SC 24 Tutorial

GRAPHCORE

GRAPHCORE'S SOLUTION

Hardware

IPU processor
designed for AI

Software

Poplar SDK and
development tools

Platform

IPU platforms
Available in the cloud

THE INTELLIGENCE PROCESSING UNIT (IPU)
WHAT MAKES IT DIFFERENT?

	CPU	GPU	IPU
Parallelism	Designed for scalar processing	SIMD/SIMT architecture. Designed for large blocks of dense contiguous data	Massively parallel MIMD architecture. High performance/efficiency for future ML trends
Memory Bandwidth	Off-chip memory	Model and Data spread across off-chip and small on-chip cache and shared memory (2TB/s for A100 HBM)	Main Model & Data in tightly coupled large locally distributed SRAM (~65 TB/s for Bow IPU)

Processor

Memory

BRIDGING SOFTWARE TO PARALLEL HARDWARE

Bulk Synchronous Parallel (BSP) is the computer science solution that bridges software to parallel processors.
- IPU implements BSP in hardware.

BSP solves parallel software issues:
- free of concurrency hazards
- efficient for highly parallel processors
- Poplar-SDK builds BSP compute on IPU hardware

[1] Pregel: a system for large-scale graph processing, 2010; [2] Large-scale graph partitioning with Apache Giraph, 2014

SCALING ACROSS DEVICES

EXCHANGE

- THE IPU EXCHANGE MODEL EXTENDS OFF DEVICE
  - ALLOWS TILE MESSAGING ACROSS IPU DEVICES
- SUPPORTS OPTIMIZED CROSS IPU SYNCHRONISATION
- DRIVEN BY COMPILER SUPPORT IN SOFTWARE

IPU-LINK™

- PROVIDES 320 GB/S IPU TO IPU BANDWIDTH
- SUPPORTS COMMUNICATION BETWEEN IPUS
- LAYOUT FULLY SOFTWARE CONFIGURABLE
- SUPPORTS POINT TO POINT TILE MESSAGING

GCD

- GRAPH CONTROL DOMAIN FOR APPLICATIONS
- CREATES A SINGLE LARGE IPU SOFTWARE TARGET
- FULLY CONFIGURABLE PARTITIONING OF IPUS
- BOTH DATA PARALLEL AND MODEL PARALLEL

UP TO 64 IPU DEVICES USABLE AS A SINGLE LARGE IPU FROM APPLICATIONS

565248 FULLY INDEPENDENT WORKERS, 57.6GB IN-PROCESSOR MEMORY™, LEVERAGING OVER 3.8 TRILLION TRANSISTORS

SCALING ACROSS SYSTEMS

256 IPU APPLICATION TARGET BUILT FROM INTERCONNECTED 64 IPU DOMAINS

- IPU EXCHANGE SUPPORT ACROSS DOMAINS
  - DRIVEN BY COMPILER SUPPORT IN SOFTWARE
- ENABLES APPLICATION COLLECTIVES SUPPORT
- ALLOWS SCALING UP TO 64000 IPU DEVICES

- IPU LINK™ CAN BE EXTENDED ACROSS DOMAINS
- SUPPORTS OPTIMIZED IPU LINK™ COLLECTIVES
- ALLOWS REPLICATION ACROSS SYSTEMS
- SUPPORTS A STANDARD IPU SOFTWARE MODEL

- IPUS CAN ACCESS MEMORY AND DEVICES OVER PCIE
- ALLOWS INTERFACING WITH HOST BASED SOFTWARE
- APPLICATIONS CAN BUILD ON HOST NETWORKING
- ALLOWS SCALING IN STANDARD SERVER PLATFORMS

PRODUCT DETAILS

GRAPHCORE

BOW IPU PROCESSOR

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

PCIe

PCI Gen4 x16
64 GB/s bidirectional bandwidth to host

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 IPU-2000

BOW-2000: THE BUILDING BLOCK OF LARGE PODS

COMPUTE

4x Bow IPUs

- 1.4 $PFLOP_{16}$ 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) @ 20 GB/s

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

x16 IPU-Link [64GB/s]

Host-Link Network I/F [100Gbps]

IPU-GW Link [100Gbps]

x8 PCIe G4 [32GB/s]

