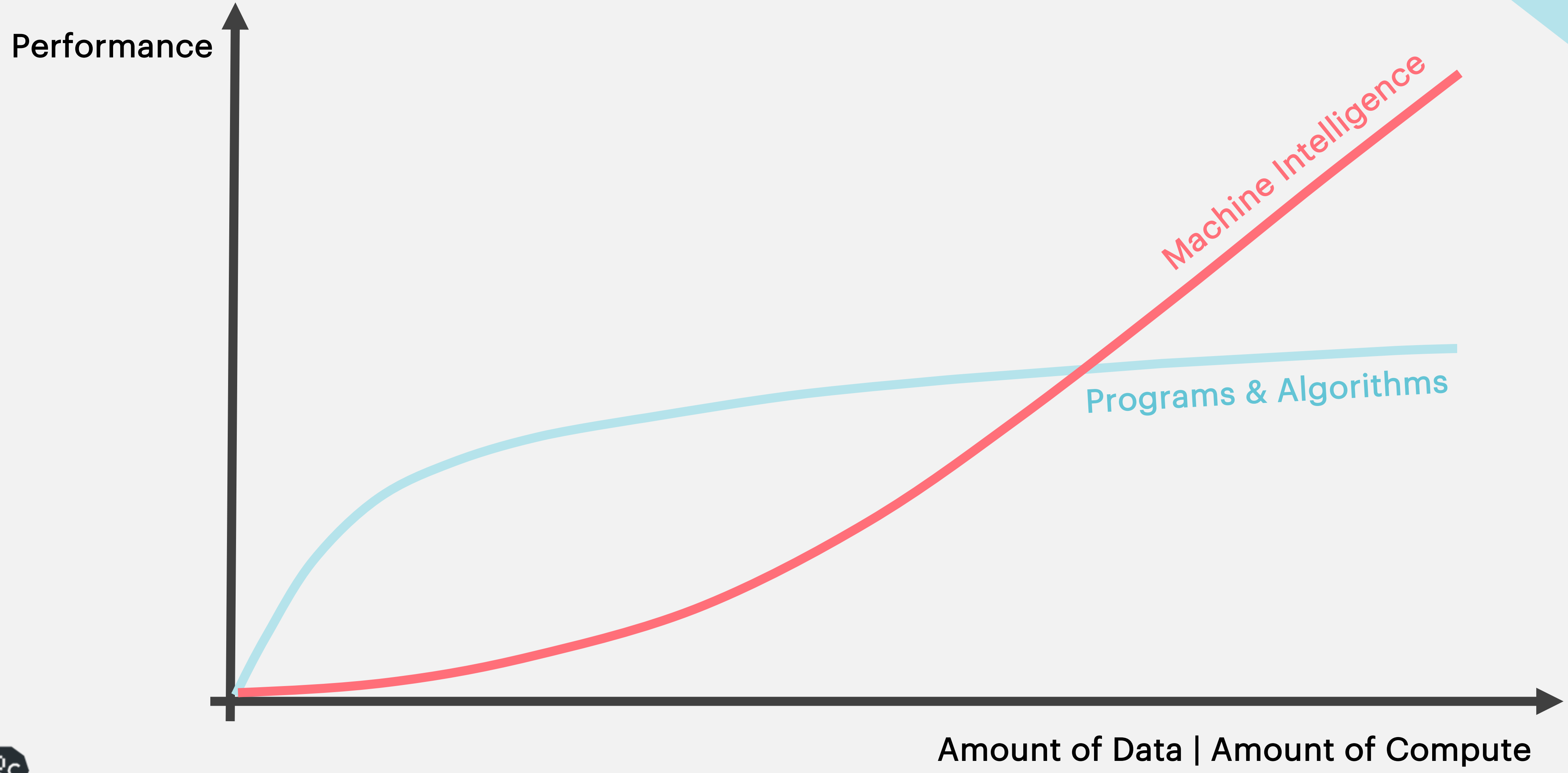
POPLAR®

Graph Toolchain for IPU

LEARNING FROM DATA : COMPUTE 2.0



A NEW APPROACH TO SOFTWARE DEVELOPMENT IS NEEDED...

