GRAPHCORE OVERVIEW AND ONBOARDING TRAINING FOR TAMU

February 21, 2023

Alexander Tsyplikhin
AGENDA

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

• 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
GRAPHCORE OVERVIEW
GRAPHCORE ENABLING MACHINE INTELLIGENCE

• Founded in 2016
• Technology: Intelligence Processor Unit (IPU)
• Team: ~500
• Offices: UK, US, China, Poland
• Raised >$710M
GRAPHCORE IPU LETS INNOVATORS CREATE THE NEXT BREAKTHROUGHS IN MACHINE INTELLIGENCE
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
IPU – ARCHITECTURED FOR AI

Massive parallelism with ultrafast memory access

**Parallelism**

**Processors**
- Designed for scalar processes

**Memory**
- Off-chip memory

**Memory Access**

**CPU**
- SIMD/SIMT architecture. Designed for large blocks of dense contiguous data

**GPU**
- Model and data spread across off-chip and small on-chip cache, and shared memory

**IPU**
- Massively parallel MIMD. Designed for fine-grained, high-performance computing

Model and data tightly coupled, and large locally distributed SRAM
PROVEN IPU ADVANTAGE
SELECT CASE STUDIES ACROSS MANY INDUSTRIES & FIELDS

HEALTHCARE
CASE STUDY: NLP

FINANCE – OPTION PRICING
CASE STUDY: SIM

AI SaaS – TEXT ANALYTICS
CASE STUDY: NLP

RESEARCH / BIG LABS
CASE STUDY

COMPUTATIONAL CHEMISTRY
CASE STUDY: GNN

SMART CITY
CASE STUDY: CV

FINANCE - INSURANCE
CASE STUDY: CV

WEATHER FORECASTING
CASE STUDY: SIM

HIGH ENERGY PHYSICS
CASE STUDY

DYNAMIC GRAPHS
CASE STUDY: GNN
IPU COMPUTATIONAL ADVANTAGES

Heterogeneous gather/scatter operations. E.g. GNNs

Group and depthwise convolutions. E.g. ResNeXt, EfficientNet

Vector operations with low arithmetic intensity. E.g. Sparse matmuls

Dense as well as Sparse Matrix Multiplication. E.g. Transformers

Hardware accelerated Random Number Generation. E.g. Random Projections

Hard to vectorize workloads. E.g. CRR algorithm for option pricing

References:
https://www.graphcore.ai/performance-results
https://www.graphcore.ai/posts/how-we-made-efficientnet-more-efficient
https://www.graphcore.ai/posts/delving-deep-into-modern-computer-vision-models
WORKLOADS THAT CAN’T EASILY BE VECTORIZED

- Workloads with while loops that continue until convergence is achieved e.g. ray tracing
- Workloads where different compute paths are required depending on the inputs e.g. CRR model
- Tree-based models with unbalanced trees of different depth
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

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

BOW IPU PROCESSOR

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
EXECUTION MODEL
BOW-2000 IPU MACHINE

IU blade form factor delivering 1.4 PetaFLOPS AI Compute

Disaggregated AI/ML accelerator platform

Excellent performance & TCO leveraging In-Processor memory & IPU-Exchange

IPU-Links scale to Bow Pod64

Expansion to Bow Pod256 and beyond with IPU-GW Links
BOW-2000: THE BUILDING BLOCK OF LARGE PODS

4x Bow IPUs
- 1.4 PFLOP₁₆ compute
- 5,888 processor cores
- > 35,000 independent parallel threads

Exchange Memory
- 3.6GB In-Processor Memory @ 260 TB/s
- 128GB Streaming Memory DRAM (up to 256GB)

IPU-Fabric managed by IPU-GW
- Host-Link – 100GE to Poplar Server for standard data center networking
- IPU-Link – 2D Torus for intra-POD64 communication
- GW-Link - 2x 100Gbps Gateway-Links for rack-to-rack – flexible topology

% x16 IPU-Link [64GB/s]
% Host-Link Network I/F [100Gbps]
% IPU-GW Link [100Gbps]
% x8 PCIe G4 [32GB/s]
HOST AND IPU-FABRIC ENABLES LARGE SCALEOUT PODS

Host-Links:
100Gbps connectivity for each Bow-2000 to host server
Enabling disaggregation of host server, with optimal server/Bow-2000 ratio.

