GRAPHCORE OVERVIEW AND ONBOARDING TRAINING FOR TAMU

May 25, 2022

Mario Michael Krell
WORKSHOP GOALS

• Explore and execute code for TensorFlow1, TensorFlow2 and PyTorch

• First insights into how to visualize and optimize IPU code

• Idea of difference of IPU and other hardware and how it might benefit your research

Disclaimer: This is my first coding lab and Graphcore's first large-scale workshop. Bear with us.
THE TEAM

Mario
Alex
Lisa
Brian
Richard
AGENDA

• Introduction to Graphcore, IPU, and Poplar
  • Hands-on: ssh into the POD, enable the SDK, clone tutorials, binary caching, run example

• TensorFlow1
  • Hands-on: Port a basic model, add infeeds, loop on device, profile a sharded/pipelined model

• TensorFlow2
  • Hands-on: Port a Keras script, leverage loop on device, replicate and run data-parallel, pipeline

• PyTorch
  • Hands-on: PopTorch example, DataLoader, options to optimize performance

• Research directions on the IPU
GRAPHCORE OVERVIEW
GRAPHCORE ENABLING MACHINE INTELLIGENCE

• Founded in 2016
• Technology: Intelligence Processor Unit (IPU)
• Team: 650+ globally
• Offices: UK, US, China, Norway, Poland
• Raised >$710M
GRAPHCORE IPU LETS INNOVATORS CREATE THE NEXT BREAKTHROUGHS IN MACHINE INTELLIGENCE
IPU ARCHITECTURE OVERVIEW
MACHINE INTELLIGENCE REPRESENTS A COMPLETELY NEW COMPUTE WORKLOAD

Massive parallelism
Sparsity in data structures
Low precision compute
Model parameter re-use
Static graph structure
LEGACY PROCESSOR ARCHITECTURES HAVE BEEN REPURPOSED FOR ML

**CPU**
Apps and Web/
Scalar

**GPU**
Graphics and HPC/
Vector
A NEW PROCESSOR IS REQUIRED FOR THE FUTURE

CPU
Apps and Web/
Scalar

GPU
Graphics and HPC/
Vector

IPU
Artificial Intelligence/
Graph
MASSIVE PARALLELISM WITH ULTRAFAST MEMORY ACCESS

**CPU**
- Designed for scalar processes
- Off-chip memory 1x

**GPU**
- SIMD/SIMT architecture, designed for dense contiguous data
- Model and Data spread across off-chip and small on-chip cache and shared mem. 5x – 32x

**IPU**
- Massively parallel MIMD, ideal for ML workloads
- Model & Data in tightly coupled large locally distributed SRAM 320x

Generalised comparisons & illustrative diagrams
Deep Trench Capacitor
Efficient power delivery
Enables increase in operational performance

Wafer-On-Wafer
Advanced silicon 3D stacking technology
Closely coupled power delivery die
Higher operating frequency and enhanced overall performance

BOW IPU PROCESSOR

IPU-Tiles™
1472 independent IPU-Tiles™ each with an IPU-Core™ and In-Processor-Memory™

IPU-Core™
1472 independent IPU-Core™
8832 independent program threads executing in parallel

In-Processor-Memory™
900MB In-Processor-Memory™ per IPU
65.4TB/s memory bandwidth per IPU

Solder Bumps

IPU-Links™
10x IPU-Links, 320GB/s chip to chip bandwidth

IPU-Exchange™
11 TB/s all to all IPU-Exchange™
Non-blocking, any communication pattern

PCIe
PCI Gen4 x16
64 GB/s bidirectional bandwidth to host
BOW-2000 IPU MACHINE

- 4 x Bow 3D Wafer-on-Wafer IPUs
- 1.4 PetaFLOPS AI Compute
- 3.6 GB In-Processor-Memory @ 260TB/s

- Up to 256 GB IPU Streaming Memory
- 2.8 Tbps IPU-Fabric™
- Same 1U blade form factor

Ideal for both Training & Inference
GROWING MODEL GARDEN

COMPUTER VISION

IMAGE CLASSIFICATION
- ResNet50 v1.5
- EfficientNet-BO
- EfficientNet-B4
- ResNeXt-101
- MobileNet v2
- MobileNet v3
- ViT

