GRAPHCORE

Porting TensorFlow models to the IPU

Version latest

Graphcore Ltd

Oct 21, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

This document is a practical guide to porting TensorFlow™ models to the Poplar SDK for running on the IPU. It is assumed that the reader is aware of the document Targeting the IPU from TensorFlow, which serves as the primary introduction to developing TensorFlow models for the IPU. It provides not only a conceptual introduction to developing models at the framework level, but also details a number of specific facets of the TensorFlow-to-Poplar API that are pivotal to running models on the IPU. In what follows, this general guide will be referred to as various topics arise.

This document will in turn focus on some of the practical considerations of developing a model for the IPU and provide some guidance on best practices. In doing so, it will attempt to identify those key elements that assist the developer in transitioning to using TensorFlow on the IPU.

Note: This document currently applies to the Graphcore port of TensorFlow 1.15.

The scope of this document includes:

- How to approach porting in general and which questions to ask up front
- Code examples that highlight IPU-specific API functions
- Preliminaries such as Bash environment setup and Python import statements
- The role of infeeds and outfeeds in boosting computational throughput
- Profile report generation to help you identify compute or memory inefficiencies
- The IPUEstimator, a TensorFlow abstraction that facilitates session handling

A working knowledge of the above elements will allow the framework developer to take their first steps in transitioning to the IPU/Poplar/TensorFlow compute stack.

An example application that demonstrates the use of IPUs to train CNNs including ResNet, ResNeXt and EfficientNet is available in the Graphcore examples repository on GitHub: https://github.com/graphcore/examples/tree/master/applications/tensorflow/cnns/training

CHAPTER

TWO

HOW TO TRANSFER AN EXISTING CODE BASE

2.1 Preliminary analysis

Before porting TensorFlow models to the IPU, it can be helpful for you to assess the problem at hand.

What is the model size? A single 2nd generation IPU has an on-chip memory/SRAM of approximately 900MB (or ~300 MB for the first generation). Will I use the IPU for training or inference? Training will require require more memory. Here, it is also good to get an idea about the minimal batch size for the model to converge correctly, for example for batch normalization. Depending on these estimates, it is possible to get an idea of whether the whole model will fit onto one IPU or if a setup with multiple IPUs is necessary.

An analysis of existing code can indicate the kind of parallelism desired. Some programs distribute the same task on multiple devices, while feeding each with a different batch of data (data parallel) and others spread one task and one batch of data across multiple devices (model parallel). There are also cases, typically seen in reinforcement learning applications, where inference and training are distributed in parallel across different devices. IPUs support all of the above schemes.

Additionally, it is important for you to assess the suitability of the existing code for porting. On one hand, if the code is mainly based on the estimator's train, evaluate, and predict functions, replacing it by the `IPUEstimator` might be all that is required. On the other hand, if instead of the estimator approach, when the `tfcompile` tool is used for code optimization together with feeds and fetches, replacing the compiler and data streams with the respective IPU counterparts could ease the transition.

If your code is very specialized for a device (for example TPU estimator or other TPU specific functions) and not general at all, it is maybe better for you to start with the existing model code and refactor the surrounding code. Graphcore provides a repository with plenty of code examples at https://github.com/graphcore/tutorials and https://github.com/graphcore/examples. It is worth exploring these to check if a similar problem has been addressed already.

The IPU documentation includes a list of supported operations (depending on the related data types) - see the supported operators chapter of the TensorFlow User Guide. If your code contains operations that are not yet supported, either a new IPU implementation will be required, or the respective part has to be processed on the CPU which is achieved by scoping. See the custom operators chapter of the TensorFlow User Guide for more information.

2.2 Planning the transfer

The following guidelines give you some ideas about to transfer your model. There are multiple non-exclusive paths which can be taken.

If your model is sufficiently small and a code transfer looks straightforward, it makes sense to start with the full model. If the model is very big or contains components that are not supported by the IPU, it makes sense to first start with a small version of the model.

A similar approach is useful when looking at the code base. If the implementation is rather simple, a direct change will probably work. Otherwise, it is good to break it down into smaller development stages. Since more complex code should come with meaningful unit tests, tests are probably a good starting point.

A different approach is to use our repository of examples. Instead of changing an existing code base, it might be easier to just transfer the model of interest to an existing example.

2.3 Next steps

The first step after planning is to port the model and work through common initial issues like operations that have to be mapped onto the CPU (see Section 5, Scoping and determining unsupported operations), or wrong input datatypes for the estimator because the original code provides iterators instead of datasets or feeds. If no estimator is used, it is recommended to use feeds to enable optimal communication between host and IPU (see Section 3, ResNeXt inference example).

The second step is to profile the code to understand potential bottlenecks that might impact processing speed or memory consumption. One approach to explore bottlenecks outside of the IPU is the intrinsic Python profiler (`cprofile`). In Section 3.4, Generating a report, we describe how the IPU can be efficiently profiled.

Thirdly, if a smaller model was used for getting started, it is now time to scale the model with potentially further profiling. There are two approaches for scaling, if the model easily fits on one IPU, the batch size can be increased and multiple IPUs can process the data in parallel. If your model does not fit on a single IPU, it is recommended to spread the model across multiple IPUs. See the document on Model Parallelism.

