_id):
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)
    self.input = ops.FileReader(file_root = image_dir)
    self.decode = ops.ImageDecoder(device = "mixed", output_type = types.RGB)
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

Define graph in imperative way

```python
def define_graph(self):
    jpegs, labels = self.input()
    images = self.decode(jpegs)
    images = self.resize(images)
    return (images, labels)
```

Build it

```python
pipe.build()
```

Run it

```python
images, labels = pipe.run()
```
**Instantiate operators**

```python
def __init__(self, batch_size, num_threads, device_id):
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)
    self.input = ops.FileReader(file_root = image_dir)
    self.decode = ops.ImageDecoder(device = "mixed", output_type = types.RGB)
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

**Define graph in imperative way**

```python
def define_graph(self):
    jpegs, labels = self.input()
    images = self.decode(jpegs)
    images = self.resize(images)
    return (images, labels)
```

**Build it**

`pipe.build()`

**Run it**

`images, labels = pipe.run()`
Instantiate operators

```python
def __init__(self, batch_size, num_threads, device_id):
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)
    self.input = ops.FileReader(file_root = image_dir)
    self.decode = ops.ImageDecoder(device = "mixed", output_type = types.RGB)
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

Define graph in imperative way

```python
def define_graph(self):
    jpegs, labels = self.input()
    images = self.decode(jpegs)
    images = self.resize(images)
    return (images, labels)
```

Build it

```
pipe.build()
```

Run it

```
images, labels = pipe.run()
```
AUGMENTATION

Rotation

Instantiate operators

def __init__(self, batch_size, num_threads, device_id):
    super(RotatedSimplePipeline, self).__init__(batch_size, num_threads, device_id, seed=12)
    self.input = ops.FileReader(file_root = image_dir, random_shuffle=True, initial_fill=21)
    self.decode = ops.ImageDecoder(device = "mixed", output_type = types.RGB)
    self.rotate = ops.Rotate(device = "gpu")
    self.rng = ops.Uniform(range = (-10.0, 10.0))

Define graph in imperative way

def define_graph(self):
    jpegs, labels = self.input()
    images = self.decode(jpegs)
    angle = self.rng()
    rotated_images = self.rotate(images.gpu(), angle = angle)
    return (rotated_images, labels)

Build it
pipe.build()

Run it
images, labels = pipe.run()
HOW TO USE DALI
Use in TensorFlow

**TensorFlow Dataset**

```python
def get_data():
    ds =
tf.data.Dataset.from_tensor_slices(files)
    ds.define_operations(...)
    return ds

classifier.train(input_fn=get_data,...)
```

**DALI TensorFlow operator**

```python
def get_data():
    dali_pipe = TrainPipe(...)
    daliop = dali_tf.DALIIterator()
    with tf.device("/gpu:0"):  
        img, labels = daliop(pipeline=dali_pipe, ...
    return img, labels

classifier.train(input_fn=get_data,...)
```
NVIDIA DATA LOADING LIBRARY (DALI)

Future

• Extract augmentation operators in a separate library (inference, custom pipeline, ...)

• Automatic Speech Processing (ASR)

• Full Video Pipeline Support (video classification, detection, slo-mo, upscale, denoise, super-res)

• 3D volumetric data (medical imaging)

• More workloads (segmentation)
NVIDIA DATA LOADING LIBRARY (DALI)

Summary

- Open source, GPU-accelerated data augmentation and image loading library
  - DALI focuses on performance, flexibility and training portability
- Deep Learning training on 8x GPUs, are likely bottlenecked on CPUs
- Download and evaluate DALI (NGC containers, pip whl, open source)
- Full pre-processing data pipeline ready for training and inference
- Easy framework integration
- Portable training workflows
DALI RESOURCES

- Official Documentation (Quick Start, Developer Guides)
  - [https://docs.nvidia.com/deeplearning/sdk/index.html#data-loading](https://docs.nvidia.com/deeplearning/sdk/index.html#data-loading)
- GitHub Documentation
- DALI Samples & Tutorial
- DALI Blog
  - Fast AI Data Preprocessing with DALI
  - Case Study: ResNet50 with DALI
- GTC talks
  - Fast AI Data Pre-processing with NVIDIA DALI
  - Integration of TensorRT with DALI for Accelerating Inferencing on Xavier
Hands on

Pipeline and Augmentation

1. Clone Github Repo:
   - git clone https://github.com/NVIDIA/DALI.git

2. Pull NGC TensorFlow container:
   - docker pull nvcr.io/nvidia/tensorflow:19.10-py3

3. Start the container:

4. Start Jupyter Notebook at /workspace/DALI/docs/examples/getting started.ipynb
   - jupyter lab --ip=0.0.0.0 --port=8888 --allow-root --no-browser --NotebookApp.token=" --NotebookApp.password="

https://docs.nvidia.com/deeplearning/sdk/dali-developer-guide/docs/examples/getting%20started.html
Distributed Training with Horovod
Why Distributed Training?

