TensorFlow

User Guide

DU-08601-001_v1.15.5, 2.5.0 | May 2025

Table of Contents

Chapter 1. Overview Of TensorFlow

TensorFlow is an open-source software library for numerical computation using data flow graphs.

Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) that flow between them. This flexible architecture lets you deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code.

TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks (DNNs) research. The system is general enough to be applicable in a wide variety of other domains, as well.

For visualizing TensorFlow results, the Docker® image also contains TensorBoard. TensorBoard is a suite of visualization tools. For example, you can view the training histories as well as what the model looks like.

For information about the optimizations and changes that have been made to TensorFlow, see the TensorFlow Release Notes.

1.1. Contents Of The NVIDIA TensorFlow Container

This image contains source and binaries for TensorFlow. The pre-built and installed version of TensorFlow is located in the `/usr/local/[bin,lib]` directories. The complete source code is located in `/opt/tensorflow`.

To achieve optimum TensorFlow performance, there are sample scripts within the container image. For more information, see Performance.

TensorFlow includes TensorBoard, a data visualization toolkit developed by Google.

Additionally, this container image also includes several built-in TensorFlow examples that you can run using commands like the following. These examples perform training of convolutional neural networks (CNNs). For more information, see MNIST For ML Beginners . The following Python commands run two of these examples:

```
python -m tensorflow.models.image.mnist.convolutional
python -m tensorflow.models.image.cifar10.cifar10_multi_gpu_train
```

The first command uses the MNIST data set, for example, THE MNIST DATABASE. The second command uses the CIFAR-10 dataset, for example, The CIFAR-10 dataset.

Chapter 2. Pulling The TensorFlow Container

To pull a TensorFlow container, see Pulling A Container

Chapter 3. Running A TensorFlow Container

About this task

To run a TensorFlow container, see Running TensorFlow.

Chapter 4. Verifying TensorFlow

The simplest way to verify that TensorFlow is running correctly, is to run the examples that are included in the `/nvidia-examples/` directory.

Each example contains a `README` that describes the basic usage.

Chapter 5. Customizing And Extending TensorFlow

The nvidia-docker images come prepackaged, tuned, and ready to run; however, you may want to build a new image from scratch or augment an existing image with custom code, libraries, data, or settings for your corporate infrastructure. This section will guide you through exercises that will highlight how to create a container from scratch, customize a container, extend a deep learning framework to add features, develop some code using that extended framework from the developer environment, then package that code as a versioned release.

About this task

By default, you do not need to build a container. The NVIDIA container repository, `nvcr.io`, has a number of containers that can be used immediately including containers for deep learning as well as containers with just the CUDA® Toolkit™ .

One of the great things about containers is that they can be used as starting points for creating new containers. This can be referred to as customizing or extending a container. You can create a container completely from scratch, however, since these containers are likely to run on GPUs, it is recommended that you at least start with a `nvcr.io` container that contains the OS and CUDA® . However, you are not limited to this and can create a container that runs on the CPUs which does not use the GPUs. In this case, you can start with a bare OS container from another location such as Docker. To make development easier, you can still start with a container with CUDA; it is just not used when the container is used.

The customized or extended containers can be saved to a user's private container repository. They can also be shared with other users but this requires some administrator help.

It is important to note that all NGC deep learning framework images include the source to build the framework itself as well as all of the prerequisites.

> **ATTENTION:** Do not install an NVIDIA driver into the Docker image at docker build time.

A best-practice is to **avoid** `docker commit` usage for developing new Docker images, and to use Dockerfiles instead. The `Dockerfile` method provides visibility and capability to efficiently version-control changes made during the development of a Docker image. The Docker commit method is appropriate for short-lived, disposable images only.

For more information on writing a Docker file, see the best practices documentation.

5.1. Benefits And Limitations To Customizing TensorFlow

You can customize a container to fit your specific needs for numerous reasons; for example, you depend upon specific software that is not included in the container that NVIDIA provides. No matter your reasons, you can customize a container.

The container images do not contain sample data-sets or sample model definitions unless they are included with the framework source. Be sure to check the container for sample data-sets or models.

5.2. Example 1: Customizing TensorFlow Using Dockerfile

For the latest instructions using Dockerfile to customize TensorFlow, see the instructions provided inside the container in the file `/workspace/docker-examples/Dockerfile.customtensorflow`.

Before customizing the container, you should ensure the TensorFlow 21.08 container has been loaded into the NGC container registry using the `docker pull` command before proceeding. For example:

```
$ docker pull nvcr.io/nvidia/tensorflow:21.08
```

The Docker containers on `nvcr.io` also provide a sample Dockerfile that explains how to patch a framework and rebuild the Docker image. In the directory, `/workspace/docker-examples`, there are two sample Dockerfiles that you can use. The first one, `Dockerfile.addpackages`, can be used to add packages to the TensorFlow image. The second one, `Dockerfile.customtensorflow`, illustrates how to patch TensorFlow and rebuild the image.

```
FROM nvcr.io/nvidia/tensorflow:21.08

# Bring in changes from outside container to /tmp
# (assumes my-tensorflow-modifications.patch is in same directory as Dockerfile)
COPY my-tensorflow-modifications.patch /tmp

# Change working directory to TensorFlow source path
WORKDIR /opt/tensorflow

# Apply modifications
RUN cd tensorflow-source \
  && patch -p1 < /tmp/my-tensorflow-modifications.patch

# Rebuild TensorFlow
```

```
RUN ./nvbuild.sh

# Reset working directory
WORKDIR /workspace
```

This DockerFile will rebuild the TensorFlow image in the same way as it was built in the original image. For more information, see Dockerfile reference.

To better understand the Dockerfile, let's walk through the major commands. The first line in the Dockerfile is the following:

```
FROM nvcr.io/nvidia/tensorflow:21.08
```

This line starts with the NVIDIA 21.08 version image for TensorFlow being used as the starting point.

The second line is the following:

```
COPY my-tensorflow-modifications.patch /tmp
```

It brings in changes from outside the container into your `/tmp` directory. This assumes that the `my-tensorflow-modifications.patch` file is in the same directory as Dockerfile.

The next important line in the file changes the working directory to the TensorFlow source path.

```
WORKDIR /opt/tensorflow
```

This is followed by the command to apply the modifications patch to the source.

```
RUN cd tensorflow-source \
 && patch -p1 < /tmp/my-tensorflow-modifications.patch
```

After the patch is applied, the TensorFlow image can be rebuilt. This is done via the `RUN` command in the `DockerFile/`.

```
RUN ./nvbuild.sh
```

Finally, the last major line in the DockerFile resets the default working directory.

```
WORKDIR /workspace
```

5.3. Example 2: Customizing TensorFlow Using `docker commit`

This example uses the `docker commit` command to flush the current state of the container to a Docker image.

About this task

This is not a recommended best practice, however, this is useful when you have a container running to which you have made changes and want to save them. In this example, we are using the `apt-get` tag to install a package that requires the user to run as root.

Note:

- The TensorFlow image release 21.07 is used in the example instructions for illustrative purposes.

