SOFTWARE STACK: TENSORFLOW PYTORCH POPLAR

June 11, 2024

Alexander Tsyplikhin

GRAPHCORE

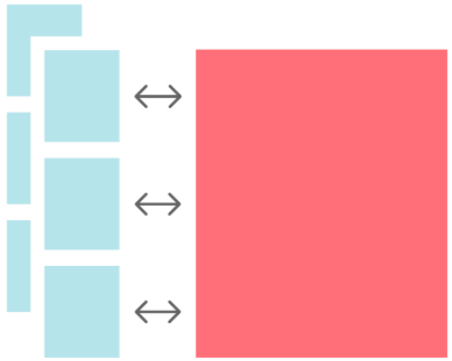
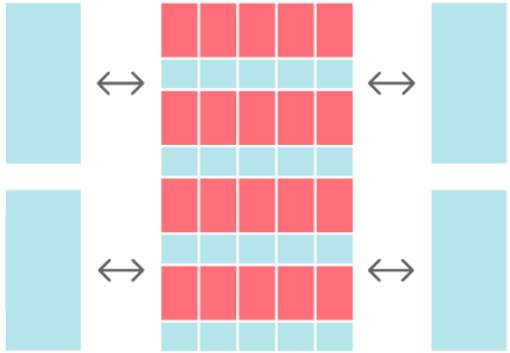
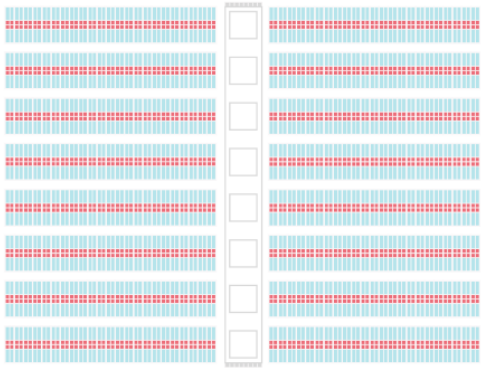
AGENDA

- Architecture Refresher
- Software Ecosystem
- TensorFlow2/Keras
- PyTorch
- Poplar

ARCHITECTURE REFRESHER

IPU – ARCHITECTURED FOR AI

Massive parallelism with ultrafast memory access

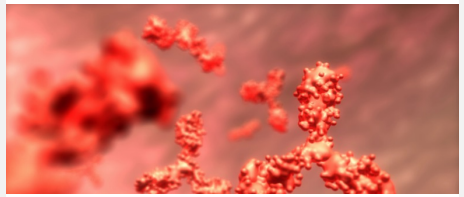
	CPU	GPU	IPU
Parallelism	Designed for scalar processes	SIMD/SIMT architecture. Designed for large blocks of dense contiguous data	Massively parallel MIMD. Designed for fine-grained, high-performance computing
Processors / Memory			
Memory Access	Off-chip memory	Model and data spread across off-chip and small on-chip cache, and shared memory	Model and data tightly coupled, and large locally distributed SRAM

PROVEN IPU ADVANTAGE

SELECT CASE STUDIES ACROSS MANY INDUSTRIES & FIELDS

LabGenius

HEALTHCARE



CASE STUDY : NLP

Man

FINANCE – OPTION PRICING

CASE STUDY : SIM

pienso

AI SaaS – TEXT ANALYTICS

CASE STUDY : NLP

Argonne NATIONAL LABORATORY

RESEARCH / BIG LABS

CASE STUDY

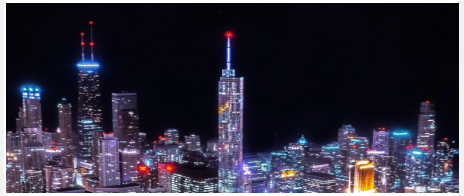
Pacific Northwest NATIONAL LABORATORY

COMPUTATIONAL CHEMISTRY

CASE STUDY : GNN

SENSORO

SMART CITY



CASE STUDY : CV

TRACTABLE

FINANCE - INSURANCE

CASE STUDY : CV

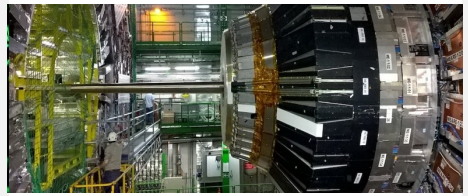
ECMWF

WEATHER FORECASTING

CASE STUDY : SIM

SLAC NATIONAL ACCELERATOR LABORATORY

HIGH ENERGY PHYSICS



CASE STUDY

UNIVERSITY OF OXFORD

DYNAMIC GRAPHS

CASE STUDY : GNN

BOW IPU

IPU-Tiles™

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

IPU-Core™

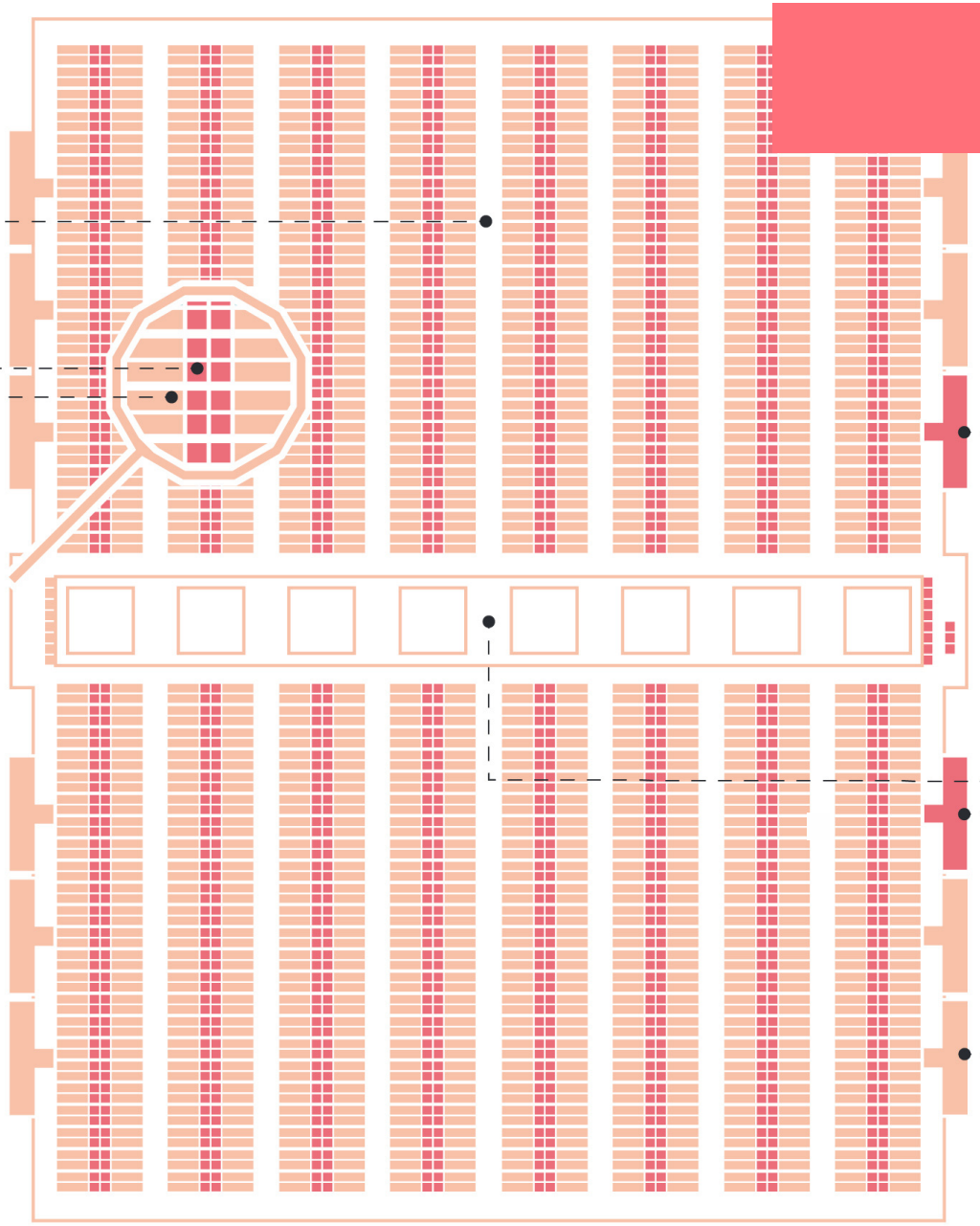
1472 independent IPU-Core™

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

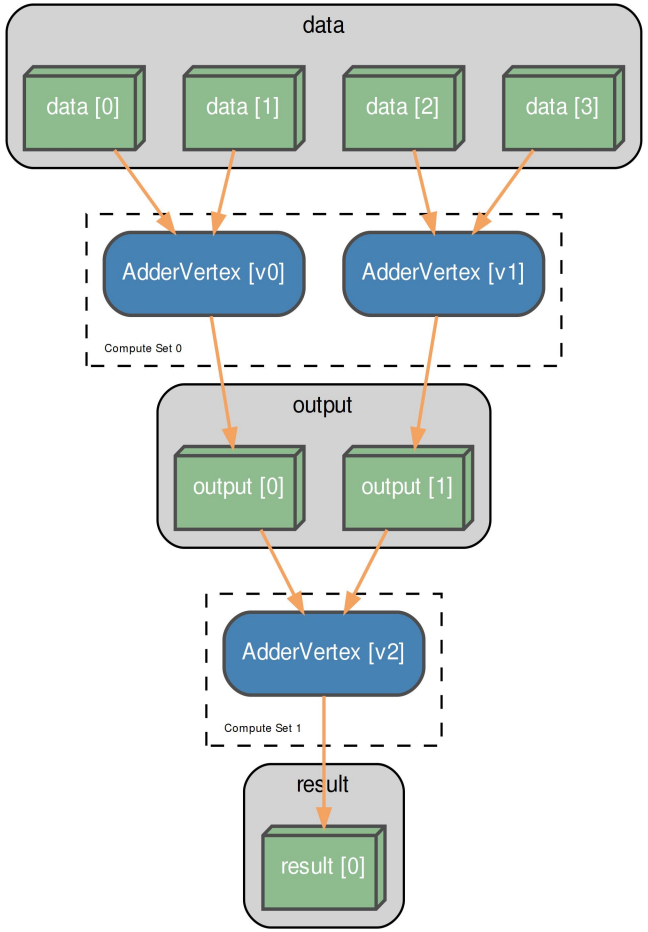
PCI Gen4 x16
64 GB/s bidirectional bandwidth to host

IPU-Links™

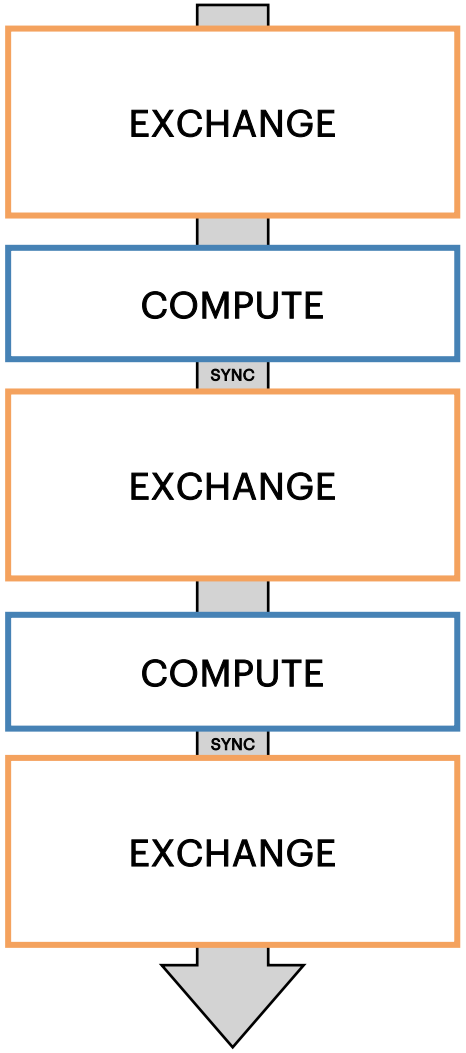
10 x IPU-Links,
320GB/s chip to chip bandwidth

EXECUTION MODEL

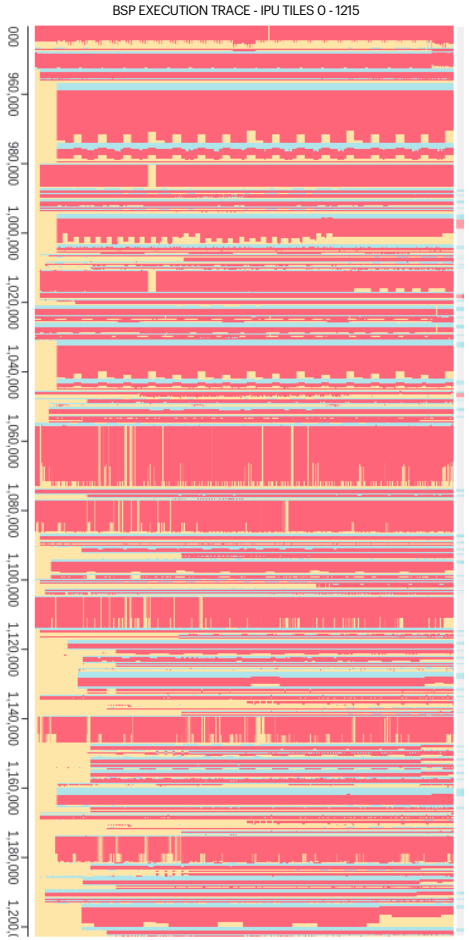
COMPUTATIONAL GRAPH



BSP SCHEDULE



OPTIMIZED IPU EXECUTION



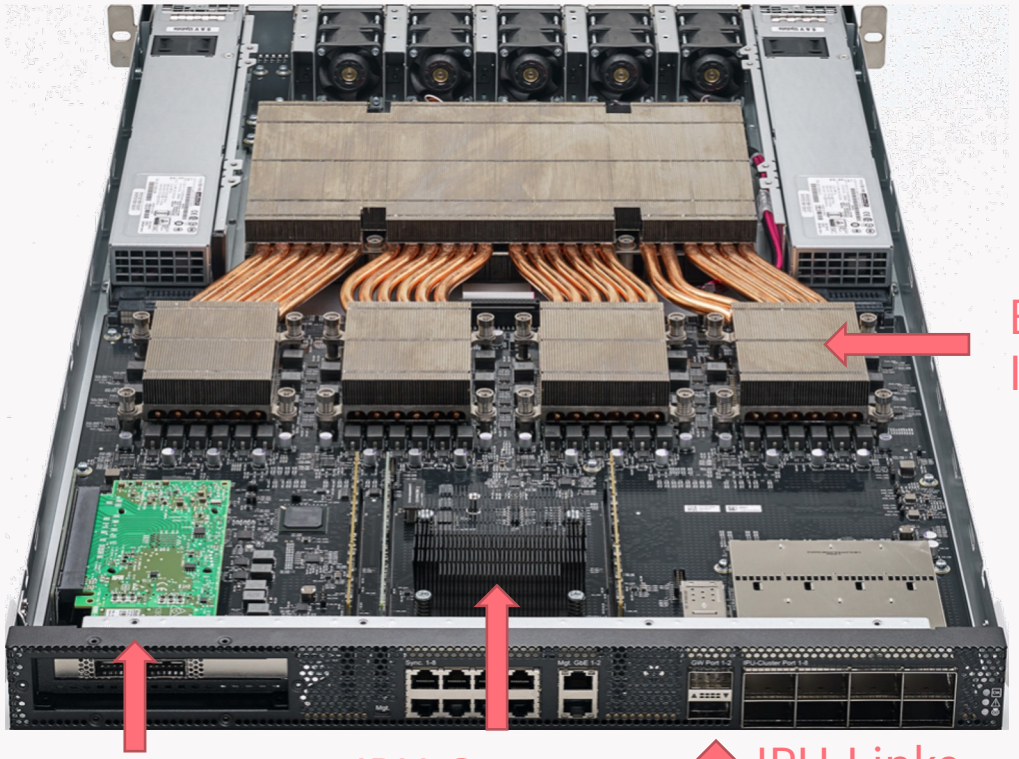
OUTPUT FROM POPVISION GRAPH ANALYSER

GRAPHCORE

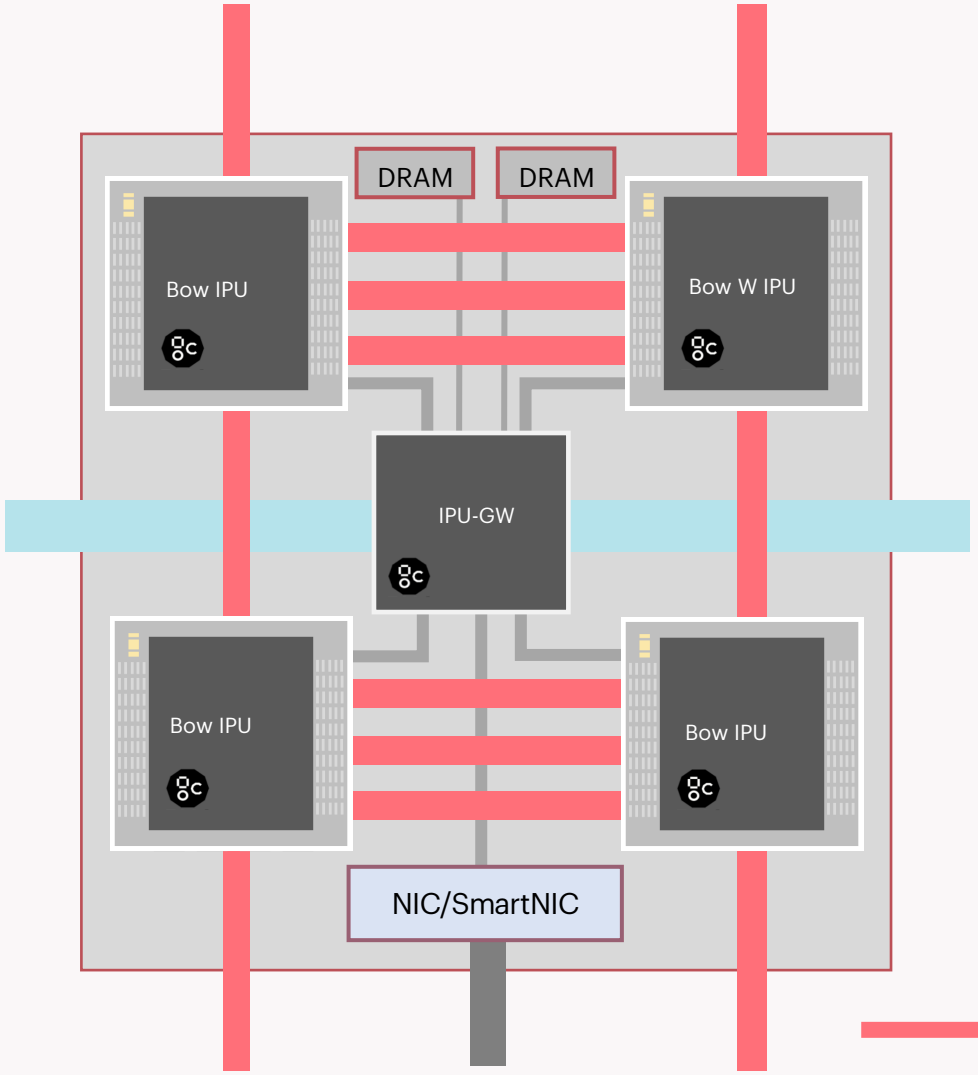
BOW-2000 IPU MACHINE

1U blade form factor delivering 1.4 PetaFLOPS AI Compute

BOW IPU-2000



BOW-2000 TOPOLIGY



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)

