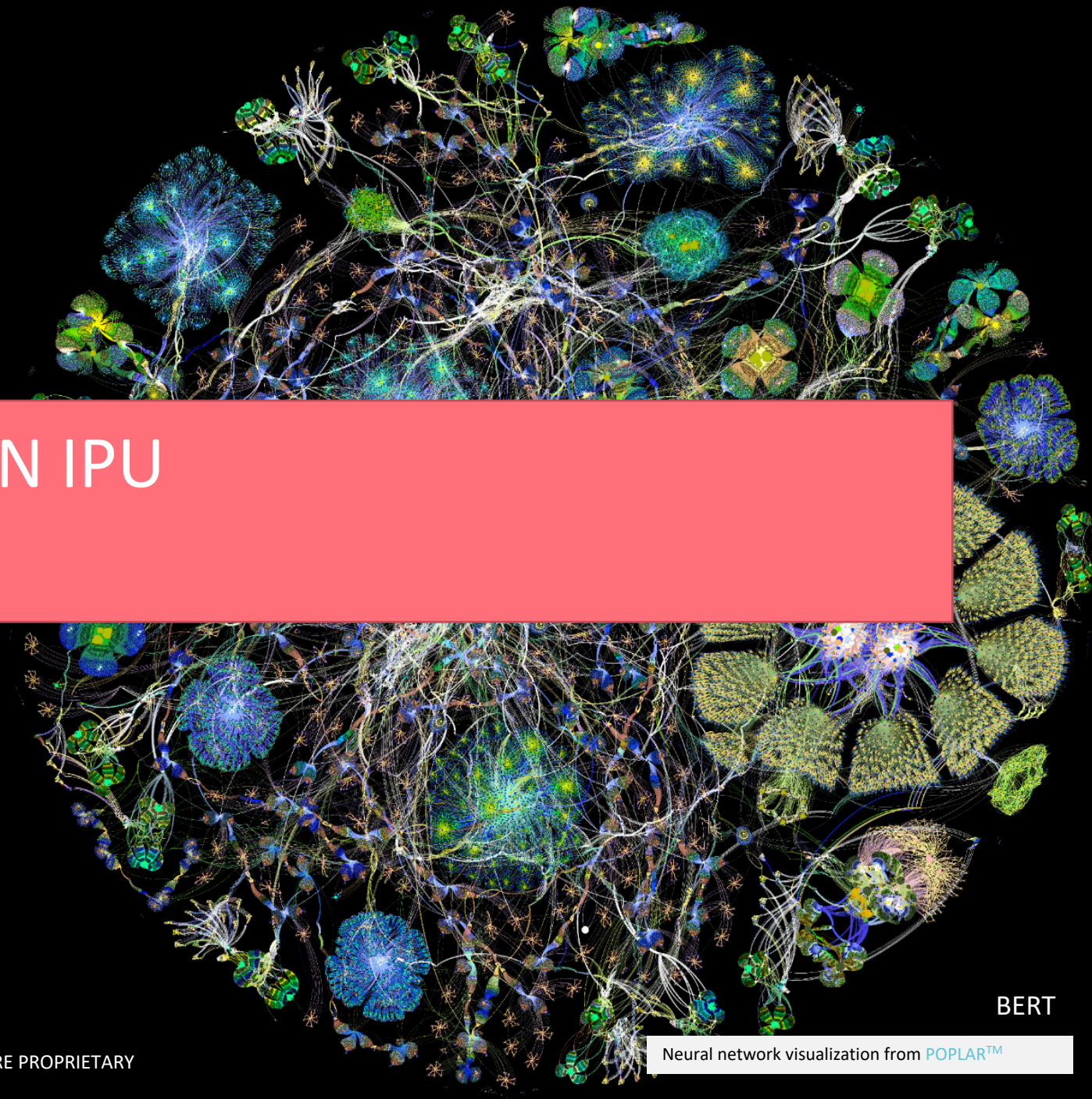
GRAPHCORE

POPART – SUPPORT ONNX ON IPU

Han Zhao 赵晗

Graphcore Tech Team Lead

hanzhao@graphcore.ai

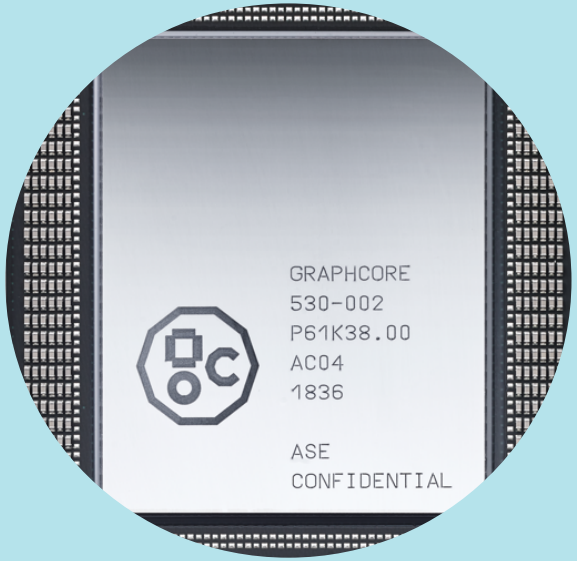


BERT

Neural network visualization from POPLAR™

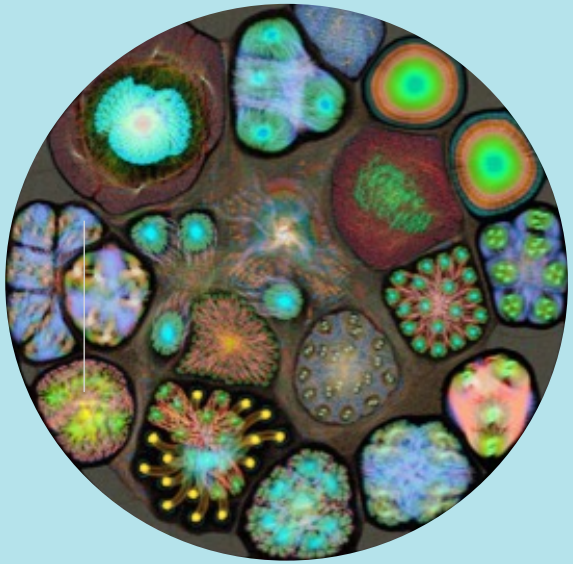
ABOUT GRAPHCORE

Hardware



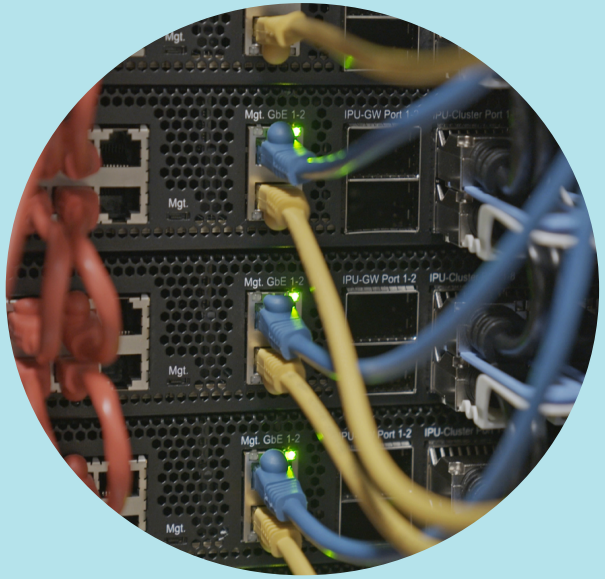
IPU processors
designed for AI

Software



Poplar® software stack &
development tools

Platforms



M2000 and Server
IPU-POD$_{64}$ scale-out

GC200 IPU PROCESSOR