- Do not use the `--rm` flag when running the container. If you use the `--rm` flag when running the container your changes will be lost when exiting the container.

Procedure

1. Pull the Docker container from the `nvcr.io` repository to your DGX™ system. For example, the following command will pull the TensorFlow container:

   ```
   $ docker pull  nvcr.io/nvidia/tensorflow:21.07
   ```

2. Run the container on your DGX.

   Note: Do not use the `--rm` flag when running the container. If you use the `--rm` flag when running the container your changes will be lost when exiting the container.

   ```
   docker run --gpus all -ti nvcr.io/nvidia/tensorflow:21.07
   ```

   ```
   ================
   == TensorFlow ==
   ================

   NVIDIA Release 21.07 (build 31239)

   Container image Copyright (c) 2021, NVIDIA CORPORATION.  All rights reserved.
   Copyright 2021 The TensorFlow Authors.  All rights reserved.

   Various files include modifications (c) NVIDIA CORPORATION.  All rights reserved.
   NVIDIA modifications are covered by the license terms that apply to the underlying
    project or file.

   NOTE: The SHMEM allocation limit is set to the default of 64MB.  This may be
      insufficient for TensorFlow.  NVIDIA recommends the use of the following flags:
      docker run --gpus all --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 ...

   root@8db6076d82c4:/workspace#
   ```

3. You should now be the root user in the container (notice the prompt). You can use the `apt` command to pull down a package and put it in the container.

   Note: The NVIDIA containers are built using Ubuntu which uses the `apt-get` package manager. Check the container release notes in [Deep Learning Documentation] for details on the specific container you are using.

   In this example, we will install octave; the GNU clone of MATLAB, into the container.

   ```
   # apt-get update
   # apt install octave
   ```

   Note: You have to first issue `apt-get update` before you install Octave using `apt`.

4. Exit the workspace.

   ```
   # exit
   ```

5. Display the list of running containers.

   ```
   $ docker ps -a
   ```

   As an example, here is some of the output from the `docker ps -a` command:

   ```
   $ docker ps -a
   CONTAINER ID     IMAGE                                 CREATED        ...
   8db6076d82c4     nvcr.io/nvidia/tensorflow:21.07      3 minutes ago   ...
   ```

6. Now you can create a new image from the container that is running where you have installed Octave. You can commit the container with the following command.

```
$ docker commit 8db6076d82c4 nvcr.io/nvidian_sas/tensorflow_octave:21.07
sha256:25198e37ae2e3416bebcf1d3084ff3a95600d978811fe7f4f184de0af3878b51
```

7. Display the list of images.

```
$ docker images
REPOSITORY                          TAG                         IMAGE ID       ...
nvidian_sas/tensorflow_octave       21.07                       25198e37ae2e   ...
```

8. To verify, run the container again and see if Octave is actually there.

```
docker run --gpus all -ti nvidian_sas/tensorflow_octave:21.07

================
== TensorFlow ==
================

NVIDIA Release 21.07 (build 31239)

Container image Copyright (c) 2021, NVIDIA CORPORATION.  All rights reserved. Copyright
 2021 The TensorFlow Authors.  All rights reserved.

Various files include modifications (c) NVIDIA CORPORATION.  All rights reserved. NVIDIA
 modifications are covered by the license terms that apply to the underlying project or
 file.

NOTE: The SHMEM allocation limit is set to the default of 64MB.  This may be insufficient
 for TensorFlow.  NVIDIA recommends the use of the following flags:
   docker run --gpus all --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 ...

root@87e8dde4be6d:/workspace# octave
octave: X11 DISPLAY environment variable not set
octave: disabling GUI features
GNU Octave, version 4.0.0
Copyright (C) 2015 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1>
```

Since the Octave prompt displayed, Octave is installed.

9. If you are using a DGX-1 or DGX Station, and you want to save the container into your private repository (Docker uses the phrase "push"), then you can use the `docker push ...` command.

```
$ docker push nvcr.io/nvidian_sas/tensorflow_octave:21.07
```

5.4. Accelerating Inference In TensorFlow With TensorRT

About this task

For step-by-step instructions on how to use TensorRT with the TensorFlow framework, see Accelerating Inference In TensorFlow With TensorRT User Guide. To view the key features, software enhancements and improvements, and known issues, see the Release Notes.

Chapter 6. TensorFlowParameters

The TensorFlow container in the NGC container registry (`nvcr.io`) comes pre-configured as defined by the following parameters. These parameters are used to pre-compile GPUs, enable support for the Accelerated Linear Algebra (XLA) backend, and disable support for Google Cloud Platform (GCP) and the Hadoop Distributed File System (HDFS).

6.1. Added And Modified Parameters

In addition to the parameters within the Dockerfile that is included in the Google TensorFlow container, the following parameters have either been added for modified with the NVIDIA version of TensorFlow.

For parameters not mentioned in this guide, see the TensorFlow documentation.

6.1.1. TF_CUDA_COMPUTE_CAPABILITIES

The TF_CUDA_COMPUTE_CAPABILITIES parameter enables the code to be pre-compiled for specific GPU architectures.

The container comes built with the following setting, which targets Pascal, Volta, Turing, and NVIDIA Ampere architecture GPUs:

```
TF_CUDA_COMPUTE_CAPABILITIES "6.0,6.1,7.0,7.5,8.0,8.6"
```

Where the numbers correspond to GPU architectures:

6.0+6.1	Pascal
7.0	Volta
7.5	Turing
8.0+8.6	NVIDIA Ampere architecture

6.1.2. TF_NEED_GCP

The TF_NEED_GCP parameter, as defined, disables support for the Google Cloud Platform (GCP).

The container comes built with the following setting, which turns off support for GCP:

```
TF_NEED_GCP 0
```

6.1.3. TF_NEED_HDFS

The TF_NEED_HDFS parameter, as defined, disables support for the Hadoop Distributed File System (HDFS).

The container comes built with the following setting, which turns off support for HDFS:

```
TF_NEED_HDFS 0
```

6.1.4. TF_ENABLE_XLA

The TF_ENABLE_XLA parameter, as defined, enables support for the Accelerated Linear Algebra (XLA) backend.

The container comes built with the following setting, which turns on support for XLA:

```
TF_ENABLE_XLA 1
```

Chapter 7. TensorFlow Environment Variables

The following environment variable settings enable certain features within TensorFlow. They change and reduce the precision of the computation slightly and are enabled by default.

7.1. Added Or Modified Variables

In addition to the variables within the Dockerfile that are included in the Google TensorFlow container, the following variables have either been added or modified with the NVIDIA version of TensorFlow.

For variables not mentioned in this guide, see the TensorFlow documentation.

7.1.1. *TF_ADJUST_HUE_FUSED*

The *TF_ADJUST_HUE_FUSED* variable enables the use of fused kernels for the image hue.

This variable is enabled by default:

```
TF_ADJUST_HUE_FUSED         1
```

To disable the variable, run the following command:

```
export TF_ADJUST_HUE_FUSED=0
```

7.1.2. *TF_ADJUST_SATURATION_FUSED*

The *TF_ADJUST_SATURATION_FUSED* variable enables the use of fused kernels for the saturation adjustment.