COMPUTE 2.0 DEVELOPMENT FLOW

COMPUTE 1.0

Integrated Design Environment:
e.g. Visual Studio/ Eclipse

Toolchain: e.g. ICC/ Cuda

Compiler: e.g. GCC/ LLVM

Debugger: e.g. GDB

Profiler: e.g. VTune

Program/ Algorithm

```
00000000   push    ebp
00000001   mov     ebp, esp
00000003   movzx   ecx, [ebp+arg_0]
00000007   pop     ebp
00000008   movzx   dx, cl
0000000C   lea     eax, [edx+edx]
0000000F   add     eax, edx
00000011   shl     eax, 2
```

COMPUTE 2.0

ML Graph Framework:
e.g. TensorFlow/ PyTorch

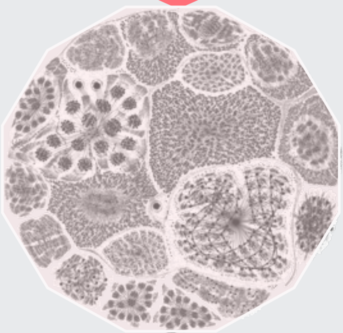
Graph Toolchain: POPLAR®

Graph Compiler

Graph Engine

Graph Debug/ Optimization

Knowledge Model



Learn from Data

IPU PROCESSOR

IPU-Tiles™

1216 IPU-Tiles™ each with an independent IPU-Core™ and tightly coupled In-Processor-Memory™

IPU-Core™

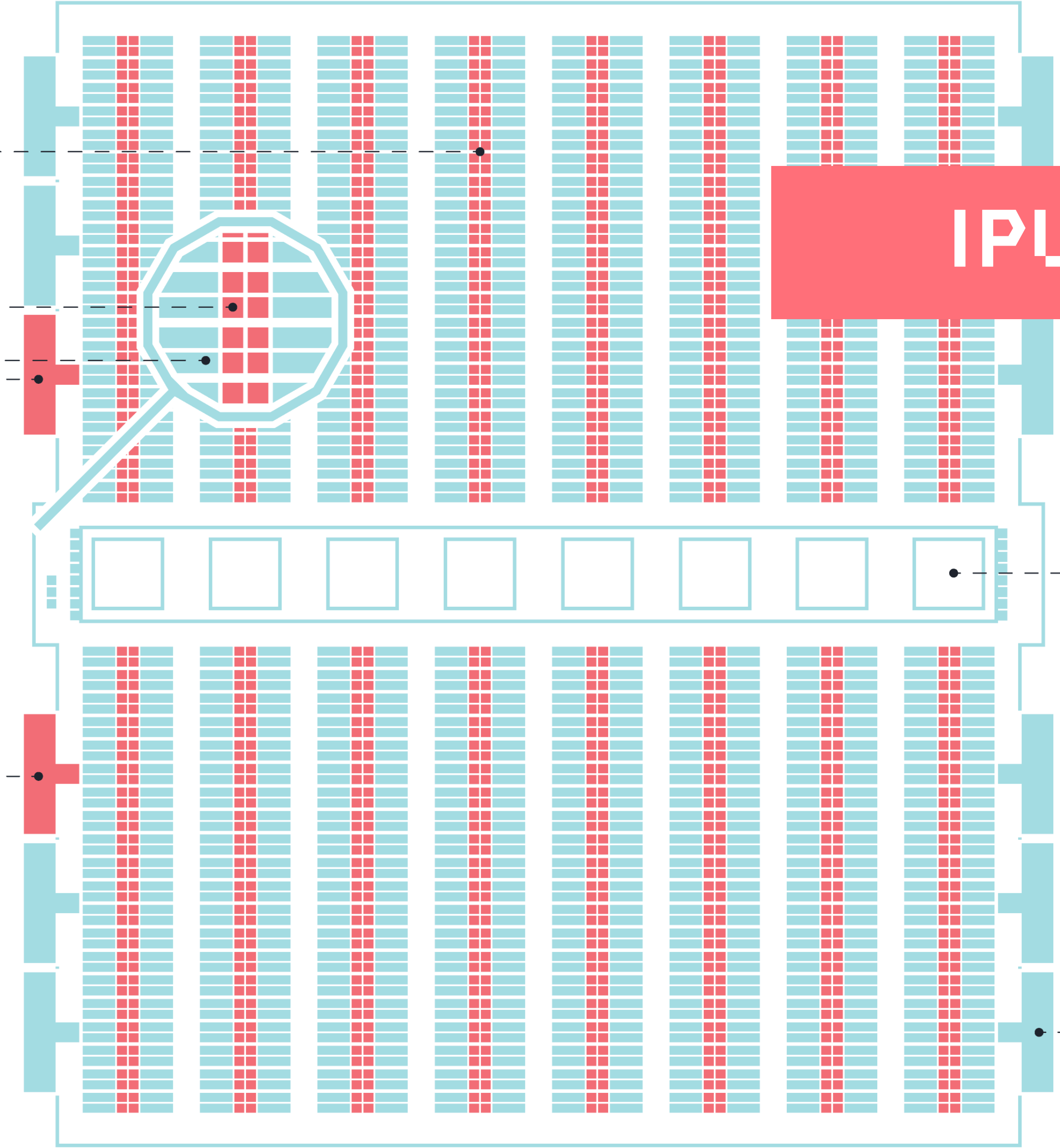
1216 IPU-Cores™ with 7296 programs executing in parallel