IPU-Tiles™

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

IPU-Core™

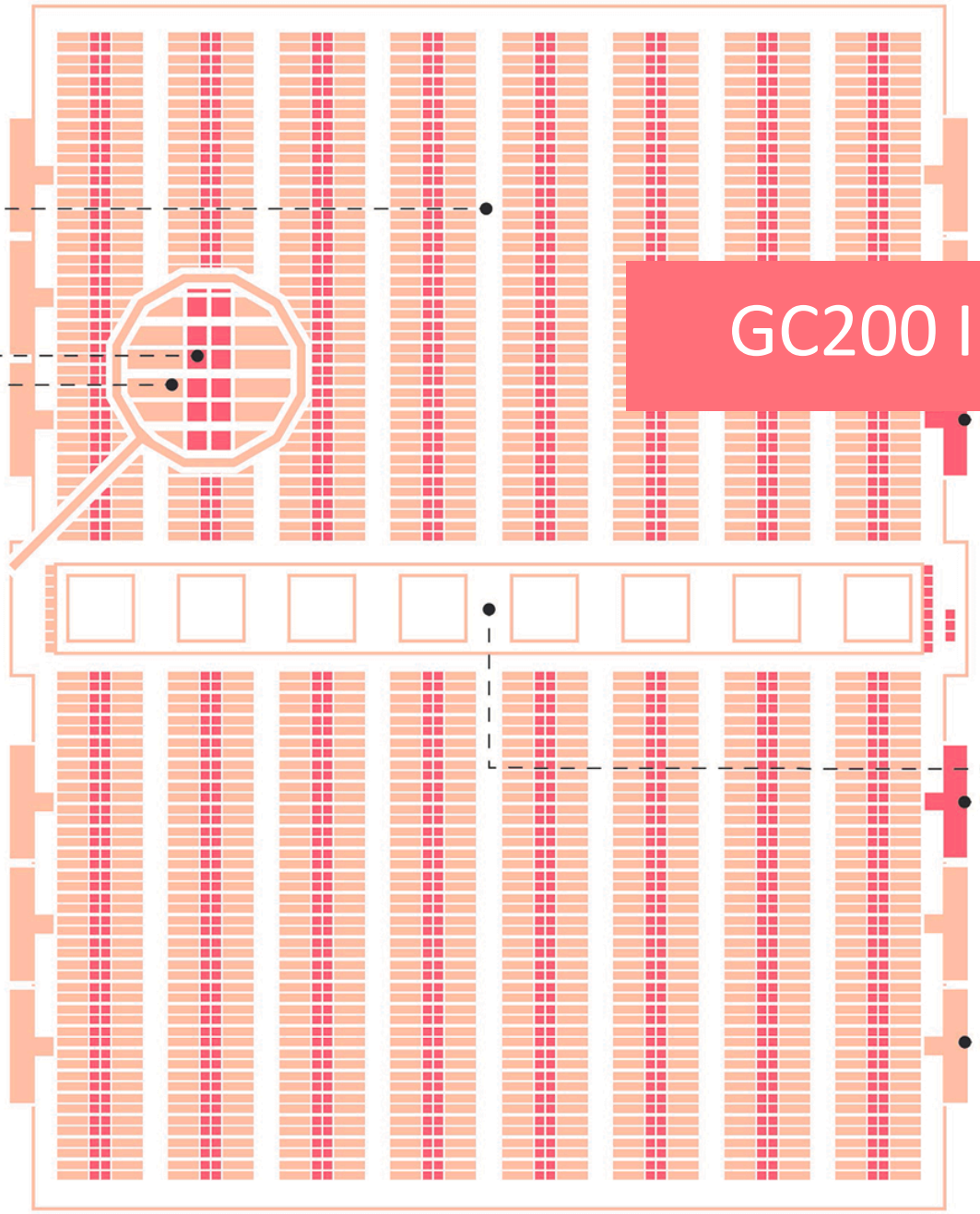
1472 independent IPU-Core™

8832 independent program threads executing in parallel

In-Processor-Memory™

900MB In-Processor-Memory™ per IPU

47.5TB/s memory bandwidth per IPU



IPU-Exchange™

8 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

IPU SCALEOUT: INNOVATIONS & DIFFERENTIATORS

DESIGNED FOR AI

MIMD ARCHITECTURE

ULTRA-FAST
IN-PROCESSOR MEMORY

TRAINING & INFERENCE