This variable is enabled by default:

```
TF_ADJUST_SATURATION_FUSED  1
```

To disable the variable, run the following command:

```
export TF_ADJUST_SATURATION_FUSED=0
```

7.1.3. *TF_ENABLE_WINOGRAD_NONFUSED*

The *TF_ENABLE_WINOGRAD_NONFUSED* variable enables the use of the non-fused Winograd convolution algorithm.

This variable is enabled by default:

```
TF_ENABLE_WINOGRAD_NONFUSED 1
```

To disable the variable, run the following command:

```
export TF_ENABLE_WINOGRAD_NONFUSED=0
```

7.1.4. *TF_AUTOTUNE_THRESHOLD*

The *TF_AUTOTUNE_THRESHOLD* variable improves the stability of the auto-tuning process used to select the fastest convolution algorithms. Setting it to a higher value improves stability, but requires a larger number of trial steps at the beginning of training before the best algorithms are found.

Within the container, this variable is set to the following:

```
export TF_AUTOTUNE_THRESHOLD=2
```

To set this variable to its default setting, run the following command:

```
export TF_AUTOTUNE_THRESHOLD=1
```

7.1.5. *CUDA_DEVICE_MAX_CONNECTIONS*

The *CUDA_DEVICE_MAX_CONNECTIONS* variable solves performance issues related to streams on Tesla K80 GPUs.

Within the container, this variable is set to the following:

```
export CUDA_DEVICE_MAX_CONNECTIONS=12
```

To set this variable to its default setting, run the following command:

```
export CUDA_DEVICE_MAX_CONNECTIONS=8
```

7.1.6. *TF_DISABLE_CUDNN_TENSOR_OP_MATH*

The *TF_DISABLE_CUDNN_TENSOR_OP_MATH* variable enables and disables Tensor Core math for cuDNN convolutions in TensorFlow.

Tensor Core math is enabled by default, but can be disabled by setting this variable to `1`. For more information, see Tensor Core Math.

This variable is disabled by default:

```
export TF_DISABLE_CUDNN_TENSOR_OP_MATH=0
```

To enable the variable, run the following command:

```
export TF_DISABLE_CUDNN_TENSOR_OP_MATH=1
```

7.1.7. *TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH*

The *TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH* variable enables and disables Tensor Core math for cuDNN RNNs in TensorFlow.

Tensor Core math is enabled by default, but can be disabled by setting this variable to `1`. For more information, see Tensor Core Math.

This variable is disabled by default:

```
export TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH=0
```

To enable the variable, run the following command:

```
export TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH=1
```

7.1.8. *TF_DISABLE_CUBLAS_TENSOR_OP_MATH*

The *TF_DISABLE_CUBLAS_TENSOR_OP_MATH* variable enables and disables Tensor Core math for cuBLAS convolutions in TensorFlow.

Tensor Core math is enabled by default, but can be disabled by setting this variable to `1`. For more information, see Tensor Core Math.

This variable is disabled by default:

```
export TF_DISABLE_CUBLAS_TENSOR_OP_MATH=0
```

To enable the variable, run the following command:

```
export TF_DISABLE_CUBLAS_TENSOR_OP_MATH=1
```

7.1.9. *TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32*

The *TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32* variable enables and disables Tensor Core math for float32 matrix multiplication operations in TensorFlow.

Tensor Core math for float32 operations is disabled by default, but can be enabled by setting this variable to `1`. For more information, see Tensor Core Math.

This variable is disabled by default:

```
export TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32=0
```

To enable this variable, run the following command:

```
export TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32=1
```

7.1.10. *TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32*

The *TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32* variable enables and disables Tensor Core math for float32 convolution operations in TensorFlow.

Tensor Core math for float32 operations is disabled by default, but can be enabled by setting this variable to `1`. For more information, see Tensor Core Math.

This variable is disabled by default:

```
export TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32=0
```

To enable this variable, run the following command:

```
export TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32=1
```

7.1.11. *TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32*

The *TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32* variable enables and disables Tensor Core math for float32 cuDNN RNN operations in TensorFlow.Tensor Core math for float32 operations is disabled by default, but can be enabled by setting this variable to `1`. For more information, see Tensor Core Math.

This variable is disabled by default:

```
export TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32=0
```

To enable this variable, run the following command:

```
export TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32=1
```

7.1.12. *TF_ENABLE_LAYOUT_NHWC*

The `TF_ENABLE_LAYOUT_NHWC` variable enforces the NHWC (channels_last) format during the model execution. It is mainly useful for float32 and NVIDIA Ampere Architecture GPUs or later, where the TF32 Tensor Core will be used and NHWC format is preferred.

This environment variable is disabled by default. When disabled, the data format will be automatically selected by TensorFlow.

To enable the variable, run the following command:

```
export TF_ENABLE_LAYOUT_NHWC=1
```

7.1.13. *TF_ENABLE_NVTX_RANGES*

NVIDIA Tools Extension (NVTX) ranges add operation name annotations to the execution timeline when profiling an application with Nsight Systems or the NVIDIA Visual Profiler.

For more information on NVTX, see https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvtx

Instructions since the 21.03 release

The `TF_ENABLE_NVTX_RANGES` variable enables and disables NVTX ranges in TensorFlow. NVTX ranges are disabled by default, but can be enabled by setting this variable to 1.

This variable is disabled by default:

```
export TF_ENABLE_NVTX_RANGES=0
```

To enable the variable, run the following command:

```
export TF_ENABLE_NVTX_RANGES=1
```

In addition to setting `TF_ENABLE_NVTX_RANGES=1`, the variable `TF_ENABLE_NVTX_RANGES_DETAILED` can be set to 1 to obtain NVTX ranges with more detailed information (such as shapes and attributes).

This variable is disabled by default:

```
export TF_ENABLE_NVTX_RANGES_DETAILED=0
```

To enable the variable, run the following command:

```
export TF_ENABLE_NVTX_RANGES_DETAILED=1
```

Note that the `TF_DISABLE_NVTX_RANGES` variable (see below) is no longer supported.

Instructions prior to the 21.03 release

The *TF_ENABLE_NVTX_RANGES* variable is disabled by default:

```
export TF_DISABLE_NVTX_RANGES=0
```

To enable this variable, run the following command:

```
export TF_DISABLE_NVTX_RANGES=1
```

7.1.14. *TF_CUDNN_CTC_LOSS*

The `TF_CUDNN_CTC_LOSS` variable enables the cuDNN CTC loss backend via `nn.ctc_loss` for Tensorflow 2.x (for example, `19.11-tf2-py3`) or `nn.ctc_loss_v2` for Tensorflow 1.x (for example, `19.11-tf1-py3`).

This variable is disabled by default:

```
export TF_CUDNN_CTC_LOSS=0
```

To enable this variable, run the following command:

```
export TF_CUDNN_CTC_LOSS=1
```

7.1.15. *TF_GPU_ALLOCATOR*

The `TF_GPU_ALLOCATOR` variable enables the memory allocator using `cudaMallocAsync` available since CUDA 11.2. It has fewer fragmentation issues than the default BFC memory allocator. Note, this could become the default in the future.