Harness the power of many GPUs!

- With complex models and large datasets, training on a single GPU can be intractable.
- Solution: distribute training workload across multiple GPUs in a system to increase throughput and obtain results more quickly → More experimentation, more data, improved models.
- With the right tools, converting existing single worker programs to data-parallel programs can be straightforward.
Horovod

Distributed Training Framework

- Originally developed by Uber to simplify distributing existing TensorFlow models
- Provides APIs for primitive collective communication operations: broadcast, allreduce, allgather
- Provides high level optimizer wrappers and training hooks for ease of use
- Can leverage NCCL for high performance collective implementation on GPU systems
- Effective across scales from single node workloads all the way up to the full Summit supercomputer!
Basic Horovod Strategy

- Each worker runs a local TensorFlow process
- Horovod intercepts optimizer gradient computation, injecting allreduce and averaging operation
- Instead of local gradients, workers receive a globally averaged gradient from `opt.compute_gradients`
- Implemented via Horovod `DistributedOptimizer`
Installation

- Horovod can be installed easily using pip:
  
  ```bash
  pip install horovod
  ```

- For best performance, we recommend installation with NCCL support:
  
  ```bash
  HOROVD_NCCL_HOME=...  HOROVD_GPU_ALLREDUCE=NCCL  pip install horovod
  ```
Using Horovod

Overview of script changes

- Applying Horovod to an existing TensorFlow model is straightforward. The basic steps are:
  - Import and initialize Horovod
  - Assign GPUs to workers
  - Synchronize initial state
  - Apply DistributedOptimizer wrapper and adjust learning rate
  - Partition training data

- Additional changes must be made to limit certain activity to a single worker (i.e. checkpointing, terminal output)
Using Horovod

Import and initialize Horovod

```python
import horovod.tensorflow as hvd
hvd.init()
```

- `hvd.init()` initializes Horovod on each worker process.
- Enables usage of Horovod routines to query basic information about current communicator like `rank`, `local rank`, and `size`

```python
rank = hvd.rank()  # unique global ID of worker
local_rank = hvd.local_rank()  # unique ID of worker per server node
size = hvd.size()  # number of workers in communicator
```
Using Horovod

Assign GPUs to workers

- To ensure that multiple workers do not attempt to use the same GPU, enforce an explicit assignment based on local rank:

```python
config = tf.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())
```
Using Horovod

Broadcast initial parameters

- Before training begins, model parameters should be synced to ensure common initialization, typically with the state of rank 0. Horovod provides a hook to do this at the start of training:

```python
hooks = [hvd.BroadcastGlobalVariablesHook(root_rank=0)]
classifier.train(..., hooks=hooks)
```
Using Horovod

Apply DistributedOptimizer and adjust learning rate

- To train with globally averaged gradients from Horovod instead of worker local gradients, wrap optimizer with DistributedOptimizer wrapper:

  ```python
  opt = tf.train.MomentumOptimizer(learning_rate, momentum)
  opt = hvd.DistributedOptimizer(opt)
  ```

- Learning rate adjustment (assuming constant batch size per worker):
  - In many scenarios, `learning_rate = base_lr * hvd.size()` is a good starting point.
  - Adding warmup can help deal with instabilities at startup with large learning rates.
  - For very large global batch sizes, more advanced optimizers like LARC can help maintain convergence.
Using Horovod

Handling data

- In data-parallel training, data should be partitioned across workers. Example pipeline using `tf.data.Dataset`:

```
import tensorflow as tf

d = tf.data.Dataset.from_tensor_slices(filenames)
d = d.shard(hvd.size(), hvd.rank())
d = d.repeat()
d = d.shuffle()
...
d = d.batch(batch_size)
```

- In multi-node training configurations, pre-staging data to node local storage (if available) is highly recommended.
Using Horovod

Additional script modifications

● Limit checkpoint output to a single worker to avoid data corruption

```python
est = tf.estimator.Estimator(
    model_dir=log_dir if hvd.rank() == 0 else None)
config=tf.estimator.RunConfig(
    ...
    save_checkpoints_steps=save_step if hvd.rank() == 0 else None))
```