1U FLEXIBILITY &
SCALABILITY

HIGH CAPACITY
EXCHANGE MEMORY

IPU-FABRIC BUILT-IN
FOR SCALEOUT

HIGHEST PERFORMANCE
PLATFORM IN 5U FORMFACTOR

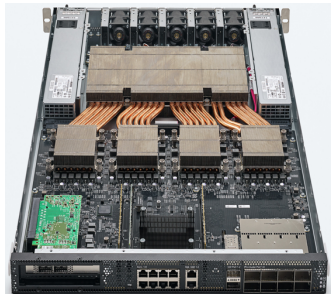
FULLY RECONFIGURABLE
FOR SCALEOUT

DISAGGREGATED HOST SERVER
FOR OPTIMAL TCO

BUILDING BLOCK FOR SCALEOUT

IPU-FABRIC
COMMUNICATION FOR
EFFICIENT
SCALEOUT
PERFORMANCE

x4

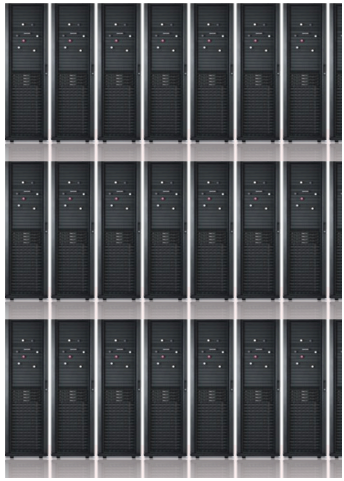


x4

IPU-POD16 Direct Attach

x4

x1024



GC200

IPU-M2000

4 IPUs
1 PetaFlop

IPU-POD$_{16}$