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

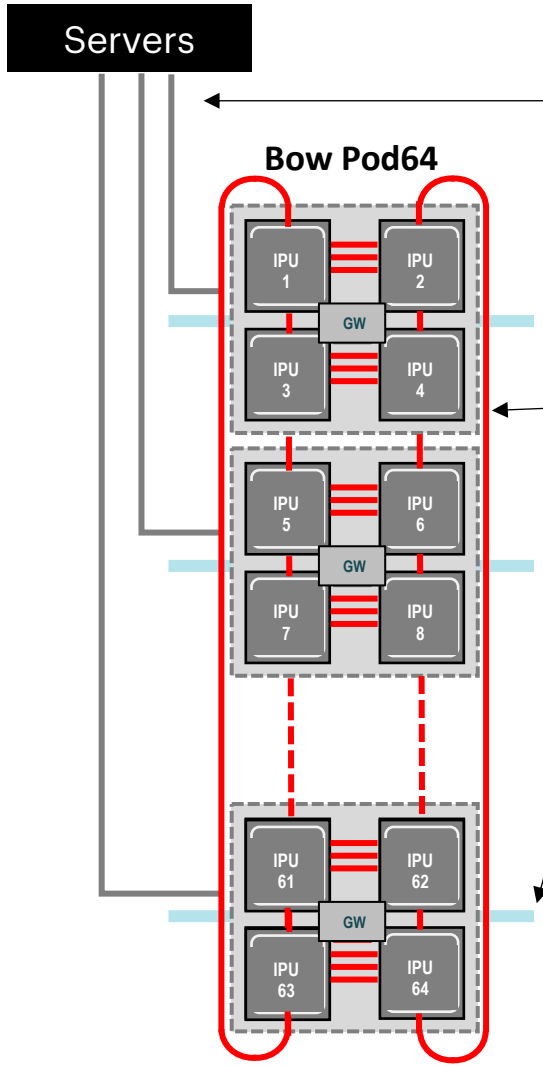
x16 IPU-Link [64GB/s]

Host-Link Network I/F [100Gbps]

IPU-GW Link [100Gbps]

x8 PCIe G4 [32GB/s]

BOW-POD64 TOPOLOGY



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

x16 IPU-Link 64GB/s

100Gbps Host-Link Network I/F

100Gbps IPU-GW Link

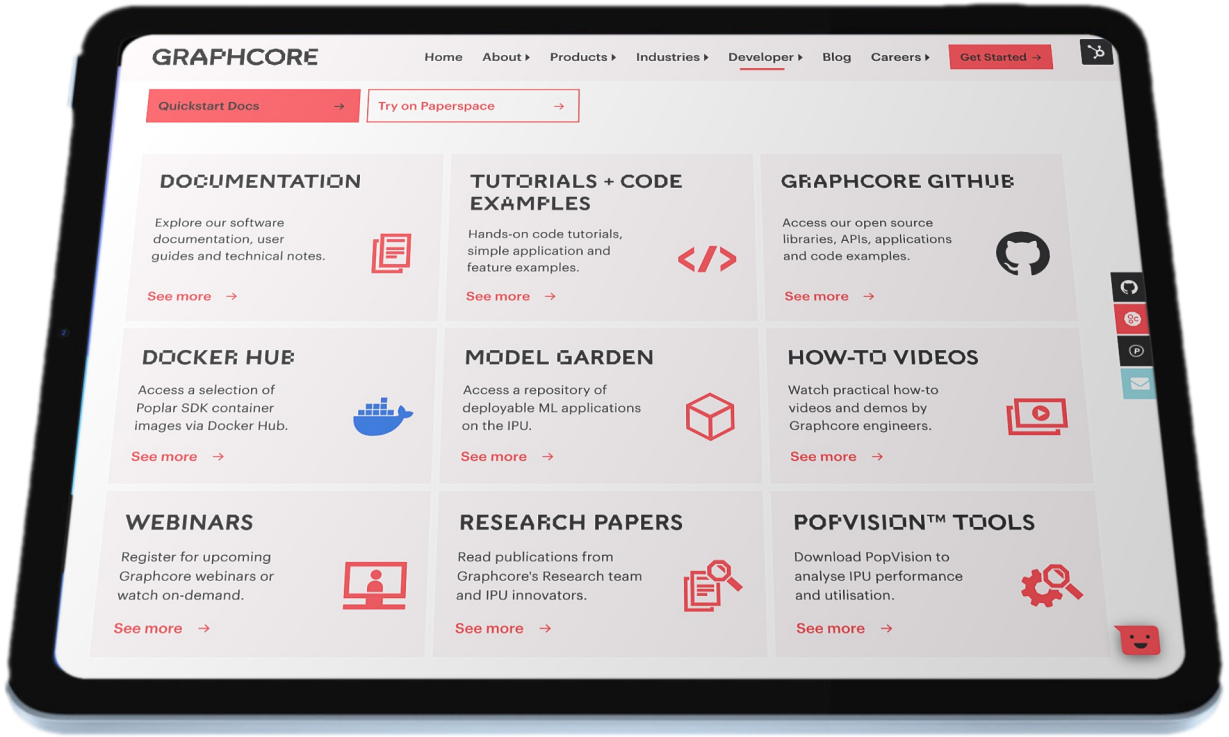
x8 PCIe G4 32GB/s

IPU DEVELOPER ECOSYSTEM

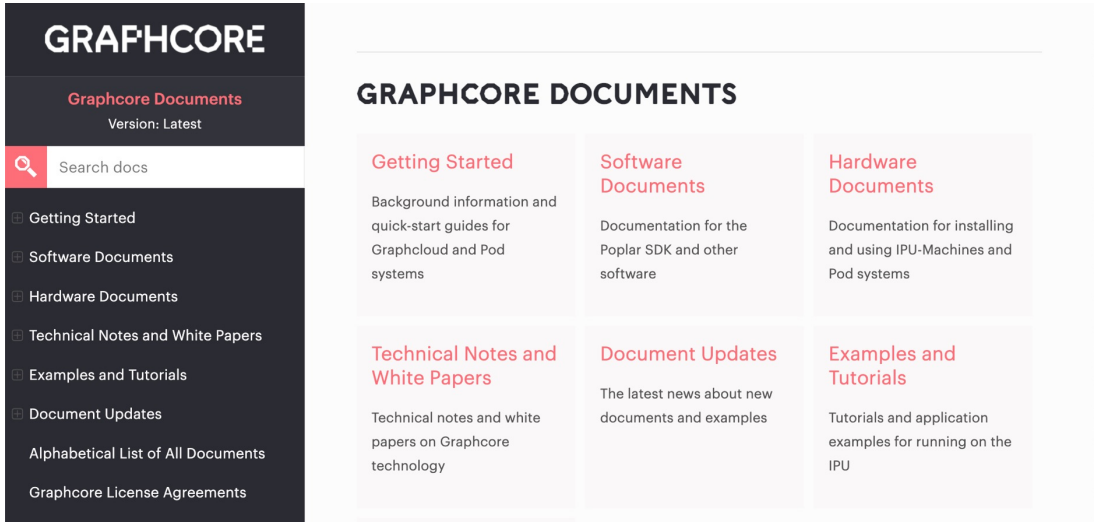
GRAPHCORE

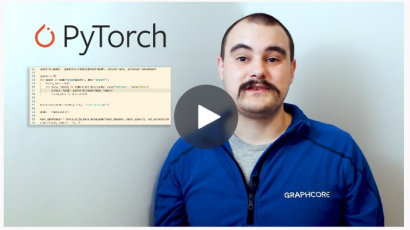
GRAPHCORE SOFTWARE ECOSYSTEM

WORLD CLASS DEVELOPER RESOURCES FOR IPU USERS



WWW.GRAPHCORE.AI/DEVELOPER





Getting started with PyTorch for the IPU

Running a basic model for training and inference

AI Customer Engineer, Chris Bogdiukiewicz introduces PyTorch for the IPU. With PopTorch™ - 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.

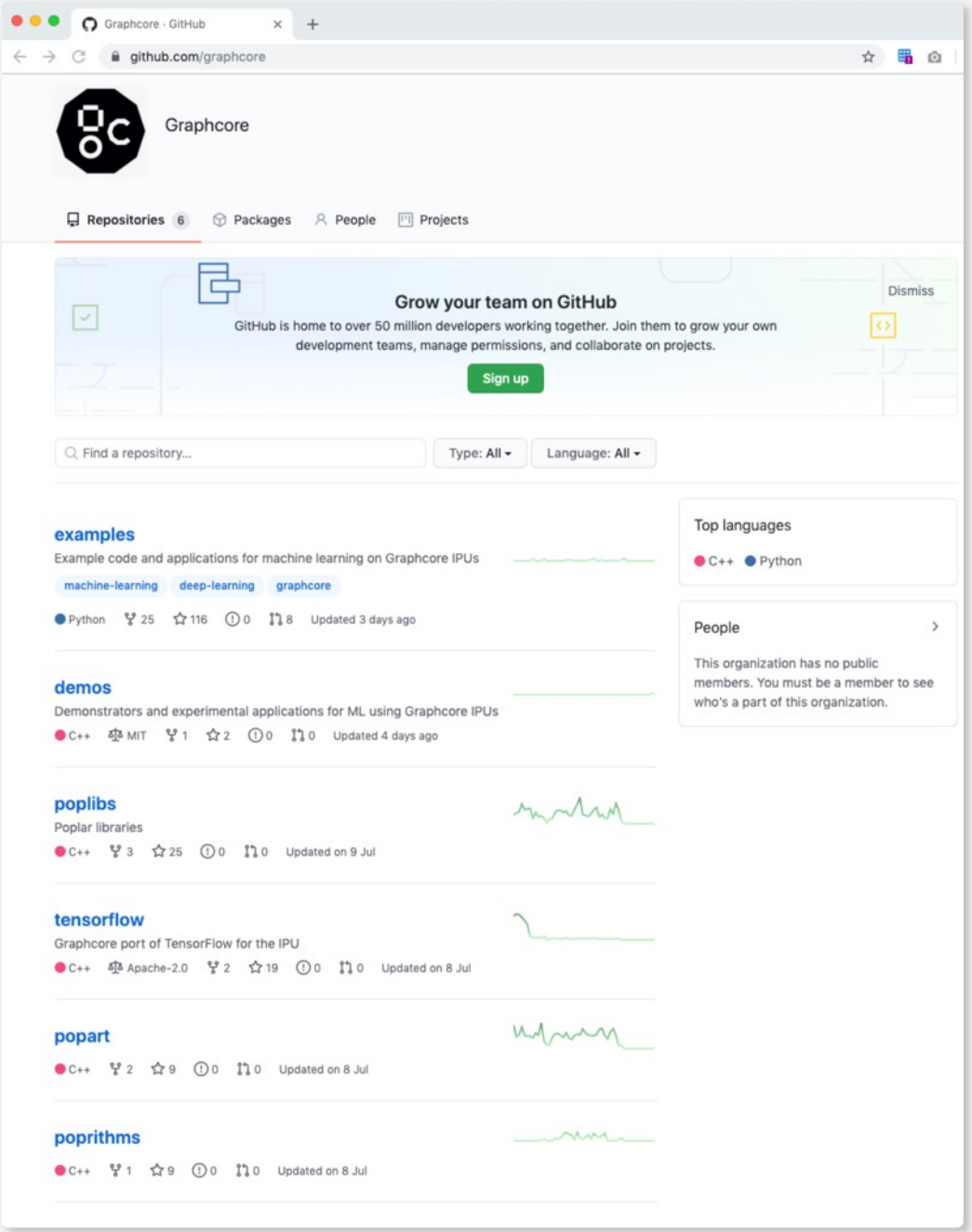
Get the Code →

Read the Guide →

OPEN SOURCE

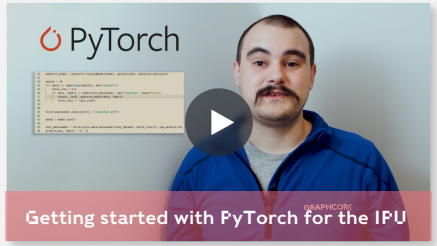
github.com/graphcore

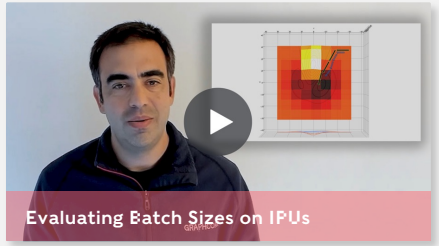
- As part of our ethos to put power in the hands of AI developers, Graphcore open sourced in 2020
- PopLibs™, PopART, PyTorch & TensorFlow for IPU fully open source and available on GitHub
- Our code is public and open for code contributions from the wider ML developer community

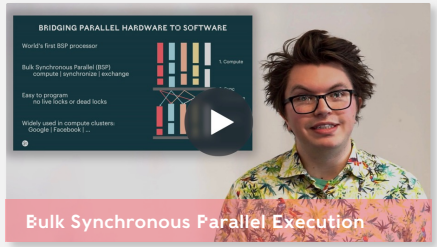


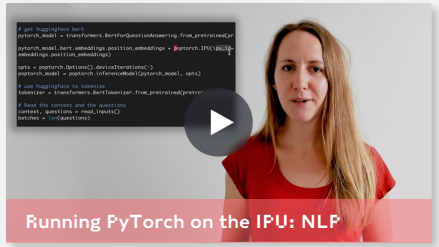
VIDEO + GITHUB TUTORIALS

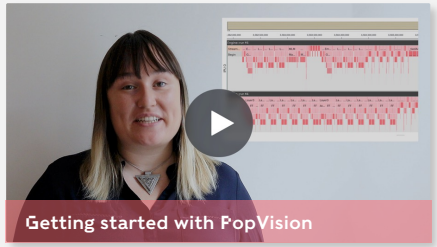
A comprehensive set of online developer training materials and educational content

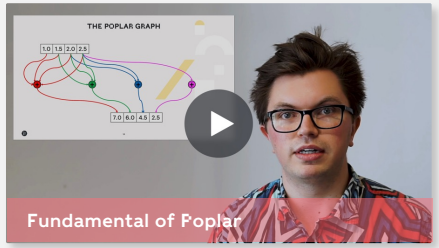


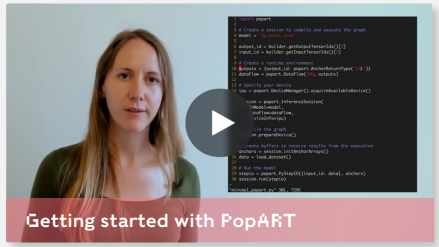


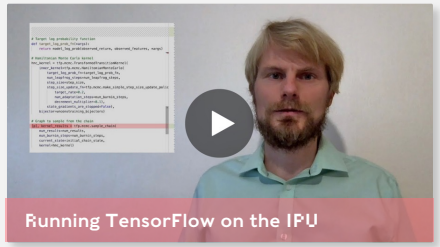












TUTORIALS

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

- Programs and Variables
- Using PopLibs
- Writing Vertex Code
- Profiling Output
- Basic Machine Learning Example
- Matrix-Vector Multiplication
- Matrix-Vector Multiplication Optimisation
- Simple PyTorch for the IPU NEW

Tutorial 1: programs and variables

Copy the file `tut1_variables/start_here/tut1.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 tutorials use a simulated target by default, so will run on any machine even if it has no Graphcore hardware attached. On systems with accelerator hardware, the header file `poplar/DeviceManager.hpp` contains API calls to enumerate and return `Device` objects for the attached hardware.