This environment variable is unset by default.

To enable the variable, run the following command:

```
export TF_GPU_ALLOCATOR=cuda_malloc_async
```

7.1.16. *TF_GRAPPLER_GRAPH_DEF_PATH*

The `TF_GRAPPLER_GRAPH_DEF_PATH` variable outputs the Graphdef files before and after the TF grappler optimizations (For more information about the grappler optimizations, see the TensorFlow graph optimization with Grappler). In checking the optimized operation graph during the TF runtime, users can specify `TF_GRAPPLER_GRAPH_DEF_PATH=/`*`path`*`/`*`to`*`/`*`graphdef`*.

This environment variable is unset by default.

To enable the variable, run the following command:

```
export TF_GRAPPLER_GRAPH_DEF_PATH="path/to/graphdef"
```

Chapter 8. Performance

To achieve optimum TensorFlow performance, for image based training, the container includes sample scripts that demonstrate efficient training of CNNs. The sample scripts may need to be modified to fit your application.

The scripts can be found in the `/opt/tensorflow/nvidia-examples/cnn/` directory. Along with the training scripts, there is also some documentation that can be found in the `/opt/tensorflow/nvidia-examples/cnn/README.md` directory.

For more information, see Performance models and Benchmarks.

8.1. Tensor Core Math

The TensorFlow container includes support for Tensor Cores starting in Volta's architecture, available on Tesla V100 GPUs. Tensor Cores deliver up to 12x higher peak TFLOPs for training. The container enables Tensor Core math by default; therefore, any models containing convolutions or matrix multiplies using the `tf.float16` data type will automatically take advantage of Tensor Core hardware whenever possible.

Tensor Core math can also be enabled for `tf.float32` matrix multiply, convolution, and RNN operations by setting the `TF_ENABLE_CUBLAS_TENSOR_OP_MATH_FP32=1`, `TF_ENABLE_CUDNN_TENSOR_OP_MATH_FP32=1`, and `TF_ENABLE_CUDNN_RNN_TENSOR_OP_MATH_FP32=1` (for RNNs that use the `cudnn_rnn op`) environment variables, respectively. This mode causes data to be internally reduced to float16 precision, which may affect training convergence.

With Tensor Core math enabled, inputs of matrix multiply, convolution, and RNN operations are implicitly down-cast from FP32 to FP16. Internal accumulation and outputs remain in FP32. This allows FP32 models to run faster by using GPU Tensor Cores when available. Additionally, users should augment models to include loss scaling (for example, by wrapping the optimizer in a `tf.contrib.mixed_precision.loss_scale_optimizer`).

For more information about the architecture, see Inside Volta and Inside Turing.

8.1.1. Float16 Training

Training with reduced precision can in some cases lead to poor or unstable convergence.

NVIDIA recommends the following strategies to minimize the effects of reduced precision during training (see `nvidia-examples/cnn/nvcnn.py` for a complete demonstration of float16 training):

1. Keep trainable variables in float32 precision and cast them to float16 before using them in the model. For example:
```
tf.cast(tf.get_variable(..., dtype=tf.float32), tf.float16)
```
2. Apply loss-scaling if the model struggles or fails to converge. Loss scaling involves multiplying the loss by a scale factor before computing gradients and then dividing the resulting gradients by the same scale again to re-normalize them. A typical loss scale factor for recurrent neural network models is 128. For example:
```
loss, params = ...
scale = 128
grads = [grad / scale for grad in tf.gradients(loss * scale, params)]
```

8.2. Automatic Mixed Precision (AMP)

Using mixed precision training requires three steps:

1. Converting the model to use the float16 data type where possible.
2. Keeping float32 master weights to accumulate per-iteration weight updates.
3. Using loss scaling to preserve small gradient values.

Using automatic mixed precision with the TensorFlow framework can be as simple as adding one line of code. It accomplishes this by automatically rewriting all computation graphs with the necessary operations to enable mixed precision training and loss scaling. See Automatic Mixed Precision for Deep Learning for more information.

8.2.1. Automatic Mixed Precision Training In TensorFlow

For models already using an optimizer from `tf.train` or `tf.keras.optimizers` for both `compute_gradients()` and `apply_gradients()` operations (for example, by calling `optimizer.minimize()` or `model.fit()`), automatic mixed precision can be enabled by wrapping the optimizer with `tf.train.experimental.enable_mixed_precision_graph_rewrite()`.

Graph-based example:
```
opt = tf.train.AdamOptimizer()
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
train_op = opt.miminize(loss)
```

Keras-based example:
```
opt = tf.keras.optimizers.Adam()
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
model.compile(loss=loss, optimizer=opt)
model.fit(...)
```

For more information on this function, see the TensorFlow documentation here.

For backward compatibility with previous container releases, AMP can also be enabled for `tf.train` optimizers by defining the following environment variable:

```
export TF_ENABLE_AUTO_MIXED_PRECISION=1
```

When enabled, automatic mixed precision will do two things:

1. Insert the appropriate cast operations into your TensorFlow graph to use float16 execution and storage where appropriate -- this enables the use of Tensor Cores along with memory storage and bandwidth savings.
2. Turn on automatic loss scaling inside the training Optimizer object.

8.2.2. Conditions And Limitations

Ensure you are familiar with the following conditions:

Additional control

It is possible to enable the automatic insertion of cast operations without automatic loss scaling. The environment variable for doing so is `TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1`.

Caveats

Model types

Convolutional architectures that rely primarily on grouped or depth-separable convolutions (MobileNet and ResNeXt are popular examples) will not presently see speedups from float16 execution. This is due to the library constraints outside the scope of automatic mixed precision, though we expect them to be relaxed soon.

Optimizers

Automatic mixed precision loss scaling requires that the model code use a subclass of the built-in `tf.train.optimizer` or `tf.keras.optimizers.optimizer` classes. Furthermore:

- TensorFlow code that directly calls `tf.gradients` and uses those gradients “by hand” will not be supported.
- Instead, automatic mixed precision requires the paired calls to `optimizer.compute_gradients` and `optimizer.apply_gradients`, or a call to the high-level function `optimizer.minimize`.
- If the optimizer class is a custom subclass of `tf.train.optimizer` (not one built into TensorFlow), then it may not be supported by automatic mixed precision loss scaling. In particular, if the custom subclass overrides either `compute_gradients` or `apply_gradients`, it must take care to also call into the superclass implementations of those methods.

Multi-GPU

Prior to TensorFlow 1.14.0, automatic mixed precision did not support TensorFlow "Distributed Strategies." Instead, multi-GPU training needed to use Horovod (or TensorFlow device primitives).

Other notes

If your code already has automatic loss scaling support built-in, it will need to be disabled in order to avoid conflicting with automatic mixed precision own automatic loss scaling. Alternatively, the automatic mixed precision graph rewrite can be enabled without enabling loss scaling by using the option described above.

8.2.3. FAQs

Q: What if my model code already supports mixed precision training?