OBJECT DETECTION
- YOLO v3
- YOLO v4
- Faster RCNN

OBJECT SEGMENTATION
- Unet (Industrial)
- Unet (Medical)

GNN

TGN
- NEW
MPNN
- NEW

PROBABILISTIC

MCMC

Sales Forecast

REINFORCEMENT

RL
- Reinforcement Learning

SPEECH

STT (ASR)
- RNN-T
- Conformer

TTS
- DeepVoice3
- FastSpeech2

NLP

BERT-Base
- BERT-Large
- GroupBERT
- GPT2
- NEW

GENERATIVE

Autoencoder
- VAE

OTHER

ETO
- ETO
- NEW

DIN

DIN

DIEN

Images from text

https://www.graphcore.ai/resources/model-garden
BENCHMARK CODE

• We publish performance benchmarks for some models on our website:
  https://www.graphcore.ai/performance-results

• The command lines needed to reproduce these performance benchmarks should be in a README in the GitHub repo.
INTRO:
GETTING STARTED
COMMAND LINE TOOLS
PoPlar SDK

• Access updates through Graphcore support portal: https://downloads.graphcore.ai/

• Unpack SDK tar and source the shell scripts to update several environment variables on your evaluation machine:

```bash
$ cd poplar_sdk-[os]-[ver]
$ source poplar-[os]-[ver]/enable.sh
$ source popart-[os]-[ver]/enable.sh
```

where [os] is the host OS (Ubuntu), [ver] is the current software version number.

You need to source the PopART enable script if you are using PopART or PopTorch.

**NOTE:** each of these scripts must be sourced every time the Bash shell is reset. If you attempt to run any Poplar software without having first enabled these scripts you’ll get an error like:

```bash
fatal error: 'poplar/Engine.hpp' file not found
```
# Consider adding these to ~/.profile

source /opt/gc/poplar_sdk-ubuntu_18_04-2.5.1+1001-64add8f33d/poplar-ubuntu_18_04-2.5.0+4748-e94d646535/enable.sh

source /opt/gc/poplar_sdk-ubuntu_18_04-2.5.1+1001-64add8f33d/popart-ubuntu_18_04-2.5.1+4748-e94d646535/enable.sh

mkdir -p /localdata/$USER/tmp

export TF_POPLAR_FLAGS=--executable_cache_path=/localdata/$USER/tmp

export POPTORCH_CACHE_DIR=/localdata/$USER/tmp

export POPLAR_LOG_LEVEL=INFO

export POPLIBS_LOG_LEVEL=INFO
# Create and activate a Python virtual env
virtualenv venv_tf2 -p python3.6
source ~/venv_tf2/bin/activate

# Install AMD TF2 wheel for IPU
pip install /opt/gc/poplar_sdk-ubuntu_18_04-2.5.1+1001-64add8f33d/tensorflow-2.5.2+gc2.5.1+193132+4673d3a1ba+amd_znver1-cp36-cp36m-linux_x86_64.whl

# Clone repo, install reqs, run example
git clone https://github.com/graphcore/tutorials.git
cd tutorials/simple_applications/tensorflow2/mnist/
pip install -r requirements.txt
python mnist.py

# Sample output:
# Epoch 4/4
# 2000/2000 [==================================] - 1s 320us/step - loss: 0.2542
HANDBOOT

bit.ly/tamu220525
**GRAPHCORE COMMAND LINE TOOLS**

- **gc-info** Determines what IPU cards are present in the system.
- **gc-inventory** Lists device IDs, physical parameters and firmware version numbers.
- **gc-reset** Resets an IPU device after reboot. Note that each IPU must be reset after the host machine is rebooted.
- **gc-exchangetest** Allows you to test the internal exchange fabric in an IPU.
- **gc-memorytest** Tests all the memory in an IPU, reporting any tiles that fail.
- **gc-links** Displays the status and connectivity of each of the IPU-Links that connect the C2 IPU-Processor cards together. See also *IPU-Link channel mapping*.
- **gc-powertest** Tests power consumption and temperature of the C2 IPU-Processor cards.
- **gc-hosttraffictest** Allows you to test the data transfer between the host machine and the IPUs (in both directions).
- **gc-iputraffictest** Allows you to test the data transfer between IPUS.
- **gc-docker** Allows you to use IPU devices in Docker containers.