BOW POD64 REFERENCE DESIGN

Pre-Qualified 64-IPU Design with Reference Server and Switches

- Up to 16 Bow-2000 platforms
- Reference architecture supports different server requirements based on workload
- Bow Pod_{64} Configuration:
  - 64 IPUs
  - 22.4 PFLOPs @ FP16.16
  - ~58GB IPU In-Processor memory
  - ~7TB Streaming Memory
- Bow Pod Host disaggregation
  - Flexibly connect required host server compute over fabric
- 2D-Torus topology
  - Maximizes bandwidth across IPU-Links
  - All-Reduce 2x faster than mesh topology
- Scalable to 64K Bow IPUs

SOFTWARE STACK: TENSORFLOW PYTORCH POPLAR

EXECUTION MODEL

COMPUTATIONAL GRAPH

BSP SCHEDULE

OPTIMIZED IPU EXECUTION

OUTPUT FROM POPVISION GRAPH ANALYSER

GRAPHCORE

GRAPHCORE SOFTWARE

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

DEVELOPER ECOSYSTEM

FRONTENDS

JUPYTER NOTEBOOKS

INFERENCE DEPLOYMENT TOOLKIT

FRAMEWORKS

ONNX
HALO
Keras
PaddlePaddle

FW BACKENDS

XLA
POPART+
POPDIST

POPLAR®

PARTITIONER
POPIR
POPIT
POPLIBS
GCL
POPLAR
GRAPH ENGINE
GRAPH COMPILER

DRIVERS

GC DEVICE ACCESS LAYER
IPUOF DRIVER
PCIe DRIVER

POPLAR® SDK

GRAPH ANALYZER
SYSTEM ANALYZER
DEBUGGER
DEVELOPMENT ENVIRONMENT

POPVISION TOOLS

V-IPU

SYSTEM MONITORING

PROMETHEUS
GRAFANA

JOB DEPLOYMENT

K8S
SLURM

SYSTEM SOFTWARE

POPVISION TOOLS

IPU MEMORY ANALYSIS

Capture memory information from your ML models when executed on IPUs. 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 IPUs 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.

INTRO TO POPTORCH

GRAPHCORE

WHAT IS POPTORCH?

PyTorch

GPU

```
    _, ind = torch.max(predictions, 1)
    # provide labels only for samples, where prediction is available (during the training, not
    ions.size()[0]:]
    rch.eq(ind, labels)).item() / labels.size(  0

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 (de
    parser.add_argument('--test-batch-size', type=int, default=8, help='batch size for testing
    parser.add_argument('--epochs', type=int, default=10, help='number of epochs to train (def
    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, t
                        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,

    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("Accuracy on test set: {:0.2f}%".format(sum_acc / len(test_data)))
```

IPU

```
    _, ind = torch.max(predictions, 1)
    # provide labels only for samples, where prediction is available (during the training, not
    labels = labels[-predictions.size()[0]:]
    accuracy = torch.sum(torch.eq(ind, labels)).item() / labels.si  100.0
    return accuracy

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 (de
    parser.add_argument('--test-batch-size', type=int, default=8, help='batch size for testing
    parser.add_argument('--epochs', type=int, default=10, help='number of epochs to train (def
    parser.add_argument('--lr', type=float, default=0.05, help='learning rate (default: 0.05)'
    parser.add_argument('--device-iterations', type=int, default=50, help='device iterations (
    args = parser.parse_args()

    opts = poptorch.Options().deviceIterations(args.device_iterations)
    training_data = poptorch.DataLoader(opts,
                        torchvision.datasets.MNIST('mnist_data/', train=True, download=True, trans
                        batch_size=args.batch_size, shuffle=True, drop_last=True)
    test_data = poptorch.DataLoader(opts,
                        torchvision.datasets.MNIST('mnist_data/', train=False, download=True, tran

    model = Network()
    training_model = TrainingModelWithLoss(model)
    optimizer=optim.SGD(model.parameters(), lr=args.lr)
    training_model = poptorch.trainingModel(training_model, opts, optimizer=optimizer)
    inference_model = poptorch.inferenceModel(model)

    # Run training
    for _ in range(args.epochs):
        for data, labels in training_data:
            preds, losses = training_model(data, labels)

    # 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: {:0.2f}%".format(sum_acc / len(test_data)))
```

UNDER THE HOOD: BSP

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

COMPUTATIONAL GRAPH

GRAPH EXECUTION MODEL

POPLAR FRAMEWORK

WHAT IS POPLAR?