Simulated devices are created with the `IPUModel` 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`:

```
// Create the IPU Model device
IPUModel ipuModel;
Device device = ipuModel.createDevice();
Target target = device.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:

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 `tut5_ml`. 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.

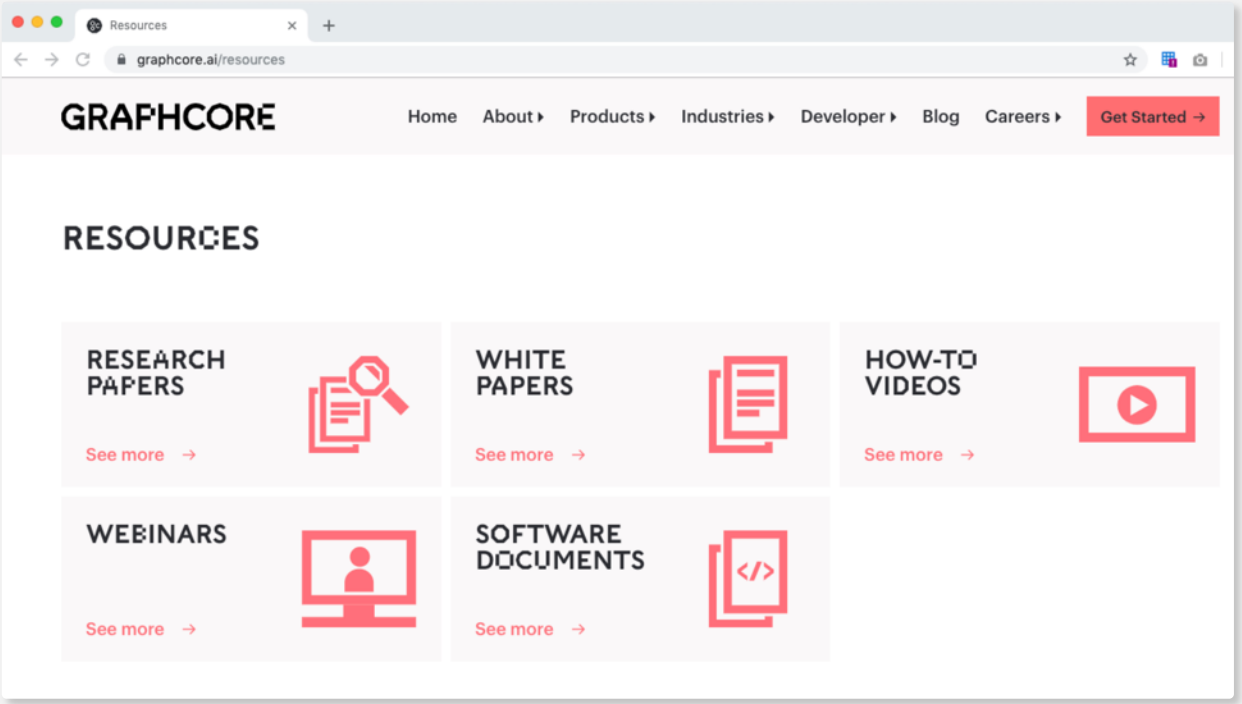
The program accepts an optional command line argument to make it use the IPU hardware instead of a simulated IPU.

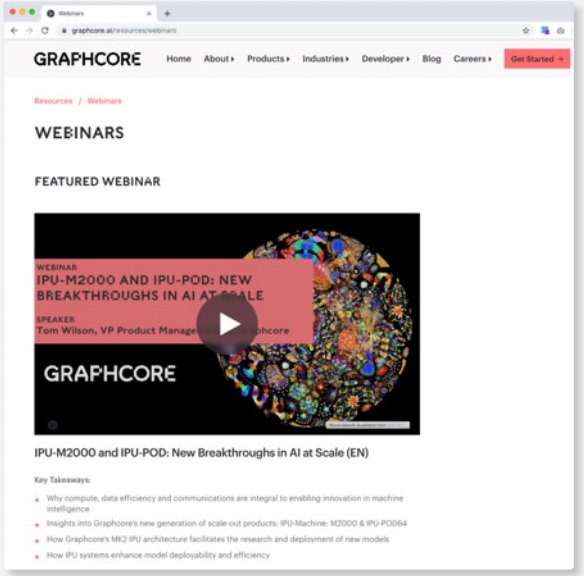
As you would expect, training is significantly faster on the IPU hardware.

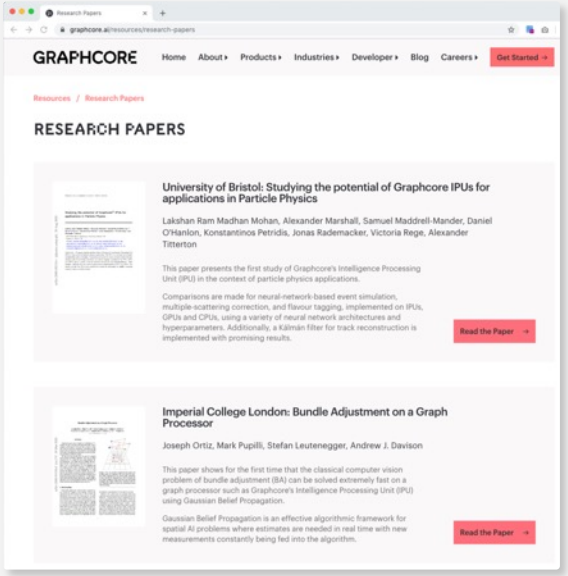
RESOURCES CENTRE

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







GRAPHCORE DEVELOPER ECOSYSTEM

TensorFlow

Keras

HUGGING FACE

PyTorch

PyTorch Lightning

PyG

jupyterhub

medium.com

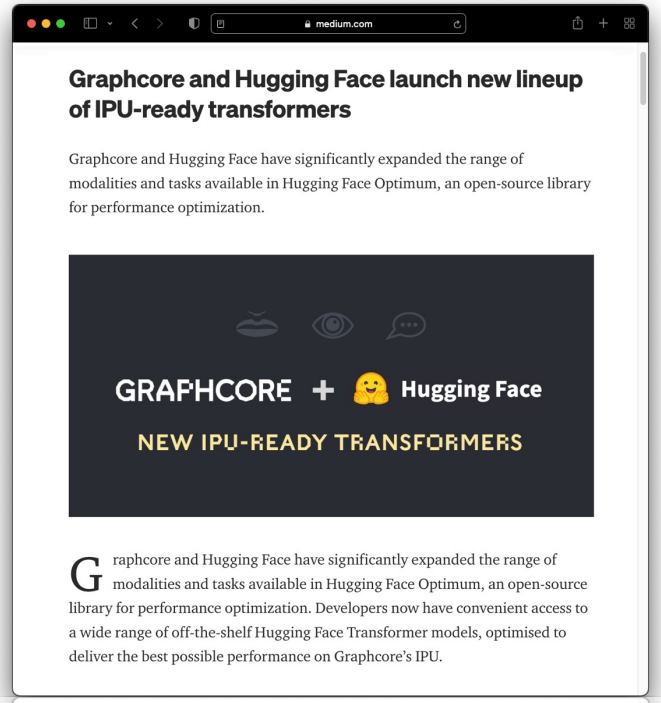
Graphcore Announces Production Release of PyTorch for IPU

PyTorch Follow
Dec 9, 2020 · 5 min read

Author: Matt Fyles, SVP Software, Graphcore

PyTorch + GRAPHCORE

Today we are introducing our first production release of PyTorch for IPU — PopTorch — combining the performance of the Graphcore IPU-M2000 system and the developer-ready accessibility of PyTorch.This will enable the fast-growing PyTorch developer community to make new breakthroughs in machine intelligence with Graphcore IPU systems, while maintaining the dynamic PyTorch experience.



medium.com

Graphcore and Hugging Face launch new lineup of IPU-ready transformers

Graphcore and Hugging Face have significantly expanded the range of modalities and tasks available in Hugging Face Optimum, an open-source library for performance optimization.

GRAPHCORE + Hugging Face

NEW IPU-READY TRANSFORMERS

Graphcore and Hugging Face have significantly expanded the range of modalities and tasks available in Hugging Face Optimum, an open-source library for performance optimization. Developers now have convenient access to a wide range of off-the-shelf Hugging Face Transformer models, optimised to deliver the best possible performance on Graphcore's IPU.

Weights & Biases

PaddlePaddle

HALO

docker

openstack.

vmware

GitHub

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

Existing models on alternative platforms

Easy port of high-level framework models

POPLAR®

IPU-Processor Platforms

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

- TF
- ONNX
- HALO
- Keras
- PaddlePaddle

FW BACKENDS

- XLA
- POPART+
- POPDIST
- PARTITIONER
- POPIR
- POPIT

POPLAR®

- 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

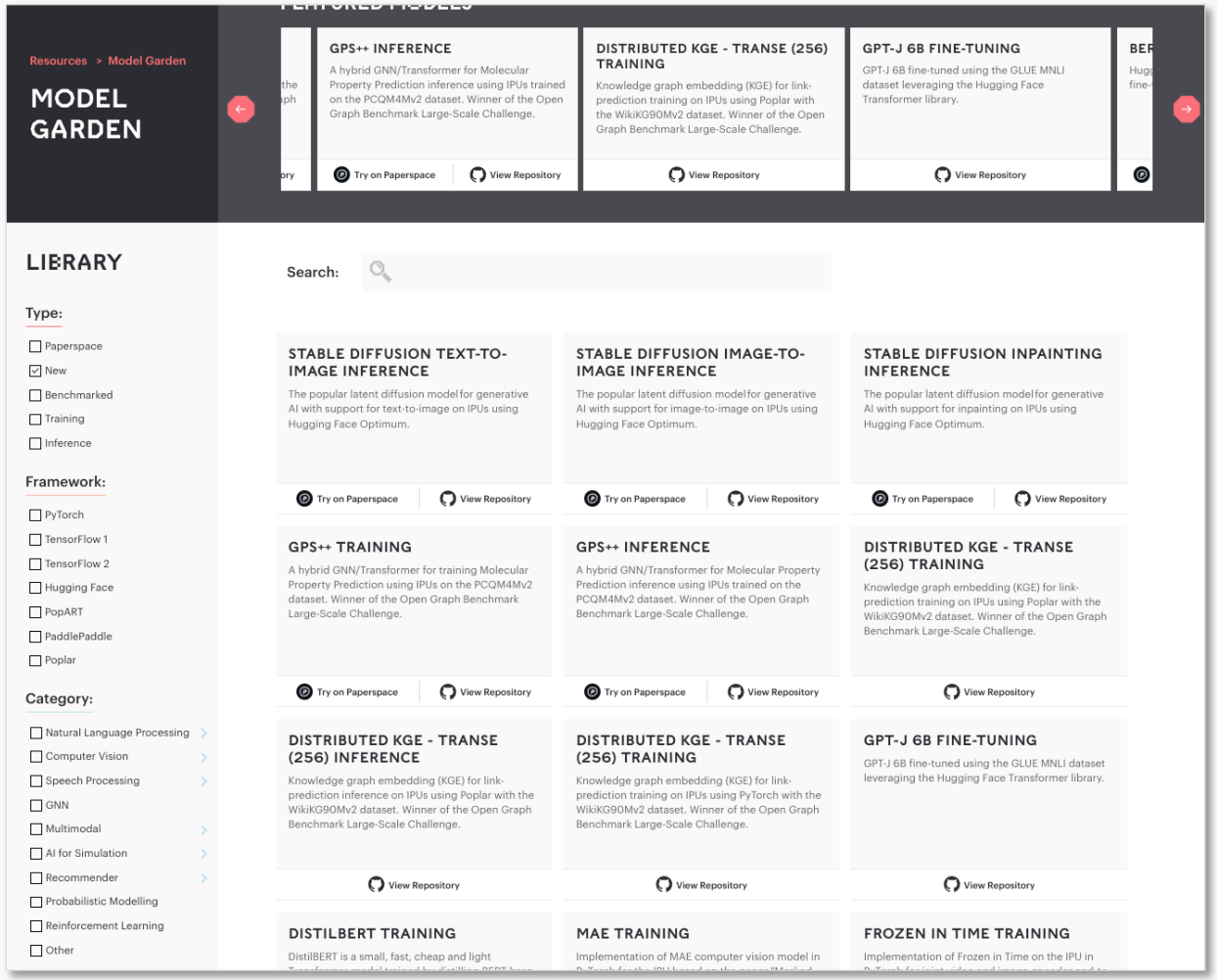
- PROMETHEUS
- GRAFANA

JOB DEPLOYMENT

- K8S
- SLURM

SYSTEM SOFTWARE

ENHANCED MODEL GARDEN



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

NEW FILTER/SEARCH CAPABILITY

DIRECT ACCESS TO GITHUB

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

MODEL GARDEN COVERAGE

COMPUTER VISION

IMAGE CLASSIFICATION
- ResNet50 v1.5
- EfficientNet-B0
- EfficientNet-B4
- ResNeXt-101
- MobileNet v2
- MobileNet v3
- ViT
- DINO
- SWIN
- MAE

OBJECT DETECTION
- YOLO v3
- YOLO v4
- Faster RCNN
- EfficientDet

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

GNN
- TGN
- MPNN-GIN
- GPS++
- Distr. KGE
- Cluster-GCN
- SchNet
- NBFNet

AI FOR SIMULATION
- DeepMD
- DeepDriveMD
- ETO
- CosmoFlow
- ABC Covid-19

REINFORCEMENT
- RL
  Reinforcement Learning

PROBABILISTIC
- MCMC
- VAE

NLP
- Dolly
- BERT
- Group | Packing
- GPT-2
- GPT-J
- RoBERTa
- Deberta
- BART
- T5
- Hubert
- DistilBERT

SPEECH

STT (ASR)
- RNN-T
- Conformer
- Wav2Vec2
- Whisper

TTS
- DeepVoice3
- FastSpeech2
- FastPitch

RECOMMENDER
- Autoencoder
- DIN
- DIEN

MULTIMODAL
- LXMERT
- CLIP
- Stable Diffusion
- Mini DALL-E
- Frozen In Time

OTHER
- Sales Forecast
- Neural Image Fields

PyG
PyTorch
TensorFlow
Hugging Face
Keras
PaddlePaddle

POPVISION®

INDUSTRY LEADING AI APPLICATION PERFORMANCE ANALYSIS TOOLS

Introduced in Q2 2020 our PopVision analysis tools provide detailed observability of IPU applications

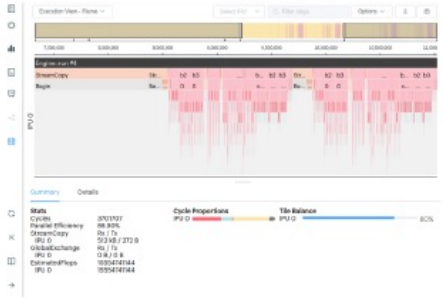
- Poplar Graph Analyser allows visual inspection of IPU execution down to the individual tile level
- Poplar System Analyser gives users the ability to view host side application and IPU interaction
- Both tools extend debug information back up into Tensorflow and Pytorch for developers

SUPPORTED PLATFORMS

POPVISION TOOLS

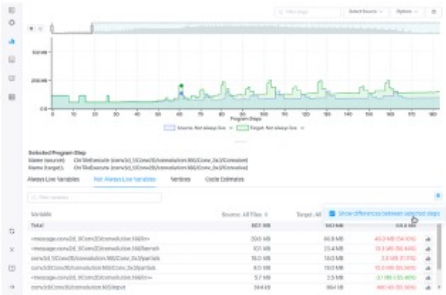
IPU MEMORY ANALYSIS

Capture memory information from your ML models when executed on IPUs. Inspect variable placement, size and liveness throughout the execution.



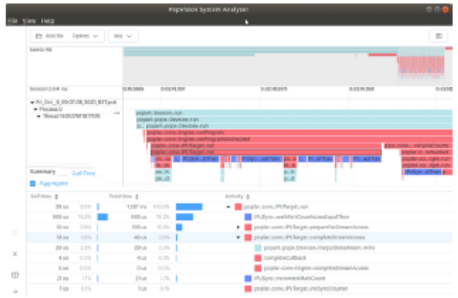
EXECUTION TRACE REPORT

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.



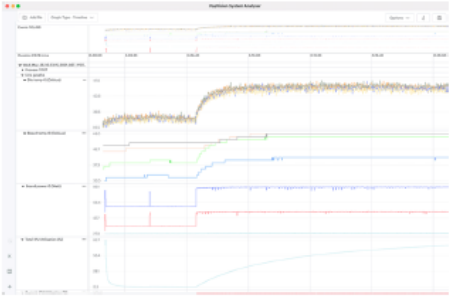
REPORT COMPARISONS

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.



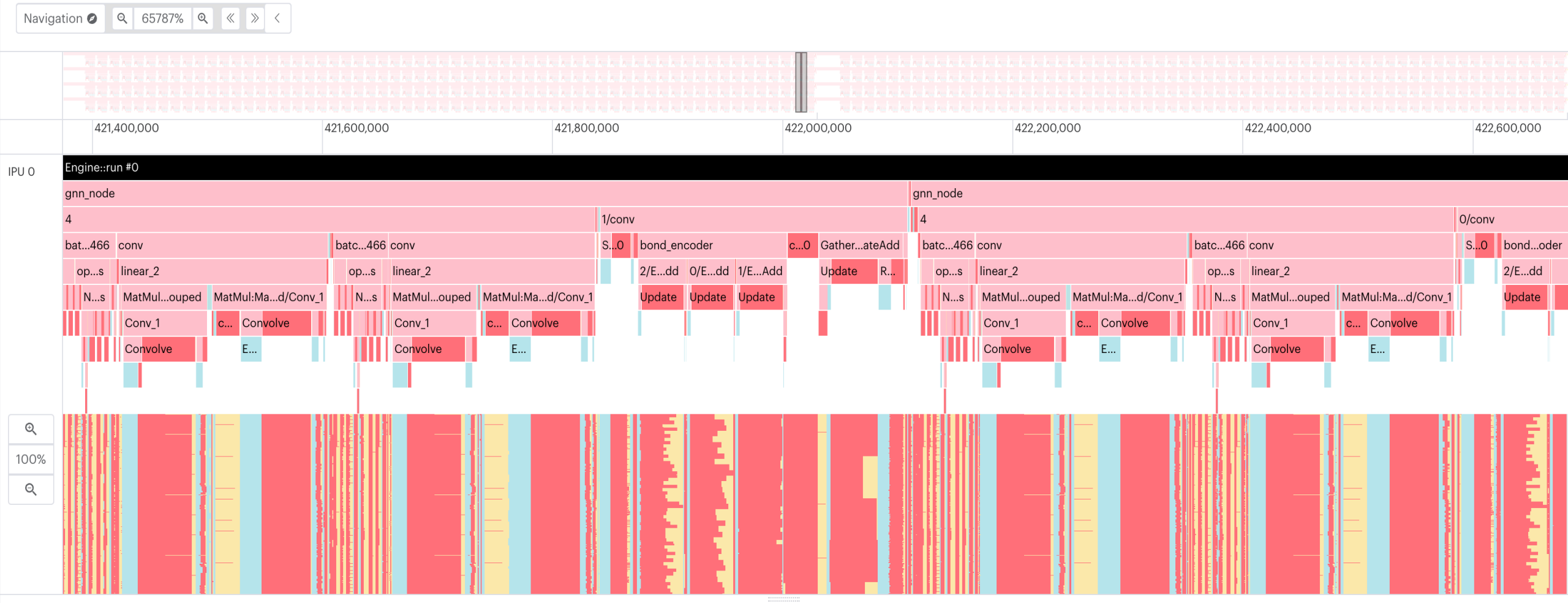
GRAPH DATA

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.

POPVISION PERFORMANCE ANALYSER



Utilization/memory map of every tile/every IPU

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.

gpu_cnn_keras.py ↔ ipu_cnn_keras.py tf_keras

Keras

GPU