If the code is already written in such a way to follow the Mixed Precision Training Guide, then automatic mixed precision will leave things as they are. For example, the CNN examples provided inside the NVIDIA TensorFlow container use mixed precision training by default. If you would like to evaluate how they work with automatic mixed precision, be sure to run them with the flag `--precision=fp32`.

Q: How much faster will my model run with automatic mixed precision?

There are no precise rules for mixed precision speedups, but here are a few guidelines:

- The more time is spent in matrix multiplication (dense layers) or convolutions, the more Tensor Cores can accelerate the model. This means that “bigger” models often see larger speedups. In particular, very small dense and convolution layers will see limited benefit from automatic mixed precision, since there is not enough math to fully exploit Tensor Cores.
- Mixed precision models use less memory than FP32, so it is possible to increase the batch size when running with automatic mixed precision. Thus, you can often increase the speedup by increasing the batch size after enabling automatic mixed precision.

Q: How can I see what changes automatic mixed precision makes to my model?

Because automatic mixed precision operates at the level of TensorFlow graphs, it can be challenging to quickly grasp the changes it makes: often it will tweak thousands of TensorFlow operations, but those correspond to many fewer logical layers. You can set the environment variable `TF_CPP_VMODULE="auto_mixed_precision=2"` to see a full log of the decisions automatic mixed precision makes (note that this may generate a lot of output).

Q: Why do I see only FP32 datatypes in my saved model GraphDef?

When you save a model graph or inspect the graph with `Session.graph` for `Session.graph_def`, TensorFlow returns the unoptimized version of the graph. Automatic mixed precision works as an optimization pass over the original graph, so its changes are not included in the unoptimized graph. You can set the environment variable

`TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_LOG_PATH="my/log/path"`, and automatic mixed precision will save out pre- and post-optimization copies of each graph it processes to that directory.

Q: Why do I see `step=0` repeated multiple times when training with automatic mixed precision?

The automatic loss scaling algorithm that automatic mixed precision enables can choose to "skip" training iterations as it searches for the optimal loss scale. When it does so, it does not increment the global step count. Since most of the skips occur at the beginning of training (usually fewer than ten iterations), this behavior manifests as multiple iterations where the step counter stays at zero.

Q: How are user-defined custom TensorFlow operations handled?

By default, automatic mixed precision will leave alone any op types it doesn't know about, including custom operations. That means the types of the op's inputs and outputs are not changed, and automatic mixed precision will insert casts as necessary to interoperate with the rest of the (possibly-changed) graph.

If you would like to make automatic mixed precision aware of a custom op type, there are three environment variables you can use:

`TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_ALLOWLIST_ADD`
: These are ops for which it is worth casting the inputs to FP16 to get FP16 execution. Mostly, they are ops that can take advantage of Tensor Cores.

`TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_INFERLIST_ADD`
: These are ops for which FP16 execution is available, so they can use FP16 if the inputs happen to already be in FP16 because of an upstream `ALLOWLIST` op.

`TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_DENYLIST_ADD`
: These are ops for which FP32 is necessary for numerical precision, and the outputs are not safe to cast back to FP16. Example ops include `Exp` and `Log`.

Each of these environment variables takes a comma-separated list of string op names. For example, you might set export `TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_ALLOWLIST_ADD=MyOp1,MyOp2`. The op name is the string name used in the call to `REGISTER_OP`, which corresponds to the name attribute on the operation's `OpDef`.

Q: Can I change the algorithmic behavior of automatic mixed precision?

The primary lever for controlling automatic mixed precision behavior is to manipulate what ops lie on each of the allowlists, inferlists, and denylists. You can add ops to each using the three environment variables above, and there is a corresponding variable `TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_{ALLOWLIST,INFERLIST,DENYLIST}_REMOVE` to take built-in ops off of each list.

Chapter 9. XLA Best Practices

9.1. XLA Introduction

XLA is an optimizing graph compiler for TensorFlow. It optimizes parts of the TensorFlow GraphDef in an attempt to improve performance.

Unlike native TensorFlow, which executes GraphDef nodes one at a time, XLA considers many GraphDef nodes at once and generates optimized code for these nodes.

9.1.1. Why Use XLA?

In many cases, XLA improves performance over native TensorFlow. The major difference between these two is the fusion optimizer in XLA. Instead of executing many small kernels back to back, XLA optimizes these into larger kernels. This greatly reduces execution time of bandwidth bound kernels. XLA also offers many algebraic simplifications, far superior to what Tensorflow offers.

9.1.2. Enabling XLA

XLA can be enabled in a few ways:

- Opt in for specific parts of the model, referred to as manual clustering.

  tf.function is one way for users to control which parts of a model to optimize with XLA. See here for TF1.X, and here for TF2.X how to use a `tf.function`.

  Alternatively, a so-called *jit_scope* can be used to control which parts of a model to optimize with XLA. See here for TF1.X, and here for TF2.X how to use a `jit_scope`. Be advised that this is merely a hint, as unsupported nodes are not compiled.

  Make TensorFlow decide which parts of the model to optimize, referred to as auto clustering.
- Opt in at the source level.

  For TF 1.X, users can opt in to auto clustering at the session level by adding the following line of code to change the session configuration argument:

  ```
  config.graph_options.optimizer_options.global_jit_level =
  tf.OptimizerOptions.ON_1
  ```

For TF 2.X, users can opt in to auto clustering at the global level by adding the following line of code at the beginning of the script:

```
tf.config.optimizer.set_jit(True)
```

- Opt in through environment variable.

  Users can opt into auto clustering, where TensorFlow decides which nodes go to XLA, without any source code changes by setting the following environment variable:

  ```
  TF_XLA_FLAGS=--tf_xla_auto_jit=1
  ```

  This environment variable works for both TF 1.X, and TF 2.X. For models that have XLA enabled through the session config, XLA can be disabled with:

  ```
  TF_XLA_FLAGS=--tf_xla_auto_jit=-1
  ```

The environment variable takes precedence over the `global_jit_level` and the `jit_scope`, but not over `tf.function`. This document focuses primarily on auto clustering. Most of it applies to manual clustering as well.

9.1.2.1. XLA Lite

TensorFlow autoclustering clusters operations based on an allow-list.

By default, this list is quite extensive. This can result in large clusters which can result in loss of performance or excessive memory usage (see below). To mitigate such problems, users can make TensorFlow only cluster “small” and fusible operations such as pointwise and reduction operations, by setting:

```
TF_XLA_FLAGS=--tf_xla_auto_jit=fusible
```

Note that since models always have matrix multiplications or convolutions, this results in more, but smaller clusters.

9.1.3. XLA Caveats

Using XLA incurs two different costs when comparing against a native TensorFlow execution:

1. Compilation time

   Code must be compiled at runtime. This takes time depending on the size of the generated clusters and the number of times it is compiled (once for every shape instance), this time might not be recoverable during execution.
2. Execution time overhead due to the TF-XLA interface.

   The way that XLA integrates back into TensorFlow adds a cost to the graph execution.

Both of these are discussed in more depth throughout the document. These caveats are mentioned here to highlight that XLA is not a silver bullet that speeds up all models under all scenarios. Short running scripts, on small batch sizes, are typically not good candidates for XLA. Models that exhibit dynamic shapes are another class of models that often don’t do well when using XLA.