● Adjust batch size and training iterations to account for multiple workers

```python
global_batchsize = batchsize * hvd.size()
nstep_per_epoch = num_training_examples // global_batchsize
```
Using Horovod

Running your script

- Horovod provides a horovodrun launcher to simplify launching scripts

  ```bash
  horovodrun -np 4 python hvd_train.py
  ```

- For greater control, can launch directly using mpirun and explicitly provide your own configuration options

  ```bash
  mpirun --allow-run-as-root --bind-to none ... -np 4 python hvd_train.py
  ```

- Run horovodrun with `--verbose` option to see the configuration options Horovod uses
Performance Tuning Tips

Additional DistributedOptimizer Options

- Compression: cast FP32 gradients to FP16 for communication
  
  \[
  \text{opt} = \text{hvd.DistributedOptimizer(} \text{opt, compression=hvd.compression.fp16})
  \]

- Sparse as Dense: treat sparse tensors as dense tensors (use allreduce, not allgather)
  
  \[
  \text{opt} = \text{hvd.DistributedOptimizer(} \text{opt, sparse_as_dense=True})
  \]
Performance Tuning Tips

Useful Horovod Configuration Knobs

- Here are several “knobs” that can be adjusted to improve Horovod performance:
  - HOROVOD_CYCLE_TIME: wait time between Horovod updates (default 5 ms)
  - HOROVOD_FUSION_THRESHOLD: size of buffer used for tensor fusion (default 64 MiB)
  - HOROVOD_HIERARCHICAL_ALLREDUCE: use mixed NCCL (intranode)/MPI (internode) allreduce algorithm (default off)
  - HOROVOD_AUTOTUNE: enable autotuning of Horovod configuration parameters (default off)

- Can be set via environment variables or as options to horovodrun (see horovodrun --help)
Performance Tuning Tips

Profiling Options

- NVIDIA Profiling tools, if familiar
- Horovod Timeline:
  - Enable with HOROVOD_TIMELINE
  - Outputs JSON file, viewable in chrome://tracing
  - Provides some deeper insight into Horovod operations and performance
Horovod Resources

Horovod on GitHub

Horovod Tutorials on GitHub

NVIDIA TensorFlow Examples
TF-TRT

Accelerating Inference in TensorFlow using TensorRT
TensorRT (TRT)

- Part of NVIDIA Deep Learning SDK: developer.nvidia.com/tensorrt
- 2016-Present
- Library for inference
- C++ and Python API
- Provides API to
  - Create neural networks (adding layers)
  - Optimize and compile graphs for a target GPU
  - Execute graphs (runtime executor)
- Optimizations include
  - Tensor Core: FP16 and INT8
  - Layer and tensor Fusion
  - Auto-tuning
  - Low latency kernels
- Does not support backward pass, only inference
TF-TRT = TF + TRT
Why TF-TRT

● Main goal: Better Performance

- Simple Python API
- Output is still a TF model
- Works even if parts of model are not supported by TRT
TF-TRT
A Graph Conversion

1) Partition
   o Partition input graph based on TRT-compatibility
   o Replace each TRT-compatible subgraph by node TRTEngineOp
2) TRT networks: For each TRTEngineOp, build a TensorRT network
3) Optimize: Optimize the network and use it to build a TensorRT engine

- TRT-incompatible subgraphs remain untouched and are handled by TF runtime
TF-TRT

Uses TensorRT Runtime

TF to TF-TRT

Input tensor

Reshape

Conv2D

BatchNorm

Add

Relu

Cast

BatchNorm

Reshape

TRT EngineOp_0

TF executor calls TRT Runtime

Input tensor

Cast
FP32 and FP16 TF-TRT API in TensorFlow <=1.13

```
import tensorflow.contrib.tensorrt as trt
converted_graph_def = trt.create_inference_graph(
    input_graph_def=frozen_graph,
    outputs=['logits', 'classes'],
    precision_mode="FP32", "FP16")
```

```
import tensorflow.contrib.tensorrt as trt
trt.create_inference_graph(
    input_saved_model_dir=input_saved_model_dir,
    output_saved_model_dir=output_saved_model_dir,
    precision_mode="FP32", "FP16")
```

Output can be used with: TF, TRTIS, TF Serving
FP32 and FP16 API in TensorFlow >1.13

Convert contrib to compiler python class

```python
from tensorflow.python.compiler.tensorrt import trt_convert as trt

converter = trt.TrtGraphConverter(
    input_saved_model_dir=input_saved_model_dir)
converter.convert()
converter.save(output_saved_model_dir)
```
from tensorflow.python.compiler.tensorrt import trt_convert as trt

converter = trt.TrtGraphConverter(
    input_saved_model_dir=input_saved_model_dir)
converter.convert()