- Parallel programming framework that targets the IPU
- Simple but powerful programming model
- Close to the metal
- General purpose, extensible

POPLAR FRAMEWORK

1. Graphs, Variables & Vertices
2. Compute Sets & Execution
3. Host IPU Execution Model

THE POPLAR GRAPH

The graph is made up of:

- Data (variables in the graph)
- Compute tasks (vertices)
- Edges that connect them

VARIABLES

3-d tensor (3 x 4 x 5)

0.3	3.22	44.5	3.13	6.49

1-d tensor (5)

0.3	3.22
24.3	9.2

2-d tensor (2 x 2)

Data is stored in the graph in fixed size multi-dimensional tensors.

VARIABLES

0.3	3.22	44.5	3.13	6.49
24.3	9.2	0.01	0.23	93.1
0.22	13.2	3.2	5.67	55.3
5.6	99.8	7.22	8.66	22.1

Variables can be distributed over multiple tiles

VERTICES

A vertex is a specific piece of work to be carried out.

The edges determine which variable elements are processed by the vertex. A vertex can connect to a single element or a range of elements.

VERTICES

Codelet A

```
Input<float>  x;
Input<Vector<float>>  y;
Output<float> z;

*z = x + sum(y);
```

Each vertex is associated with a codelet.

VERTICES

A vertex runs on a single tile

VERTICES

Many vertices are needed to fully utilize the device

POPLAR FRAMEWORK

1. Graphs, Variables & Vertices
2. Compute Sets & Execution
3. Host IPU Execution Model

COMPUTE SETS

Compute sets specify sets of vertices to execute in parallel

Poplar verifies the compute set is free of data races

COMPUTE SET EXECUTION

A compute sets execute in 3 steps:

1. Exchange
   Transfer inputs
2. Compute
   Run vertices in parallel
3. Exchange
   Transfer outputs

Exchange code is generated by Poplar

Sync

Exchange inputs

Compute

Sync

Exchange outputs

COMPUTE SET EXECUTION

Exchange is required when a vertex in a compute set needs to read or write data which is stored on another tile's memory.

COMPUTE SET EXECUTION

For each compute set, each tile will have a number of vertices to execute.

COMPUTE SET EXECUTION

All tiles start by syncing.

COMPUTE SET EXECUTION

The tiles then move to exchange: Required vertex input data is copied between memory.

COMPUTE SET EXECUTION

Tiles will move to compute when they have finished exchange.

During compute vertices will read from and write to local tile memory.

COMPUTE SET EXECUTION

Each tile processor has several independent hardware threads (workers) to execute code.

Once exchange is complete, a hardware scheduler (supervisor) dispatches vertices onto the workers to run.

The tiles will run all vertices and then sync.

SUMMARY

A graph is made up of:

- Data (variables in the graph)
- Compute tasks (vertices)
- Edges that connect them

Vertices:

- Are associated to a codelet (code)
- Run on a single tile

Compute sets:

- Specify sets of vertices to execute in parallel
- Are executed in 3 steps: Exchange inputs, Compute, Exchange outputs

Control program:

- Specifies the order of operations

The program resides on the chip:

- The host takes care of compilation and of the data stream preparation

THE HOST PROGRAM

Host programs use the poplar library.

The Graph class is used to build up the computation graph.

The Engine class represents a fully compiled program ready to run on hardware.

```
#include <poplar/Engine.hpp>

using namespace poplar;
using namespace poplar::program;

...

Graph graph(target);
graph.addCodelets("my-codelets.cpp");

Program prog1, prog2;

constructMyGraph(graph, &prog1, &prog2);

Engine eng(device, graph, {prog1, prog2});
...

eng.run(0);
```

Codelets are loaded into the graph.

Control programs are built up out of instances of the Program class.

CODELET DEFINITIONS

Each codelet is defined as a C++ class that inherits from the Vertex class.

The fields of the vertex specify its inputs, outputs and internal data.

```
class AdderVertex : public Vertex {
public:
 Input<float> x;
 Input<float> y;
 Output<float> z;
 float bias;

 bool compute() {
   *z = x + y + bias;
   return true;
 }
}
```

The compute method specifies the vertex execution behaviour.