In-Processor-Memory™

300MB In-Processor-Memory™
45TB/s memory bandwidth
Whole model held on-chip

PCIe

PCIe Gen4 x16
64 GB/s bidirectional bandwidth to host



IPU-Exchange™

8 TB/s all to all IPU-Exchange™
Non-blocking, any communication pattern

IPU-Links™

80 IPU-Links, 320GB/s chip to chip bandwidth

BULK SYNCHRONOUS PARALLEL (BSP)

Software bridging model for parallel computing

Compute

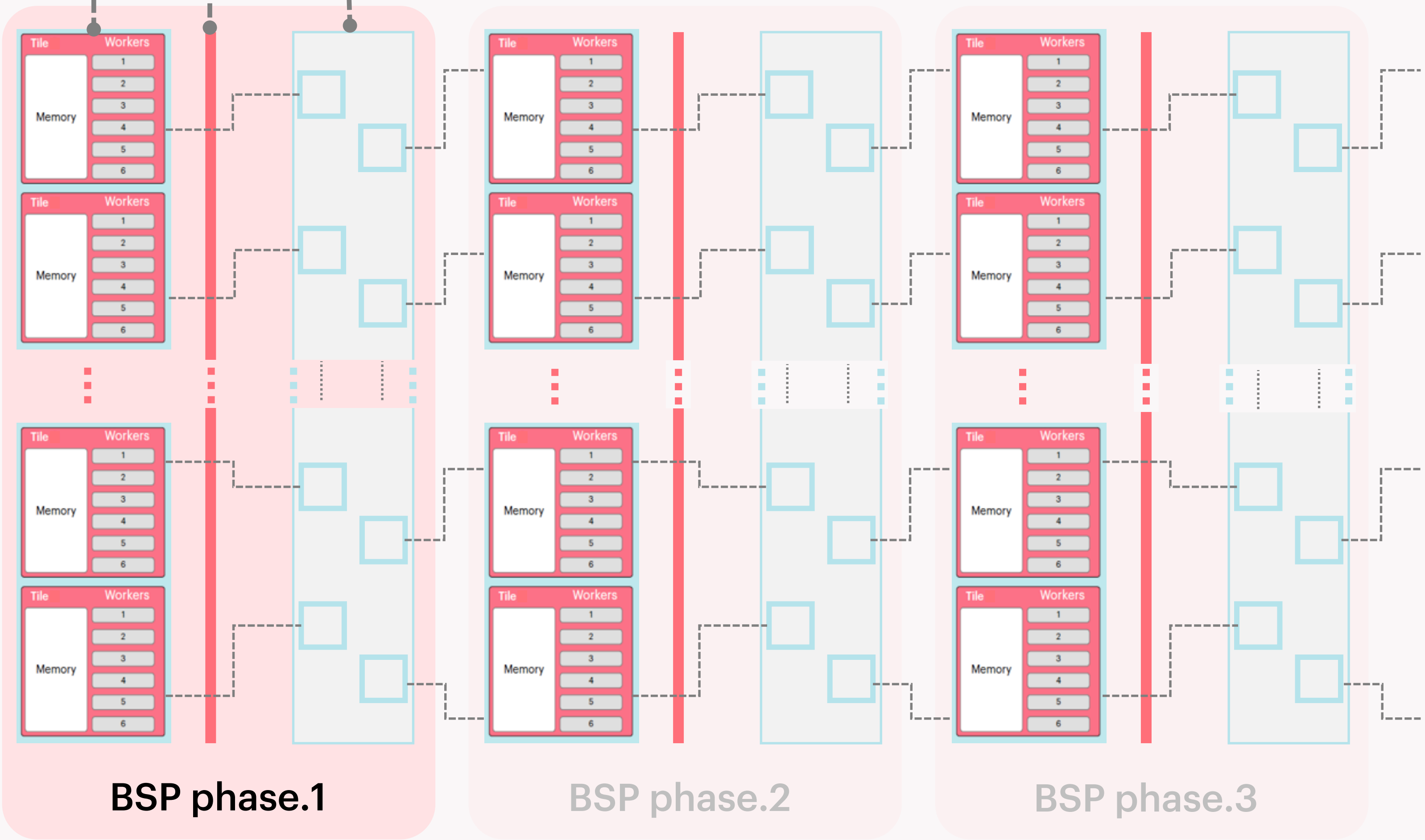
10,000s of compute threads
all operating in parallel
each with all the data that
they need, held locally