CHAPTER

THREE

RESNEXT INFERENCE EXAMPLE

When being introduced to a new API, it is often helpful to have a working example of code to get a general overview of the key elements involved. A particular model which is useful to review given its simple, but non-trivial topology is ResNeXt.

ResNeXt is an Inception inspired model based on ResNet with repeated computational blocks interspersed with residual connections. Its primary distinguishing characteristic is the use of group convolutions in its module compute structure. Group convolutions, as opposed to conventional convolutional layers, partition the output channels of the operation into segregated groups. The number of segregated groups is referred to as the model's cardinality, which the authors state allows for more robust syntax extraction by allowing for more complex transformations. An illustration of cardinality is given in Fig. 3.1 below, where each of the [256, 1, 1, 4] streams in the graph represent a distinct convolution set, while the structure as a whole is a group convolution.

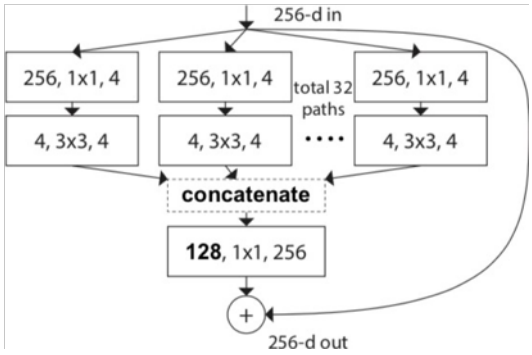

Fig. 3.1: ResNeXt cardinality expressed via group convolutions. Refer to Xie (2017) for more details.

3.1 Abridged sample code

In the code sample that follows, only those facets that are specific to the IPU API are explicitly documented, while other general items of TensorFlow development are identified but mostly omitted. A full working version of the code can be found in Section 7, ResNeXt full code example. Omissions are indicated by … and accompanied by comments to define the nature of what has been redacted.

```
import tensorflow as tf
... # Various additional import statements

# Import IPU API
from tensorflow.python import ipu

# Create a dataset for in-feed interface into session call
def create_input_data(batch_size=1, height=224, width=224, channels=4):
    # Synthetic input data follows NHWC format
    input_data = np.random.random((batch_size, height, width, channels))
    input_data = tf.cast(input_data, DTYPE)

    ds = tf.data.Dataset \
```

(continues on next page)

(continued from previous page)

```
        .range(1) \
        .map(lambda k: {"features": input_data}) \
        .repeat() \
        .prefetch(BATCHES_PER_STEP).cache()
    return ds

... # Various layer wrappers required by model definition,
    # (e.g. convolution, max pool)

# Group convolution definition
def group_conv(x, ksize, stride, filters_in, filters_out, index=0, groups=1,
               dtype=tf.float16, name='conv'):
    with tf.variable_scope(name, use_resource=True):
        # Define a weight variable
        W = tf.get_variable("conv2d/kernel" + str(index),
                            shape=[ksize, ksize,
                                   filters_in.value / groups,
                                   filters_out], dtype=dtype,
                                   trainable=True,
                                   initializer=tf.variance_scaling_initializer())
        # Implicit group convolution since channels of W are fraction of x
        return tf.nn.conv2d(x, filters=W, strides=[1, stride, stride, 1],
                            padding='SAME')

def group_conv_block(x, first_stride, filters, count, name='', cardinality=4):
    ... # Define the modular group convolution block as described in the paper
    return x

# Define the ResNext model
def resnext101_model():
    def body(features):
        with tf.variable_scope("VanillaResNeXt"):
            ... # Elements of model definition
            output = fc(x, num_units_out=1000)
            outfeed = outfeed_queue.enqueue(output)
            return outfeed
    return tf.python.ipu.loops.repeat(n=BATCHES_PER_STEP, body=body,
                                      infeed_queue=infeed_queue)

if __name__ == '__main__':
    # no simulation
    IPU_MODEL = False

    ... # Various additional variables

    # Create input data using randomized numpy arrays
    dataset = create_input_data(batch_size=BATCH_SIZE, height=224, width=224, channels=4)

    if IPU_MODEL:
        os.environ['TF_POPLAR_FLAGS'] = "--use_ipu_model"

    # Setup infeed queue
    with tf.device('cpu'):
        infeed_queue = ipu.ipu_infeed_queue.IPUInfeedQueue(dataset,
                                           feed_name="inference_infeed")

    # Setup outfeed
    outfeed_queue = ipu.ipu_outfeed_queue.IPUOutfeedQueue(feed_name="outfeed")

    # Compiles graph and targets IPU(s)
    with ipu.scopes.ipu_scope('/device:IPU:0'):
        res = ipu.ipu_compiler.compile(resnext101_model, inputs=[])

    # Setup IPU configuration and build session
    cfg = ipu.config.IPUConfig()
    cfg.convolutions.poplar_options["availableMemoryProportion"] = "0.3"
    cfg.auto_select_ipus = 1
    cfg.configure_ipu_system()
    ipu.utils.move_variable_initialization_to_cpu()
    outfeed = outfeed_queue.dequeue()
```