BUILDING THE COMPUTE GRAPH

```
Graph g(device);
g.addCodelets("codelets.cpp");

Tensor t1 = g.addVariable(FLOAT, {4, 5});
Tensor t2 = g.addVariable(FLOAT, {4});

ComputeSet cs = g.addComputeSet("myComputeSet")

VertexRef v1 = g.addVertex(cs, "AdderVertex");
VertexRef v2 = g.addVertex(cs, "AdderVertex");

g.connect(t1[1][1], v1["x"]);
g.connect(t1.slice({3, 1}, {4, 3}), v1["y"]);

g.connect(t2[0], v1["z"]);

g.connect(t1[0][3], v2["x"]);
g.connect(t1.slice({2, 2}, {3, 4}), v2["y"]);
g.connect(t2[3], v2["z"]);

g.setTileMapping(t1.slice({0, 0}, {4, 2}), 0);
g.setTileMapping(t1.slice({0, 2}, {4, 5}), 1);
g.setTileMapping(t2, 2);

g.setTileMapping(v1, 0);
g.setTileMapping(v2, 1);
```

CREATING CONTROL PROGRAMS

```
Graph g(device);
g.addCodelets("codelets.cpp");

...

auto prog = Sequence();
prog.add(Execute(cs1));
prog.add(Execute(cs2));
```

prog

```
Execute(cs1);
Execute(cs2);
```

CREATING THE ENGINE

```
Graph g(device);
g.addCodelets("codelets.cpp");

...

auto prog = Sequence();
prog.add(Execute(cs1));
prog.add(Execute(cs2));

Engine eng(device, graph, {prog});
```

prog

```
Execute(cs1);
Execute(cs2);
```

eng

SUMMARY

- Poplar lets you define your own operations by writing codelets
- Poplar generates “glue code” required to synchronize / exchange data
- Frees you to concentrate on parallel algorithm design

LIBRARIES = MODULAR GRAPH BUILDING

POPLIBS™

C / C++ and Python language bindings

poputil	popops	poplin	poprandom	popnn
Utility functions for building graphs	Pointwise and reduction operators	Matrix multiply and convolution functions	Random number generation	Neural network functions (activation fns, pooling, loss)

POPLAR®

GitHub

github.com/graphcore/poplibs

PORTING APPLICATIONS

June 11, 2024

Alexander Tsyplikhin

GRAPHCORE

INTRO TO POPTORCH

GRAPHCORE

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.

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.

TRAINING A MODEL

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

GPU

```
    _, ind = torch.max(predictions, 1)
    # provide labels only for samples, where prediction is available (during the training, not
    ions.size()[0]:]
    rch.eq(ind, labels)).item() / labels.size(  0

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 (de
    parser.add_argument('--test-batch-size', type=int, default=8, help='batch size for testing
    parser.add_argument('--epochs', type=int, default=10, help='number of epochs to train (def
    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, t
                        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,

    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("Accuracy on test set: {:0.2f}%".format(sum_acc / len(test_data)))
```

IPU

```
    _, ind = torch.max(predictions, 1)
    # provide labels only for samples, where prediction is available (during the training, not
    labels = labels[-predictions.size()[0]:]
    accuracy = torch.sum(torch.eq(ind, labels)).item() / labels.si  100.0
    return accuracy

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 (de
    parser.add_argument('--test-batch-size', type=int, default=8, help='batch size for testing
    parser.add_argument('--epochs', type=int, default=10, help='number of epochs to train (def
    parser.add_argument('--lr', type=float, default=0.05, help='learning rate (default: 0.05)'
    parser.add_argument('--device-iterations', type=int, default=50, help='device iterations (
    args = parser.parse_args()

    opts = poptorch.Options().deviceIterations(args.device_iterations)
    training_data = poptorch.DataLoader(opts,
                        torchvision.datasets.MNIST('mnist_data/', train=True, download=True, trans
                        batch_size=args.batch_size, shuffle=True, drop_last=True)
    test_data = poptorch.DataLoader(opts,
                        torchvision.datasets.MNIST('mnist_data/', train=False, download=True, tran

    model = Network()
    training_model = TrainingModelWithLoss(model)
    optimizer=optim.SGD(model.parameters(), lr=args.lr)
    training_model = poptorch.trainingModel(training_model, opts, optimizer=optimizer)
    inference_model = poptorch.inferenceModel(model)

    # Run training
    for _ in range(args.epochs):
        for data, labels in training_data:
            preds, losses = training_model(data, labels)

    # 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: {:0.2f}%".format(sum_acc / len(test_data)))
```

LARGE LANGUAGE MODELS

README.md

Optimum Graphcore

🤗 Optimum Graphcore is the interface between the 🤗 Transformers library and Graphcore IPUs. It provides a set of tools enabling model parallelization and loading on IPUs, training, fine-tuning and inference on all the tasks already supported by 🤗 Transformers while being compatible with the 🤗 Hub and every model available on it out of the box.

