GRAPHCORE WORKSHOP FOR TAMU

October 3, 2023

Alexander Tsyplikhin
AGENDA

• Introduction to Graphcore, IPU, and Poplar
  • Hands-on: access the POD, enable the SDK, run an example

• TensorFlow2/Keras
  • 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 IPU LETS INNOVATORS CREATE THE NEXT BREAKTHROUGHS IN MACHINE INTELLIGENCE
IPU – ARCHITECTURED FOR AI

Massive parallelism with ultrafast memory access

**Parallelism**
- Processors
- Memory

**Memory Access**
- CPU: Designed for scalar processes
  - Off-chip memory
- GPU: SIMD/SIMT architecture. Designed for large blocks of dense contiguous data
  - 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. DFT in Computational Chemistry

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. DFT or CRR model

- Tree-based models with unbalanced trees
PySCF on IPU

Installation guide | Example DFT Computations | Generating data | Training SchNet | QM18 dataset

Port of PySCF to Graphcore IPU.

Limitations

- Restricted Kohn Sham DFT (based on RKS, KohnShamDFT and hf.RHF).
- Number of atomic orbitals less than 70 mol.nao_nr() <= 70.
- Larger numerical errors due to np.float32 instead of np.float64.
- Limited support for jax.grad().

Installation

PySCF on IPU requires Python 3.8, JAX IPU experimental, TessellateIPU library and Graphcore Poplar SDK 3.2.

To run this package on a standard CPU machine (laptop or server), install the base Python requirements:

```bash
pip install -r requirements.txt
```

On IPU machines, please additionally use the IPU requirements file:

```bash
pip install -U pip
pip install -r requirements_ipu.txt
```

This will configure Graphcore research experimental JAX support in your python environment.

We recommend upgrading pip to the latest stable release when using the IPU requirements. This may be an optional step depending on the overall configuration of your python environment.

And finally, make our sub-packages available:

```bash
pip install -e .
```
Try changing the `mo_index` variable to select the different molecular orbitals benzene.

```python
mo_index = 5
orbital = molecules_orbitals[mo_index]
mol_view = plot_orbital(orbital, mol)
mol_view.spin()```

```
 1 2 3 4 5
```

```python
mol_view = plot_orbital(orbital, mol)
mol_view.spin()```
MIMD for
\[ E \in \mathbb{R}^{N^2 \times N^2} \]

65 TB/s for
\[
\begin{align*}
J &= E \cdot v \in \mathbb{R}^{N^2} \\
K &= E^T \cdot v \in \mathbb{R}^{N^2}
\end{align*}
\]
**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
65TB/s memory bandwidth per IPU

**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

**IPU-Links™**
10 x IPU-Links,
3200GB/s chip to chip bandwidth
**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

**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

**Solder Bumps**

**PCIe**
PCI Gen4 x16
64 GB/s bidirectional bandwidth to host
EXECUTION MODEL

COMPUTATIONAL GRAPH

data
- data [0]
- data [1]
- data [2]
- data [3]

AdderVertex [v0]
AdderVertex [v1]
AdderVertex [v2]

output
- output [0]
- output [1]

result
- result [0]

BSP SCHEDULE

EXCHANGE

COMPUTE

SYNC

EXCHANGE

SYNC

EXCHANGE

OPTIMIZED IPU EXECUTION

OUTPUT FROM POPVISION GRAPH ANALYSER

GRAPHCORE
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
BOW-2000 IPU MACHINE

1U 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

**COMPUTE**
- 4x Bow IPUs
  - 1.4 PFLOP\textsuperscript{16} compute
  - 5,888 processor cores
  - > 35,000 independent parallel threads

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

**COMMUNICATIONS**
- 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

---

**Diagram Details**
- x16 IPU-Link [64GB/s]
- Host-Link Network I/F [100Gbps]
- IPU-GW Link [100Gbps]
- x8 PCIe G4 [32GB/s]
BOW POD16 DIRECT ATTACH

Simple design with server with 4 Bow-2000 and 16 Bow IPUs
HANDBS-ON:

GET STARTED

RUN AN EXAMPLE
GRAPHCORE SOFTWARE ECOSYSTEM
WORLD CLASS DEVELOPER RESOURCES FOR IPU USERS

WWW.GRAPHCORE.AI/DEVELOPER

GRAPHCORE DOCUMENTS

Getting Started
Software Documents
Hardware Documents
Technical Notes and White Papers
Examples and Tutorials
Document Updates
Alphabetical List of All Documents
Graphcore License Agreements

Getting started with PyTorch for the IPU

Running a basic model for training and inference

AI Customer Engineer, Chris Bogdalskiwicz introduces PyTorch for the IPU. With PopVision™, a simple Python wrapper for PyTorch programs, developers can easily run models, directly on Graphcore IPUs with a few lines of extra code.