16 IPUs
4 PetaFlop
4x IPU-M2000

IPU-POD$_{64}$

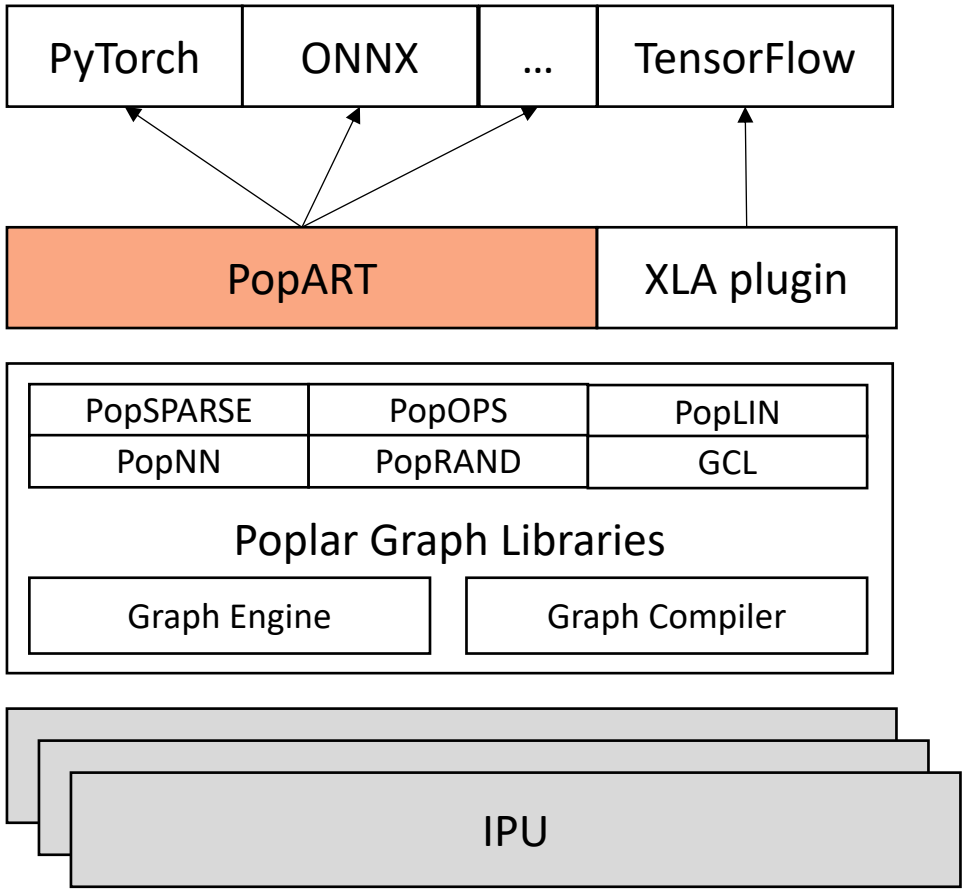
64 IPUs
16 PetaFlop
16x IPU-M2000

IPU-POD$_{64k}$

64k IPUs
16 ExaFlop
<7.4 PB

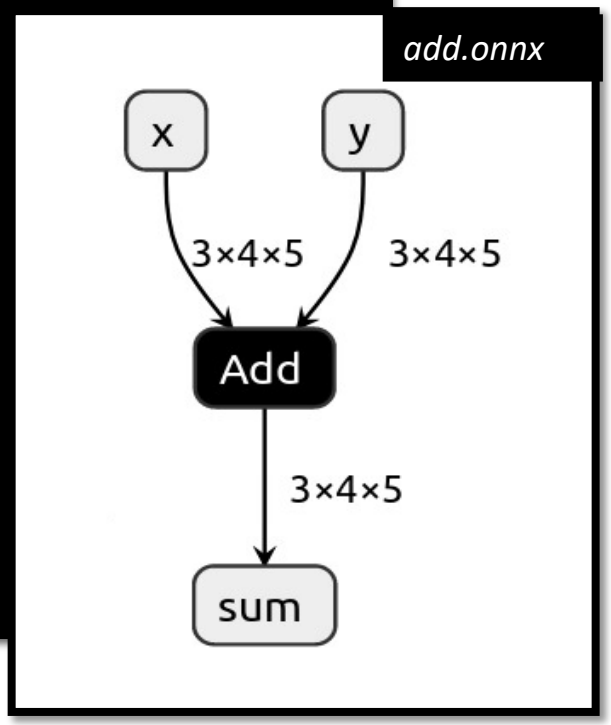
PopART

- PopART : Poplar Advanced Runtime
- PopART is a runtime which can execute ONNX models on IPU(s)
- Implements ONNX IR version 4 , Opset Versions 6 to 11
- Supports inference & training(*)
- Provides a C++ & Python API
- Provides a builder API to construct ONNX models programmatically