What is an Intelligence Processing Unit (IPU)?

Quote from the Hugging Face blog post:

> IPUs are the processors that power Graphcore's IPU-POD datacenter compute systems. This new type of processor is designed to support the very specific computational requirements of AI and machine learning. Characteristics such as fine-grained parallelism, low precision arithmetic, and the ability to handle sparsity have been built into our silicon.

> Instead of adopting a SIMD/SIMT architecture like GPUs, Graphcore's IPU uses a massively parallel, MIMD architecture, with ultra-high bandwidth memory placed adjacent to the processor cores, right on the silicon die.

> This design delivers high performance and new levels of efficiency, whether running today's most popular models, such as BERT and EfficientNet, or exploring next-generation AI applications.

+ 12

Contributors 36

+ 25 contributors

Languages

- Python 52.1%
- Jupyter Notebook 47.8%
- Makefile 0.1%

README.md

How to use Optimum Graphcore

To immediately use a model on a given input (text, image, audio, ...), we support the `pipeline` API:

```
->>> from transformers import pipeline
+>>> from optimum.graphcore import pipeline

# Allocate a pipeline for sentiment-analysis
->>> classifier = pipeline('sentiment-analysis', model="distilbert-base-uncased-finetuned-sst-2-engl
+>>> classifier = pipeline('sentiment-analysis', model="distilbert-base-uncased-finetuned-sst-2-engl
>>> classifier('We are very happy to introduce pipeline to the transformers repository.')
[{'label': 'POSITIVE', 'score': 0.9996947050094604}]
```

It is also super easy to use the `Trainer` API:

```
-from transformers import Trainer, TrainingArguments
+from optimum.graphcore import IPUConfig, IPUTrainer, IPUTrainingArguments

-training_args = TrainingArguments(
+training_args = IPUTrainingArguments(
     per_device_train_batch_size=4,
     learning_rate=1e-4,
+    # Any IPUConfig on the Hub or stored locally
+    ipu_config_name="Graphcore/bert-base-ipu",
+)
+
+# Loading the IPUConfig needed by the IPUTrainer to compile and train the model on IPUs
+ipu_config = IPUConfig.from_pretrained(
+    training_args.ipu_config_name,
 )

 # Initialize our Trainer
-trainer = Trainer(
+trainer = IPUTrainer(
     model=model,
+    ipu_config=ipu_config,
     args=training_args,
```

ai_for_simulation
finance
gnn
multimodal
nlp
bert
bloom/popxl
dolly/popxl
gpt2/pytorch
gpt3_175B/popxl
gpt3_2.7B/popxl
gpt_j/popxl
t5/popxl
preview
probability
speech
tutorials
utils
vision
.git-blame-ignore-revs

GPT-3 training on IPUs using PopXL

Framework	Domain	Model	Datasets	Tasks	Training	Inference	Reference
PopXL	NLP	GPT-3	Wikipedia	Next sentence prediction, Question/Answering	✅ Min. 256 IPUs (POD256) required	❌	Language Models are Few-Shot Learners

This README describes how to run GPT-3 models for NLP pre-training on Graphcore IPUs using the PopXL library. A combination of phased execution, tensor model parallelism, data parallelism, and remote tensor sharding are utilised to train the models.

This application shows how to run larger models on IPU. The techniques to do this mean that performance is lower than for models that fit in IPU memory. Large model training or fine-tuning requires a big Pod installation. The minimum to run pre-training with this model is a Pod256. PopXL is an experimental framework and may be subject to change in future releases.

Instructions summary

1. Install and enable the Poplar SDK (see Poplar SDK setup)
2. Install the system and Python requirements (see Environment setup)
3. Download the WIKI-103 dataset (See Dataset setup)

Poplar SDK setup

To check if your Poplar SDK has already been enabled, run:

```
echo $POPLAR_SDK_ENABLED
```

PROFILING WITH POPVISION

June 12, 2024

Alexander Tsyplikhin

GRAPHCORE

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

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.