See: [https://documents.graphcore.ai/](https://documents.graphcore.ai/)
TENSORFLOW ON THE IPU

• Graphcore supplies its own branch of TensorFlow that supports the IPU.

• TensorFlow 1.15 and TensorFlow 2.4 are supported.

• There are 2 main differences in the Graphcore implementation of TensorFlow:

  (1) Some machine-learning ops are optimised for the IPU hardware. For example, our custom dropout op is designed to use less memory by not storing the dropout mask between forward and backward passes.

  (2) It provides extra IPU-specific functions, such as those for selecting and configuring IPUs.
PYTORCH ON THE IPU

• PopTorch is a set of extensions for PyTorch to enable PyTorch models to run directly on Graphcore IPU hardware.

• PopTorch supports both inference and training. To run a model on the IPU, you wrap your existing PyTorch model in either a PopTorch inference wrapper or a PopTorch training wrapper.
POPART – POPLAR ADVANCED RUNTIME

• PopART enables you to import models using the Open Neural Network Exchange (ONNX) and run them using the Poplar tools.

• PopART has three main features:

  1) It can import ONNX graphs into a runtime environment.

  2) It provides a simple interface for constructing ONNX graphs without needing a third party framework.

  3) It runs imported graphs in inference, evaluation or training modes, by building a Poplar engine, connecting data feeds and scheduling the execution of the Engine.
PROGRAMMING ON IPU

DOCS AND TUTORIALS
USEFUL ENV VARIABLES
MULTI-IPU CONSTRUCTS
FRAMEWORKS
POPVISION
DEVELOPER RESOURCES
Graphcore developer portal launched in May 2020

- Public hub for developers to access:
  - Software documentation
  - How-to videos
  - Code tutorial walkthroughs
  - Performance Benchmarks
  - Community support
  - Developer news

- Learn about the Poplar® SDK and how to easily run ML models on IPU systems

graphcore.ai/developer
As part of our ethos to put power in the hands of AI developers, Graphcore open sourced in July 2020.

PopLibs™, PopART, PyTorch & TensorFlow for IPU fully open source and available on GitHub.

Our code is public and open for code contributions from the wider ML developer community.

github.com/graphcore
VIDEO + GITHUB TUTORIALS
A comprehensive set of online developer training materials and educational content

TUTORIALS
Learn how to create and run programs using Poplar and PopLibs with our hands-on programming tutorials.

<table>
<thead>
<tr>
<th>Programs and Variables</th>
<th>Using PopLibs</th>
<th>Writing Vertex Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling Output</td>
<td>Basic Machine Learning Example</td>
<td>Matrix-Vector Multiplication</td>
</tr>
<tr>
<td>Matrix-Vector Multiplication Optimisation</td>
<td>Simple PyTorch for the IPU</td>
<td>NEW</td>
</tr>
</tbody>
</table>

Tutorial 1: programs and variables

Copy the file \texttt{mnist\_example\_mnist\_mnist.cu} to your working directory and open it in an editor. The file contains the outline of a C++ program including some Poplar library headers and a namespace.

**Graphs, variables and programs**

All Poplar programs require a \texttt{Graph} object to construct the computation graph. Graphs are always created for a specific target (where the target is a description of the hardware being targeted, such as an IPU). To obtain the target you need to choose a device.

The tutorials use a simulated target by default, so you can run on any machine even if it has no Graphcore hardware attached. On systems with accelerator hardware, the header file \texttt{poplar\_accelerator.h} contains API calls to enumerate and return a \texttt{Device} object for the attached hardware.

Simulated devices are created with the \texttt{mockDevice}, which models the functionality of an IPU on the host. The \texttt{createDevice} function creates a new virtual device to work with. Once we have this device we can create a \texttt{Graph} object to target it.

- Add the following code to the body of \texttt{main}:

  ```cpp
  // Create the IPU device
  // (requires \texttt{poplar\_accelerator} header)
  Device device = getDevice("ipu\_accelerator\_0\_0\_0\_0");
  Target target = device.getTarget();
  // Create a Graph object...
  ```

Any program running on an IPU needs data to work on. These are defined as variables in the graph.