BSP Sync

All threads are
synchronized

Exchange

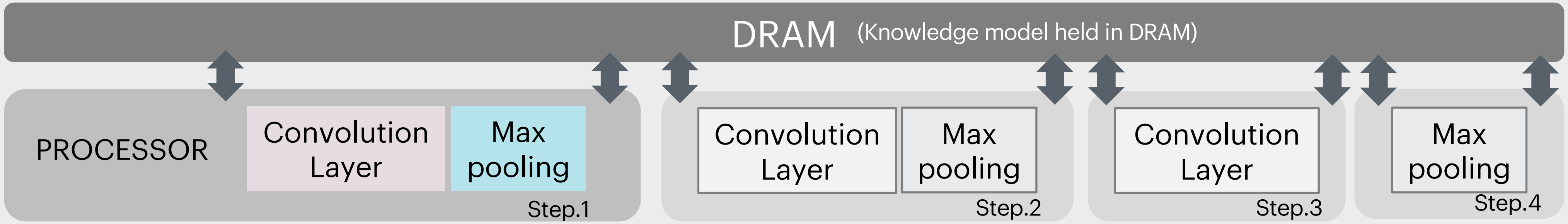
Data is exchanged so that every thread
has all the data that it needs for the
next phase of Compute



WHY IS BSP IMPORTANT?

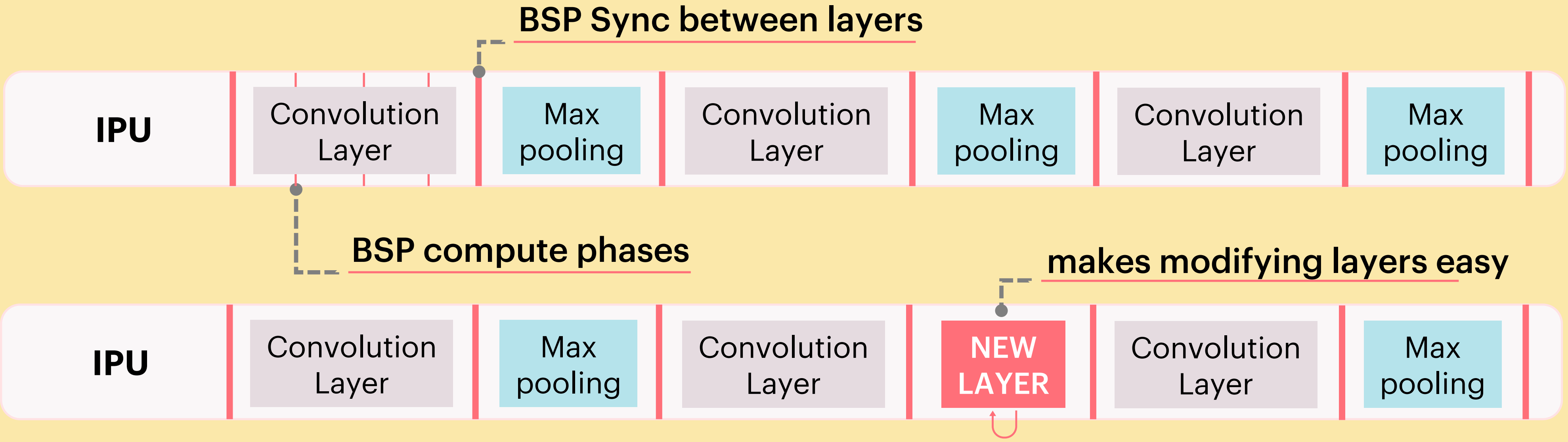
CPU/GPU

- Optimization is difficult
- Timing hazards occur
- Hard to modify or change
- Data handling challenging