```
as tf
ras.layers import *

(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_remainde

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(32, (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)
```

IPU

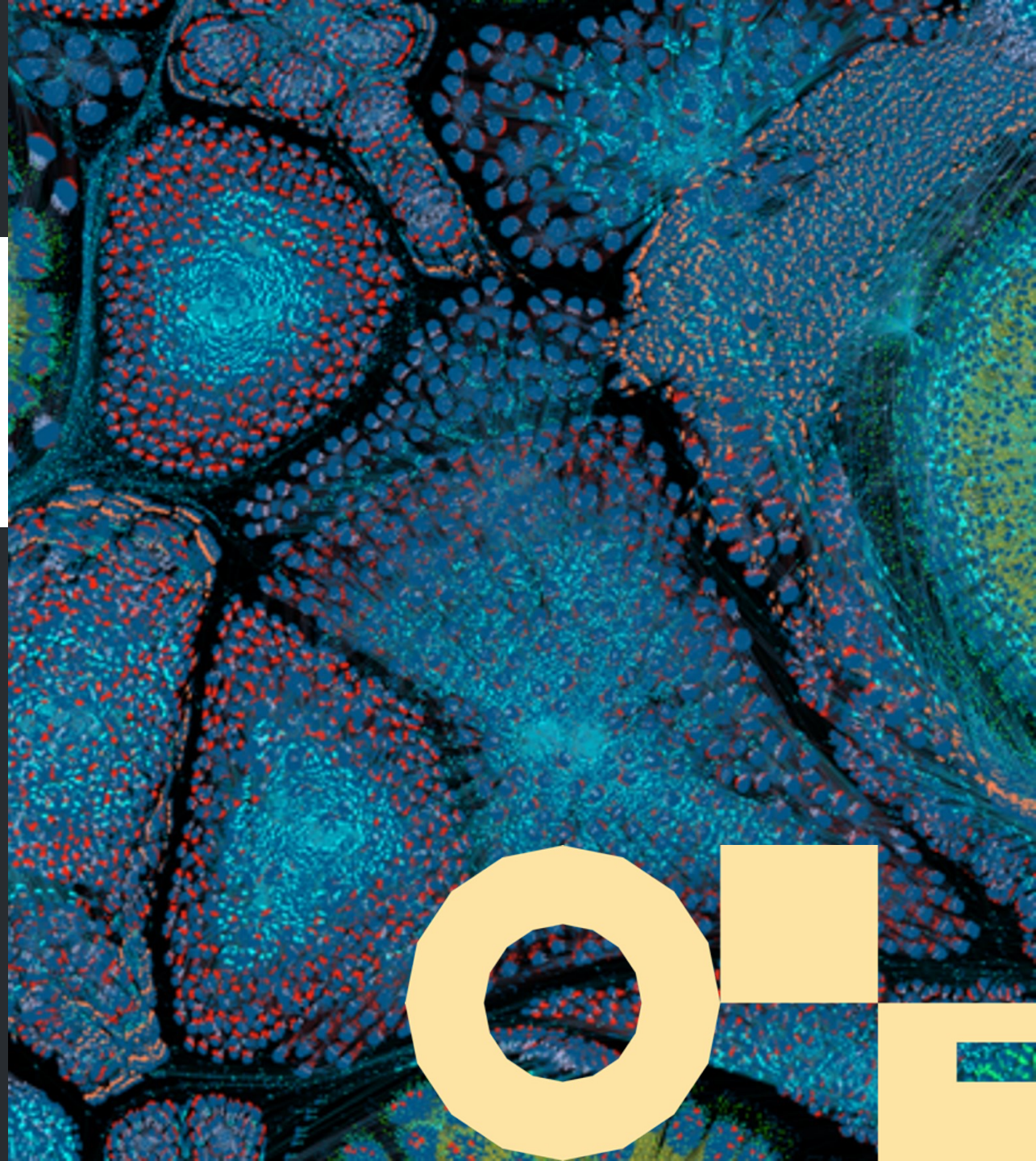
```
import tensorflow as tf
from tensorflow.keras.layers import *
+ from tensorflow.python import ipu

+ 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_rema

    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(32, (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

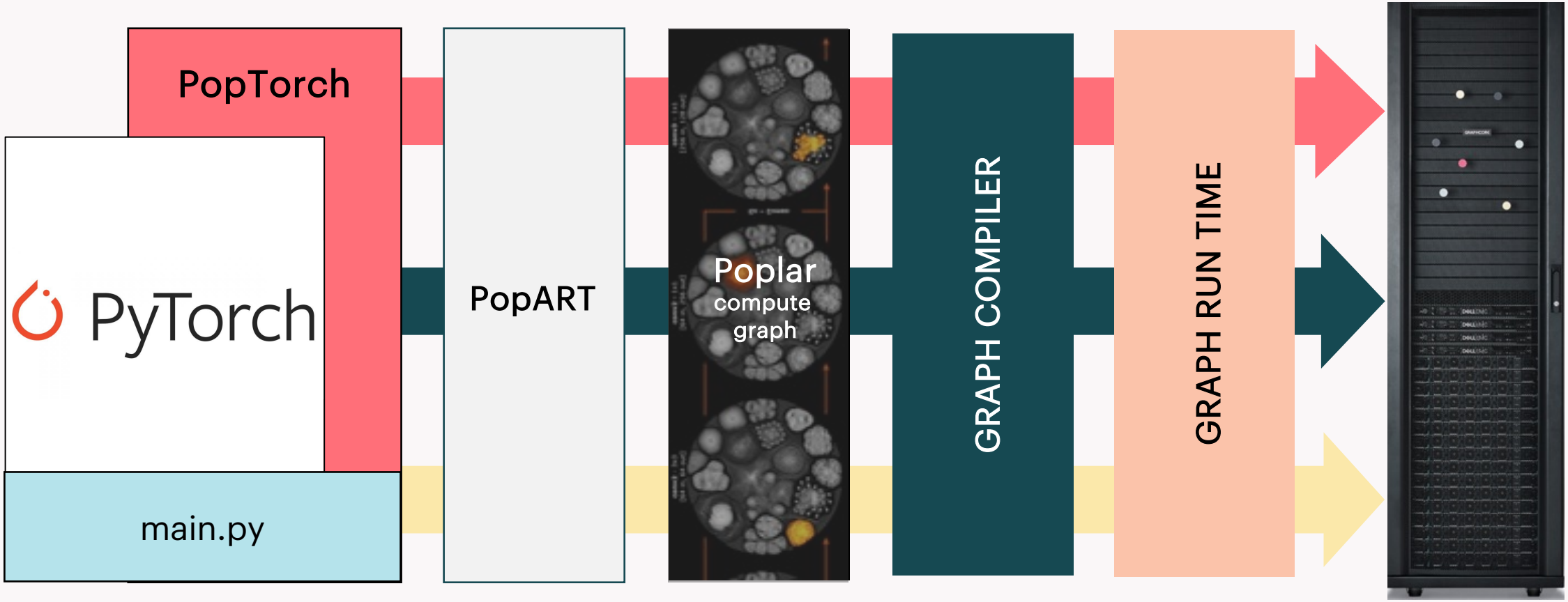
github.com/graphcore/examples/tree/master/tutorials/tutorials/tensorflow2