- Add the following code to create the first variable in the program:

  ```cpp
  // Create the first variable...
  ```

Tutorial 5: a basic machine learning example

This tutorial contains a complete training program that performs a logistic regression on the MNIST data set, using gradient descent. The files for the demo are in \texttt{mnist\_sh}.

There are no coding steps in the tutorial. The task is to understand the code, build it and run it. You can build the code using the supplied makefile.

Before you can run the code you will need to run the \texttt{get\_mnist\_sh} script to download the MNIST data.

The program accepts an optional command line argument to make it use the IPU hardware instead of a simulated IPU.

As you would expect, training is significantly faster on the IPU hardware.

Copyright © 2018 Graphcore Ltd. All rights reserved.
**RESOURCES CENTRE**

- New resources hub made available in September 2020
- Central source of research papers, white papers, videos, on-demand webinars and documentation
- Product resources for ML Engineers & IT / Infrastructure Managers now available

[graphcore.ai/resources]
USEFUL ENV VARIABLES
USEFUL ENV VARIABLES

LOGGING

Logging messages can be generated when your program runs. This is controlled by the environment variables described below. For more detailed information see the docs: https://docs.graphcore.ai/projects/poplar-user-guide/en/latest/env-vars.html

POPLAR_LOG_LEVEL: Enable logging for Poplar

POPLAR_LOG_DEST: Specify the destination for Poplar logging ("stdout", "stderr" or a file name)

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OFF&quot;</td>
<td>No logging information. The default.</td>
</tr>
<tr>
<td>&quot;ERR&quot;</td>
<td>Only error conditions will be reported.</td>
</tr>
<tr>
<td>&quot;WARN&quot;</td>
<td>Warnings when, for example, the software cannot achieve what was requested (for example, if the convolution planner can’t keep to the memory budget, or Poplar has determined that the model won’t fit in memory but the debug.allowOutOfMemory option is enabled).</td>
</tr>
<tr>
<td>&quot;INFO&quot;</td>
<td>Very high level information, such as PopLibs function calls.</td>
</tr>
<tr>
<td>&quot;DEBUG&quot;</td>
<td>Useful per-graph information.</td>
</tr>
<tr>
<td>&quot;TRACE&quot;</td>
<td>The most verbose level. All useful per-tile information.</td>
</tr>
</tbody>
</table>
SYNTHETIC-DATA

TF_POPLAR_FLAGS= "--use_synthetic_data --synthetic_data_initializer=random"

Used for measuring the IPU-only throughput and disregards any host/CPU activity.
USING POPVISION (MORE ON THIS LATER)

POPLAR ENGINE OPTIONS = '{"autoReport.all": "true", "autoReport.directory": "/tommyFlowers"}’

- The PopVision Graph Analyser uses report files generated during compilation and execution by the Poplar SDK.
- These files can be created using POPLAR ENGINE OPTIONS.
- In order to capture the reports needed for the PopVision Graph Analyser you only need to set POPLAR ENGINE OPTIONS = '{"autoReport.all": "true"}' before you run a program. By default this will enable instrumentation and capture all the required reports to the current working directory.
A NOTE ON COMPILe TIME AND EXECUTABLE CACHING

• Our compiler technology consumes input from the high-level frameworks e.g. PyTorch, and generates a massively parallel computational graph. This graph is then compiled down to target the IPUs MIMD architecture.

• It can take a long time to compile a large fused graph into an executable suitable for the IPU. E.g. ~20 mins for BERT-L pre-training on IPU-POD16.

• Reducing compile time is something we are focused on this year.

• To prevent the need for compiling every time a new process is started, you can enable an executable cache: more on the next slide.
EXECUTABLE CACHE

If you often run the same models you might want to enable executable caching to save time:

POPTORCH:

• You can do this by either setting the POPTORCH_CACHE_DIR environment variable or by calling poptorch.Options.enableExecutableCaching.

TENSORFLOW:

• You can use the flag --executable_cache_path to specify a directory where compiled files will be placed. Fused XLA/HLO graphs are hashed with a 64-bit hash and stored in this directory.