# For INT8 - Run calibration 10 times.
converted_graph_def = converter.calibrate(
    fetch_names=['output:0'],
    num_runs=10,
    feed_dict_fn=lambda: {'input:0': my_next_data()})
converter.save(output_saved_model_dir)
from tensorflow.python.compiler.tensorrt import trt_convert as trt

converter = trt.TrtGraphConverterV2(
    input_saved_model_dir=input_saved_model_dir)

converter.convert()

# optionally build TRT engines before deployment
converter.build(input_fn=my_input_fn)

converter.save(output_saved_model_dir)
from tensorflow.python.compiler.tensorrt import trt_convert as trt

converter = trt.TrtGraphConverterV2(
    input_saved_model_dir=input_saved_model_dir)
converter.convert(calibration_input_fn=my_input_fn)

# optionally build TRT engines before deployment
converter.build(input_fn=my_input_fn)
converter.save(output_saved_model_dir)
TF-TRT Installation

- Pip packages
  - `pip install tensorflow-gpu==1.15`
  - `pip install tensorflow-gpu==2.0`
- NVIDIA Docker containers
  - `docker pull nvcr.io/nvidia/tensorflow:19.10-py3`
- Compile from source
  - Do you wish to build TensorFlow with TensorRT support? [y/N]: y
TF-TRT Demo on AWS VM

Jupyter lab in NVIDIA TF containers

1. Login to VM: `ssh nvidia@IP`

2. Ensure you have tensorrt examples by running `ls ~/tensorrt`


4. Run Jupyter: `jupyter lab --ip=0.0.0.0 --port=5000 --allow-root --no-browser --NotebookApp.token=' ' --NotebookApp.password=' '`

5. Open this address in browser: `http://IP_ADDRESS:5000/`

6. Open this notebook:
   `tftrt/examples/image-classification/TFv2-TF-TRT-inference-from-Keras-saved-model.ipynb`
TF-TRT Log

Lots of useful information

- List of unconverted ops
- Number of TRT subgraphs
- Increase log verbosity:
  - TF_CPP_VMODULE=segment=2,convert_graph=2,convert_nodes=2,trt_engine=1,trt_logger=2
  python ...
- Why ops are not converted
- TRT verbose logs
NVIDIA Deep Learning Profiler
Background

- Training Deep Learning Models is a **time** and **compute** intensive process
- Optimizing Deep Learning Models involves iterating through:
  - **Collecting current performance metrics** (Profiling)
  - **Visualizing and analyzing** Key areas of improvement
  - Taking **Action**

**Profiling** helps to find where to update code to accelerate training time on GPUs without loss of accuracy
PROFILING TOOLS & TECHNOLOGIES

Researchers
- NVTX
- Nsight Systems
- Nsight Compute

Data Scientists & Applied Researchers
- DLProf
- Tensorboard
  <Nsight Systems w/ NVTX>

Sysadmins & DevOps
- Data Center Monitoring Tools
  DCGM, NVML
  <Nsight Systems>

Skills in Algorithms --------> Skills in Domains & Applications --------> Skills in Systems
Data Scientist & Researcher Challenges

- Using Tensorboard but no GPU time or usage

- How to locate opportunities for speed ups using reduced precision

- Reason why some operations are not using Tensor cores could be as easy as dimensions of the matrix are wrong, need to be divisible by 8

How to discover where performance can be improved?
*Nsight Systems and Nsight Compute have been built using CUDA Profiling Tools Interface (CUPTI). They rely on NVTX markers to focus on sections of code.

*NVTX Nvidia Tools Extension Library is a way to annotate source code with markers.

*DLProf calls Nsight systems to collect the profile data and correlate with the graph.
DLProf User Workflow