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)))
```

POPTORCH TUTORIALS

github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch

UNDER THE HOOD: BSP

BULK SYNCHRONOUS PARALLEL (BSP)

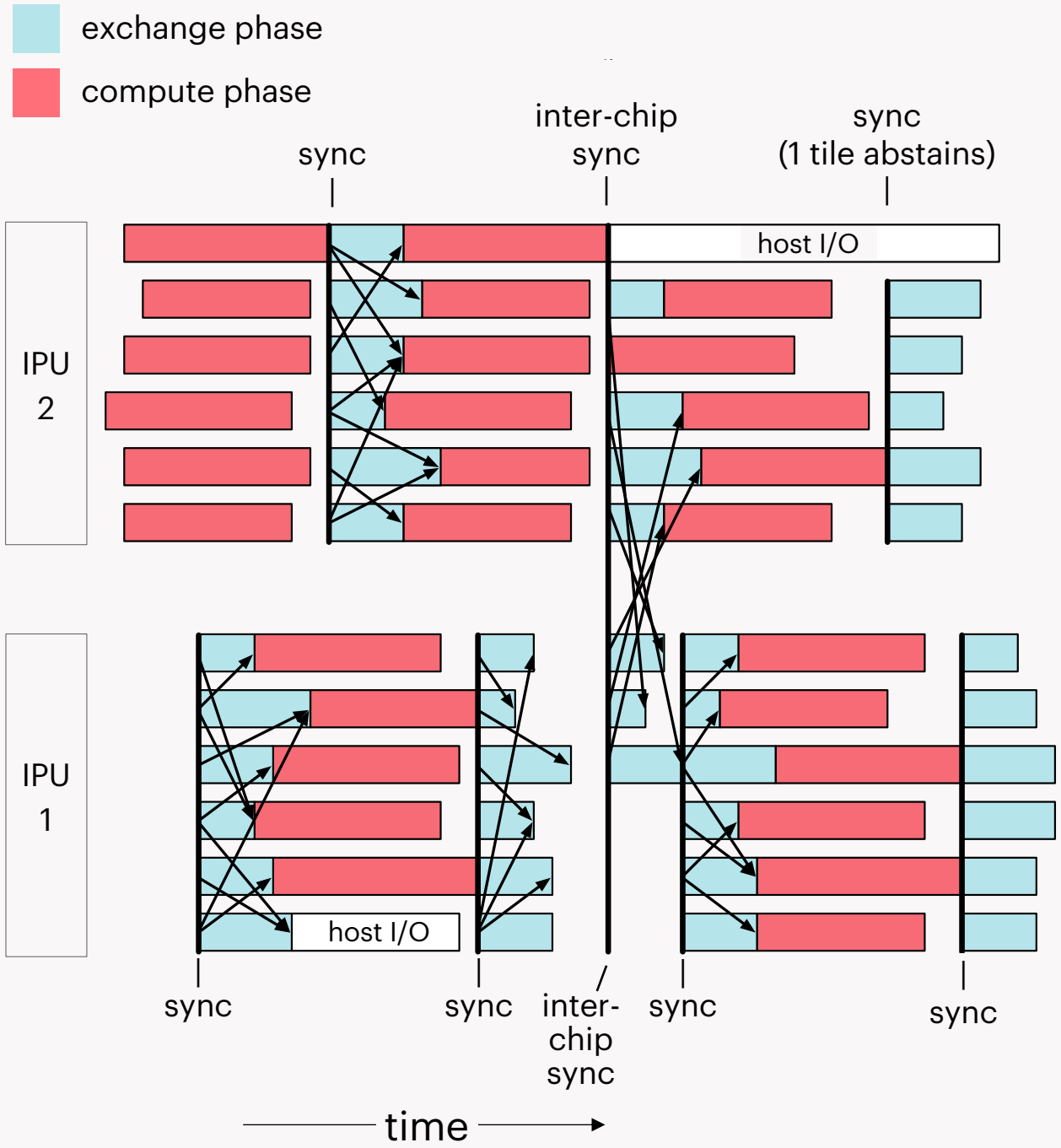
BSP software bridging model – massively parallel computing with no concurrency hazards

3 phases: compute, sync, exchange

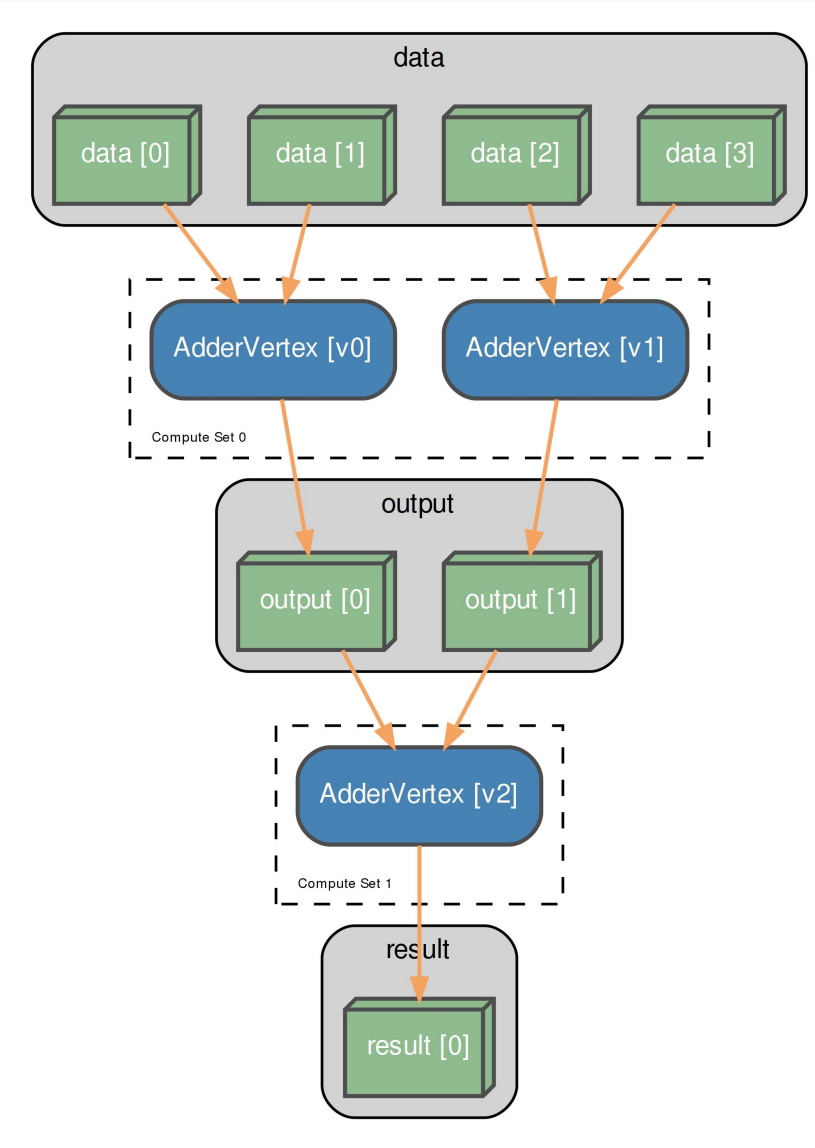
Easy to program – no live-locks or dead-locks

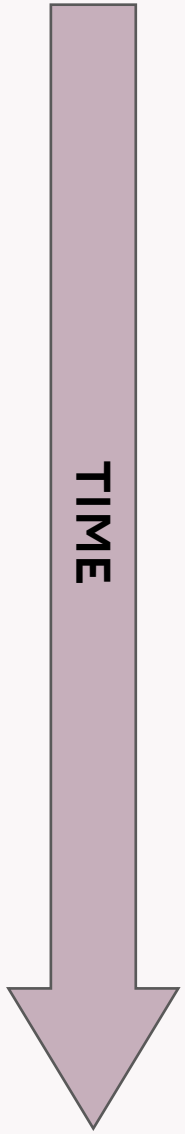
Widely-used in parallel computing – Google, FB, ...

First use of BSP inside a parallel processor

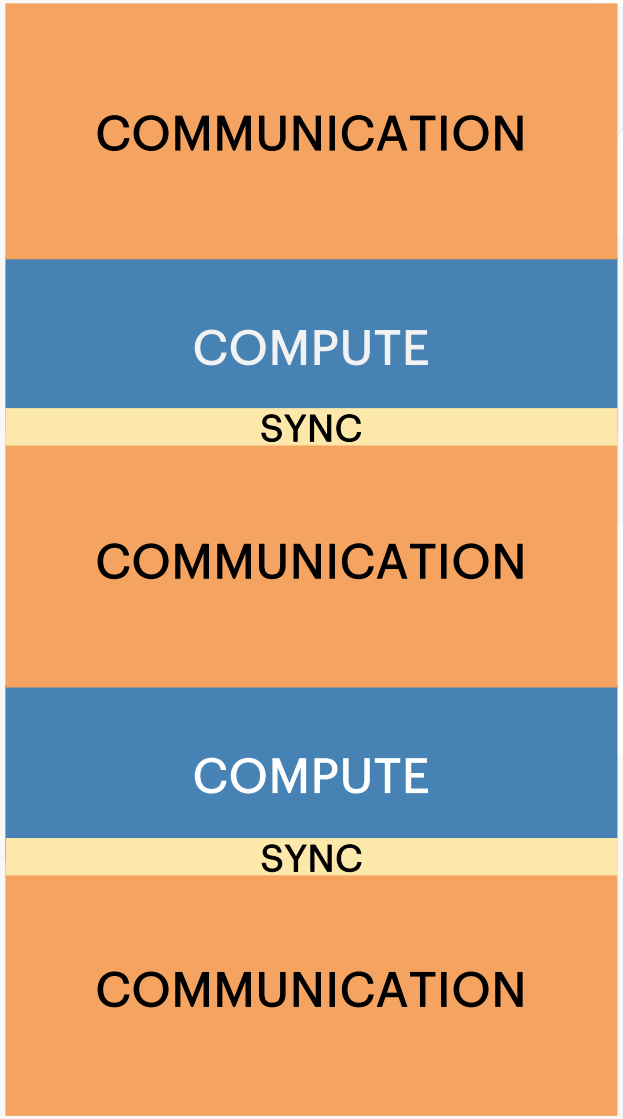


COMPUTATIONAL GRAPH





GRAPH EXECUTION MODEL



POPLAR FRAMEWORK

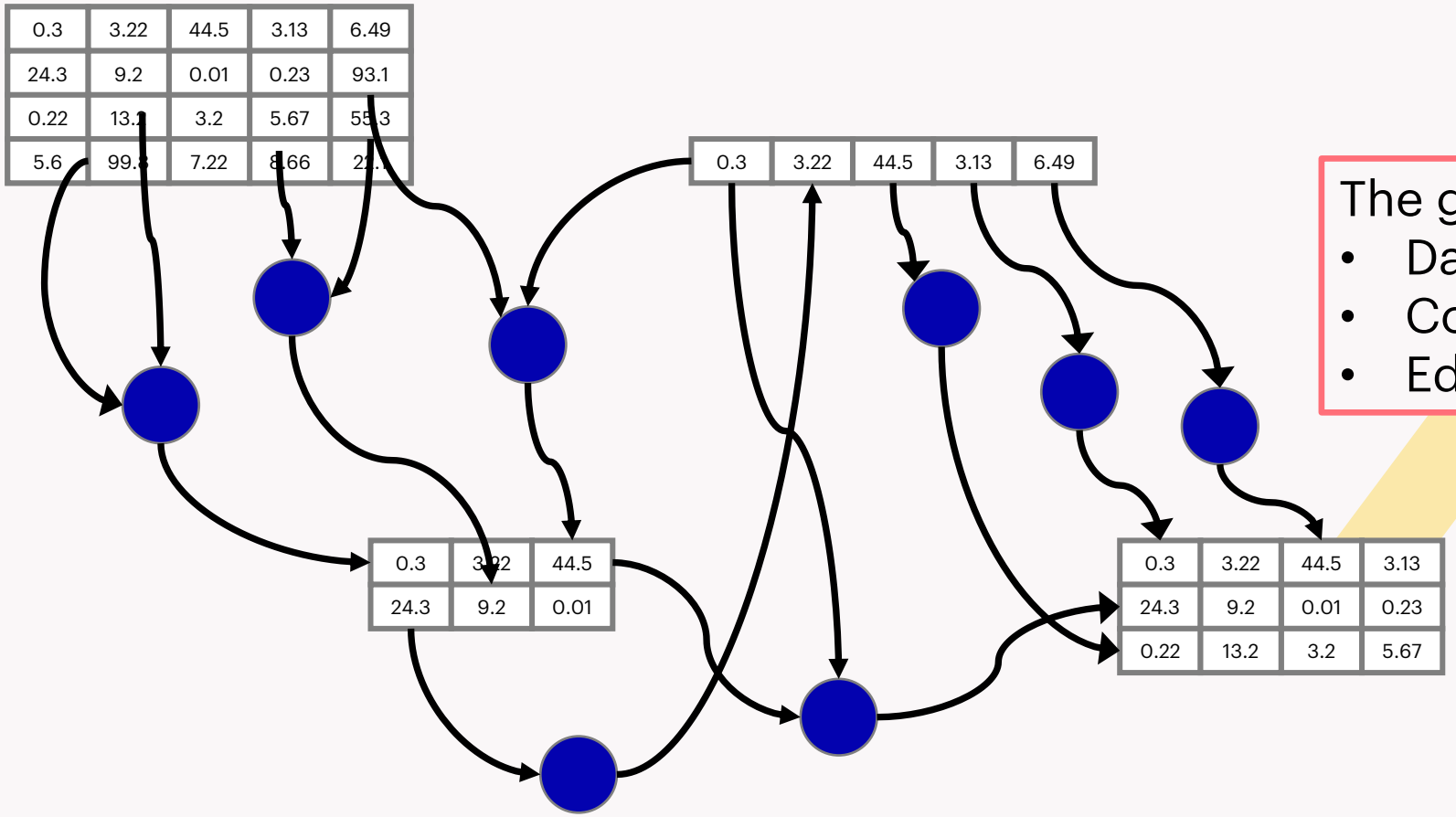
WHAT IS POPLAR?

- 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

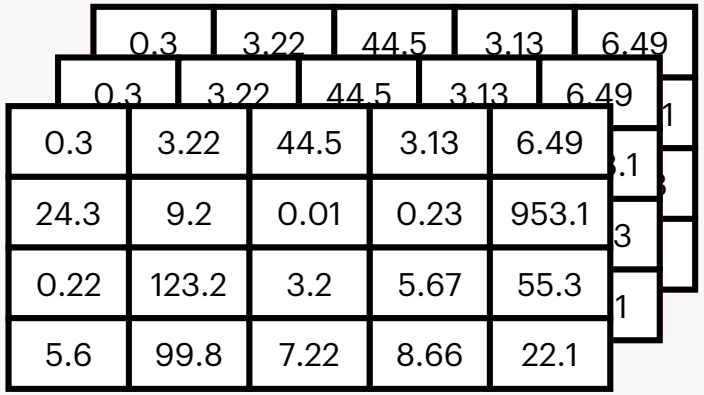
THE POPLAR GRAPH



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

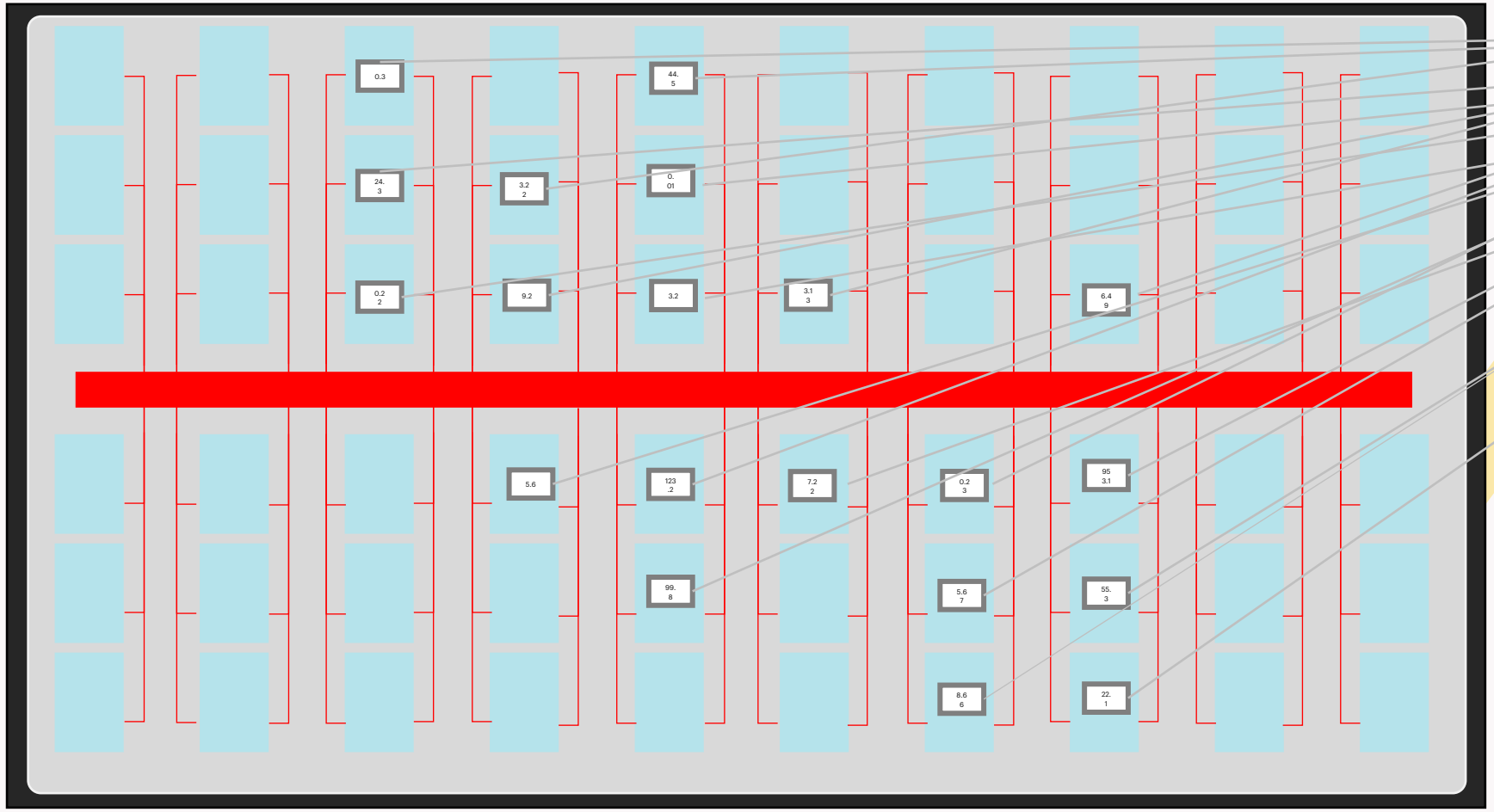
1-d tensor (5)

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.

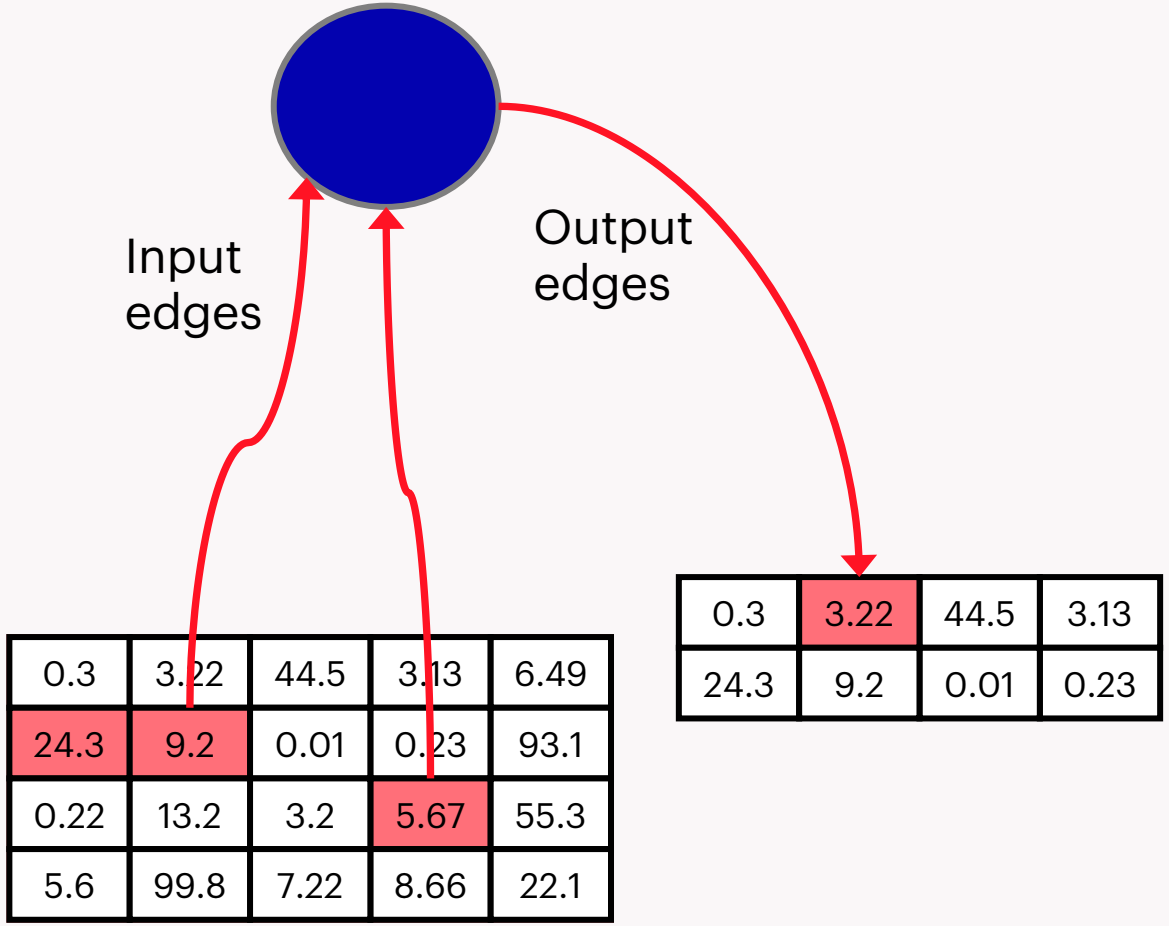
VARIABLES



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

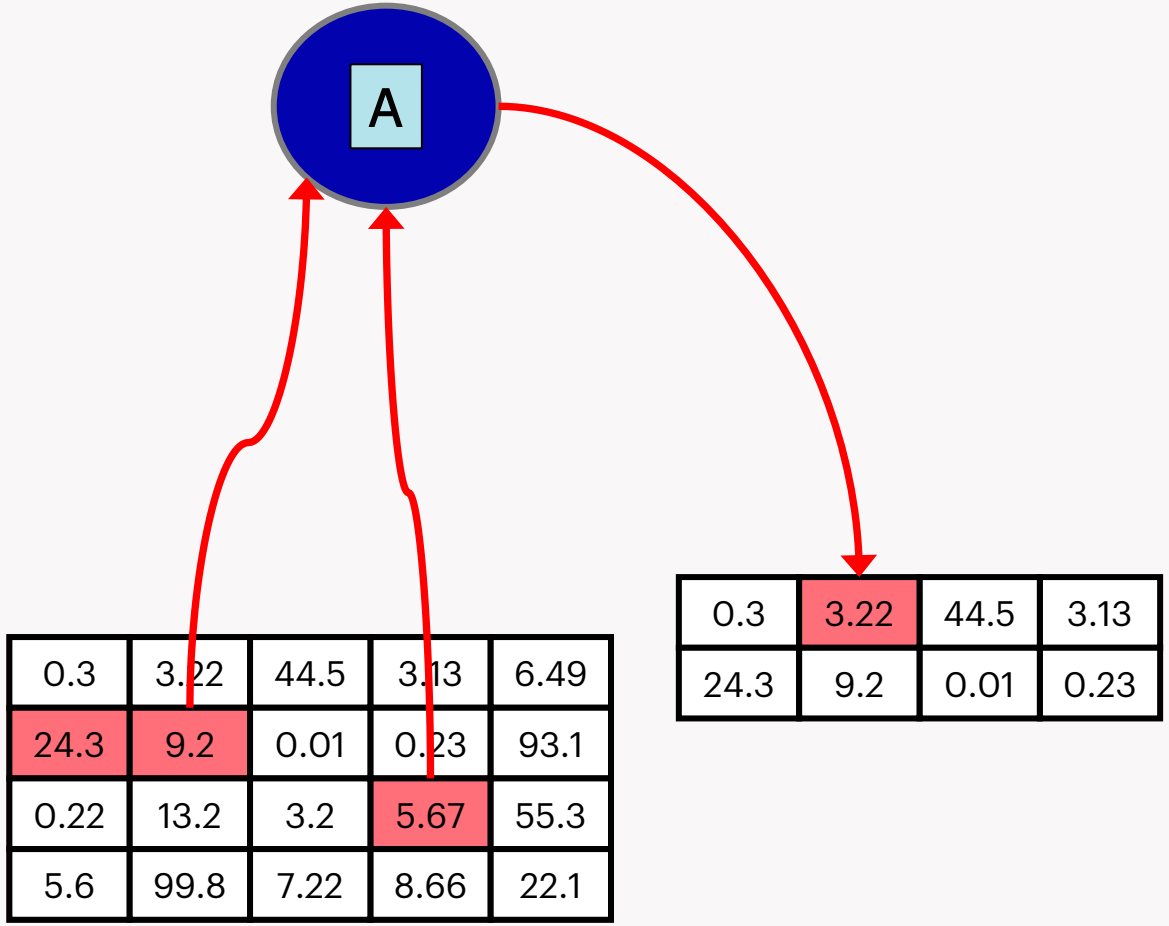
VERTICES



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

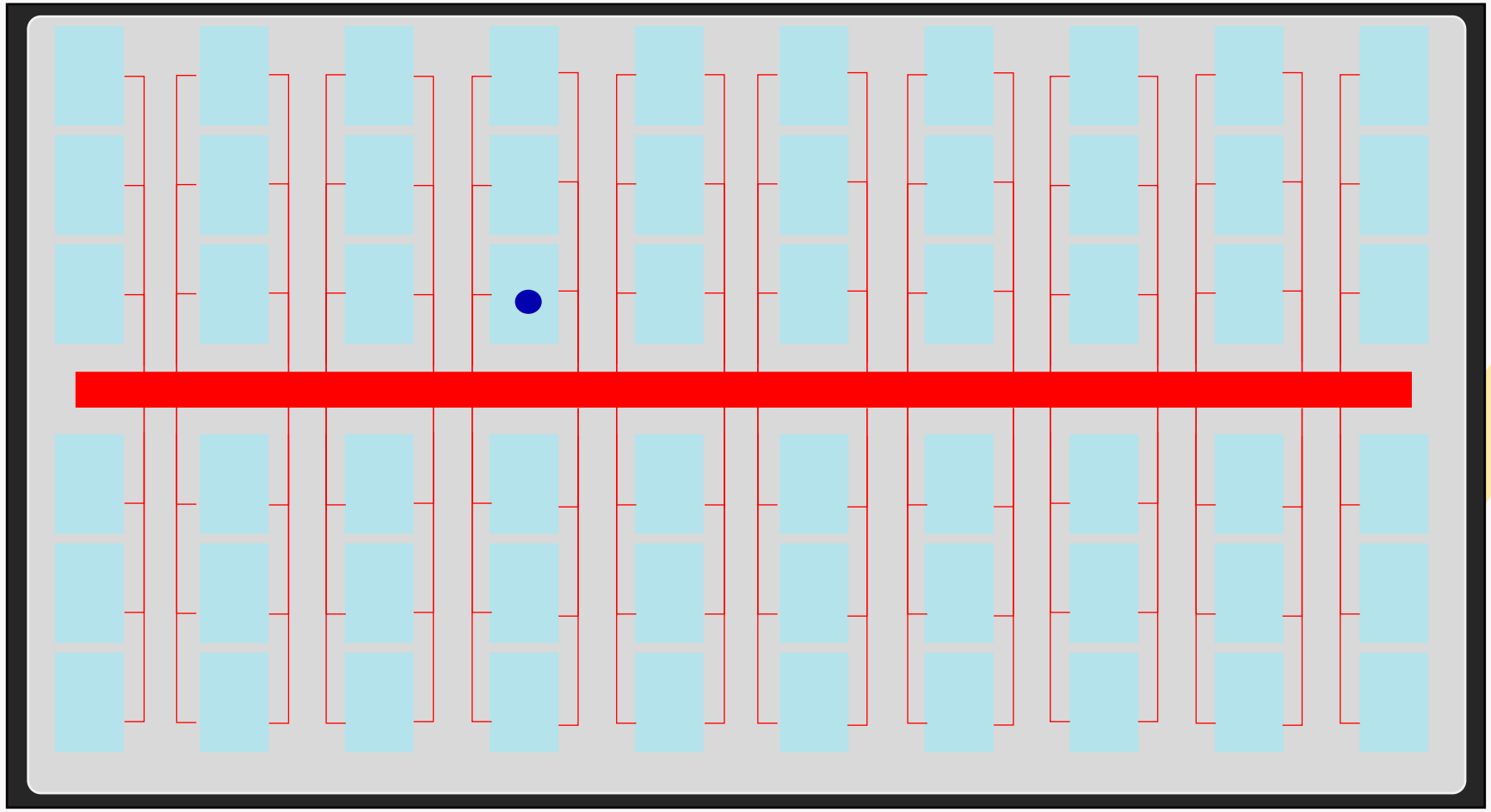