IPU

- High performance straight out of the box
- No timing hazards
- Easy to modify
- Whole Knowledge Model is held inside the IPU



USE YOUR EXISTING KNOWLEDGE MODELS

TensorFlow ONNX PYTORCH

POPLAR®
Graph Toolchain

IPU-Processor PCIe Card

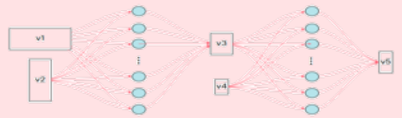
IPU Servers and System

POPLAR® : Graph Toolchain

POPLAR® Graph Libraries

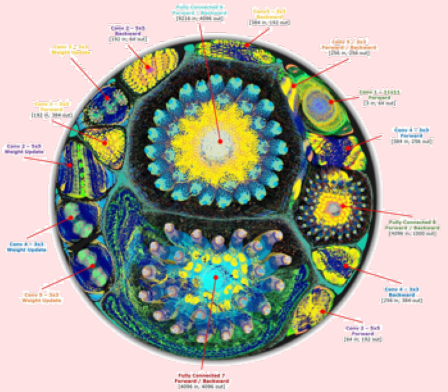
popnn	poputils	poplin
popops	poprandom	popsparse

POPLAR® Framework

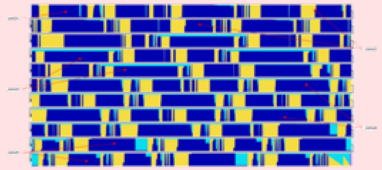


C++ / Python Graph Framework

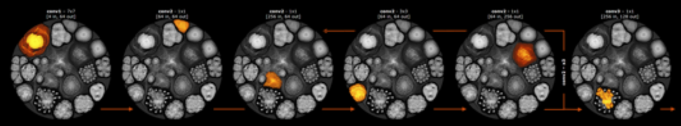
POPLAR® Compute Graph

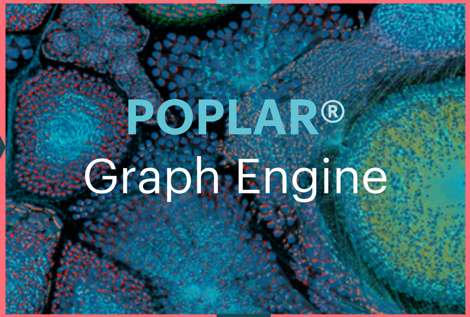


POPLAR® Visualization & Debug Tools



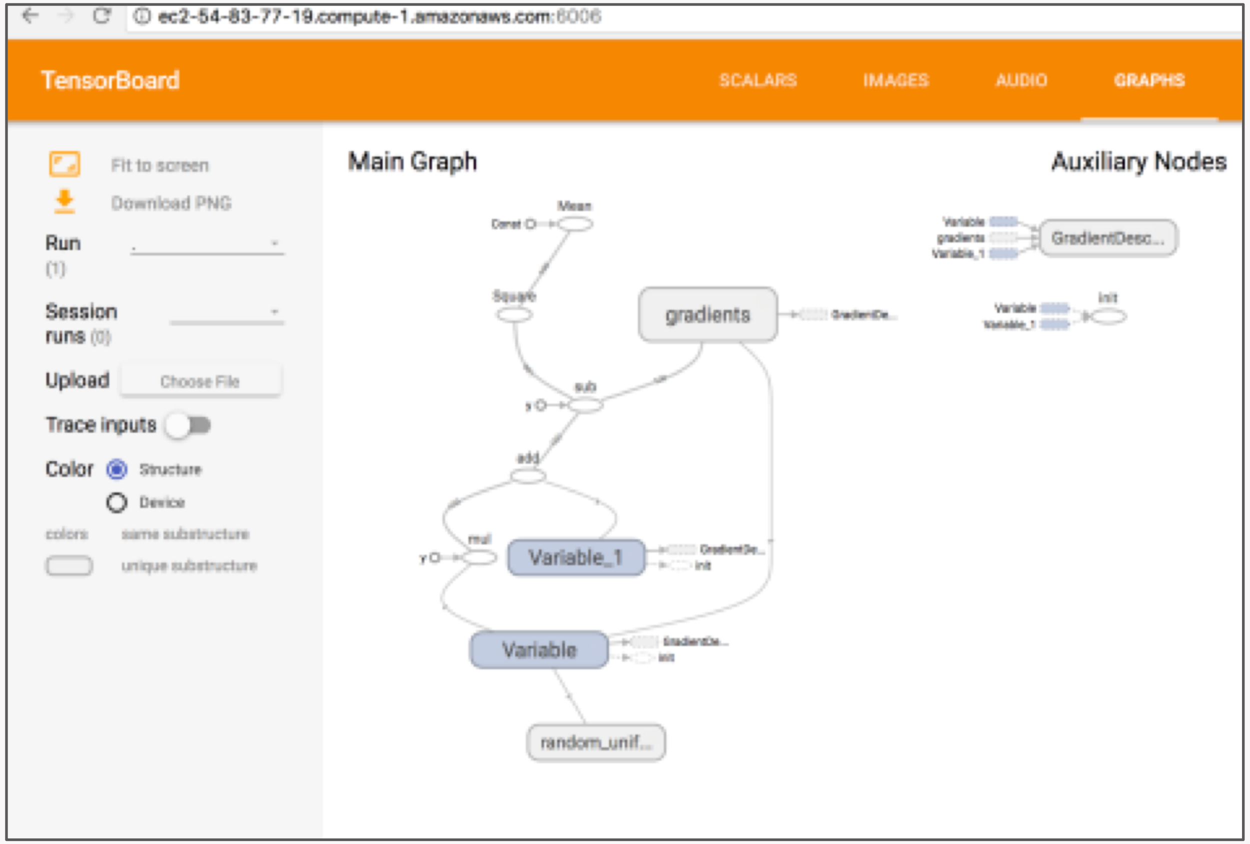