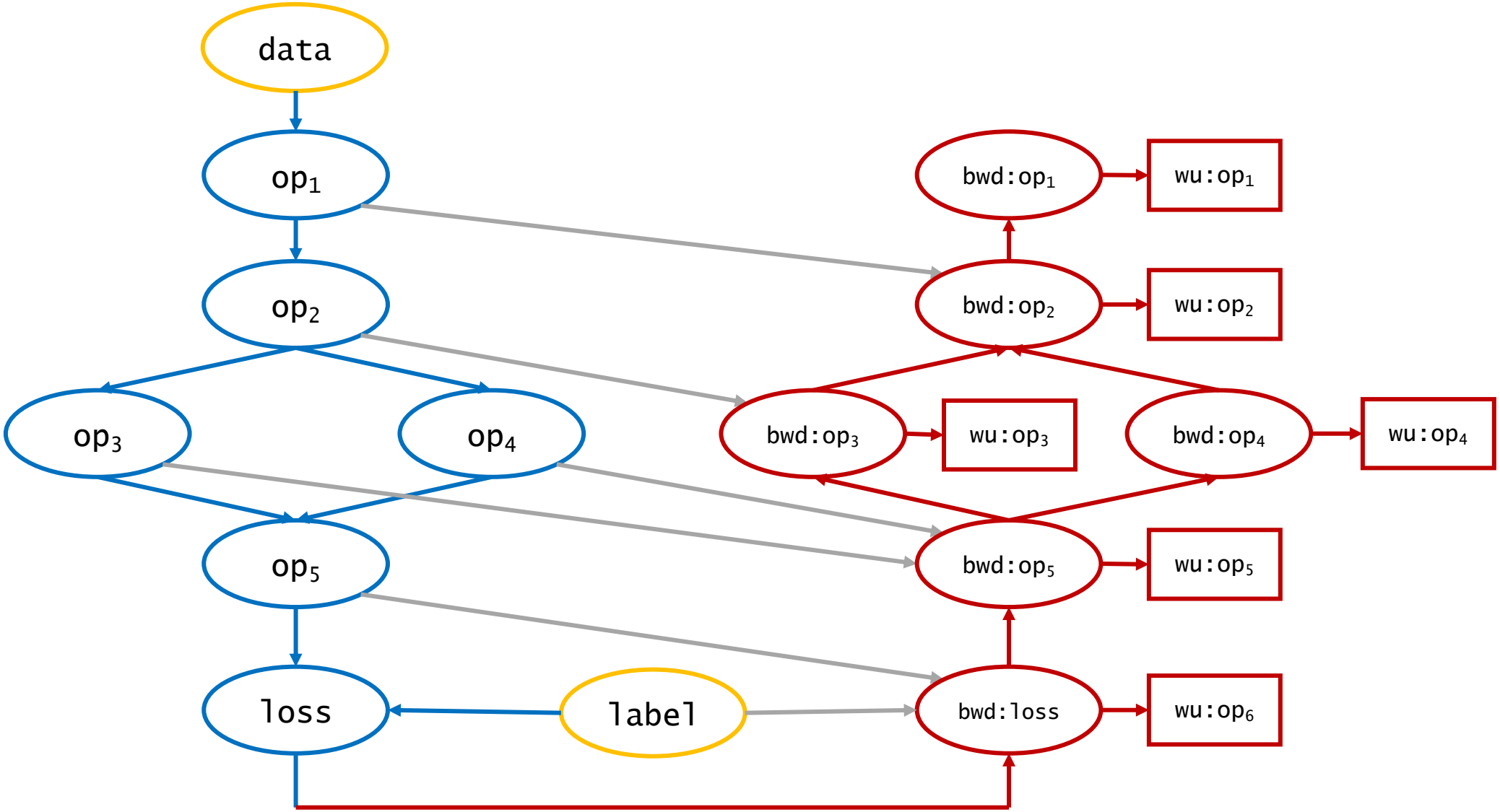


PopART INFERENCE example

```python
import numpy as np
from popart import *


# Create inference session for "add.onnx"
session = InferenceSession(fnModel="add.onnx",
                           dataFeed=DataFlow(batchesPerStep=1, {"sum": AnchorReturnType("FINAL")}),
                           deviceInfo=DeviceManager().acquireAvailableDevice(numIpus=1))

# Create buffers to receive the output data from the execution of the model
anchors = session.initAnchorArrays()