Use NVIDIA Optimized Tensorflow Framework container

$ dlprof --in_graphdef=graphdef.pbtxt /usr/bin/python
nvidia-examples/cnn/resnet.py --layers=50 --num_iter=100 --iter_unit=batch
--display_every=50 --data_dir=/data/train-images --batch=128
--precision=fp16

generate graph definition

prefix training script with dlprof

visualize with Tensorboard or text reports

How Does Deep Learning Profiler Work

**INPUT**
- Graphdef file generated in Tensorflow

**PROFILE**
- Use **Nsight** tools to gather kernel and timing profile data

**CORRELATE**
- Correlate profile data with Tensorflow model

**OUTPUT**
- Generate TensorBoard event files and detailed reports

**ANALYZE**
- Analyze in TensorBoard or other 3rd party tools
EASY TO USE
Simply prefix training script with dlprof

1. Use the Tensorflow Container
2. Generate graphdef and run dlprof
3. Collection of timing data occurs leveraging Nsight Systems behind the scenes
4. Visualize on Tensorboard
   a. Check Model summary report to understand how much time was spent in operations that were eligible to use Tensor cores but did not, what was the performance like in each iteration
   b. Check iterations report to understand each iteration of the training run
   c. Read top 10 ops that were consuming the most GPU and CPU time
   d. Switch to csv reports and back to Tensorboard as needed
5. Find clues to where optimization may be possible
EASY TO USE