(continues on next page)

(continued from previous page)

```
84
85      # Create a session initiation and run the model
86      with tf.Session() as sess:
87          fps = []
88          latency = []
89          sess.run(infeed_queue.initializer)
90          sess.run(tf.global_variables_initializer())
91          # Warm up
92          print("Compiling and Warmup...")
93          start = time.time()
94          sess.run(res)
95          outfeed = sess.run(outfeed)
96          for iter_count in range(NUM_ITERATIONS):
97              print("Running iteration for benchmarking: ", iter_count)
98              sess.run(res)
99              sess.run(outfeed)
100             ... # Various summary statistics
```

In the following three sections, we review specific elements of the code presented, using the line numbers to identify the pertinent code elements.

3.2 Preliminaries: getting up and running

Before running the script, it is necessary to ensure a Poplar SDK has been downloaded and extracted (for more information see the Getting Started guide for your IPU system) on an IPU-enabled platform and that the environment variables are set appropriately.

```
1  # Export statements
2  export TMPDIR=/mnt/data/username/tmp/
3  export TF_POPLAR_FLAGS="--executable_cache_path=/mnt/data/username/ipu_cache/"
4  export POPLAR_LOG_LEVEL=INFO
```

Moving on to the export statements, Poplar and TensorFlow's XLA backend, both cache parts of the compilation to speed up graph construction. It is important to make sure there is enough space for it, which is why the first export statement points to a temp scratch directory. The second export sets flags useful during development. The IPU works on static graphs that need to be compiled before execution. There can be a significant time spent on compiling the computational graph for the IPU. Caching the binary makes sure that when you run the same program again, the binary is loaded instead of being recompiled. In addition, for repeated calls of `session.run` or `estimator.train`, it will speed up processing time after the first run. The last export item is to increase verbosity of the IPU compilation process.

Returning to the sample code, line 5 imports the IPU API. The next paragraph details how to set up the optional arguments to configure the hardware and software stack for the run.

3.3 Configuring the IPU

There are several configuration parameters that are available to you, and the document Targeting the IPU from TensorFlow provides valuable insight into these settings. Here, we review some that are frequently required and explain their role. From lines 77 to 82, the script sets the working configuration for the IPU.

Line 79 sets the `availableMemoryProportion` for convolutions. This parameter represents the proportion of tile memory to be made available as temporary memory for convolutions - it can vary between 0 and 1.0. Less temporary memory allocated will result in a higher number of cycles to compute a given convolution task, but too much memory allocation may oversubscribe the tile. Profiling, discussed in the next section, will provide insight into how this parameter affects model compilation. Finally, line 80 determines how many IPUs are required to compile and run the model.

3.4 Generating a report

In developing TensorFlow models for the IPU, it is critical to profile the compiled graph when it is deployed to hardware. A variety of key elements can be documented by doing so that include the total size of the compiled model, the tile balance of consumed memory, and the cycle counts of the various compute processes. To generate profile data, it is sufficient to set the following environmental variable:

```
POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true"}'
```

Note: When profiling, it is recommended to set `BATCHES_PER_STEP` (controlling infeeds) and `NUM_ITERATIONS` to small values (~2) to control the size of the trace.

The following files will be generated:

- `debug.cbor` - in the current working directory
- `archive.a`, `framework.json`, `profile.pop`, `vars.capnp` - in a sub-directory that contains the ISO date/time and process ID in its name.

These can be used by the PopVision Graph Analyser tool (available from the Graphcore downloads portal). The profile data includes information on the memory breakdown, the tile usage, graph structure, compute operations, and respective length of processing cycles. See the PopVision User Guide for details.

Note that the `debug.cbor` file should be moved into the sub-directory with the other files, or a symbolic link created, in order for the PopVision Graph Analyser to use the information it contains. (For other applications, multiple sub-directories may be created, one per Poplar executable. A link to the debug.cbor file should be made in each sub-directory.)

Two sample visualisations from the PopVision Graph Analyser are given below.