CREATE PROFILE

```
(3.3.0+1403_poptorch) alext@bpod16:~/work/examples/tutorials/tutorials/pytorch/basics$ POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true", "autoReport.directory":"./
report", "debug.allowOutOfMemory":"true", "profiler.includeFlopEstimates":"true"}' python walkthrough.py
epochs:   0%|                                                                                                                    | 0/5 [00:00<?, ?it/s[
22:25:58.232] [poptorch:cpp] [warning] [DISPATCHER] Type coerced from Long to Int for tensor id 12                                | 0/3750 [00:00<?, ?it/s]
Graph compilation: 100%|                                                                                                          | 100/100 [00:39<00:00]
epochs: 100%|                                                                                                                     | 5/5 [01:21<00:00, 16.21s/it]
/nethome/alext/venvs/poplar_sdk-ubuntu_20_04-3.3.0+1403-208993bbb7/3.3.0+1403_poptorch/lib/python3.8/site-packages/torch/nn/modules/module.py:1802: UserWarning: P
ositional args are being deprecated, use kwargs instead. Refer to https://pytorch.org/docs/master/generated/torch.nn.Module.html#torch.nn.Module.state_dict for de
tails.
  warnings.warn(
Graph compilation: 100%|                                                                                                          | 100/100 [00:19<00:00]
Eval accuracy: 89.25%
Graph compilation: 100%|                                                                                                          | 100/100 [00:09<00:00]
tensor([[-4.4754, -0.9084, -3.2791, -5.8905, -3.4738, -1.4554, -1.8643, -6.6169,
         -2.1170, -5.5300]])
IPU predicted class: Trouser
CPU predicted class: Trouser
```

PopVision® Graph Analyser [10.129.111.38 : /nethome/alext/work/examples/tutorials/tutorials/pytorch/basics/report/training]

- Summary
- Insights
- Memory Report
- Liveness Report
- Program Tree
- Operations Summary
- Operations Graph
- Execution Trace
- Reload Report
- Close Report
- Documentation
- Minimise

Graph Type - Total Memory | Breakdown - None | 201 | Options

Navigation 100%

Details | Compute Sets | Vertices | Exchanges | Variables

	Tile 201
Physical Tile Id	522
Memory Including Gaps	209 KiB
Memory Excluding Gaps	28.3 KiB
By Memory Region	
Non-interleaved	27.4 KiB
Interleaved	960 B
Overflowed	0 B
By Data Type	
Not Overlapped	
Variables	1.6 KiB
Constants	414 B
Control Code	2.4 KiB
Vertex Code	8.8 KiB
Internal Exchange Code	7.2 KiB
Host Exchange Code	640 B
Global Exchange Code	0 B
Vertex Instances	1.4 KiB

PopVision® Graph Analyser [10.129.111.38 : /nethome/alext/work/examples/tutorials/tutorials/pytorch/basics/report/training]

- Summary
- Insights
- Memory Report
- Liveness Report
- Program Tree
- Operations Summary
- Operations Graph
- Execution Trace
- Reload Report
- Close Report
- Documentation
- Minimise

Filter steps | Select Source | Options

Navigation 100%

Selected Program Steps

Step 312 OnTileExecute (conv2/Conv:ConvDataGrad/218/convolution/Conv_5x5/Convolve)

Always-Live Variables | Not-Always-Live Variables | Vertices | Cycle Estimates | FLOP Estimates

Filter variables

Variable	All Tiles
Total	**29.3 MiB**
vertexCode	11.8 MiB
internalExchangeCode	8.5 MiB
controlCode	3.0 MiB
vertexInstanceState	1.9 MiB
hostExchangeCode	924 KiB
stack	535 KiB
instrumentationResults	402 KiB
Accl1__fc1.weight	380 KiB
fc1.weight	380 KiB
vertexFieldData	241 KiB
copyDescriptor	223 KiB
hostExchangePacketHeader	198 KiB