Generate the graphdef

If the TensorFlow script doesn't contain an option to create a graphdef, the following code can be inserted into your TensorFlow python script after the TensorFlow session has been created:

```python
graph_def = session.graph.as_graph_def()
with open('graphdef.pb', 'wb') as f:
    f.write(graph_def.SerializeToString())
with open('graphdef.pbtxt', 'w') as f:
    f.write(str(graph_def))```
Example: DLProf with & without AMP

Impact of Enabling Automatic Mixed Precision

https://github.com/tensorflow/models.git

https://github.com/tensorflow/benchmarks
Example: DLProf with & without AMP

Impact of Enabling Automatic Mixed Precision

$ cd ./DLProf

// Run the NVIDIA TensorFlow Container
$ nvidia-docker run \
--privileged --rm -it -p5000:5000 \n-v $PWD/data:/data \n-v $PWD/results:/results \n-v $PWD/tensorflow-examples:/workspace/tensorflow-examples \nvcr.io/nvidia/tensorflow:19.10-py3
Example: DLProf with & without AMP

Impact of Enabling Automatic Mixed Precision

$ cd /workspace/tensorflow-examples/benchmarks/scripts/tf_cnn_benchmarks

// Create the graph_def

$ PYTHONPATH=/workspace/tensorflow-examples/models /usr/bin/python tf_cnn_benchmarks.py \
--num_gpus=1 --batch_size=128 --model=resnet50_v1.5 \
--device=gpu --gpu_indices=1 --data_name=imagenet \
--data_dir=/data/train-val-tfrecord-480 \
--num_batches=1 \
--use_fp16 --fp16_enable_auto_loss_scale \
--graph_file=/results/resnet_graph.pb
Example: DLProf with & without AMP

Impact of Enabling Automatic Mixed Precision

// Run DLProf
$ dlprof --force --in_graphdef=/results/resnet_graph.pb
   --key_node=global_step
   /usr/bin/python tf_cnn_benchmarks.py --num_gpus=1 --batch_size=128
   --model=resnet50_v1.5 --device=gpu --gpu_indices=1
   --data_name=imagenet --data_dir=/data/train-val-tfrecord-480
   --num_batches=50

$ tensorboard --logdir ./event_files --port 5000
Example: DLProf with & without AMP

Impact of Enabling Automatic Mixed Precision

// Run DLProf
$ dlprof --force --in_graphdef=/results/resnet_graph.pb --key_node=global_step \
/usr/bin/python tf_cnn_benchmarks.py --num_gpus=1 --batch_size=128 \ 
--model=resnet50_v1.5 --device=gpu --gpu_indices=1 --data_name=imagenet \ 
--data_dir=/data/train-val-tfrecord-480 --num_batches=50 \ 
--use_fp16 --fp16_enable_auto_loss_scale

$ tensorboard --logdir ./event_files --port 5000
VISUALIZATION

NVIDIA Modifications to Tensorboard to reflect GPU details

- Puts a **GPU Lens** on Tensorboard
- **At a glance**, Tensor Core Compatibility shows where to focus
  - Eligible operations that could use Tensor cores
- Drill down to examine deeper on the green colored nodes
  - Ops that used Tensor cores
  - Ops that did not

Which operations are not using tensorcores?

*Note: DLProf version of Tensorboard is included in the container in addition to the existing one*
### Generating text reports

DLProf can generate reports in csv and json formats if specified on commandline.

**Option:** `--reports=detail,iteration --file_formats=csv`

<table>
<thead>
<tr>
<th>Name</th>
<th>Node Op</th>
<th>Origin</th>
<th>No.</th>
<th>TC Elig</th>
<th>Util</th>
<th>Total CP Time (ns)</th>
<th>Avg. CPL Time (ns)</th>
<th>Min CPU Time (ns)</th>
<th>Max CPU% Time (ns)</th>
<th>Total GP Time (ns)</th>
<th>Avg. GF Time (ns)</th>
<th>Min Time</th>
<th>Max GFI Time (ns)</th>
<th>Total CPU Overhead (ns)</th>
<th>Avg. CPU Overhead (ns)</th>
<th>Min CPU Overhead (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tower_O/v/cg/MobileNetV2/expanded_conv_10/expand/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>60</td>
<td>yes</td>
<td>no</td>
<td>5831205</td>
<td>97186.8</td>
<td>59381</td>
<td>247570</td>
<td>4690040</td>
<td>78167.3</td>
<td>52959</td>
<td>203295</td>
<td>3279247</td>
<td>54654.1</td>
<td>33864</td>
</tr>
<tr>
<td>tower_O/v/cg/MobileNetV2/expanded_conv_6/expand/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>60</td>
<td>yes</td>
<td>no</td>
<td>8055630</td>
<td>134260.7</td>
<td>59348</td>
<td>212492</td>
<td>6096820</td>
<td>101613.7</td>
<td>52928</td>
<td>150751</td>
<td>4751284</td>
<td>79188.1</td>
<td>32234</td>
</tr>
<tr>
<td>tower_O/v/cg/MobileNetV2/expanded_conv_9/expand/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>60</td>
<td>yes</td>
<td>no</td>
<td>6770314</td>
<td>112838.6</td>
<td>61427</td>
<td>212544</td>
<td>5213941</td>
<td>80899</td>
<td>54720</td>
<td>171486</td>
<td>3973637</td>
<td>66227.3</td>
<td>34516</td>
</tr>
<tr>
<td>tower_O/v/cg/Affine4/xw_plus_b/MatMul</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>60</td>
<td>yes</td>
<td>no</td>
<td>6260904</td>
<td>104330.9</td>
<td>58893</td>
<td>232099</td>
<td>5533812</td>
<td>92230.2</td>
<td>61984</td>
<td>196381</td>
<td>3381811</td>
<td>56363.9</td>
<td>33096</td>
</tr>
</tbody>
</table>

92
Reading Tensorboard and Reports

Use all reports to perform your analysis of the model

Detailed Report csv file
- Check the detailed report and sort by TC eligible, find those ops that say “yes” for TC eligible column and “no” for Using TC
- Go back to Tensorboard to visualize these operations by using the search capability
- Check model summary to see that 15 nodes were eligible but not using TCs

Tensorboard GPU Summary Tab
- Model Summary: GPU time summary for all iterations
  - 15 nodes accounted for 33.6 milliseconds out of total GPU time of 2.62 seconds so less than 3 %