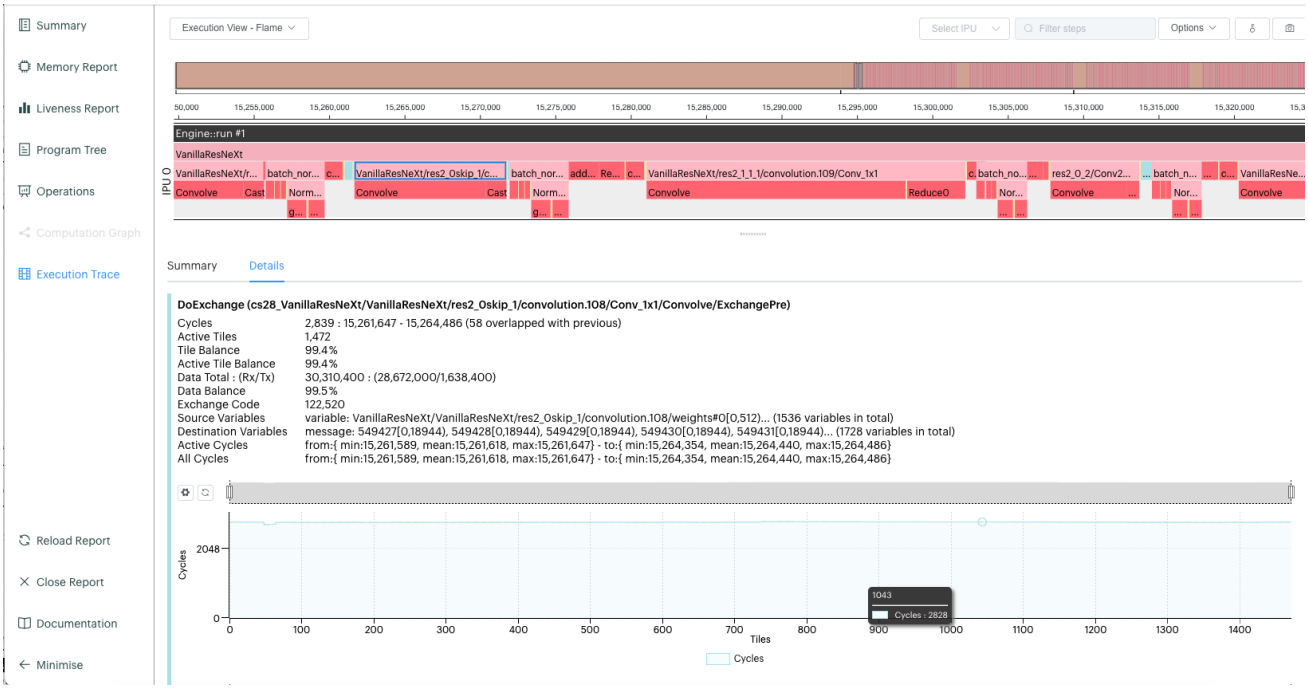

Fig. 3.2: Cycle breakdown of inference pipeline. The beginning section is waiting time for the pre-processing. The orange section is the data transfer. Both can be decreased using infeeds and outfeeds as described in Section 3.5, Infeeds and outfeeds.

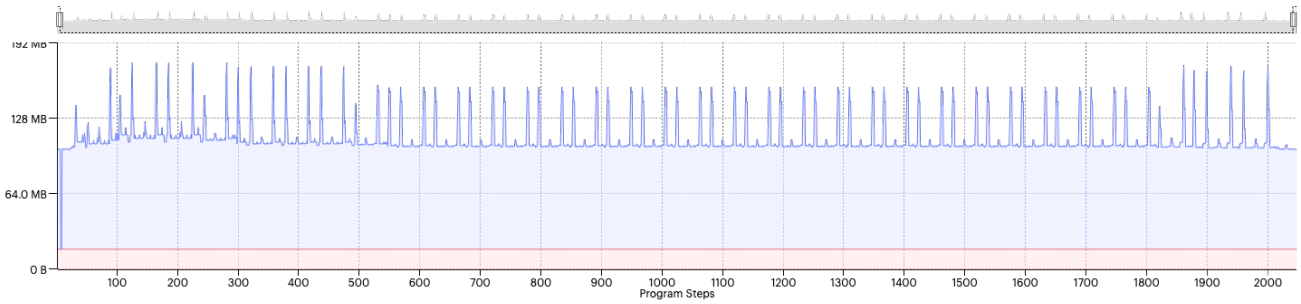

Fig. 3.3: Total memory usage of the IPU over processing time. The constant offset memory comes from the control code. The spikes in live-memory are characteristic of convolution or fully connected layers.

3.5 Infeeds and outfeeds

Infeeds and outfeeds are framework constructs that allow data to be streamed directly into and out ofa TensorFlow session. This creates a significant boost in data throughput since the host-to-device transfer is an active stack making data available as required. The concept is illustrated below.

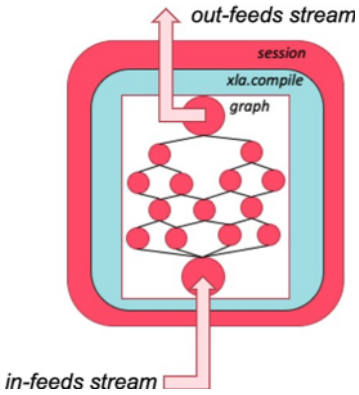

Fig. 3.4: The infeeds/outfeeds construction in relation to a session

The Targeting the IPU from TensorFlow guide provides a description of data feeds. Here, we mainly summarize the data feeds construction in the sample script.

The first step is to create a dataset (lines 8 to 18) where the input data (in this case a synthetic tensor of [batch size, h, w, channels] dimension), is packaged into the dataset `ds`. Lines 66 to 71 instantiate the infeed and outfeed queues.