IPU-Links (part of IPU-Fabric):
2D Torus for IPU communication
Providing high bandwith connectivity across IPUs up to Pod64

GW-Links (part of IPU-Fabric):
2x 100Gbps Gateway-Links for rack-to-rack communication
Redundant rack-to-rack communication for large scaleout beyond Pod64

2.8 Tbps* ultra-low latency fabric designed for AI

*Bandwidth for a Bow-2000

x16 IPU-Link 64GB/s
100Gbps Host-Link Network I/F
100Gbps IPU-GW Link
x8 PCIe G4 32GB/s
HANDBS-ON:

GET STARTED

RUN AN EXAMPLE
bit.ly/tamu2302221
STANDARD ML FRAMEWORK SUPPORT

Develop models using standard high-level frameworks or port existing models

- PyTorch
- TensorFlow
- PyTorch Lightning
- HUGGING FACE
- PyG
- Keras
- PaddlePaddle
- HALO

Easy port of high-level framework models

Existing models on alternative platforms

POPLAR®

IPU-Processor Platforms
GRAPHCORE SOFTWARE MATURITY

- NLP/TRANSFORMERS
- IMAGE CLASSIFICATION/CNNs
- OBJECT DETECTION
- LARGE MODELS
- MLPERF
- CONDITIONAL SPARSITY
- GNNS
- ML APPLICATIONS

- TUTORIALS
- CODE EXAMPLES
- DOCUMENTATION
- VIDEOS
- NATIVE IPU CODERS PROGRAM
- APPS PORTFOLIO

- FRONTENDS
  - JUPYTER NOTEBOOKS

- FRAMEWORKS
  - Keras
  - ONNX

- FW BACKENDS
  - XLA
  - POPART+
  - POPDIST

- PARTITIONER
- POPIR
- POPIT

- POPLIBS
- GCL
- POPLAR

- GRAPH ENGINE
- GRAPH COMPILER

- GC DEVICE ACCESS LAYER

- DRIVERS
  - IPUOF DRIVER
  - PCIe DRIVER

- POPLAR® SDK

- DEVELOPER ECOSYSTEM

- INFERENCE DEPLOYMENT TOOLKIT

- POPVISION TOOLS

- GRAPH ANALYZER
- SYSTEM ANALYZER
- DEBUGGER
- DEVELOPMENT ENVIRONMENT

- SYSTEM MONITORING
  - PROMETHEUS
  - GRAFANA

- JOB DEPLOYMENT
  - K8S
  - SLURM

- SYSTEM SOFTWARE
PROGRAMMING ON IPU

DOCS AND TUTORIALS
USEFUL ENV VARIABLES
FRAMEWORKS
POPVISION
DEVELOPER RESOURCES
DEVELOPER PORTAL

graphcore.ai/developer

- 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
• As part of our ethos to put power in the hands of AI developers, Graphcore open sourced in 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
### 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 Optimization</td>
<td>Simple PyTorch for the IPU</td>
<td></td>
</tr>
</tbody>
</table>

#### Tutorial 1: programs and variables

Copy the file `mnist_tut/tutorials/start_tutorial.cpp` 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 `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 we need to choose a device.

The tutorial uses a simulated target by default, so we can run any machine even if it has no Graphcore hardware attached. On systems with accelerator hardware, the `headerfile` path/`accessor` path contains API calls to enumerate and return devices - objects for the attached hardware.

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

- **Add the following code to the body of `main()`:**
  ```cpp
  auto graph = createGraph();
  ```

---

#### 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 `1x15.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 `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 (c) 2018 Graphcore Ltd. All rights reserved.**
ENHANCED MODEL GARDEN

PUBLIC ACCESS TO WIDE VARIETY OF MODELS, READY TO RUN ON IPU

NEW FILTER/SEARCH CAPABILITY

DIRECT ACCESS TO GITHUB

PAPERSPACE NOTEBOOK LINKS
RESOURCES CENTRE

graphcore.ai/resources

- Central source of research papers, white papers, videos, on-demand webinars and documentation
- Product resources for ML Engineers & IT / Infrastructure Managers now available
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>Log Level</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.
CREATE EXECUTION PROFILE

POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true", "autoReport.directory": "/report"}’

- 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.
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.
**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-powerertest** 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-putraficctest** 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/)
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.
import tensorflow as tf
from tensorflow.keras.layers import *