POPLAR® Graph Compiler





POPLAR® Graph Engine

DEVELOP YOUR MODEL USING INDUSTRY STANDARD ML FRAMEWORKS



TensorFlow

PYTORCH

ONNX

mxnet

Optimized support for inference and training

POPLAR® GRAPH LIBRARIES

Highly optimized ***open source*** libraries partition work and data efficiently across IPU devices

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 functions	Neural network functions (activation fns, pooling, loss)

POPLAR®

GitHub

OPEN-SOURCE GRAPH LIBRARIES

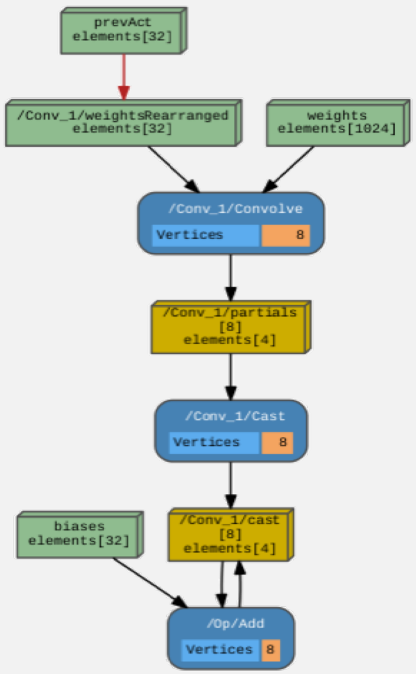
> **50** open-source **GRAPH FUNCTIONS**
  available inlcuding (matmul, conv, etc) built from...

> **750** optimized **COMPUTE ELEMENTS**
  such as (ReduceAdd, AddToChannel, Zero, etc)