Codelet A

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

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

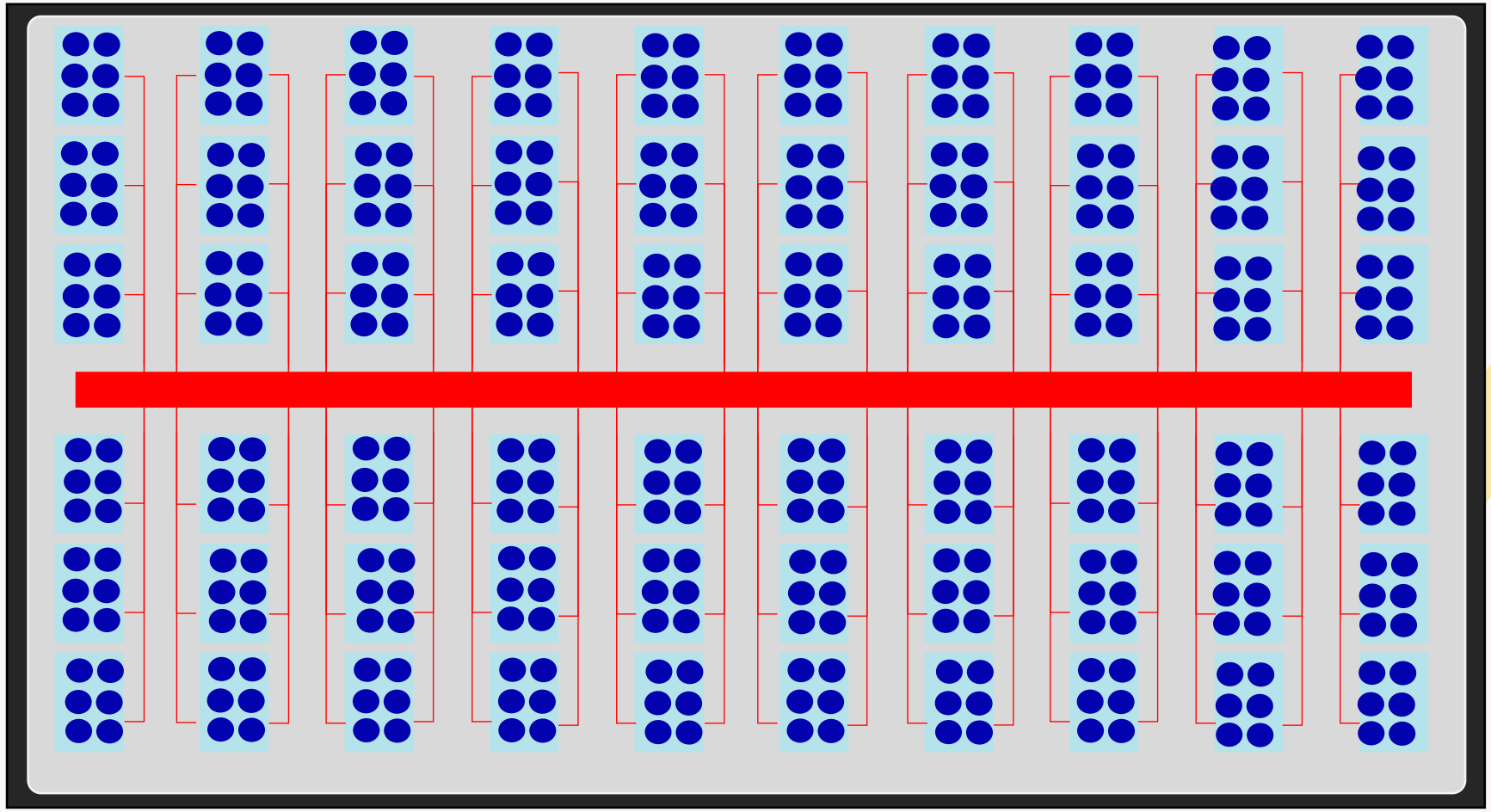
Each vertex is associated with a **codelet.**

VERTICES



A vertex runs on a single tile

VERTICES

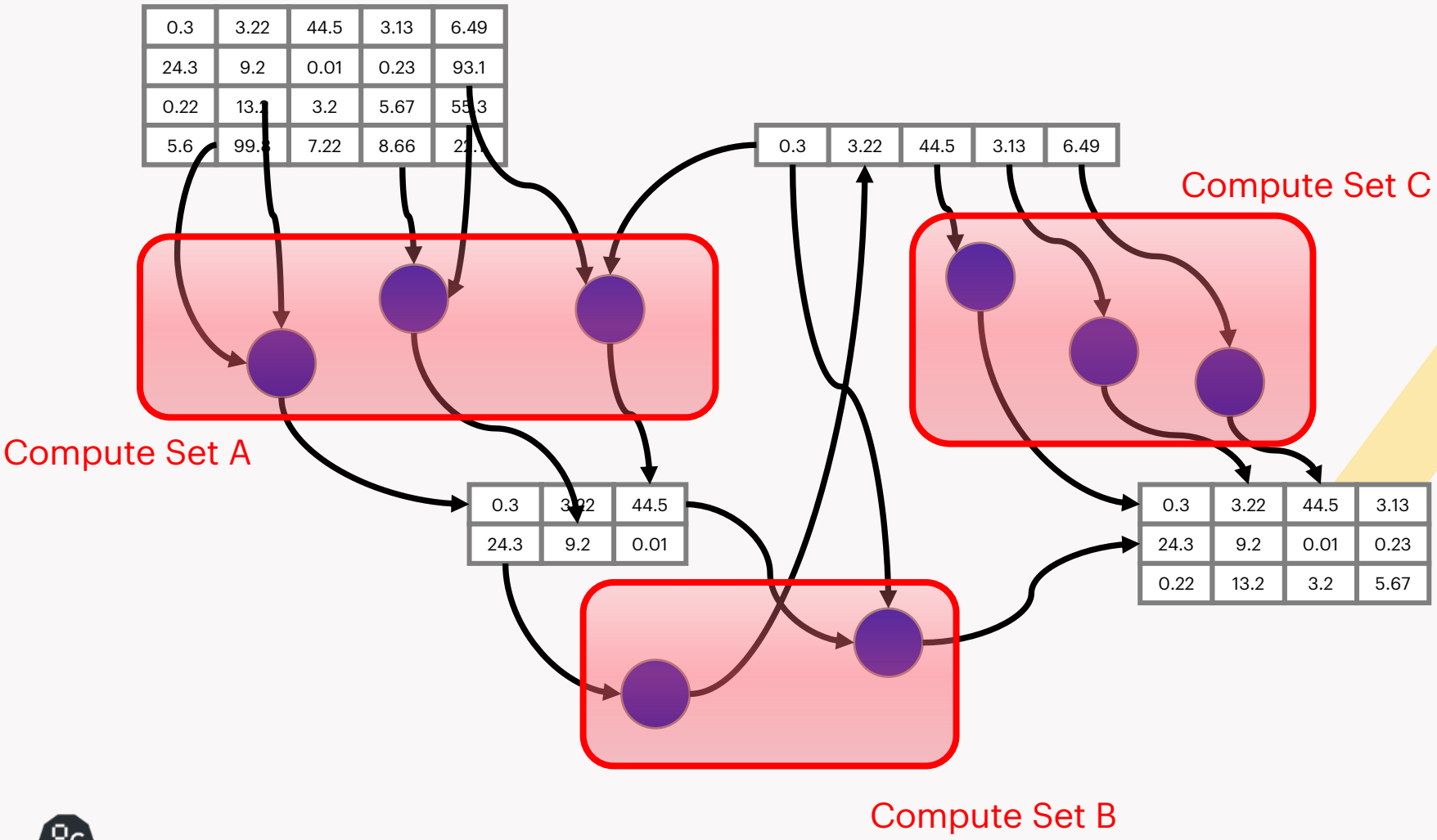


Many vertices are needed to fully utilize the device

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

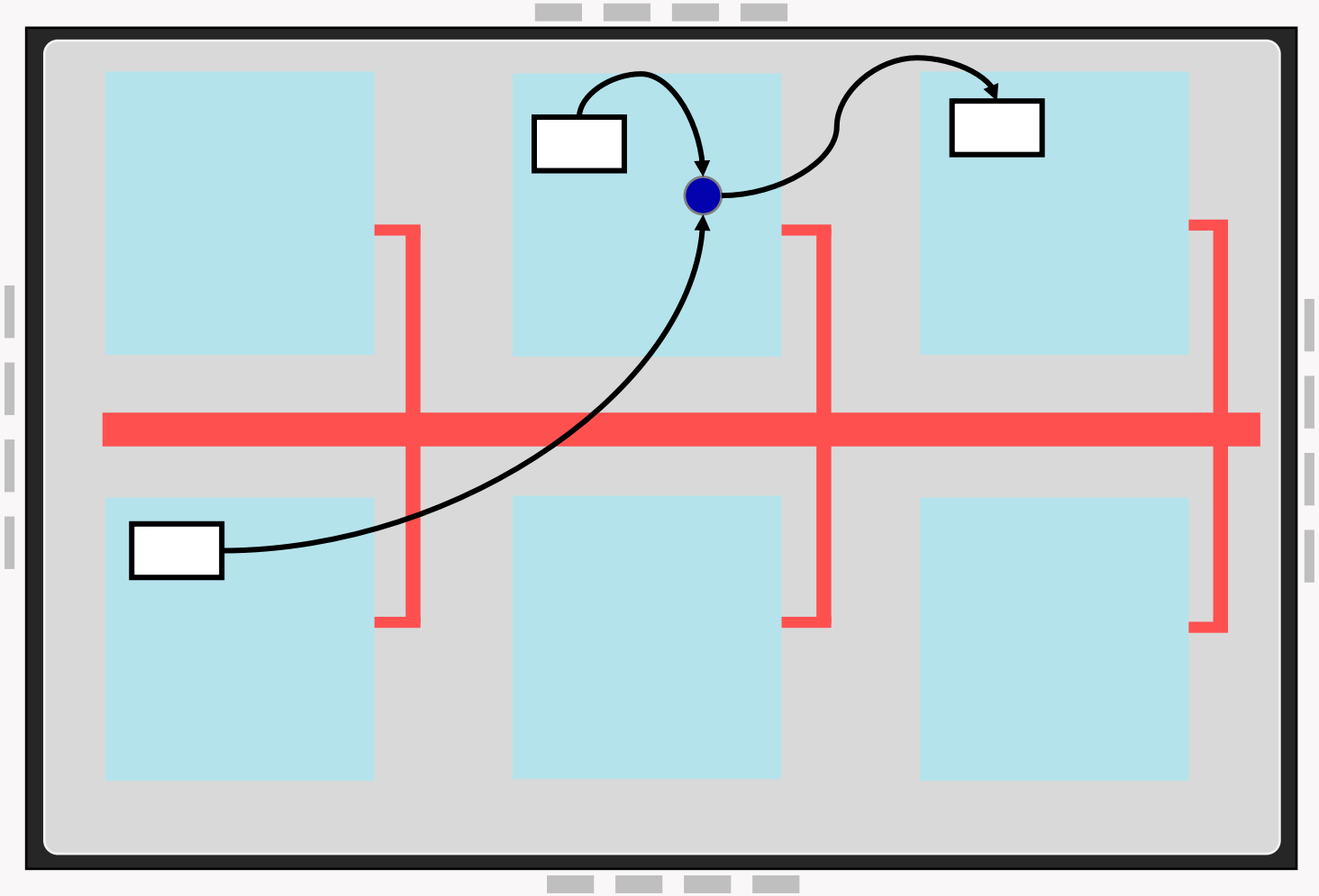
Exchange inputs

Compute

Sync

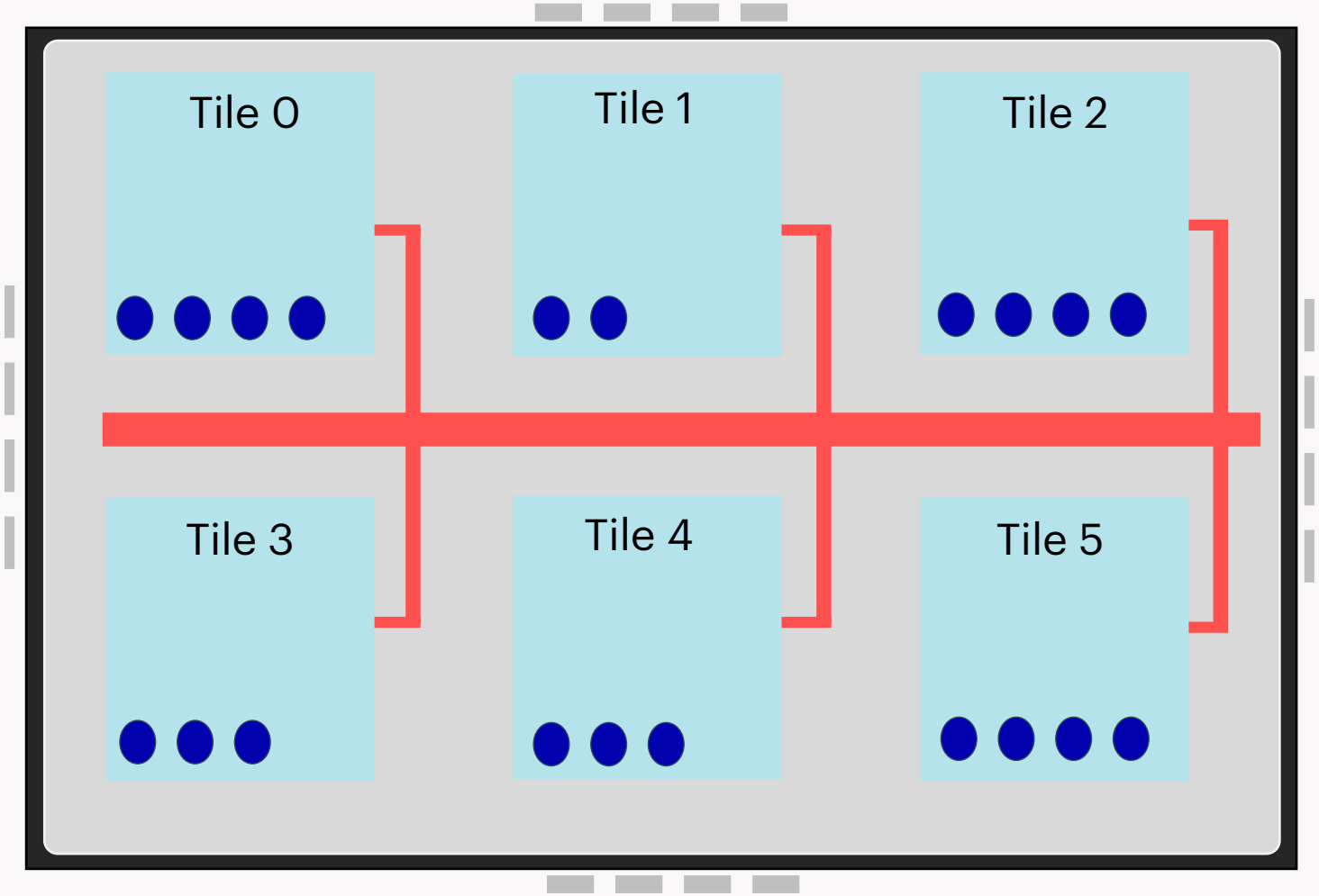
Exchange outputs

COMPUTE SET EXECUTION



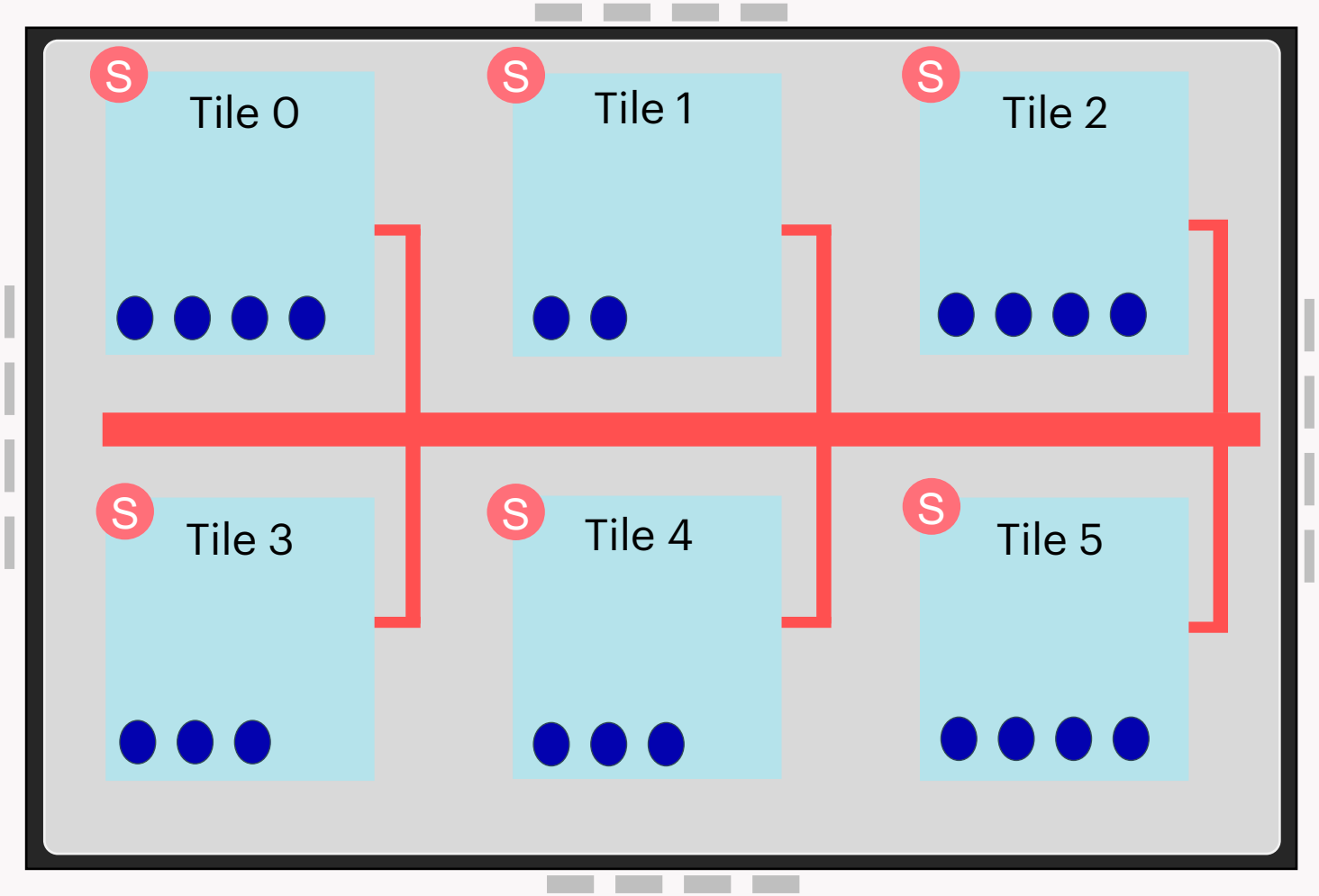
Exchange is required when a vertex in a compute set needs to read or write data which is stored on another tile's memory.

COMPUTE SET EXECUTION



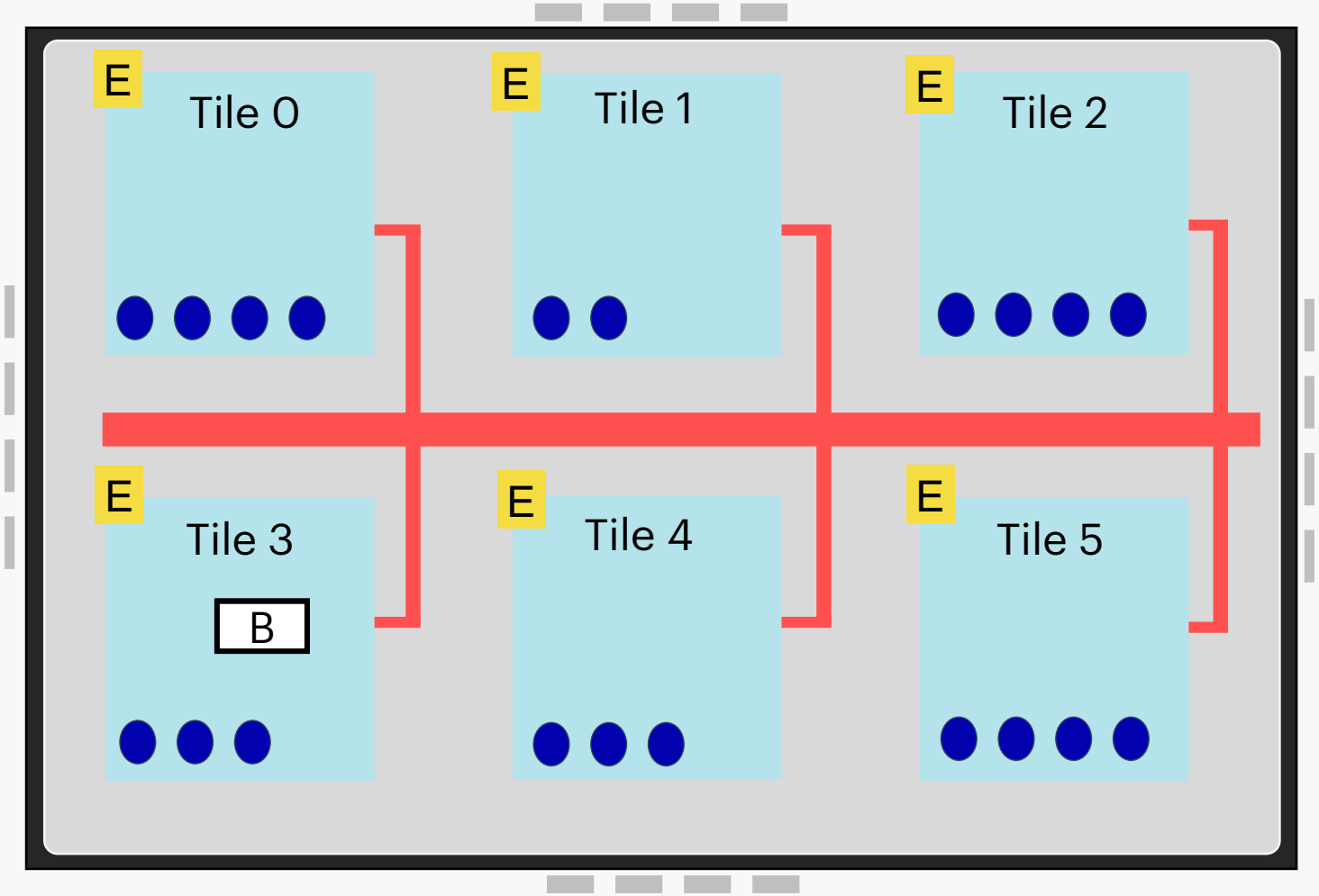
For each compute set, each tile will have a number of vertices to execute.

COMPUTE SET EXECUTION



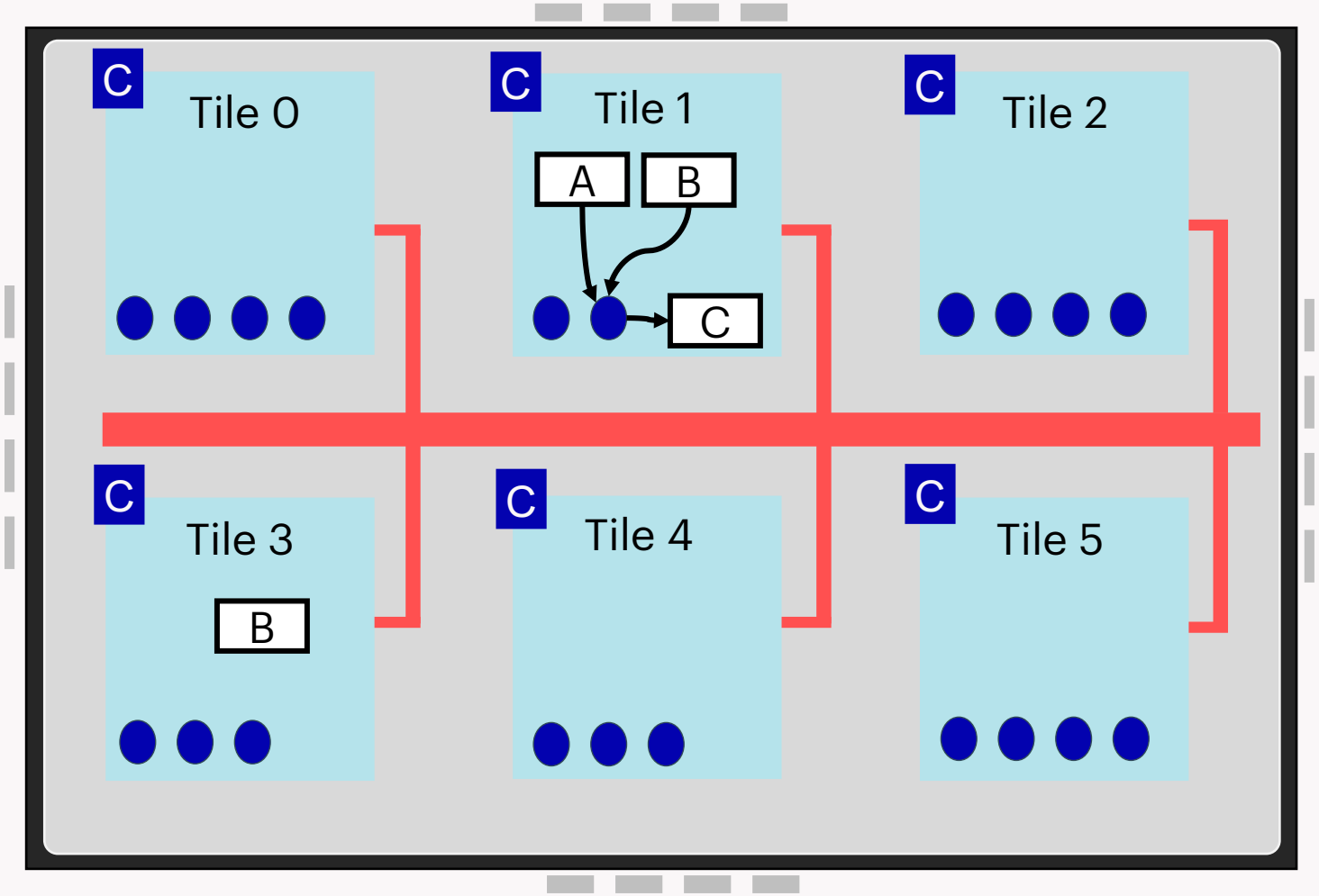
All tiles start by **syncing**.

COMPUTE SET EXECUTION



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

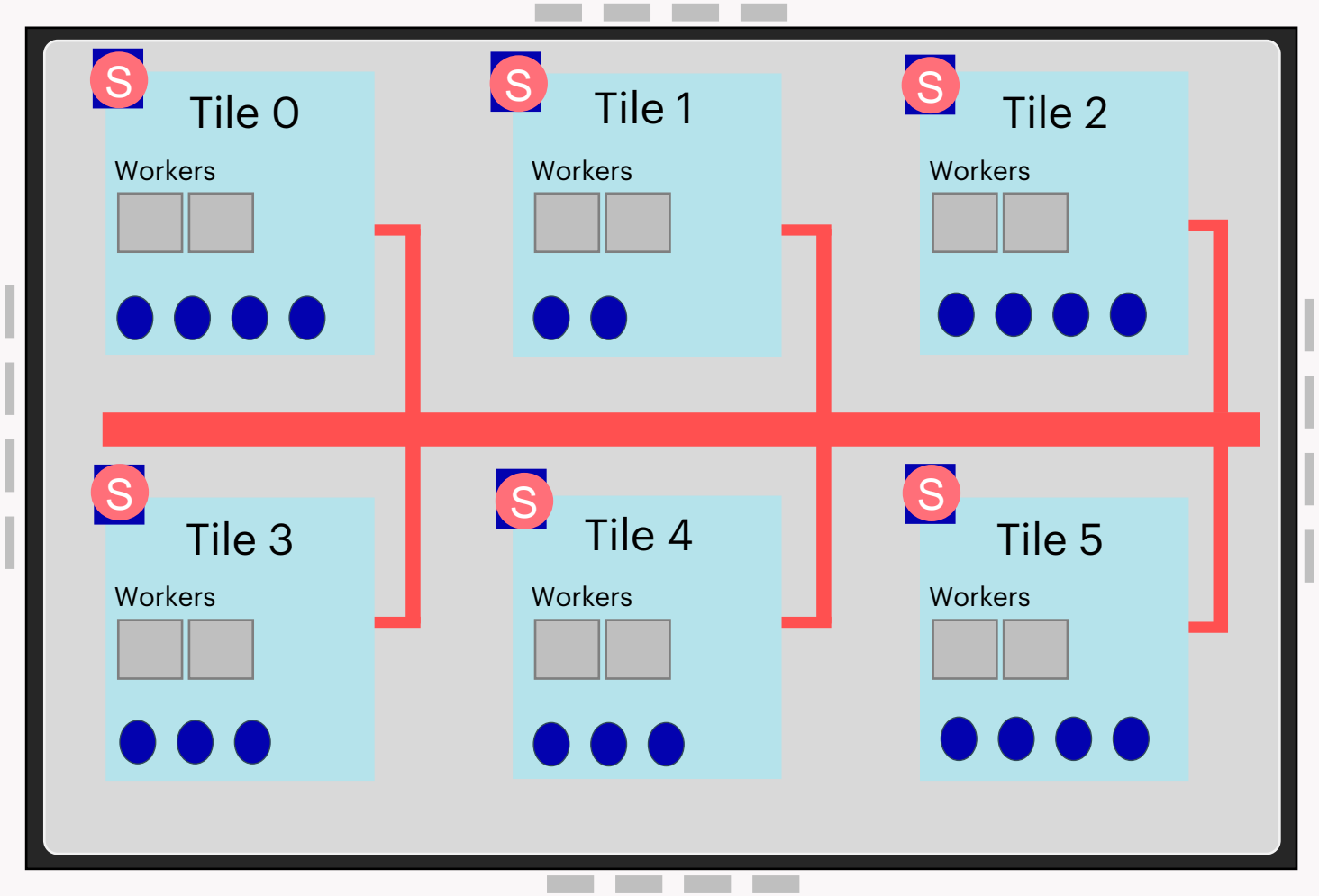
COMPUTE SET EXECUTION



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.

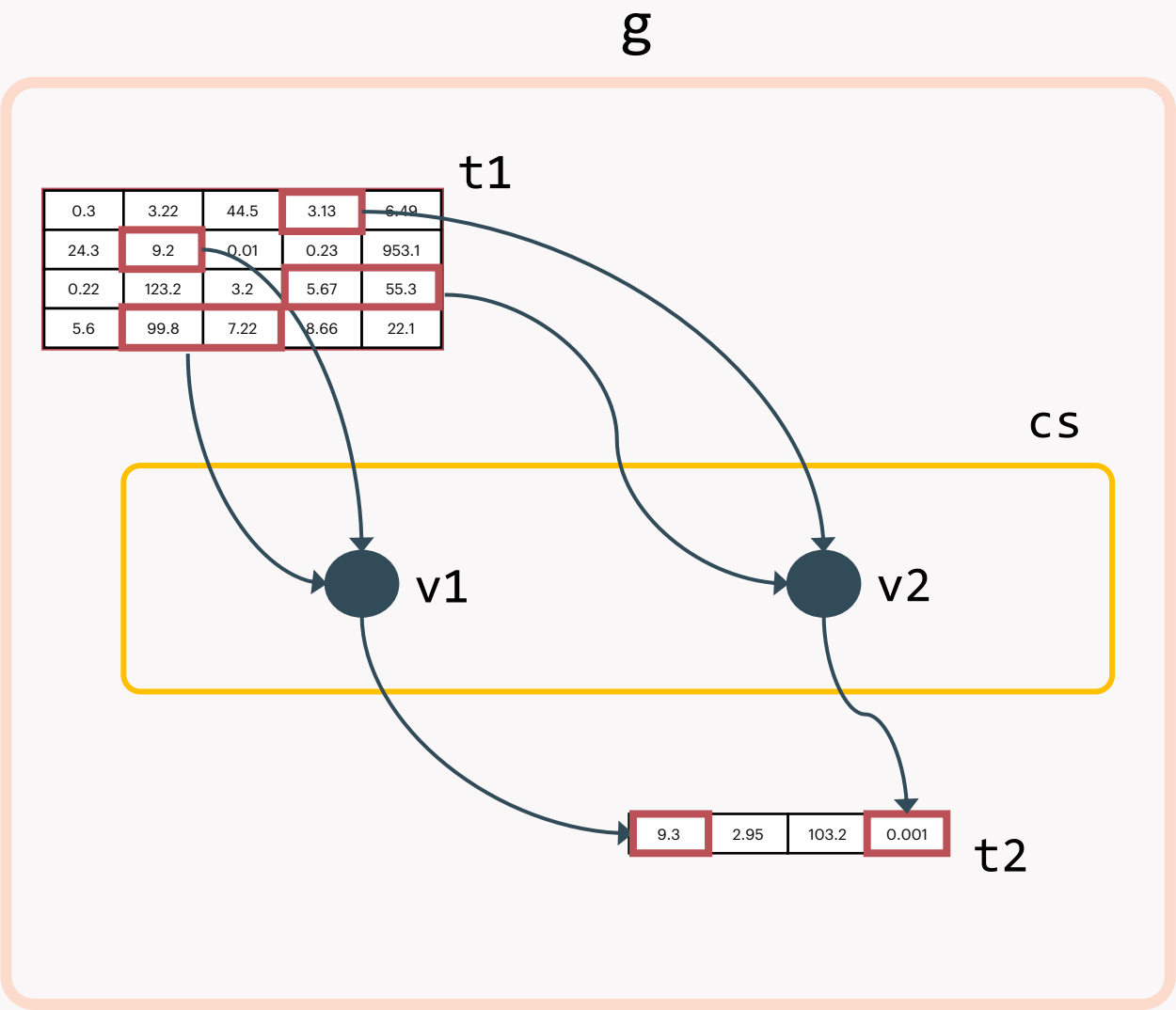
The fields of the vertex specify its inputs, outputs and internal data.