- Top 10 panel shows timing info for Top 10 operations shows among the Top10, the nodes eligible were already using TCs
CUDA Occupancy

## GPU Summary: Top 10

### GPU Summary tab

- **Top 10 operations**
- **Duration:** All of the operations in your code with respect to time, sortable, filterable and whether Tensor cores were used
- **Iterations report:** Kernels executed and time used correlated with op names
Example continued

Visualising the interesting node in Tensorboard

- Out of the nodes eligible but not using Tensor cores, find one and paste in Tensorboard search box to investigate
- Check the input and output types
- There may be an operation that is not multiple of 8
- Get to the python code that was contributing to any slowness in training
- Workflow ends at this point, it is up to user to modify their python code

Overall conclusion: This model is mostly optimized to use mixed precision tensor cores
## Comparing results

### Model Summary

### FP32

<table>
<thead>
<tr>
<th></th>
<th>GPU Time</th>
<th>#Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Nodes</td>
<td>2.36 s</td>
<td>4018</td>
</tr>
<tr>
<td>Nodes Using TC</td>
<td>0 μs</td>
<td>0</td>
</tr>
<tr>
<td>Nodes Eligible For TC, But Not Using</td>
<td>495 ms</td>
<td>110</td>
</tr>
<tr>
<td>All Other Nodes</td>
<td>1.86 s</td>
<td>3908</td>
</tr>
</tbody>
</table>

### AMP ON

<table>
<thead>
<tr>
<th></th>
<th>GPU Time</th>
<th>#Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Nodes</td>
<td>2.73 s</td>
<td>4897</td>
</tr>
<tr>
<td>Nodes Using TC</td>
<td>547 ms</td>
<td>94</td>
</tr>
<tr>
<td>Nodes Eligible For TC, But Not Using</td>
<td>33.9 ms</td>
<td>16</td>
</tr>
<tr>
<td>All Other Nodes</td>
<td>2.15 s</td>
<td>4787</td>
</tr>
</tbody>
</table>
### Top 10 nodes

#### Before AMP

<table>
<thead>
<tr>
<th>GPU Time (µs)</th>
<th>CPU Time (µs)</th>
<th>Op Name</th>
<th>Op Type</th>
<th>Origin</th>
<th>Calls</th>
<th>TC Eligible</th>
<th>Using TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>133745</td>
<td>298429</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_15/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>133454</td>
<td>299570</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_14/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>96537</td>
<td>221625</td>
<td>tower_D/v/gradadients/tower_0/v/cg/MobileNetV2/expanded_conv_15/depthwise/</td>
<td>DepthwiseConv2dNativeBackpropInput</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>96418</td>
<td>221176</td>
<td>tower_D/v/gradadients/tower_0/v/cg/MobileNetV2/expanded_conv_16/depthwise/</td>
<td>DepthwiseConv2dNativeBackpropInput</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>84885</td>
<td>193880</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_13/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>80552</td>
<td>181258</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_12/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>86453</td>
<td>181206</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_11/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>58268</td>
<td>133232</td>
<td>tower_D/v/gradadients/tower_0/v/cg/MobileNetV2/expanded_conv_12/depthwise/</td>
<td>DepthwiseConv2dNativeBackpropInput</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>58210</td>
<td>130000</td>
<td>tower_D/v/gradadients/tower_0/v/cg/MobileNetV2/expanded_conv_11/depthwise/</td>
<td>DepthwiseConv2dNativeBackpropInput</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>24589</td>
<td>2116868</td>
<td>tower_D/v/cg/MobileNetV2/Conv/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>60</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

#### With AMP

<table>
<thead>
<tr>
<th>GPU Time (µs)</th>
<th>CPU Time (µs)</th>
<th>Op Name</th>
<th>Op Type</th>
<th>Origin</th>
<th>Calls</th>
<th>TC Eligible</th>
<th>Using TC</th>
<th>Kernel Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>194723</td>
<td>534939</td>
<td>tower_D/v/gradadients/tower_0/v/cg/MobileNetV2/expanded_conv_16/depthwise/depthwise</td>
<td>DepthwiseConv2dNativeBackpropInput</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>57728</td>
</tr>
<tr>
<td>137005</td>
<td>364633</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_15/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>38522</td>
</tr>
<tr>
<td>136265</td>
<td>366239</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_14/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>38522</td>
</tr>
<tr>
<td>97950</td>
<td>305310</td>
<td>tower_D/v/gradadients/tower_0/v/cg/MobileNetV2/expanded_conv_15/depthwise/depthwise</td>
<td>DepthwiseConv2dNativeBackpropInput</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>28924</td>
</tr>
<tr>
<td>96799</td>
<td>282716</td>
<td>tower_D/v/gradadients/tower_0/v/cg/MobileNetV2/expanded_conv_14/depthwise/depthwise</td>
<td>DepthwiseConv2dNativeBackpropInput</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>28924</td>
</tr>
<tr>
<td>86323</td>
<td>226384</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_13/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>18616</td>
</tr>
<tr>
<td>80330</td>
<td>207948</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_11/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>23162</td>
</tr>
<tr>
<td>80322</td>
<td>206170</td>
<td>tower_D/v/cg/MobileNetV2/expanded_conv_12/depthwise/depthwise</td>