easily create new **GRAPH FUNCTIONS**
using the library of **COMPUTE ELEMENTS**

modify and create new **COMPUTE ELEMENTS**

example GRAPH FUNCTION
32in_32out_Fully_Connected_Layer

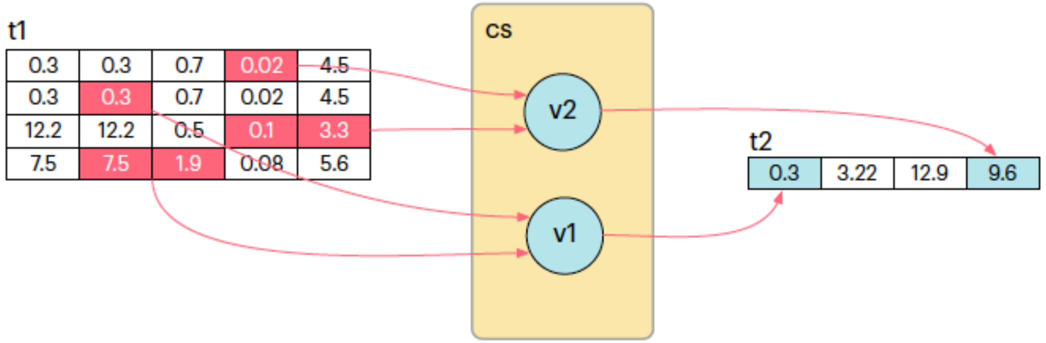


GitHub share library elements and new innovations

POPLAR® GRAPH FRAMEWORK

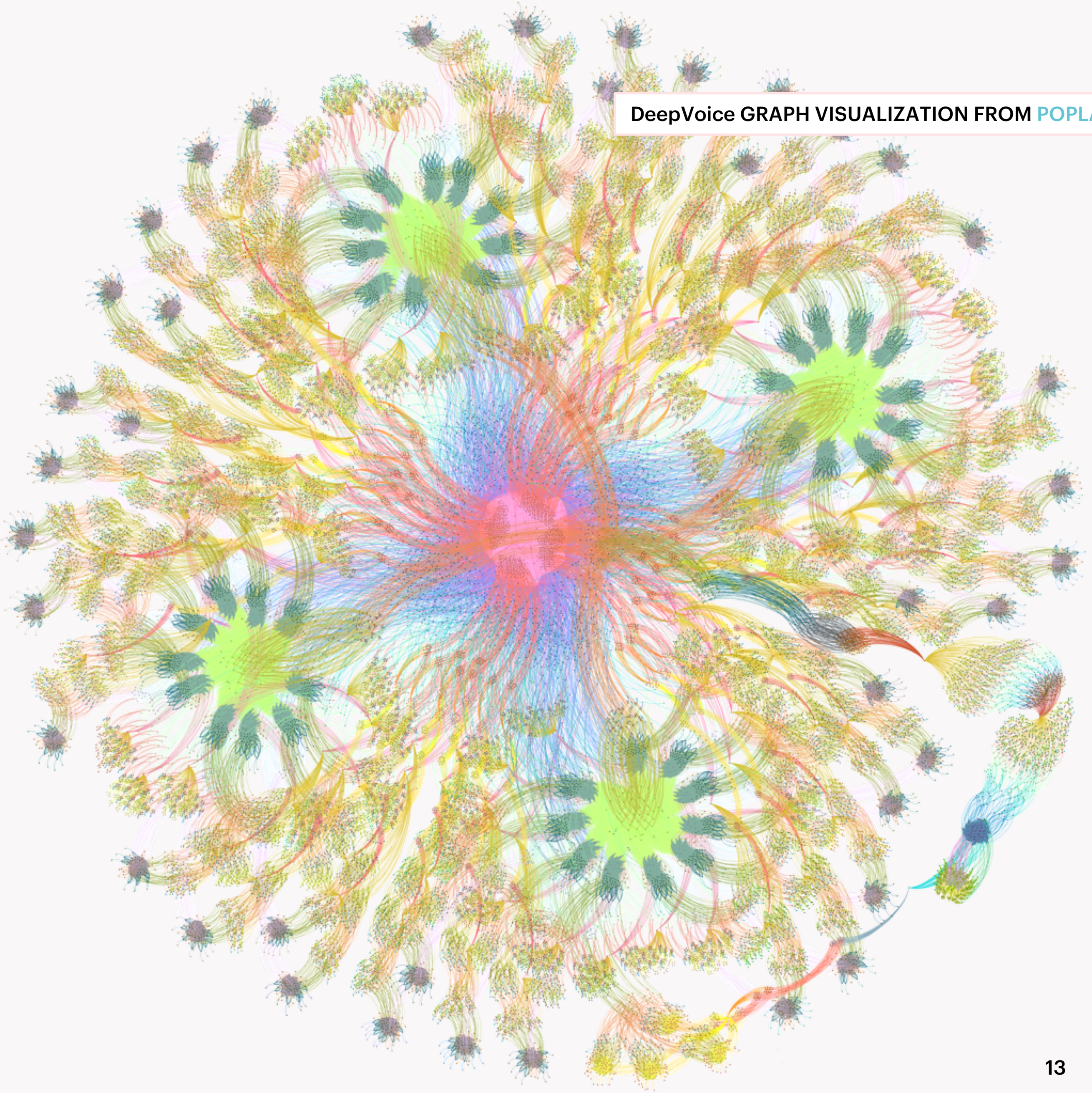
C++ / PYTHON – POPLAR® GRAPH FRAMEWORK
LETS YOU EASILY MODIFY OR CREATE YOUR OWN GRAPH FUNCTIONS

```
Graph g(device); g.addCodelets(“codelets.cpp");
Tensor t1 = g.addTensor(“float”, {4, 5});
Tensor t2 = g.addTensor(“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"]);
```