# Compile graph - where the magic happens
session.prepareDevice()

# Create the input data
x = np.random.rand(3, 4, 5).astype(np.float32)
y = np.random.rand(3, 4, 5).astype(np.float32)

# Run the inference session. The PyStepIO defines the inputs and outputs for the run
session.run(PyStepIO({"x": x "y": y}, anchors))

# Print the result
print(anchors["sum"])

```



PopART TRAINING example



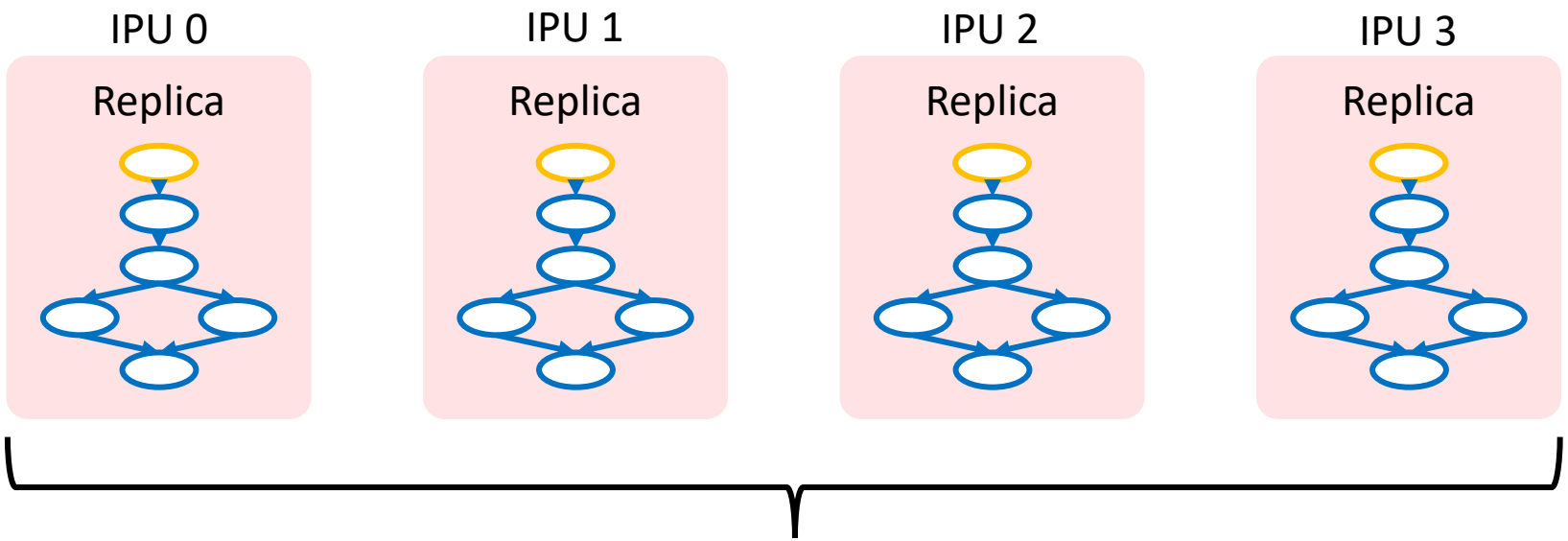
bwd = calc gradient of input to op and gradient of weights
wu = weights update