Warning
The cache directory might grow large quickly. Poplar doesn’t evict old models from the cache and, depending on the number and size of your models and the number of IPUs used, the executables might be quite large. It is your responsibility to delete the unwanted cache files.
PRECOMPIILATION

• PopTorch and TensorFlow support precompilation: This means you can compile your model on a machine which doesn’t have an IPU and export the executable to a file. You can then reload and execute it on a different machine which does have an IPU.

More details in the documentation.
TFI FOR IPU
TENSORFLOW PROGRAMS ON THE IPU

**Minimum code to run on IPU**
- Configure the IPU: Single or multiple IPUs. Graph optimization and profiling options.
- Compile graph to XLA: Fix input and output tensors. Compiles static graph to Poplar executable.

**Now, run it fast**
- Optimise data flow: Minimise host IO by looping on IPU. Use Datasets, infeeds & outfeeds.
MINIMUM CODE CHANGES TO RUN ON IPU

1. Configure IPU system
2. Functionalize your model to be placed on IPU
3. Compile on IPU
MINIMUM CODE CHANGES TO RUN ON IPU

1. Configure IPU system

```python
from tensorflow.python import ipu

# Create a default configuration
ipu_configuration = ipu.config.IPUConfig()

# Select an IPU automatically
ipu_configuration.auto_select_ipus = 1

# Apply the configuration
ipu_configuration.configure_ipu_system()
```
MINIMUM CODE CHANGES TO RUN ON IPU

2. Functionalize your model to be placed on IPU

```python
# Do basic addition with tensors
o1 = pa + pb
o2 = pa + pc
simple_graph_output = o1 + o2

def basic_graph(pa, pb, pc):
    # Do basic addition with tensors
    o1 = pa + pb
    o2 = pa + pc
    simple_graph_output = o1 + o2
    return simple_graph_output
```
MINIMUM CODE CHANGES TO RUN ON IPU

3. Compile on IPU

```python
from tensorflow.python.ipu.scopes import ipu_scope
with ipu_scope("/device:IPU:0"):  
xla_result = ipu.ipu_compiler.compile(basic_graph, [pa, pb, pc])
```
TENSORFLOW PROGRAMS ON THE IPU

Configure the IPU
Single or multiple IPUs. Graph optimization and profiling options.

Compile graph to XLA
Fix input and output tensors. Compiles static graph to Poplar executable.

Optimise data flow
Minimise host IO by looping on IPU. Use Datasets, infeeds & outfeeds.
WHY DO WE NEED TRAINING LOOPS?

• Communication between the host and IPU is slow compared to execution on-device, so we see this overhead if calling the hardware for each batch.

• By placing the training operations inside a loop, they can be executed multiple times without returning control to the host.
WHY DO WE NEED DATA FEEDS?

• When a training operation is placed into a loop, the inputs to that training operation need to provide a stream of values.

• Standard TensorFlow Python feed dictionaries cannot provide data in this form, so when training in a loop, data must be fed from a TensorFlow DataSet.
POPVISION™ TOOLS
POPVISION GRAPH ANALYSER

• You can use the PopVision Graph Analyser tool to debug IPU programs and generate reports on compilation and execution of the program.

• This tool can be downloaded from the Graphcore customer support portal: https://downloads.graphcore.ai/.

• There is a built-in help system within the tool for any questions you might have about producing and analysing reports.
PopVision Graph Analyser
v2.2

Getting started video available on the developers portal

Several new features including:
- A new file format for the graph and execution profile, resulting in a 50% file size reduction
- Enhanced PopLibs debug information

Liveness Report
The debug information shown for a variable now displays enhanced information. For each variable that has debug information, you can now see the PopLibs API that created it, its arguments and its outputs.

Enhanced debug information has been added to program steps. Program steps show Poplar and PopLibs debug information such as which PopLibs API created that program step, its arguments and its outputs.

Check out the integrated help or visit our developer portal for more information.
PopVision System Analyser v1.0

The PopVision System Analyser allows developers to understand the execution of programs running on the host processor which control the IPU(s). The System Analyser shows the interaction between the host and the IPU(s) so that developers can understand where the bottlenecks are in the execution of their applications.