Within the network definition, three additions are required. Firstly, on line 44, a body(features) wrapper is defined to hold the model definition. Secondly, the output of the model is fed into the outfeed construct on line 49. Finally, the return statement of the model definition is a call to `ipu.loops.repeat` on line 50. On line 86, an outfeed dequeue is instantiated, which is the final preamble before a session definition. Within the session scope, an infeed queue is initialized on line 92, and after the `session.run` call to the compiled graph, the outfeed is dequeued. Data transfer thus follows a sequence of data upload into the infeed queue; session run of graph; data return via the outfeed queue.

Further aspects of this are presented within the guide and should be reviewed for greater insight.

CHAPTER

FOUR

TRAINING WITH THE ESTIMATOR API

The TensorFlow estimator is a high-level API that encapsulates several common functions that are useful during the development of deep learning models. These functions include (but are not limited to) training and evaluation methods. They provide the primary interface between the user and the underlying model. Given the level of abstraction, it is somewhat of a departure from the session scope paradigm so common to many TensorFlow scripts and is reviewed as a separate topic. The Estimator is supported on Graphcore's TensorFlow release.

There are two primary facets of the estimator: the `train` method and the `evaluate` method. In the `train` method, the user provides an input pipeline which is a function that is called for fetching a mini-batch from the training dataset. The number of iterations in the training loop can be specified by providing a steps parameter. The state of the model is captured in the checkpoint (.ckpt) file, which is stored in a specified model directory. The `evaluate` method is typically used for model quality assessment, where certain metrics are produced to quantify the performance of a model based on validation data. The model under evaluation is fetched via a specified checkpoint (.ckpt) file. Similar to the `train` method, the user is expected to provide the input pipeline function and the number of steps when the `evaluate` method is invoked.

4.1 Instantiate an estimator for the IPU

The estimator is named `IPUEstimator` in the API. The following provides the essential arguments for instantiating an `IPUEstimator`:

- `config`: set the configuration of the estimator, which includes:
  - IPU profiling configuration
  - IPU selection configuration (number of IPUs to target, ID of IPU and so on)
  - graph placement configuration: number of shards, number of replicas and so on
  - logging configuration: parameters which control the logging frequency
  - output configuration: directory for output checkpoint files
- `model_fn`: definition of the model function. Refer to Write a model function for details on how to write this function.
- `model_dir`: directory to save model parameters, graph and other data
- `params`: hyperparameters to pass to the model function
- `warm_start_from`: optional path to checkpoint file that you can use for a warm start

4.2 Abridged code sample for the estimator

The following is an abridged sample script that instantiates an estimator:

```
from tensorflow.python import ipu

def create_ipu_estimator(model_fn, model_dir, params):
   # Create IPU configuration
   ipu_options = ipu.config.IPUConfig()

   #  IPU selection configuration
   ipu_options.auto_select_ipus = params['num_devices']

   # Graph placement configuration
   ipu_run_config = ipu.ipu_run_config.IPURunConfig(
         iterations_per_loop=params['iterations_per_loop'],
         ipu_options=ipu_options,
         num_shards=1,
         num_replicas=params['num_devices'],
         autosharding=False)

   # logging and output configuration
   config = ipu.ipu_run_config.RunConfig(
         ipu_run_config=ipu_run_config,
         log_step_count_steps=params['log_interval'] ,
         save_summary_steps=params['summary_interval'] ,
         model_dir=model_dir)

   # return an IPUEstimator instance
   return ipu.ipu_estimator.IPUEstimator(
         config=config,
         model_fn=model_fn,
         params=params)

# Instantiate an IPUEstimator
estimator = create_ipu_estimator(
            model_fn,
            model_dir=params['model_dir'],
            params=params)
```

4.3 Train and evaluate methods

Once an `IPUEstimator` is instantiated, you can run training and evaluation with the `train` and `evaluate` methods. You can run these as follows:

```
# partial is used to configure the training mode of the input function (input_fn)
train_input_fn = functools.partial(input_fn, params=params, is_training=True)

eval_input_fn = functools.partial(input_fn, params=params, is_training=False)

estimator.train(train_input_fn, steps=train_steps)
estimator.evaluate(eval_input_fn,
                   checkpoint_path=estimator.latest_checkpoint(),
                   steps=eval_steps)
```

For multiple epochs of training, you can calculate a number of steps according to the number of samples in the data set and the batch size. Then you invoke the `train` and `evaluate` methods in a loop of epochs as shown below:

```
for i in range(args.epochs):
   print("Training epoch {}/{}".format(i, args.epochs))
   estimator.train(train_input_fn, steps=train_steps)

   estimator.evaluate(eval_input_fn,
                      checkpoint_path=n_est.latest_checkpoint(),
                      steps=eval_steps)
```

The `train_input_fn` and `evaluate_input_fn` parameters, which are supplied to the estimator methods, are the inputs to the pipeline. Their major functionality is to extract feature and label pairs from examples in the dataset. For more information about how these input pipelines can be composed, search for `input_fn` in the evaluate section of the TensorFlow Estimator documentation.