PopART TRAINING example

```
01 import numpy as np
02 from popart import *
03
04
05 builder = Builder("model.onnx")
06
07 # Add a softmax layer to the output of the model & add a label input
08 probs = builder.aiOnnx.softmax(["output"])
09 label =  builder.addInputTensor(TensorInfo("INT32", [1]), "label")
10
11 # Create training session (batchesPerStep = 32)
12 session = TrainingSession(fnModel=builder.getModelProto(),
13                           dataFeed=DataFlow(batchesPerStep=32, {"loss": AnchorReturnType("ALL")}),
14                           losses=[NllLoss("output", label, "loss")],
15                           optimizer=SGD(learning_rate=0.001),
16                           deviceInfo=DeviceManager().acquireAvailableDevice(numIpus=1))
17
18 # Create buffers to receive results from the execution
19 anchors = session.initAnchorArrays()
20
21 # Compile graph
22 session.prepareDevice()
23
24 # Write the weights & optimizer to the device
25 session.weightsFromHost()
26 session.optimizerFromHost()
27
28 # Run the training session
29 for i in range(8):
..   << Read the data (image_data [32, 4, 224, 224] & label_data [32, 4, 1]) >>
36   session.run(PyStepIO({"input": image_data, "label": label_data }, anchors))
37
38 # Write the trained model to file
39 session.modelToHost("trained_model.onnx")
```

DATA PARALLELISM: REPLICATION

```
options = SessionOptions()
options.enableReplicatedGraphs = True
options.replicationFactor = 4
session = TrainingSession(fnModel=builder.getModelProto(), userOptions=options, ...)
```



Data parallel scale-out – each replica has a copy of the model and different data

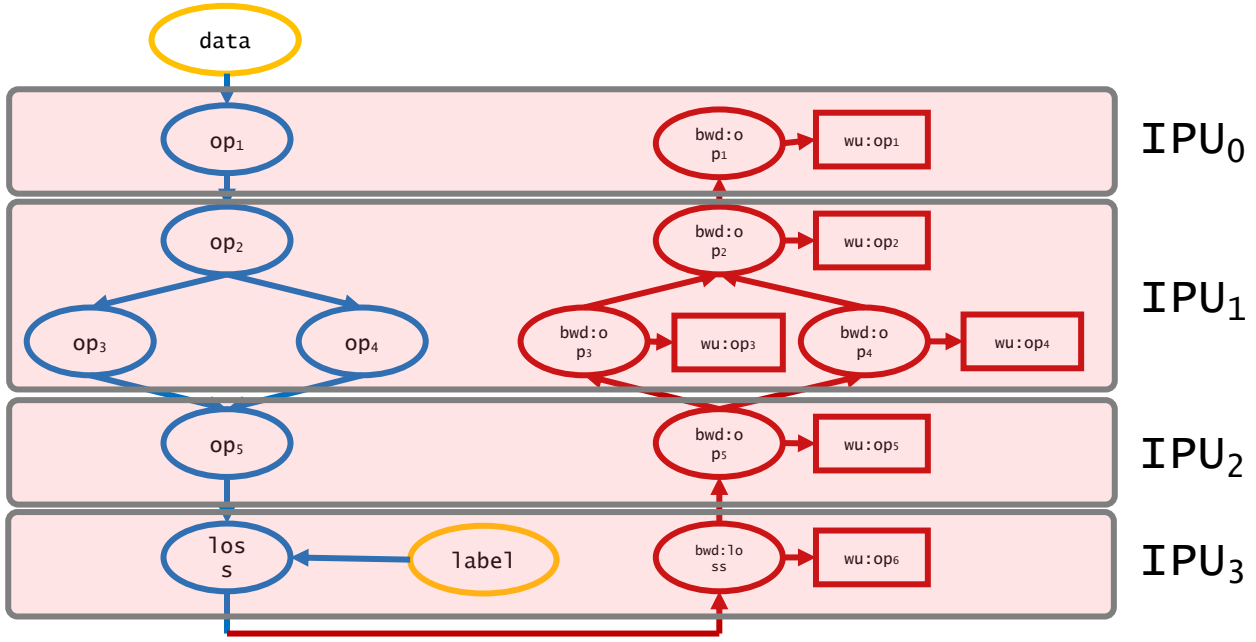
MODEL PARALLELISM: SHARDING LAYERS ACROSS IPUS