In this video, Chris provides a quick demo on running a basic model for both training and inference using a MNIST based example.
github.com/graphcore

- 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
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 PopLabs with our hands-on programming tutorials.

<table>
<thead>
<tr>
<th>Programs and Variables</th>
<th>Using PopLabs</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

Cut the file `tutorial/tutorial_code/` 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 tutorials use a simulated target by default, so you can run any machine even if it has no Graph core hardware attached. On systems with accelerator hardware, the header file `#include <debugmemgraph.h>` contains API calls to examine and return a device object for the attached hardware.

Simulated devices are created with the `Device` class, which models the functionality of an IPU on the host. The `UniversalDevice` 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
  // Create the IPU model device
  GraphDevice graphDevice;
  Target target = graphDevice.getTarget();
  // Create the Graph object
  Graph graph(target);
  ```

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
  // Add a tensor variable to the graph
  GraphTensor tensor = graph.addTensor(Shape{4, 3, 2});
  ```

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 the `example/demos/`.

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.
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
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

https://www.graphcore.ai/resources/model-garden
USEFUL ENV VARIABLES
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>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“OFF”</td>
<td>No logging information. The default.</td>
</tr>
<tr>
<td>“ERR”</td>
<td>Only error conditions will be reported.</td>
</tr>
<tr>
<td>“WARN”</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>“INFO”</td>
<td>Very high level information, such as PopLibs function calls.</td>
</tr>
<tr>
<td>“DEBUG”</td>
<td>Useful per-graph information.</td>
</tr>
<tr>
<td>“TRACE”</td>
<td>The most verbose level. All useful per-tile information.</td>
</tr>
</tbody>
</table>
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.
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.
GRAPHCORE COMMAND LINE TOOLS

**gc-docker**  Allows you to use IPU devices in Docker containers using the Docker container engine.

**gc-flops**  Allows you to benchmark the number of floating point operations per second on one or more IPU processors.

**gc-info**  Determines what IPU cards are present in the system.

**gc-inventory**  Lists device IDs, physical parameters and firmware version numbers.

**gc-links**  Displays the status and connectivity of each of the IPU-Links that connect IPUs. See also [IPU-Link channel mapping](https://docs.graphcore.ai/projects/command-line-tools/en/latest/index.html) for connectivity in an IPU Server containing C2 cards.

**gc-monitor**  Monitors IPU activity on shared systems.

**gc-reset**  Resets IPU devices.

**gc-exchangetest**  Allows you to test the internal exchange fabric in an IPU.

**gc-exchangewritetest**  Tests direct writes to the IPU’s tile memory via the host.

**gc-gwlinkstrafficstest**  Tests GW-Links on multi-rack IPU-POD systems.

**gc-hostsynclatencytest**  Reports the latency of transfers between the host machine and the IPUs (in both directions).

**gc-hosttrafficstest**  Allows you to test the data transfer between the host machine and the IPUs (in both directions).

**gc-podman**  Allows you to use IPU devices in Docker containers using the Podman container engine.

**gc-powerstest**  Tests power consumption and temperature of the IPU processors.

TF2/KERAS ON IPU
Keras on IPU

IPU optimized Keras Model and Sequential with 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 *
+ 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)
TF2/KERAS TUTORIALS

Continued in the repositories below (follow the READMEs)

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

GRAPHCORE
WHAT IS POPTORCH?