PopVision® Graph Analyser [10.129.111.38 : /nethome/alext/work/examples/tutorials/tutorials/pytorch/basics/report/training]
Summary
Insights
Memory Report
Liveness Report
Program Tree
Operations Summary
Operations Graph
Execution Trace
Reload Report
Close Report
Documentation
Minimise
Execution View - Flame & BSP
Navigation
58957%
All IPUs
Search
Options
1,490,000
1,500,000
1,510,000
1,520,000
1,530,000
1,540,000
1,550,000
1,560,000
1,570,000
IPU 0
Engine::run #0
100%
Summary
Details
Total Cycles 94,526 Total FLOPs 31,844,509
IPU
Proportions
Tile Utilisation
Stream Copy (Rx/Tx)
Global Exchange (Rx/Tx)
0
59.16%
0 B / 0 B
0 B / 0 B
One or more step types are hidden. These step types are excluded from the tile balance, tile utilisation, and cycle proportions shown and so the statistics may not be representative. Step types can be shown or hidden using the view options dropdown.

PopVision® Graph Analyser [10.129.111.38 : /nethome/alext/work/examples/tutorials/tutorials/tensorflow2/infeed_outfeed/completed_code/report/tf_report__2023-07-25__01-03-33.553__93433 vs 10.129.111.38 : /nethome/alext/work/examples/tutorials/tutorials/tensor...

- Summary
- Memory Report
- Liveness Report
- Program Tree
- Operations Summary
- Execution Trace
- Swap Reports
- Reload Report
- Close Report
- Documentation
- Minimise

Filter steps | Select Source | Options

Navigation 100%

Navigation 100%

Selected Program Step

Name (source): None selected. Click on the graph to select a source program step

Name (target): None selected. Click on the graph to select a target program step

Always-Live Variables | Not-Always-Live Variables | Vertices | Cycle Estimates

Filter variables

Variable	Source: All Tiles	Target: All Tiles	Diff: All Tiles
Total	**11.3 MiB**	**11.2 MiB**	**-113 KiB**
vertexCode	6.0 MiB	6.0 MiB	-16 B (-0.00%)
controlCode	1.5 MiB	1.5 MiB	-37.7 KiB (-2.41%)
internalExchangeCode	1.2 MiB	1.1 MiB	-47.6 KiB (-3.99%)
hostExchangeCode	595 KiB	556 KiB	-39.2 KiB (-6.58%)
stack	540 KiB	529 KiB	-11.0 KiB (-2.04%)
vertexInstanceState	527 KiB	527 KiB	20 B (0.00%)
while/sequential/dense/MatMul/dot.35/rhs	392 KiB	—	-392 KiB (-100.00%)
sequential/dense/MatMul/dot/rhs	—	392 KiB	392 KiB (100.00%)
instrumentationResults	180 KiB	180 KiB	4 B (0.00%)
hostExchangePacketHeader	114 KiB	137 KiB	23.7 KiB (17.25%)

PopVision® Graph Analyser [10.129.111.38 : /nethome/alext/work/examples/tutorials/tutorials/pytorch/pipelining/report/training]

- Summary
- Insights
- Memory Report
- Liveness Report
- Program Tree
- Operations Summary
- Operations Graph
- Execution Trace
- Reload Report
- Close Report
- Documentation
- Minimise

Execution View - Flame | Navigation | 260775% | All IPUs | Search | Options

2,600,000 2,800,000 3,000,000 3,200,000 3,400,000 3,600,000 3,800,000 4,000,000 4,200,000

IPU 0

Engine::run #0

St...y StreamCopy StreamCopy StreamCopy StreamCopy StreamCopy Ex...c StreamCopy StreamCopy StreamCopy Stre...Copy E...

Mid Mid Mid Mid Mid Mid Mid Mid Mid Mid

IPU 1

Engine::run #0

St...y StreamCopy StreamCopy StreamCopy StreamCopy StreamCopy StreamCopy Stre...Copy StreamCopy Stre...Copy E...

End End End End End End End End End End

IPU 2

Engine::run #0

St...y StreamCopy StreamCopy StreamCopy StreamCopy StreamCopy Ex...c StreamCopy Ex...c Stre...Copy Ex...c StreamCopy Ex...c Stre...Copy Ex...c Ex...c E... E... E... Ex...c E...