```
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.

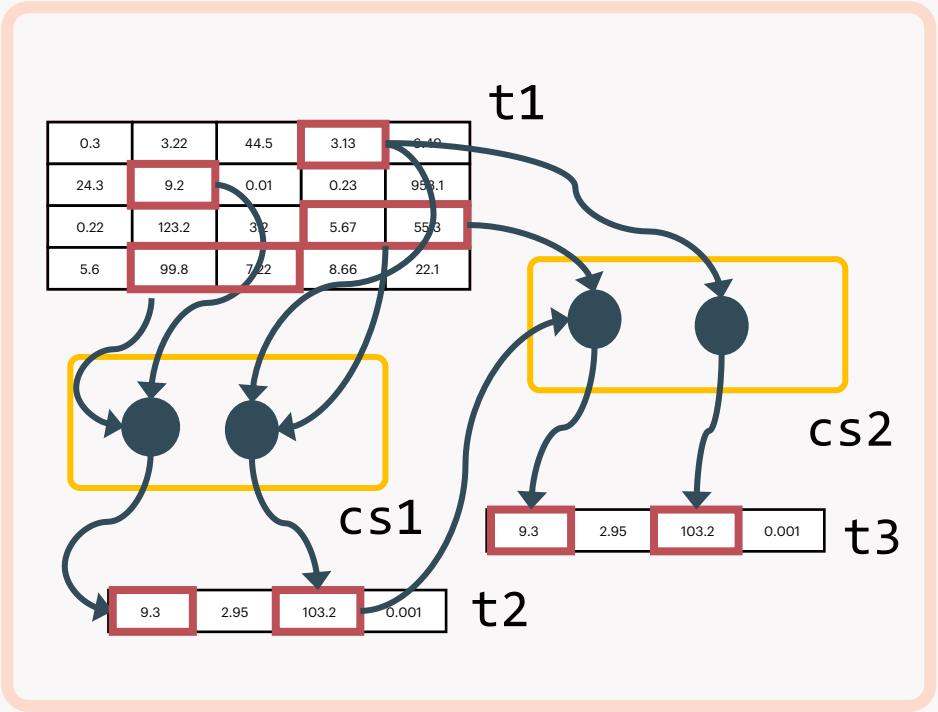
BUILDING THE COMPUTE GRAPH

```
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));
```