CHAPTER

FIVE

SCOPING AND DETERMINING UNSUPPORTED OPERATIONS

Porting a model to the IPU usually requires an explicit call to `ipu_scope` where you define which part of the graph goes onto the IPU. In its simplest form, this means taking the model definition and scoping it in its entirety on the IPU:

```
with ipu_scope('/device:IPU:0'):
   original_graph_definition
```

or as defined in lines 73-75 of the ResNext example in Section 3.1, Abridged sample code:

```
74 # Compiles graph and targets IPU(s)
75    with ipu.scopes.ipu_scope('/device:IPU:0'):
76       res = ipu.ipu_compiler.compile(resnext101_model, inputs=[])
```

The `ipu_compiler.compile` wrapper compiles the model graph and optimizes it for the IPU. The IPUEstimator, explained in Section 4, Training with the estimator API, uses the same approach internally and does not require explicit scoping or compilation.

In general, you can scope most facets of the model definition to run on the IPU, which leads to a simplified deployment process. If an underlying operation in the graph is unsupported, however, such an approach is not possible and TensorFlow will provide an error that elaborates on which operation cannot run on the IPU and which are the supported and unsupported data types. In certain cases, you will need to place a component of the graph on the host:

```
with tf.device('/device:CPU:0'):
   not_supported_op
```

You can also use the `allow_soft_placement` option to place unsupported operations on the CPU. If you use this option when constructing the `Session` then there's no need to explicitly place operations on the host:

```
with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
```

CHAPTER

SIX

CHECKING HARDWARE AVAILABILITY

Using the `gc-monitor` command, you can check the availability of IPUs on the host machine.

For more information, see the command-line tools documentation.

gc-monitor	Partition: 'p_ipuof' has 8 reconfigurable IPUs							
IPU-M	Serial	ICU FW	Type	Server version	ID	PCIe ID	Routing	GWSW
10.44.44.253	0001.0002.8204221	2.1.3	M2000	1.5.4	0	3	DNC	
10.44.44.253	0001.0002.8204221	2.1.3	M2000	1.5.4	1	2	DNC	
10.44.44.253	0001.0001.8204221	2.1.3	M2000	1.5.4	2	1	DNC	
10.44.44.253	0001.0001.8204221	2.1.3	M2000	1.5.4	3	0	DNC	
10.44.44.244	0055.0002.8204121	2.1.3	M2000	1.5.4	4	3	DNC	
10.44.44.244	0055.0002.8204121	2.1.3	M2000	1.5.4	5	2	DNC	
10.44.44.244	0055.0001.8204121	2.1.3	M2000	1.5.4	6	1	DNC	
10.44.44.244	0055.0001.8204121	2.1.3	M2000	1.5.4	7	0	DNC	

Attached processes				IPU			Board	
PID	Command	Time	User	ID	Clock	Temp	Temp	Power
198633	python3	9m4s	alext	0	1330MHz	N/A	N/A	N/A
198633	python3	9m4s	alext	1	1330MHz	N/A	N/A	N/A
198633	python3	9m4s	alext	2	1330MHz	N/A	N/A	N/A
198633	python3	9m4s	alext	3	1330MHz	N/A	N/A	N/A

Fig. 6.1: An example of a system with 2 IPU-M2000s, 4 of the 8 IPUs are active

CHAPTER

SEVEN

RESNEXT FULL CODE EXAMPLE

```
import tensorflow as tf
import os
import numpy as np
import time
import logging

# Import IPU API
from tensorflow.python import ipu
from tensorflow_core.python.keras.backend import unique_object_name
from typing import Any, Optional, Tuple, Union


def create_input_data(batch_size=1, height=224, width=224, channels=4):
    """
        Create the input dataset for in-feeds

        :param batch_size: size of the batches to process
        :param height: height of input image
        :param width: width of input image
        :param channels: channels (RGB) in the input image
        :return: Constructed dataset
    """
    # Synthetic input data follows NHWC format
    input_data = np.random.random((batch_size, height, width, channels))
    input_data = tf.cast(input_data, DTYPE)

    # Prepare dataset for ipu_infeeds
    ds = tf.data.Dataset \
        .range(1) \
        .map(lambda k: {"features": input_data}) \
        .repeat() \
        .prefetch(BATCHES_PER_STEP)
    return ds


def conv(input_tensor: tf.Tensor,
         kernel_size: Union[int, Tuple[int, int]],
         filters_out: int,
         stride: Optional[int] = 1,
         padding: Optional[str] = 'SAME',
         add_bias: Optional[bool] = True,
         dtype: Optional[Any] = tf.float16,
         name: Optional[str] = None,
         weight_suffix: Optional[str] = "kernel",
         bias_suffix: Optional[str] = "conv/bias",
         *_):
    """
        Apply convolutional layer and optional bias on input tensor.

        Args:
            input_tensor: Input data
            kernel_size: Filter size (assumes equal height and width)
            filters_out: Number of output filters
            stride: Stride of the filter
            padding: Type of padding to use
```

(continues on next page)