Show the execution of the software on the host processor enabling users to identify bottlenecks in execution between CPU & IPU(s).

Provide profile insights as you scale models to multiple CPUs / IPUs.

The PopVision System Analyser visualises the information collected by the PopVision Trace Instrumentation Library which is part of the Poplar SDK.

Visit our developer portal for more information and the latest documentation: https://www.graphcore.ai/developer
**BULK SYNCHRONOUS PARALLEL (BSP)**

BSP software bridging model – massively parallel computing with no concurrency hazards

3 phases: compute, sync, exchange

Easy to program – no live-locks or dead-locks

Widely-used in parallel computing – Google, FB, ...

First use of BSP inside a parallel processor
TF2/KERAS ON IPU
Keras On IPU

- IPU optimized Keras Model and Sequential are available for the IPU. These have the following features:
  
  * On-device training loop for reduction of communication overhead.
  * Gradient accumulation for simulating larger batch sizes.
  * Automatic data-parallelisation of the model when placed on a multi-IPU device.
```python
import tensorflow as tf
from tensorflow.keras.layers import *
from tensorflow.python import ipu

cfg = ipu.config.IPUConfig()
cfg.auto_select_ipus = 1
cfg.configure_ipu_system()
with ipu.ipu_strategy.IPUStrategy().scope():
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
    x_train = x_train.astype('float32') / 255.0
    y_train = tf.keras.utils.to_categorical(y_train, 10)
    ds_train = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(64, drop_remainder=True)

model = tf.keras.Sequential([  
    Conv2D(32, (3, 3), padding='same', input_shape=x_train.shape[1:]),  
    Activation('relu'),  
    Conv2D(32, (3, 3)),  
    Activation('relu'),  
    MaxPooling2D(pool_size=(2, 2)),  
    Dropout(0.25),  
    Conv2D(64, (3, 3), padding='same'),  
    Activation('relu'),  
    Conv2D(64, (3, 3)),  
    Activation('relu'),  
    MaxPooling2D(pool_size=(2, 2)),  
    Dropout(0.25),  
    Flatten(),  
    Dense(512),  
    Activation('relu'),  
    Dropout(0.5),  
    Dense(10),  
    Activation('softmax')
])

model.compile(loss='categorical_crossentropy',  
               optimizer=tf.optimizers.SGD(learning_rate=0.016),  
               metrics=['accuracy'])

model.fit(ds_train, epochs=40)
```
$ python3 gpu_keras_cnn.py

$ python3 ipu_keras_cnn.py
Train for 1560 steps

Epoch 1/40
1560/1560 [==================================] - 8s 5ms/step - loss: 2.168
Epoch 2/40
1560/1560 [==================================] - 5s 3ms/step - loss: 1.880
Epoch 3/40
1560/1560 [==================================] - 5s 3ms/step - loss: 1.652
Epoch 4/40
75/1560 [>...............................] - ETA: 5s - loss: 1.5328 -