Furthermore, the overhead incurred from using XLA is hard to pin-point. Some overhead occurs at the beginning of the model, others occur throughout the execution with fixed cost, and others yet occur irregularly during execution. It is important to be aware of this when instructing the code to get timing information from a model execution.

Besides these performance caveats, XLA sometimes causes an out-of-memory during execution. This is discussed in more detail later as well.

9.2. TF-XLA Integration

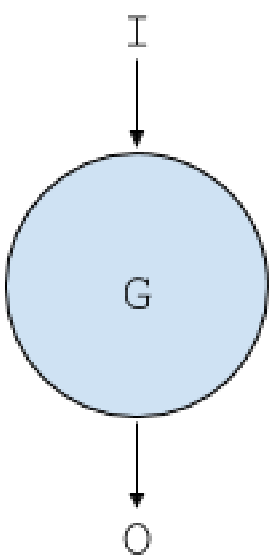
This section describes how XLA currently changes the executed graph, and how it affects the execution of the graph.

This helps developers understand how TensorFlow interfaces with XLA, and how this affects performance. This section, as well as the remainder of this document, assumes the reader to have a rudimentary knowledge of TensorFlow GraphDef, and how the TensorFlow stream executor executes it.

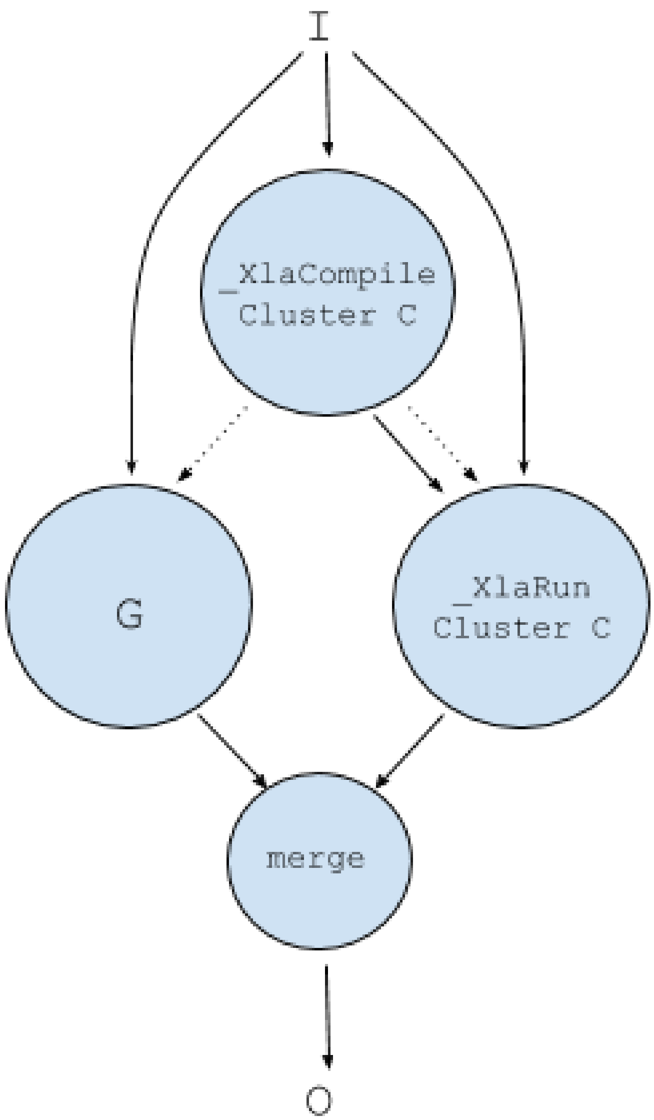
9.2.1. Changes to the TensorFlow Graph

9.2.1.1. Clustering

When auto clustering is enabled, a part of the graph is chosen to be compiled with XLA. This part (`G`), has a set of inputs (`I`) and a set of outputs (`O`).



When XLA is enabled, this graph is transformed into the following graph for execution:



The part (G) has been clustered into a cluster (C). Be aware that C is merely a cluster of operations and has not yet been compiled. It cannot be compiled yet, as the actual shapes of the input and output tensors are not available until execution time.

When auto clustering, each part of the model that is clustered is replaced with a graph depicted above, referring to a cluster that represents that part. Note that the original GraphDef representation of the graph is still there to serve as a fallback path. The merge nodes (in the actual graph, there is one merge node per output) forward the outputs from either the native TensorFlow execution or the XLA execution.

9.2.1.2. TensorFlow Graph Execution with XLA

During the graph execution, when the above graph is reached, the following happens *for each unique shape instance* of the set of inputs.[1]

1. The first two times the `_XlaCompile` node is executed, it falls back to the original GraphDef execution of `G`.
2. The third time the `_XlaCompile` node is executed, cluster C is compiled into an XLA binary. After the compilation has finished, the binary is stored in a cache, accessible through a key that encodes the cluster, and the shapes and types of the inputs. This key is passed to the `_XlaRun` node, and the binary is executed. If compilation of the cluster took longer than 30 seconds, it will not be compiled for any other shape instance in the future.
3. Any subsequent time the `_XlaCompile` node is executed, compilation is skipped and the key of the cached binary is passed to `_XlaRun` node to execute.

   Note that in order for the `_XlaCompile` node to execute, all inputs (`I`) must be ready. Similarly, none of the outputs (O) are ready until either `G` or `_XlaRun` are done executing.

   Note that the above steps are for each unique shape instance. If the shape of any input changes throughout the model execution, recompilation of the cluster happens.

9.2.2. Symptoms of XLA Issues

This section mentions the most common issues observed when using XLA, and why these can happen. The next section focuses on how to address them.

9.2.2.1. Functional Issues

The most common functional issues fall into two categories:

1. Different output.

   XLA is a compiler that performs semantically equivalent transformations only.[2] Since these transformations are on floating point tensors, the outcome is most likely (slightly) different compared to the native TensorFlow execution. Developers should be aware of this.
2. Out of memory.

   XLA, by nature of its design, increases the amount of memory needed to execute. As mentioned earlier, all inputs must be ready before execution, and all outputs are ready at the same time after execution. This means memory for all inputs and outputs must be allocated for the execution. Furthermore, all memory required to hold the intermediate tensors in the XLA binary must be allocated.

[1] This is based on TF 1.15.2+nv. The numbers in the heuristic can change between releases.
[2] XLA bugs excluded. XLA does not change the accuracy of the operations.

9.2.2.2. Performance Issues

The most common performance issues fall into four categories:

1. TF-XLA integration.

   The way that TF interfaces with XLA (as described in 9.2.1), comes at a performance cost, partly due to the extra nodes inserted in the graph.
2. Compile time overhead.

   When a model has input data of varying shapes (for example sentences for an NLP model), compilation overhead can add up. Large clusters contribute to this as well.
3. Compute/communication overlap.

   XLA requires all inputs to be ready before execution, and the outputs are ready when the execution finishes. This synchronization can contribute to a performance when compared to Tensorflow. TensorFlow has a fine-grained execution model, that allows for overlap of copy and computation, which is not possible when using XLA.
4. XLA optimization and code generation.