(continued from previous page)

```
            add_bias: Should bias be added
            dtype: Data type of parameters
            name: Optional name for this op
            weight_suffix: String to weight name with
            bias_suffix: String to suffix the bias name with

        Returns: Output of convolution operator.
    """

    # Assumes input in NHWC format.
    filters_in = input_tensor.get_shape()[-1]
    if isinstance(kernel_size, int):
        w_shape = [kernel_size, kernel_size, filters_in, filters_out]
    else:
        w_shape = kernel_size + (filters_in, filters_out)
    w_init = tf.contrib.layers.xavier_initializer(dtype=dtype)
    if name is None:
        name = unique_object_name("conv2d", zero_based=True)

    name_scope = tf.get_default_graph().get_name_scope()
    if name_scope not in ["", None]:
        name = name_scope + "/" + name

    with tf.get_default_graph().as_default():
        with tf.variable_scope(name):
            weights = tf.get_variable(weight_suffix,
                                      shape=w_shape,
                                      initializer=w_init,
                                      dtype=dtype)

    output_tensor = tf.nn.conv2d(input_tensor,
                                 weights, [1, stride, stride, 1],
                                 padding=padding.upper(),
                                 name=name)

    if add_bias:
        b_shape = [filters_out]
        b_init = tf.zeros_initializer()
        with tf.variable_scope(name):
            biases = tf.get_variable(bias_suffix,
                                     shape=b_shape,
                                     initializer=b_init,
                                     dtype=dtype)
        output_tensor += biases
    return output_tensor


# Block definitions for ResNeXt
def input_block(x):
    x = conv(x, kernel_size=7, stride=2, filters_out=64, name="conv1")
    x = norm(x, training=False)
    x = relu(x)
    x = maxpool(x, size=3, stride=2)
    return x


def maxpool(x, size=3, stride=2):
    x = tf.nn.max_pool(x,
                       ksize=[1, size, size, 1],
                       strides=[1, stride, stride, 1],
                       padding='SAME')
    return x


def reduce_mean(x, indices=(1, 2)):
    x = tf.reduce_mean(x, reduction_indices=indices)
    return x


def fc(x, num_units_out):
```

(continues on next page)

(continued from previous page)

```
    num_units_in = x.get_shape()[1]
    w_init = tf.contrib.layers.xavier_initializer(dtype=tf.float16)
    b_init = tf.constant_initializer(0.0)

    weights = tf.get_variable('weights',
                              shape=[num_units_in, num_units_out],
                              initializer=w_init,
                              dtype=tf.float16)
    biases = tf.get_variable('biases',
                             shape=[num_units_out],
                             initializer=b_init,
                             dtype=tf.float16)

    x = tf.nn.xw_plus_b(x, weights, biases)
    return x


def norm(x, training=False):
    x = tf.layers.batch_normalization(x,
                                      fused=True,
                                      center=True,
                                      scale=True,
                                      training=training,
                                      trainable=training,
                                      momentum=0.997,
                                      epsilon=1e-5)
    return x


def relu(x):
    return tf.nn.relu(x)


def group_conv(x,
               ksize,
               stride,
               filters_in,
               filters_out,
               index=0,
               groups=1,
               dtype=tf.float16,
               name='conv'):
    """
    Apply group convolutions by leveraging XLA implementation.

    """
    with tf.variable_scope(name, use_resource=True):
        W = tf.get_variable(
            "conv2d/kernel" + str(index),
            shape=[ksize, ksize, filters_in.value / groups, filters_out],
            dtype=dtype,
            trainable=True,
            initializer=tf.variance_scaling_initializer())
        return tf.nn.conv2d(x,
                            filters=W,
                            strides=[1, stride, stride, 1],
                            padding='SAME')


def group_conv_block(x, first_stride, filters, count, name='', cardinality=4):
    """
        Group convolution block implementation.

        :param x: Input tensor
        :param first_stride: Initial tensor
        :param filters: List of number of filters for various convolution blocks
        :param count: Number of times block is repeated
        :param name: Name of block
        :param cardinality: Number of groups = outputchannels/cardinality
        :return: Layer x after application of all the ops within the block
```

(continues on next page)

(continued from previous page)

```
    """
    for i in range(count):
        shortcut = x
        stride = (first_stride if (i == 0) else 1)

        # First vanilla convolution
        x = conv(x,
                 kernel_size=1,
                 stride=stride,
                 filters_out=filters[0],
                 add_bias=False,
                 name=name + str(i) + "_1",
                 dtype=tf.float16)
        x = norm(x)
        x = relu(x)

        # Group convolution evaluation
        x = group_conv(x,
                       ksize=3,
                       stride=1,
                       filters_in=x.get_shape()[-1],
                       filters_out=filters[0],
                       index=1,
                       name=name + str(i) + "_2",
                       groups=cardinality,
                       dtype=tf.float16)
        x = norm(x)
        x = relu(x)

        # Second vanilla convolution
        x = conv(x,
                 kernel_size=1,
                 stride=1,
                 filters_out=filters[1],
                 add_bias=False,
                 name=name + str(i) + "_3",
                 dtype=tf.float16)
        x = norm(x)
        if i == 0:
            shortcut = conv(shortcut,
                            kernel_size=1,
                            stride=stride,
                            filters_out=filters[1],
                            add_bias=False,
                            name=name + str(i) + "skip",
                            dtype=tf.float16)
            shortcut = norm(shortcut)
        x = shortcut + x
        x = relu(x)
    return x


def resnext101_model():
    """
    Define ResNext-101 network graph

    """
    def body(features):
        with tf.variable_scope("VanillaResNeXt"):
            x = input_block(features)
            x = group_conv_block(x,
                                 first_stride=1,
                                 filters=[128, 256],
                                 count=3,
                                 cardinality=CARDINALITY,
                                 name='res2_')  # 112
            x = group_conv_block(x,
                                 first_stride=2,
                                 filters=[256, 512],
                                 count=4,
```