$ python3 gpu_keras_cnn.py
2020-05-12 16:40:32.449285: I tensorflow/compiler/plugin/poplar/driver/platform_driver.cc:3517: Poplar backend is configured to load
2020-05-12 16:40:35.357131: I tensorflow/compiler/plugin/poplar/driver/platform_driver.cc:3517: Poplar backend is configured to load
2020-05-12 16:40:35.898895: I tensorflow/compiler/jit/xla_compilation_cache.cc:256: Empty compile_cache directory, will compile to:
most once for the lifetime of the process.
1560/1560 [==================================] - 2s 2ms/step - loss: 0.0500 - accuracy: 0.99978
Epoch 2/40
1560/1560 [==================================] - 1s 593us/step - loss: 0.0408 - accuracy: 0.9997
Epoch 3/40
1560/1560 [==================================] - 1s 592us/step - loss: 0.0357 - accuracy: 0.99994
Epoch 4/40
1560/1560 [==================================] - 1s 597us/step - loss: 0.0325 - accuracy: 0.99997
Epoch 5/40
1560/1560 [==================================] - 1s 600us/step - loss: 0.0299 - accuracy: 0.99997
Epoch 6/40
1560/1560 [==================================] - 1s 600us/step - loss: 0.0278 - accuracy: 0.99997
Epoch 7/40
1560/1560 [==================================] - 1s 599us/step - loss: 0.0258 - accuracy: 0.99997
Epoch 8/40
1560/1560 [==================================] - 1s 598us/step - loss: 0.0241 - accuracy: 0.99997
Epoch 9/40
1560/1560 [==================================] - 1s 600us/step - loss: 0.0224 - accuracy: 0.99997
Epoch 10/40
1560/1560 [==================================] - 1s 601us/step - loss: 0.0208 - accuracy: 0.99997
Epoch 11/40
1560/1560 [==================================] - 1s 601us/step - loss: 0.0193 - accuracy: 0.99997
Epoch 12/40
1560/1560 [==================================] - 1s 600us/step - loss: 0.0178 - accuracy: 0.99997
Epoch 13/40
1560/1560 [==================================] - 1s 601us/step - loss: 0.0164 - accuracy: 0.99997
Epoch 14/40
1560/1560 [==================================] - 1s 601us/step - loss: 0.0150 - accuracy: 0.99997
Epoch 15/40
1560/1560 [==================================] - 1s 601us/step - loss: 0.0136 - accuracy: 0.99997
Epoch 16/40
1560/1560 [==================================] - 1s 601us/step - loss: 0.0122 - accuracy: 0.99997
Epoch 17/40
1560/1560 [==================================] - 1s 601us/step - loss: 0.0109 - accuracy: 0.99997
KERAS TUTORIAL

https://github.com/graphcore/tutorials/tree/master/tutorials/tensorflow2/keras
INTRO TO POPTORCH

GRAPHCORE
WHAT IS POPTORCH?
WHAT IS POPTORCH?

• PopTorch is a set of extensions for PyTorch to enable PyTorch models to run on Graphcore's IPU hardware.

• PopTorch supports both inference and training. To run a model on the IPU you wrap your existing PyTorch model in either a PopTorch inference wrapper or a PopTorch training wrapper.

• You can provide further annotations to partition the model across multiple IPUs. Using the user-provided annotations, PopTorch will use PopART to parallelise the model over the given number of IPUs.

• Additional parallelism can be expressed via a replication factor which enables you to data-parallelise the model over more IPUs.

• Under the hood PopTorch uses TorchScript, an intermediate representation (IR) of a PyTorch model, using the torch.jit.trace API. To learn more about TorchScript and JIT, you can go through PyTorch’s tutorial: https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html

• Not all PyTorch operations have been implemented by the backend yet and you can find the list of supported operations here: https://docs.graphcore.ai/projects/poptorch-user-guide/en/latest/supported_ops.html
Define a model within PyTorch

Create an IPU execution wrapper around the model and run as normal

PopTorch uses the `torch.jit.trace` API to trace the model to PyTorch IR

Compile the graph in PopART and then run on one or more IPUs

Examples available from https://github.com/graphcore/examples
GETTING STARTED: TRAINING A MODEL
1. Import packages

PopTorch is a separate package from PyTorch, and must be imported.

2. Load dataset using torchvision.datasets and poptorch.DataLoader

In order to make data loading easier and more efficient, PopTorch offers an extension of torch.utils.data.DataLoader class: poptorch.DataLoader class is specialised for the way the underlying PopART framework handles batching of data.

3. Define model and loss function using torch API

The only difference here from pure PyTorch is the loss computation, which has to be part of the forward function. This is to ensure the loss is computed on the IPU and not on the CPU, and to give us as much flexibility as possible when designing more complex loss functions.
4. Prepare training

Instantiate compilation and execution options, these are used by PopTorch’s wrappers such as `poptorch.DataLoader` and `poptorch.trainingModel`.

5. Train the model

Define the optimizer using PyTorch’s API.