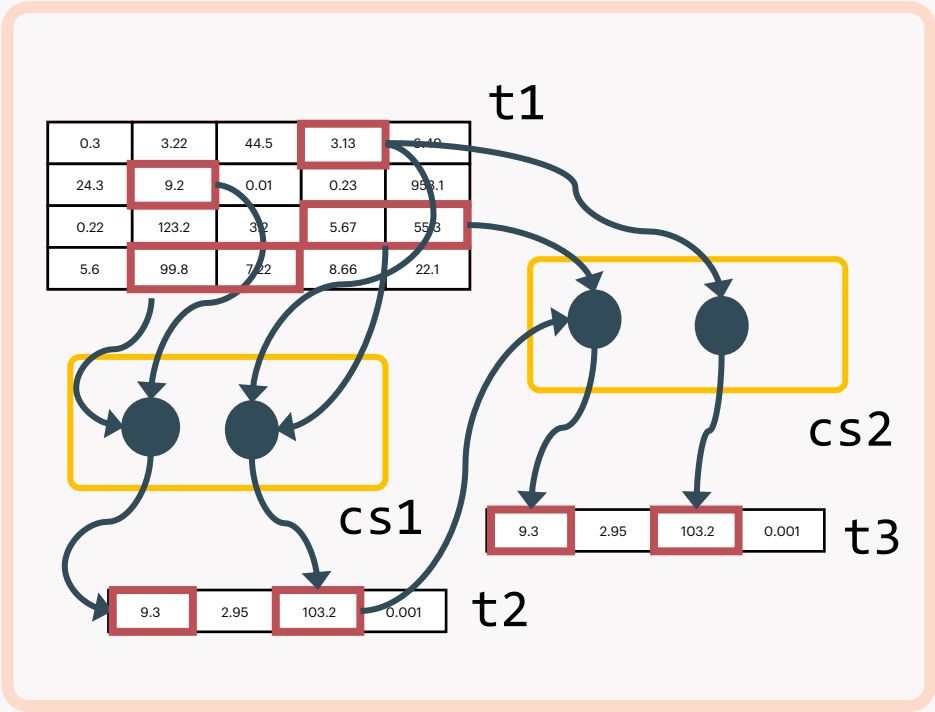


prog

```
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

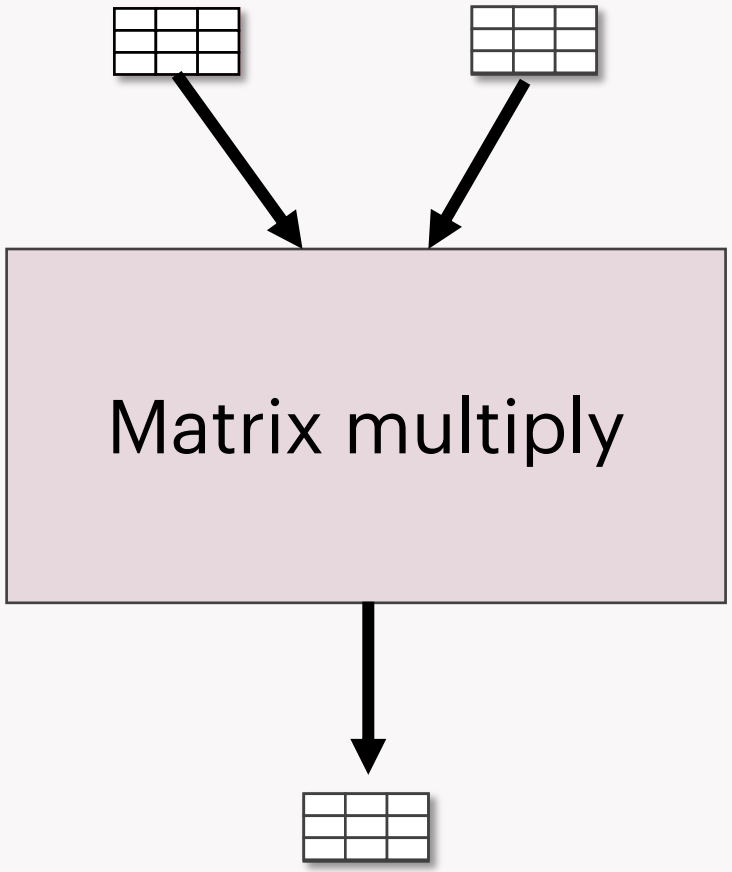
```
Execute(cs1);
Execute(cs2);
```

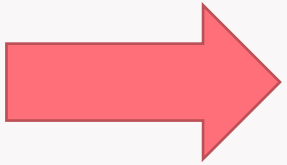
eng

SUMMARY

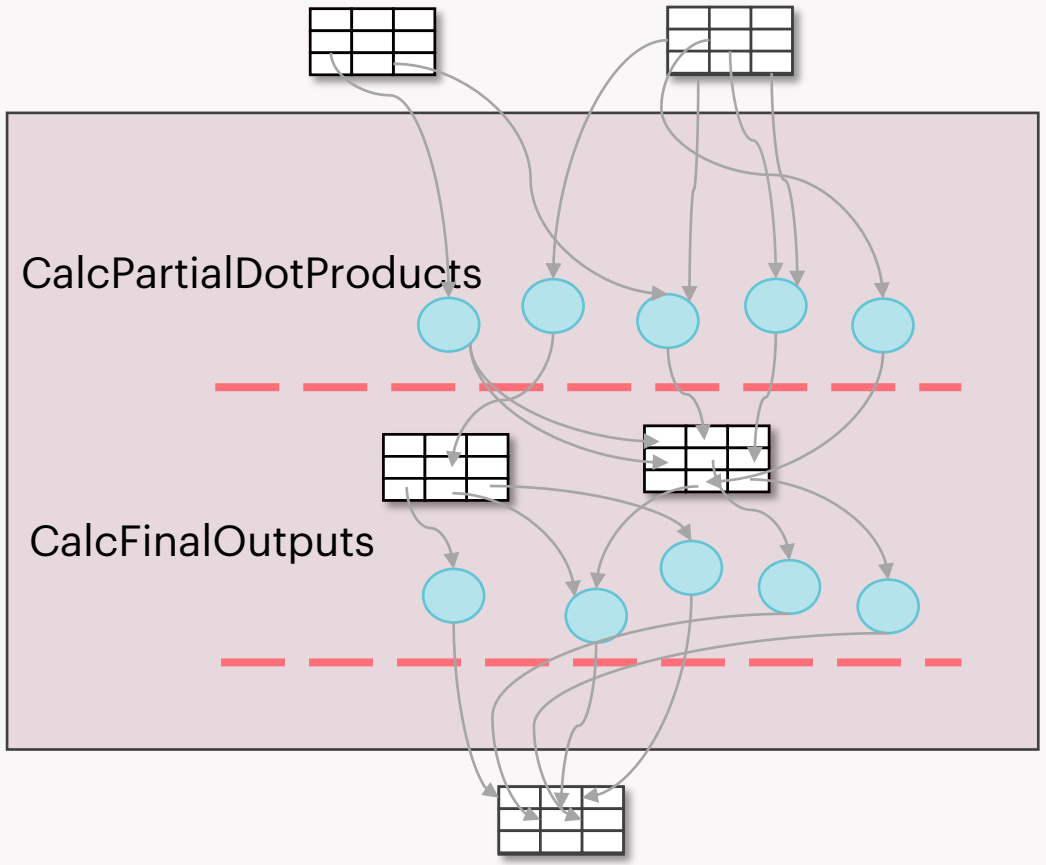
- Poplar lets you define your own operations by writing codelets
- Poplar generates “glue code” required to synchronize / exchange data
- Frees you to concentrate on parallel algorithm design

LIBRARIES = MODULAR GRAPH BUILDING





Library
call



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

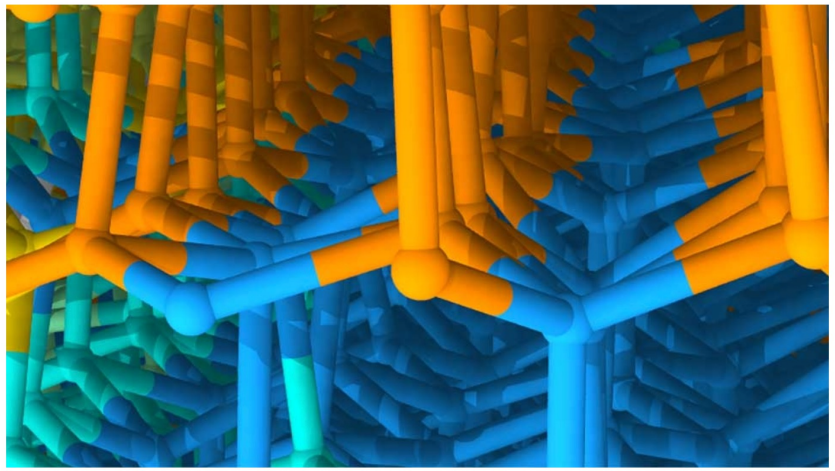
APPLY AND JOIN TODAY

Argonne Leadership Computing Facility

HOME / SCIENCE

Director's Discretionary Allocation Program

The ALCF Director's Discretionary program provides "start up" awards to researchers working to achieve computational readiness for for a major allocation award.



Molecular dynamics simulations based on machine learning help scientists learn about the movement of the boundary between ice grains (yellow/green/cyan) and the stacking disorder that occurs when hexagonal (orange) and cubic (blue) pieces of ice freeze together. Image: Henry Chan and Subramanian Sankaranarayanan, Argonne National Laboratory

Apply at alcf.anl.gov/science/directors-discretionary-allocation-program

general

charlieb 6:05 AM

🎉 Pleased to share with you all some new work from the Graphcore research team! 🎉

Our paper *Unit Scaling* introduces a new method for low-precision number formats, making FP16
We've managed to train BERT in these formats for the first time without loss scaling.

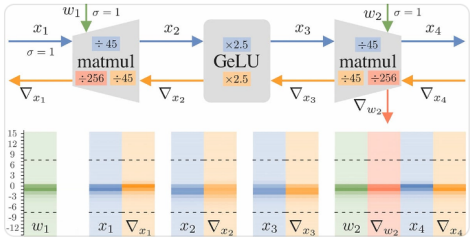
- You can find our blog post here: https://www.graphcore.ai/posts/simple-fp16-and-fp8-trainir
- Paperspace notebook (try it yourself!): https://ipu.dev/qXfm2a
- Arxiv paper: https://arxiv.org/abs/2303.11257

(& we were also featured on Davis Blalock's popular ML newsletter this week) (edited)

graphcore.ai

Simple FP16 and FP8 training with unit scaling

Unit Scaling is a new low-precision machine learning method able to train language models in FP16 and FP8 without loss scaling. (69 kB)



arXiv.org

Unit Scaling: Out-of-the-Box Low-Precision Training

We present unit scaling, a paradigm for designing deep learning models that simplifies the use of low-precision number formats. Training in FP16 or the recently proposed FP8 formats offers substantial efficiency gains, but can lack sufficient range for out-of-the-box training. Unit scaling addresses this by introducing a principled approach to model numerics: seeking unit variance of

Show more

🔥 8 🎉 8 💡 7 💥 1 👍 1 🙂 1

Join at graphcore.ai/join-community

Tuesday, 11 June

1:00 PM	→ 1:15 PM	Introduction	15m
1:15 PM	→ 1:45 PM	Graphcore BowPod64 Hardware	30m
1:45 PM	→ 2:30 PM	Software Stack: TensorFlow, PyTorch, and Poplar	45m
2:30 PM	→ 2:45 PM	Break	15m
2:45 PM	→ 3:15 PM	Porting applications with Poplar	30m
3:15 PM	→ 4:00 PM	How to use Bow Pod64@ ALCF	45m

Wednesday, 12 June

1:00 PM	→ 1:45 PM	Deep Dive on Graph neural networks and Large Language Models	45m
1:45 PM	→ 2:15 PM	Profiling with PopVision	30m
2:15 PM	→ 2:30 PM	Break	15m
2:30 PM	→ 3:15 PM	Hands-on session	45m
3:15 PM	→ 4:00 PM	Best Practices, Q&A	45m

THANK YOU

Alexander Tsyplikhin
alext@graphcore.ai