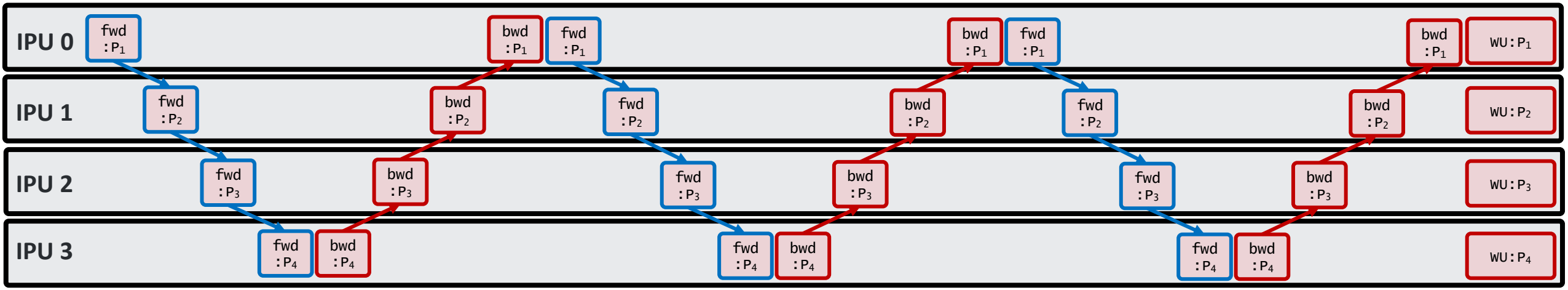
Auto sharding:

```
options.virtualGraphMode = VirtualGraphMode::Auto
```

Manual sharding:

```
options.virtualGraphMode = VirtualGraphMode::Manual

with builder.virtualGraph(1):
  o = builder.aiOnnx.add([o1, o2])
```

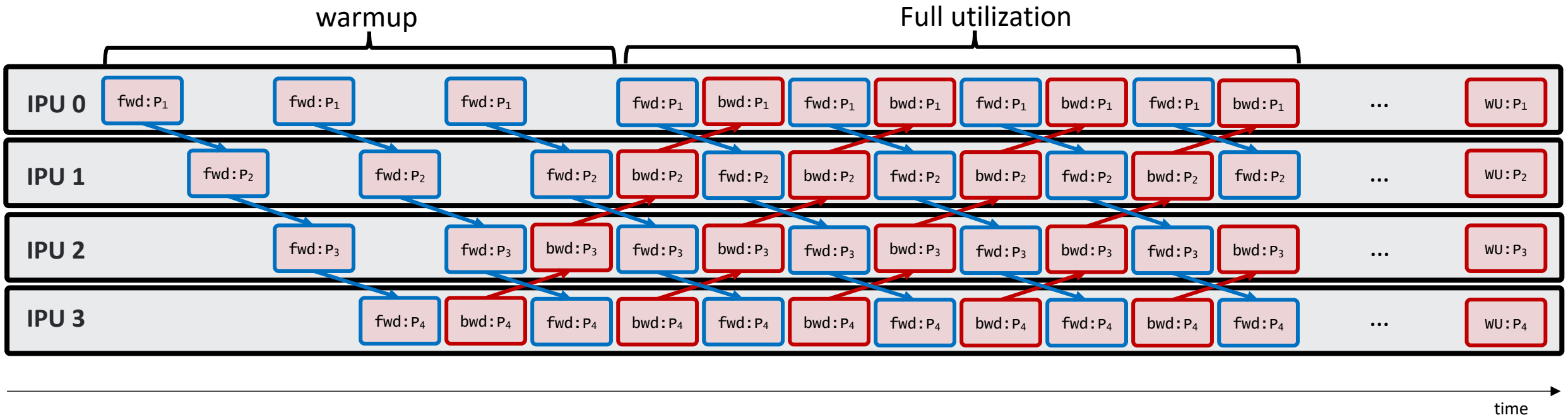




MODEL PARALLELISM: PIPELINING

Enable pipelining:

```
options.virtualGraphMode = VirtualGraphMode::Manual
options.enablePipelining = True

with builder.virtualGraph(0):
  o3 = builder.aiOnnx.add([o1, o2])

with builder.virtualGraph(1):
  o5 = builder.aiOnnx.add([o3, o4])
```



THANK YOU

www.graphcore.ai

www.graphcore.cn

hanzhao@graphcore.ai / info_china@graphcore.ai

github.com/graphcore