<td>DepthwiseConv2dNative</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>23162</td>
</tr>
<tr>
<td>38870</td>
<td>224592</td>
<td>tower_D/v/gradadients/tower_0/v/cg/MobileNetV2/expanded_conv_0/depthwise/depthwise</td>
<td>DepthwiseConv2dNativeBackpropInput</td>
<td>GraphDef</td>
<td>60</td>
<td>false</td>
<td>false</td>
<td>11644</td>
</tr>
<tr>
<td>22811</td>
<td>2403342</td>
<td>tower_D/v/cg/MobileNetV2/Conv/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>60</td>
<td>true</td>
<td>true</td>
<td>300</td>
</tr>
</tbody>
</table>
Visualization

Before

After: using AMP
ResNet50 example

Without using AMP
ResNet50 example continued

Without AMP

---

**TensorBoard**

**GPU SUMMARY**

<table>
<thead>
<tr>
<th>GPU Time (μs)</th>
<th>CPU Time (μs)</th>
<th>Op Name</th>
<th>Op Type</th>
<th>Origin</th>
<th>Calls</th>
<th>TC Eligible</th>
<th>Using TC</th>
<th>Kernel Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>73803</td>
<td>3131817</td>
<td>tower_0/vgg/conv2d/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>501</td>
</tr>
<tr>
<td>63136</td>
<td>66684</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>45</td>
</tr>
<tr>
<td>46226</td>
<td>44221</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>69</td>
</tr>
<tr>
<td>45223</td>
<td>47468</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>33</td>
</tr>
<tr>
<td>45134</td>
<td>46647</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>69</td>
</tr>
<tr>
<td>43270</td>
<td>47142</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>45</td>
</tr>
<tr>
<td>43238</td>
<td>48385</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>45</td>
</tr>
<tr>
<td>43156</td>
<td>48129</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>45</td>
</tr>
<tr>
<td>42504</td>
<td>46638</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>45</td>
</tr>
<tr>
<td>41918</td>
<td>42430</td>
<td>tower_0/vgg/merge_v1.51/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>GraphDef</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td>45</td>
</tr>
</tbody>
</table>

**Kernel Summaries**

<table>
<thead>
<tr>
<th>#</th>
<th>Kernel Name</th>
<th>Using TC</th>
<th>Calls</th>
<th>GPU Time (μs)</th>
<th>Avg (μs)</th>
<th>Min (μs)</th>
<th>Max (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>void t/s/tensorflow/constant:0</td>
<td>false</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>void cudnn/align/convolve:0</td>
<td>false</td>
<td>2</td>
<td>7099</td>
<td>3949</td>
<td>3949</td>
<td>3990</td>
</tr>
<tr>
<td>2</td>
<td>cudnn/gemm/convolution:0</td>
<td>false</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>void w/s/tensorflow/constant:0</td>
<td>false</td>
<td>3</td>
<td>8045</td>
<td>3682</td>
<td>3679</td>
<td>5085</td>
</tr>
<tr>
<td>4</td>
<td>void im2col/keras/constant:0,0</td>
<td>false</td>
<td>2</td>
<td>5474</td>
<td>2737</td>
<td>2737</td>
<td>2737</td>
</tr>
<tr>
<td>5</td>
<td>void cudnn/align/convolve:0,0,0,0</td>
<td>false</td>
<td>2</td>
<td>8694</td>
<td>4343</td>
<td>4343</td>
<td>4343</td>
</tr>
<tr>
<td>6</td>
<td>void f16/0.5(x,0.5(x,0))</td>
<td>false</td>
<td>2</td>
<td>36</td>
<td>19</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>
ResNet50 example continued

With AMP enabled
Before & After

No Tensor core info

Tensor core candidates and usage details per node operation
Generating other reports besides Tensorboard

DLProf can generate reports in csv and json formats if specified

for example --out_detail_report_csv=detailreport.csv
NVIDIA TOOLS EXTENSION LIBRARY (NVTX)

- NVTX is a platform agnostic, tools agnostic API
- Allows developers to annotate (mark) source code, events, code ranges etc
- NVIDIA optimized Tensorflow, have NVTX annotations built in
- Easy to use plugin for Tensorflow is also available:
  
  https://github.com/NVIDIA/nvtx-plugins/

*Nsight Systems, Nsight Compute and Deep Learning Profiler make use of NVTX markers

https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvtx
TensorFlow NVTX Annotation*

https://github.com/NVIDIA/nvtx-plugins/ *DLProf will support in an upcoming version

Nvidia framework containers on NGC already have NVTX markers which are used by DLProf. For easily defining your own markers, use this library in addition

- Library developed specifically for annotating python code to help visualize network better in Nsight Systems

- Workflow:
  - Import nvtx_tf library
  - Annotate python code
  - Run tensorflow
  - Get data through a profiler such as Nsight Systems
Roadmap

- Support for running profiler with XLA
- Tensorboard 1.15 support
- Running with Tensorflow 2.0
- Adding your own user defined NVTX markers to generate profiles with DLProf
- Comparing consecutive runs in Tensorboard
- More friendly recommendation steps for actions that improve performance (Expert Systems)

*Note: Subject to change*
Getting Started: Deep Learning Profiler

- Try DLProf alpha included in Optimized Tensorflow container on NGC starting June
  docker pull nvcr.io/nvidia/tensorflow:19.09-py3
- Have graphdef definitions of your models ready for input
- Prefix training script with dlprof command and get the profile output
- Run a few iterations and smaller batch size to collect fast profile data of your entire model

**Note:** There are two versions of Tensorboard included in the docker container
DLProf version of Tensorboard is only used if user wishes to.