   Sometimes the XLA optimizer or code generator don't do as well as they could. In our experience this is the least common scenario when dealing with performance regressions.

9.3. Identifying and Managing the Issues

For developers, XLA is mostly a black box. This section sheds some light on how to identify the symptoms, and take appropriate action in an attempt to solve these issues.

9.3.1. Controlling XLA with Environment Variables

XLA exposes many ways to either retrieve information from it, or control the behavior of it. These options are not (well) documented, and can even change from release to release (behavior, format, and even existence). Even then, they offer lots of insights into XLA's innards. They come in two classes:

- TF_XLA_FLAGS environment variable

  This variable can be set to options that control the TF-XLA boundary. It can be set to a space separated list of options, which can be found in `tensorflow/compiler/jit/flags.cc`
- XLA_FLAGS environment variable

  This variable can be set to options that control the XLA optimizer and code generator. It can be set to a space separated list of options, which can be found in `tensorflow/compiler/xla/debug_options_flags.cc`

Most of the options in these source files are primarily used by XLA developers and go beyond the scope of this document. Some are useful to model developers, and are mentioned below.

9.3.2. Out of Memory Issue

This is very easy to identify, as it is reported by the TensorFlow allocator. The reason for out-of-memory has been described in Functional Issues. When this happens, run the model with this extra parameter:

`TF_XLA_FLAGS=--tf_xla_always_defer_compilation=true` [3]

This option instructs TensorFlow to never compile a cluster, but always execute the fallback path. If this still results in an out-of-memory error, the problem is that the inputs (`I`) and outputs (`O`) don't all fit in memory simultaneously. If the execution succeeds, the problem is that the inputs (`I`), outputs (`O`) and all intermediate tensors in the XLA binary don't all fit in memory simultaneously.

In both cases, the only way to guranteed address an out-of-memory issue in XLA is by reducing the number of operations in a cluster. The number of operations (in)directly affects the number of inputs, outputs, and intermediate tensors.

One way to reduce the cluster sizes is by enabling XLA-Lite. This pretty much guarantees this issue to disappear, as the cluster sizes are reduced rather drastically.

Another way is to limit the maximum size of each cluster. By default, there is no upper bound to the size of a cluster. The sizes of clusters can by retrieved with:

```
TF_CPP_VMODULE=mark_for_compilation_pass=2
```

Look for occurrences of " *** Clustering info for graph" to get the sizes of the generated clusters.

When running with:

```
TF_XLA_FLAGS=--tf_xla_max_cluster_size=<n>
```

where `<n>` is smaller than the largest cluster size, an upper bound can be found that avoids the out-of-memory.

Reducing the mini-batch size is another proactive way an out-of-memory issue can possibly be avoided. This requires no changes to XLA. A smaller mini-batch size, in combination with XLA, can still outperform native TensorFlow.

The TensorFlow stream executor executes many operations in parallel. This increases the memory requirement as more intermediate tensors are alive concurrently. Limiting the amount of parallelism in the stream executor often reduces the amount of memory needed. The amount of parallelism is controlled by specifying the number of threads the executor can use, by setting the following environment variable:

```
TF_NUM_INTEROP_THREADS=<n>
```

[3] The real setting should be: `TF_XLA_FLAGS='--tf_xla_auto_jit=1,--tf_xla_always_defer_compilation=true'`. The remainder of the document uses this shorthand. It is assumed that XLA is on in all these cases.

9.3.2.1. Memory Fragmentation

By default, XLA allocates the required memory for the intermediate tensors in one allocation. Because of memory fragmentation, it can happen that the TensorFlow allocator can not allocate a contiguous memory buffer eventhough ample memory is still available.

By setting

```
XLA_FLAGS=--xla_multiheap_size_constraint_per_heap=<n>
```

a user can break up the monolithic memory allocation into multiple ones, each with a maximum size of `n` bytes.

Alternatively, TensorFlow-XLA Integration Issues describes an option that sometimes successfully circumvents memory fragmentation.

9.3.3. TensorFlow-XLA Performance Issues

9.3.3.1. TensorFlow-XLA Integration Issues

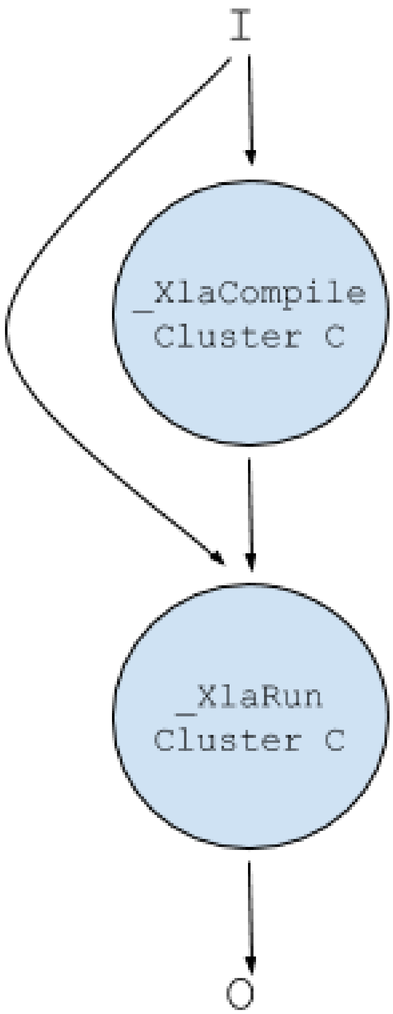
When XLA performs worse than native TensorFlow, one of first things to try is to run it again with:

```
TF_XLA_FLAGS=--tf_xla_always_defer_compilation=true
```

If the performance is still worse, the TF-XLA integration is the issue, as this option will run the fallback path without any compilation. In that case, run again with:

```
TF_XLA_FLAGS=--tf_xla_enable_lazy_compilation=false
```

This will completely change the way that TensorFlow interfaces with XLA. Unlike the diagram in Clustering, the graph now looks like:



Note that the fallback path is no longer present. This option often delivers good performance results for nodes that are executed many times for a given shape instance. It avoids the merge nodes and the compilation heuristics, as it must compile on first execution. A direct side effect of this option is a very different graphDef node execution (execute a single `_XlaRun` node instead of many nodes in the fallback path), and therefore a very different usage of the TensorFLow memory allocator. There are scenarios where this option successfully avoids the memory fragmentation issue mentioned before. This is a mere accidental side-effect, and not a fix.

A second manner in which the TX-XLA interface can manifest itself as a performance issue is when lots of smaller clusters occur. When this happens, the overhead of the extra nodes in the graph, the compilation time (little as it is), and the XLA executor is worse than the performance gain achieved by the XLA compiler. By default, the lower bound for a cluster size is 4 operations. When running with:

```
TF_XLA_FLAGS=--tf_xla_min_cluster_size=<n>
```

where `<n>` is larger than 4, any cluster with less than `<n>` operations will be ignored, and the graph remains unchanged.