(continues on next page)

(continued from previous page)

```
                                cardinality=CARDINALITY,
                                name='res3_')  # 224
            x = group_conv_block(x,
                                first_stride=2,
                                filters=[512, 1024],
                                count=23,
                                cardinality=CARDINALITY,
                                name='res4_')  # 448
            x = group_conv_block(x,
                                first_stride=2,
                                filters=[1024, 2048],
                                count=3,
                                cardinality=CARDINALITY,
                                name='res5_')  # 896
            x = reduce_mean(x)
            output = fc(x, num_units_out=1000)
            outfeed = outfeed_queue.enqueue(output)
            return outfeed

    return tf.python.ipu.loops.repeat(n=BATCHES_PER_STEP,
                                      body=body,
                                      infeed_queue=infeed_queue)


if __name__ == '__main__':
    print("ResNeXt-101 Inference")

    IPU_MODEL = False

    # Number of steps
    NUM_ITERATIONS = 5
    BATCHES_PER_STEP = 1000

    # Model
    MODEL = 'ResNeXt-101'
    CARDINALITY = 32
    BATCH_SIZE = 4

    # Precision
    DTYPE = tf.float16

    # Create input data using randomized numpy arrays
    dataset = create_input_data(batch_size=BATCH_SIZE,
                                height=224,
                                width=224,
                                channels=4)

    if IPU_MODEL:
        os.environ['TF_POPLAR_FLAGS'] = "--use_ipu_model"

    # Setup infeed queue
    if BATCHES_PER_STEP > 1:
        with tf.device('cpu'):
            infeed_queue = ipu.ipu_infeed_queue.IPUInfeedQueue(
                dataset, feed_name="inference_infeed")
    else:
        raise NotImplementedError("batches per step == 1 not implemented yet.")

    # Setup outfeed
    outfeed_queue = ipu.ipu_outfeed_queue.IPUOutfeedQueue(feed_name="outfeed")

    # Compiles graph and targets IPU(s)
    with ipu.scopes.ipu_scope('/device:IPU:0'):
        res = ipu.ipu_compiler.compile(resnext101_model, inputs=[])

    # Setup IPU configuration and build session
    cfg = ipu.config.IPUConfig()
    cfg.convolutions.poplar_options["availableMemoryProportion"] = "0.3"
    cfg.ipu_model.compile_ipu_code = False
    cfg.auto_select_ipus = 1
```

(continues on next page)

(continued from previous page)

```
    cfg.configure_ipu_system()
    ipu.utils.move_variable_initialization_to_cpu()
    outfeed = outfeed_queue.dequeue()

    with tf.Session() as sess:
        fps = []
        latency = []
        sess.run(infeed_queue.initializer)
        sess.run(tf.global_variables_initializer())
        # Warm up
        print("Compiling and Warmup...")
        start = time.time()
        sess.run(res)
        outfed = sess.run(outfeed)
        duration = time.time() - start
        print("Duration: {:.3f} seconds\n".format(duration))
        for iter_count in range(NUM_ITERATIONS):
            print("Running iteration: ", iter_count)
            # Run
            start = time.time()
            sess.run(res)
            sess.run(outfeed)
            stop = time.time()
            fps.append((BATCHES_PER_STEP * BATCH_SIZE) / (stop - start))
            logging.info(
                "Iter {3}: {0} Throughput using real data = {1:.1f}"
                " imgs/sec at batch size = {2}".format(
                    str(MODEL), fps[-1], BATCH_SIZE, iter_count))
            latency.append(1000 * (stop - start) / BATCHES_PER_STEP)
            logging.info(
                "Iter {3}: {0} Latency using real data = {2:.2f} msecs "
                "at batch_size = {1}".format(str(MODEL), BATCH_SIZE,
                                             latency[-1], iter_count))

        print("Average statistics over {0} iterations, excluding the 1st "
              "iteration.".format(NUM_ITERATIONS))
        print("-------------------------")
        fps = fps[1:]
        latency = latency[1:]
        print(
            "Throughput at bs={} of {}: min={}, max={}, mean={}, std={}.".
            format(BATCH_SIZE, str(MODEL), min(fps), max(fps),
                   np.mean(fps), np.std(fps)))
        print("Latency at bs={} of {}: min={}, max={}, mean={}, std={}.".
              format(BATCH_SIZE, str(MODEL), min(latency), max(latency),
                     np.mean(latency), np.std(latency)))
```

CHAPTER

EIGHT

TRADEMARKS & COPYRIGHT