- PopTorch
- PopART
- GRAPH COMPILER
- GRAPH RUN TIME

PyTorch
main.py

Poplar compute graph

40
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.
Define a model within PyTorch

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

PopTorch uses PyTorch dispatcher to trace the model

Compile the graph in PopART and then run on one or more IPUs
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)
                        or testing (default: 10)
                        or epochs (default: 10)
                        or learning rate (default: 0.05)
                        or device iterations (default: 50)
    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:
            pred, loss = training_model(data, labels)
            optimizer.zero_grad()
            loss.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("Accuracy on test set: {}\%", format(sum_acc / len(test_data)))
```

---

```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)
                        or testing (default: 10)
                        or epochs (default: 10)
                        or learning rate (default: 0.05)
                        or device iterations (default: 50)
    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:
            pred, loss = training_model(data, labels)
            optimizer.zero_grad()
            loss.backward()
            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: {}\%", format(sum_acc / len(test_data)))
```
POPTORCH.OPTIONS

- The compilation and execution on the IPU can be controlled using `poptorch.Options`.
- Full list of options available here: https://docs.graphcore.ai/projects/poptorch-user-guide/en/latest/overview.html#options
- Some examples:
  1. `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.
  2. `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.
POPTORCH TUTORIALS

Continued in the repositories below (follow the READMEs)

- [github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/basics](https://github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/basics)
- [github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/mixed_precision](https://github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/mixed_precision)
- [github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/efficient_data_loading](https://github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/efficient_data_loading)
- [github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/pipelining](https://github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/pipelining)
Graphium integrates state-of-the-art Graph Neural Network (GNN) architectures and a user-friendly API, enabling the easy construction and training of custom GNN models.

Run Graphium on IPU with Paperspace Jupyter Notebook

- **Domain:** Molecules
- **Tasks:** Multitask
- **Model:** GCN/GIN/GINE
- **Datasets:** QM9, Zinc, Tox21
- **Workflow:** Training, validation, inference
- **Execution time:** 20 mins
“The suitability of IPUs for running GNNs and the kind of performance advantage that Graphcore and its customers have demonstrated is really helping to accelerate the uptake of this exciting model class”

Matthias Fey – PyG creator & founder of Kumo.ai
**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
**POPVISION TOOLS**

**IPU MEMORY ANALYSIS**
Capture memory information from your ML models when executed on IFUs. Inspect variable placement, size and liveness throughout the execution.

**EXECUTION TRACE REPORT**
View the output of instrumenting a Poplar program, capturing cycle counts for each step. See execution statistics, tile balance, cycle proportions and compute-set details.

**REPORT COMPARISONS**
Open two reports at once to compare their memory, execution, liveness and operations. Visualise where efficiencies can be made with different model parameters.

**HOST EXECUTION ANALYSIS**
Understand the execution of IPU-targeted software on your host system processors. Identify any bottlenecks between CPUs and IFUs across a visual interactive timeline.

**GRAPH DATA**
Plot graph data of any numerical data points from the host or IPU processor systems, such as board temperature, power consumption and IPU utilisation.

**LOCAL + REMOTE REPORTS**
Ability to open reports either on your local machine, or remotely on the host machine. The Graph Analyser also supports local and remote report access.
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 graphcore.ai/downloads
- There is a built-in help system within the tool for any questions you might have about producing and analysing reports.
PopVision Graph Analyser

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

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.

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.

Visit our developer portal for more information and the latest documentation: https://www.graphcore.ai/developer
CREATE PROFILE
POPVISION TUTORIALS

Continued in the repositories below (follow the READMEs)

tutorials/tensorflow2/infeed_outfeed

tutorials/pytorch/pipelining

tutorials/popvision/system_analyser_instrumentation
Prepare requests: overview

We've organized your options for requesting access to advanced research computing resources into a set of opportunities designed to support needs ranging from new or entry-level exploration to the largest-scale computational experiments. We welcome you to find the opportunity that aligns with your best estimate of your resource needs. And don't worry about starting too small. As you clarify your needs, you can upgrade to a larger-scale opportunity when you're ready.

There are four opportunities, which are described below and compared side-by-side in this table:

- Explore ACCESS
- Discover ACCESS

Prepare requests: overview
Explore ACCESS
Discover ACCESS
Accelerate ACCESS
Maximize ACCESS
Comparison Table
THANK YOU

Alexander Tsyplikhin
alext@graphcore.ai

Request an IPU allocation for your research today!
allocations.access-ci.org/prepare-requests-overview