9.3.3.2. Compilation Overhead

Compile time overhead can cause severe performance degradation, especially in models with varying shapes of input data.

To know how much time is spent on compilation, run the model with:

```
TF_CPP_VMODULE=xla_compilation_cache=1
```

This dumps information after each compilation happens. It shows how long the last compilation took, and the accumulated time of all compilations up to that moment. When running with:

```
TF_CPP_VMODULE=nvptx_compiler=2
```

it also dumps information regarding the size of the .ptx files and .cub files. Compile time overhead can be reduced by either increasing the lower bound for clusters (when there are many small compilations):

```
TF_XLA_FLAGS=--tf_xla_min_cluster_size=<n>
```

or by decreasing the upper bound for cluster sizes when some compilations take too long:

```
TF_XLA_FLAGS=--tf_xla_max_cluster_size=<n>
```

XLA can also perform cluster compilations in the background, while execution of the fallback path can make progress. These background compilations take CPU resources away from the main execution pipeline, but for many cases, not blocking the execution while waiting for compilation to finish can lower the end to end latency. Asynchronous compilation can be opted into by setting

```
TF_XLA_FLAGS=--tf_xla_async_compilation=true
```

Asynchronous compilation is not compatible with disabling of lazy compilation. When lazy compilation is disabled, it takes precedence over asynchronous compilation.

XLA can persistently cache parts of a cluster compilation to disk. This cache can be used to speed up the compilation time of subsequent runs by reusing cached compilation results. Persistent caching is enabled by setting

```
TF_XLA_FLAGS=--xla_gpu_persistent_cache_dir=<dir_name>
```

`dirname` must refer to an existing directory, with read and write permissions. It will be used to store new entries, and retrieve existing entries. The recommended usage is a small cache per model, as all cache entries are read at initialization time and entries not used by the model will therefore be read, but never used. This is an experimental feature, and compatibilty of the cache with respect to the TensorFlow version is not guaranteed.

9.3.3.3. Compute/Communication Overlap

As mentioned earlier, as a direct side effect of how XLA integrates with TensorFlow, all outputs of an `_XlaRun` node are ready at the same time.

This means that any consumer of any XLA output must wait until all outputs are produced. This limits the TensorFlow executor to execute computation and communication in parallel. The fact is that inside an XLA cluster it is often the case that outputs have already been computed, and could be passed on to a consumer. XLA has an optimization referred to as AyncIO, that allows consumers of XLA outputs to start execution the moment the output is available.

If XLA on one GPU outperforms native TensorFlow, but performs worse on multi-GPU systems, it could be because of the lack of compute and computation overlap. Run TensorFlow with:

```
TF_XLA_FLAGS=--tf_xla_async_io_level=1
```

to enable AsyncIO.

9.3.4. XLA Optimizer and Code Generation

In most cases, the TF-XLA interface issues disappear when:

- Running long enough. Compilation overhead decreases over time, as most shape instances of the clusters will hit in the cache.
- Running with large enough tensors, i.e., a large enough batch size.
- Filtering out tiny clusters.

If XLA compiled clusters run slower than the fallback path, the XLA compiler could be the reason for the performance degradation.

9.3.4.1. XLA Autotune

By default, XLA performs autotuning over the GEMM and CONV algorithms in cuBLAS and cuDNN. For each GEMM, all possible alternatives in cuBLAS are tried, and the fastest one is picked; similarly for each CONV operation into cuDNN. Autotuning is controlled with a level [0..4]. Level 4 is the default and it performs some functional tests between the different alternatives to catch cuBLAS and cuDNN problems.

Try running with:

```
XLA_FLAGS=--xla_gpu_autotune_level=2
```

This does full autotuning, but skips some initialization and comparisons done at level 4. For short running scripts, even disabling autotuning altogether can improve performance:

```
XLA_FLAGS=--xla_gpu_autotune_level=0
```

9.3.4.2. Various Options

cuDNN Batch Normalization

By default the XLA breaks down the batch normalization layer into many smaller operations that then get optimized by XLA. For Training only, XLA has an option to preserve batch normalization as a single operation that targets the cuDNN batch normalization API to execute it. Run with:

```
XLA_FLAGS=--xla_gpu_use_cudnn_batchnorm_level=2
```

to use cuDNN to execute the batch normalization layer for both forward and backward layers.

tf.enable_resource_variables()

TensorFlow resource variables are improved versions of TensorFlow variables. XLA does not cluster variables, whereas is does cluster resource variables. Clustering resource variables therefore clusters more graphDef nodes, including *assign* nodes. We've seen at least one model in which many *assign* nodes were all depending on outputs from an XLA cluster. Enabling resource variables made these nodes a part of the XLA cluster, improving the latency, as these *assign* nodes did not have to wait for the XLA cluster to finish. Enabling resource variables is achieved by adding:

```
tf.enable_resource_variables()
```

to the model source before any variables have been created.

--xla_backend_optimization_level

```
XLA_FLAGS=--xla_backend_optimization_level=0
```

--xla_gpu_disable_ptxas_optimizations

```
XLA_FLAGS=--xla_gpu_disable_ptxas_optimizations=true
```

9.4. XLA Options Reference

This section lists the options mentioned throughout this document with a short description.

TF_XLA Options

- `--tf_xla_auto_jit`

  Controls clustering. 0: off , 1: on, fusible: Xla-Lite

- `--tf_xla_always_defer_compilation`

  Controls deferred compilation. true: fallback path, false: default strategy

- `--tf_xla_max_cluster_size`

  Sets upper bound for cluster size

- `--tf_xla_min_cluster_size`

  Sets lower bound for cluster size

- `--tf_xla_enable_lazy_compilation`

  Controls lazy compilation. false: always compile, true: default strategy

- `--tf_xla_async_compilation`

  Controls asynchronous compilation. false: default strategy, true: compile asynchronously

  Available as of 20.10

- `--xla_gpu_persistent_cache_dir`

Controls a persistent compilation cache. When set to an existing directory, that directory is used as the compilation cache.

Available as of 21.04

- `--tf_xla_async_io_level`

  Enable asynchronous IO. 0: off, 1: on

XLA Options

- `--xla_gpu_autotune_level`

  Controls autotune level, 0: off, 1: w/o initialization, 2: w/ initalizalization, 3: w/ re-initialization, 4: w/ functional check

- `--xla_multiheap_size_constraint_per_heap`

  Controls whether to multiple heaps. -1: single heap, `n`: use multiple allocations with a maximum of `n` bytes each.

  Available as of 20.11

- `--xla_gpu_use_cudnn_batchnorm+level`

  Controls whether to use cuDNN batchnorm. 0 no, 1: BatchNorm+Bias+Act only, 2: all cases.

Chapter 10. Troubleshooting

10.1. Support

For more information about TensorFlow, including tutorials, documentation, and examples, see:

- TensorFlow tutorials
- TensorFlow API
- Automatic mixed precision
- Tensor Cores
- Deep Learning Examples optimized for Tensor Cores

For the latest TensorFlow Release Notes, see the https://docs.nvidia.com/deeplearning/frameworks/tensorflow-release-notes/index.html.

Notice

HDMI

OpenCL

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com