# GPU
(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), padding='same'),
    Activation('relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Conv2D(64, (3, 3), padding='same'),
    Activation('relu'),
    Conv2D(64, (3, 3), padding='same'),
    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)

# IPU
+ cfg = ipu.config.IPUConfig()
+ cfg.auto_select_ipus = 1
+ cfg.configure_ipu_system()

(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), padding='same'),
    Activation('relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Conv2D(64, (3, 3), padding='same'),
    Activation('relu'),
    Conv2D(64, (3, 3), padding='same'),
    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
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
2020-05-12 16:40:32.449285: I tensorflow/compiler/plugin/poplar/driver/pPrincipalParserPlugin: F66a6e6ce3
2020-05-12 16:40:34.523582: I tensorflow/core/platform/proto_tools/cpu_utils.cc:250
1560/1560 [======================================] - 2s 2ms/step - loss: 0.0500 - accuracy: 0.997
Epoch 2/40
1560/1560 [======================================] - 1s 593us/step - loss: 0.0408 - accuracy: 1.000
Epoch 3/40
1560/1560 [======================================] - 1s 592us/step - loss: 0.0357 - accuracy: 1.000
Epoch 4/40
1560/1560 [======================================] - 1s 597us/step - loss: 0.0325 - accuracy: 1.000
Epoch 5/40
1560/1560 [======================================] - 1s 600us/step - loss: 0.0299 - accuracy: 1.000
Epoch 6/40
1560/1560 [======================================] - 1s 600us/step - loss: 0.0278 - accuracy: 1.000
Epoch 7/40
1560/1560 [======================================] - 1s 599us/step - loss: 0.0258 - accuracy: 1.000
Epoch 8/40
1560/1560 [======================================] - 1s 598us/step - loss: 0.0241 - accuracy: 1.000
Epoch 9/40
1560/1560 [======================================] - 1s 600us/step - loss: 0.0224 - accuracy: 1.000
Epoch 10/40
1560/1560 [======================================] - 1s 601us/step - loss: 0.0208 - accuracy: 1.000
Epoch 11/40
1560/1560 [======================================] - 1s 601us/step - loss: 0.0193 - accuracy: 1.000
Epoch 12/40
1560/1560 [======================================] - 1s 600us/step - loss: 0.0178 - accuracy: 1.000
Epoch 13/40
1560/1560 [======================================] - 1s 601us/step - loss: 0.0164 - accuracy: 1.000
Epoch 14/40
1560/1560 [======================================] - 1s 601us/step - loss: 0.0150 - accuracy: 1.000
Epoch 15/40
1560/1560 [======================================] - 1s 601us/step - loss: 0.0136 - accuracy: 1.000
Epoch 16/40
1560/1560 [======================================] - 1s 601us/step - loss: 0.0122 - accuracy: 1.000
Epoch 17/40
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.
```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='MNIST training in PopTorch')
    parser.add_argument('--batch-size', type=int, default=8, help='Batch size for training (default: 8)
    parser.add_argument('--test-batch-size', type=int, default=10, help='Batch size for testing
    parser.add_argument('--epochs', type=int, default=10, help='Number of epochs to train (default: 10)
    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=1, 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:
            preds, losses = training_model(data, labels)
            optimizer.zero_grad()
            losses.backward()
            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(f'Accuracy on test set: {sum_acc / len(test_data):.2f}')
```
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.
INFERENCEx

• 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-ipu
MORE INFO

• 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

Getting started with PyTorch for the IPU

Running a basic model for training and inference
POPLAR™ POPVISION TOOLS

GRAPH ANALYSER
Useful for analysing and optimising the memory use and execution performance of ML models on the IPU

SYSTEM ANALYSER
Graphical view of the timeline of host-side application execution steps

“Our team was very impressed by the care and effort Graphcore has clearly put into the PopVision graph and system analysers. It’s hard to imagine getting such a helpful and comprehensive profiling of the code elsewhere, so this was really a standout feature in our IPU experience.”

Dominique Beaini, Valence Discovery, a leader in AI-first drug design
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

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.

Getting started video available on the developers portal

Check out the integrated help or visit our developer portal for more information
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.

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
ANY QUESTIONS, REQUESTS, BUGS...

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

Alexander Tsyplikhin
alext@graphcore.ai