End End End End End End End End End End End

IPU 3

Engine::run #0

St...y StreamCopy StreamCopy StreamCopy StreamCopy StreamCopy E...c StreamCopy E...c StreamCopy E...c StreamCopy E...c StreamCopy E...c E...c S... Ext...nc Ext...nc E...c

End End End B... End End End End End End End End

Summary Details

Total Cycles 1,700,636 Total FLOPs 2,227,345,013

IPU	Proportions	Tile Utilisation	Stream Copy (Rx/Tx)	Global Exchange (Rx/Tx)
0		56.02%	1.1 MiB / 172 KiB	4.1 MiB / 4.2 MiB
1		54.73%	0 B / 0 B	5.0 MiB / 5.3 MiB
2		26.01%	0 B / 0 B	1.8 MiB / 1.4 MiB
3		18.01%	768 B / 320 B	640 KiB / 640 KiB

PopVision® System Analyser
Add file
Flame Graph Type - Aggregated
Line Graph Type - Line
Heatmap Options
Navigation
100%
Search
Options
Events 405
Duration 17.561 secs
Collapse all charts
00:00:00
00:00:02
00:00:04
00:00:06
00:00:08
00:00:10
00:00:12
00:00:14
00:00:16
Tue_Jul_25_00:06:49_2023_UTC_57...
Aggregated
po...ad
pop...ch
pop...ack
p...
poplar::cor...::operator
po...r
IPUSync...alThan
poplar::core:...repareAccess
pop...ss
IP...t
poplar::co...aitForAny
poplar::core::IPUTarget::processHspSync
po...h
poplar::core::IPUTarget::run
poplar::core::IPUTarget::run
P...e
poplar::core::Engine::runProgramUnchecked
p...e
pop...oad
poplar::core::Engine::runProgram
p...e
popl...oad
popart::popx::Devicex::run
popa...ble
popart:...treams
popart::popx::Devicex::run
p...h
popa...ice
popart:...treams
popart::Session::run
compileWi...ITracing
loadEn...treams
execute
poptorch.training...modelCompilation
poptorch.trainingModel.modelExecution
po...et
Prep...ble
popl...are
poplar...::load
popl...are
poplar:...::load
popar...able
popart::pop...ectStreams
popl...ach
popar...vice
popart::Ses...ectStreams
compileWith...nualTracing
loadEngineA...ectStreams
poptorch.inferenceModel.modelCompilation
Summary
Call Tree
Aggregate
Self time
Total time
Activity
3.421 secs 38.0% 8.996 secs 100.0% poplar::core::IPUTarget::run
565.996 ms 6.3% 3.615 secs 40.2% poplar::core::IPUTarget::processHspSync
366.950 ms 4.1% 1.945 secs 21.6% poplar::core::SyncGroupFlow::prepareAccess
352.631 ms 3.9% 1.578 secs 17.5% poplar::core::Fetch::operator
58.174 ms 0.6% 658.362 ms 7.3% poplar::core::Fetch::doCallback
43.665 ms 0.5% 600.188 ms 6.7% popart::popx::Devicex::PrefetchCallback::fetch
556.523 ms 6.2% 556.523 ms 6.2% popart::popx::Devicex::InputDatastream::read
306.832 ms 3.4% 306.832 ms 3.4% poplar::core::Fetch::doHostGatewaySync
192.610 ms 2.1% 192.610 ms 2.1% poplar::core::Fetch::doHostGatewayMirror
36.681 ms 0.4% 67.629 ms 0.8% popart::popx::Devicex::PrefetchCallback::complete
147.590 ms 1.6% 599.345 ms 6.7% poplar::core::SyncGroupFlow::completeAccess
331.532 ms 3.7% 331.532 ms 3.7% IPUSync::incrementMarkCount
96.082 ms 1.1% 173.335 ms 1.9% poplar::core::SyncGroupFlow::releaseAccess
313.409 ms 3.5% 1.503 secs 16.7% poplar::core::IPUDevice::waitForAny

PopVision® System Analyser
Add file
Flame Graph Type - Timeline
Line Graph Type - Line
Heatmap Options
Search
Options
Events 3,163,853
Duration 58.214 secs
Collapse all charts
00:00:00
00:00:10
00:00:20
00:00:30
00:00:40
00:00:50
Tue_Jul_25_00:42:31_2023_UTC_762...
Derived Info
IPU(s) Activity
Heatmaps
conv1.weight
conv2.weight
Process 76244
Track 0
×10
Epoch:0
Epoch:1
Epoch:2
Epoch:3
Epoch:4
Epoch:5
Epoch:6
Epoch:7
Epoch:8
Epoch:9
popt...tion
co...g
lo...s
Line Graphs
Loss
2.6