Use `poptorch.trainingModel` wrapper, to wrap your PyTorch model. This wrapper will trigger the compilation of our model, using TorchScript, and manage its translation to a program the IPU can run. Then run your training loop.
# PyTorch

```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='MNIST training in PopToch')
    parser.add_argument('--batch-size', type=int, default=8, help='batch size for training (device: GPU)
                         batch size for testing (device: IPU)')
    parser.add_argument('--epochs', type=int, default=10, help='number of epochs to train (device: GPU)
                         number of epochs to train (device: IPU)')
    parser.add_argument('--lr', type=float, default=0.05, help='learning rate (default: 0.05)')
    args = parser.parse_args()

    training_data = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST('mnist_data/', train=True, download=True),
        batch_size=args.batch_size, shuffle=True, drop_last=True)

    test_data = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST('mnist_data/', train=False, download=True),
        batch_size=args.batch_size, shuffle=True, drop_last=True)

    model = Network()
    training_model = TrainingModelWithLoss(model)
    optimizer = optim.SGD(model.parameters(), lr=args.lr)

    # Run training
    for _ in range(args.epochs):
        for data, labels in training_data:
            optimizer.zero_grad()
            optimizer.step()

    # Run validation
    sum_acc = 0.0
    with torch.no_grad():
        for data, labels in test_data:
            output = model(data)
            sum_acc += accuracy(output, labels)

    print("Accuracy on test set: {:.2f}%".format(sum_acc / len(test_data)))
```

```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='MNIST training in PopToch')
    parser.add_argument('--batch-size', type=int, default=8, help='batch size for training (device: GPU)
                         batch size for testing (device: IPU)')
    parser.add_argument('--epochs', type=int, default=10, help='number of epochs to train (device: GPU)
                         number of epochs to train (device: IPU)')
    parser.add_argument('--lr', type=float, default=0.05, help='learning rate (default: 0.05)')
    args = parser.parse_args()

    training_data = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST('mnist_data/', train=True, download=True),
        batch_size=args.batch_size, shuffle=True, drop_last=True)

    test_data = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST('mnist_data/', train=False, download=True),
        batch_size=args.batch_size, shuffle=True, drop_last=True)

    model = Network()
    training_model = TrainingModelWithLoss(model)
    optimizer = optim.SGD(model.parameters(), lr=args.lr)

    # Run training
    for _ in range(args.epochs):
        for data, labels in training_data:
            optimizer.zero_grad()
            optimizer.step()

    # Detach the training model so that the same IPU could be used for validation
    training_model.detachFromDevice()

    # Run validation
    sum_acc = 0.0
    with torch.no_grad():
        for data, labels in test_data:
            output = inference_model(data)
            sum_acc += accuracy(output, labels)

    print("Accuracy on test set: {:.2f}%".format(sum_acc / len(test_data)))
```

POPTORCH TUTORIAL

https://github.com/graphcore/tutorials/tree/master/tutorials/pytorch/tut1_basics
The compilation and execution on the IPU can be controlled using `poptorch.Options`.


Some examples:

(i) **deviceIterations**
This option specifies the number of batches that is prepared by the host (CPU) for the IPU. The higher this number, the less the IPU has to interact with the CPU, for example to request and wait for data, so that the IPU can loop faster. However, the user will have to wait for the IPU to go over all the iterations before getting the results back. The maximum is the total number of batches in your dataset, and the default value is 1.

(ii) **replicationFactor**
This is the number of replicas of a model. We use replicas as an implementation of data parallelism. To achieve the same behavior in pure PyTorch, you'd wrap your model with `torch.nn.DataParallel`, but with PopTorch, this is an option.
To run inference, you use `poptorch.inferenceModel` class, which has a similar API to `poptorch.trainingModel` except that it doesn't need an optimizer.

See tutorial example here: [https://github.com/graphcore/tutorials/tree/master/tutorials/pytorch/tut1_basics#running-our-model-for-inference-on-an-ipy](https://github.com/graphcore/tutorials/tree/master/tutorials/pytorch/tut1_basics#running-our-model-for-inference-on-an-ipy)
• PyTorch for the IPU: User Guide

• GitHub tutorial
https://github.com/graphcore/examples/tree/master/tutorials/pytorch/tut1_basics

• Code examples on GitHub
https://github.com/graphcore/examples/tree/master/code_examples/pytorch/mnist

• Video tutorial on our developer page
https://www.graphcore.ai/developer
ANY QUESTIONS, REQUESTS, BUGS...

https://www.graphcore.ai/support
THANK YOU

Mario Michael Krell
mariok@graphcore.ai