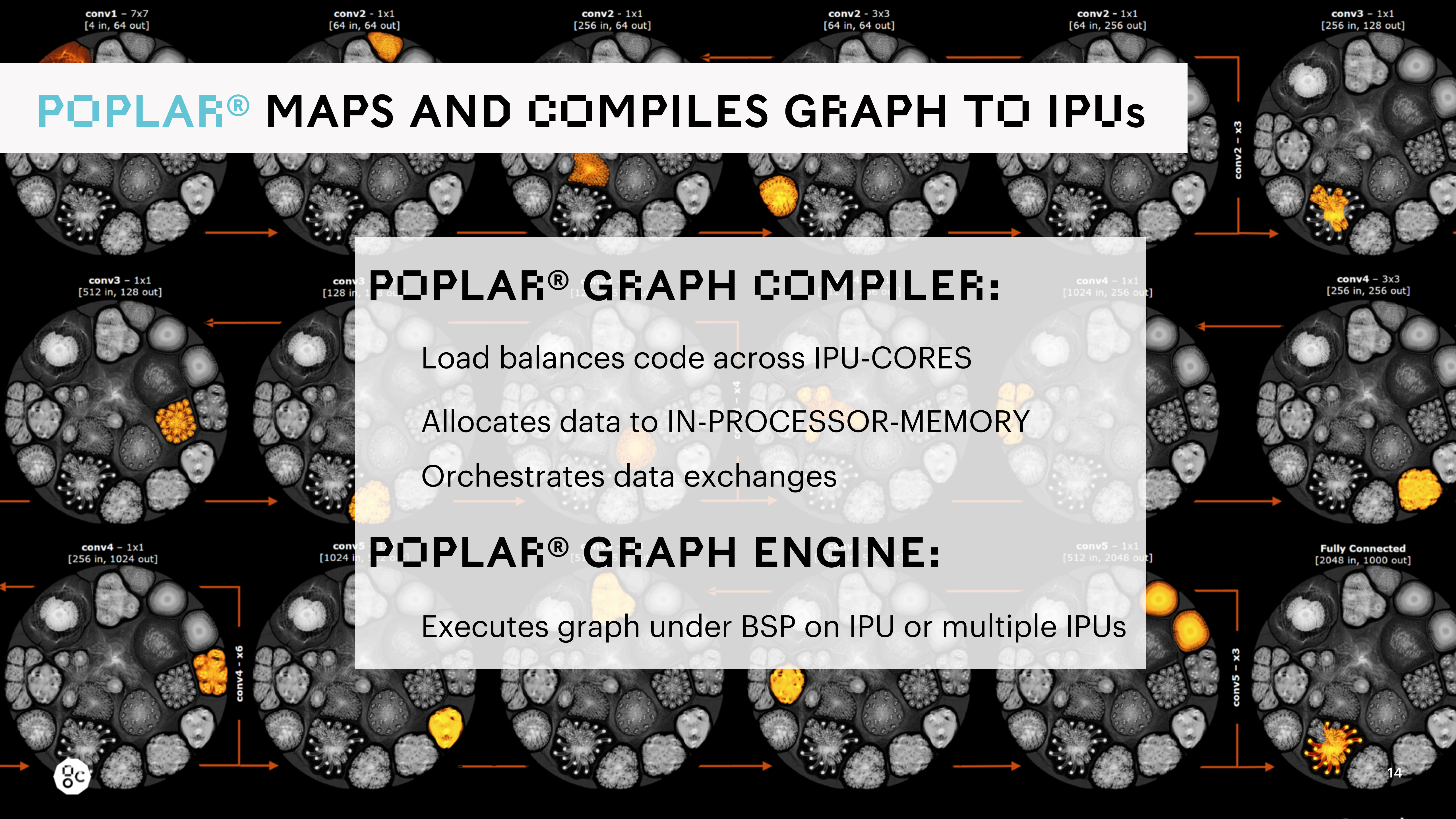


POPLAR®

expands the ML Framework output to a full compute graph



POPLAR® MAPS AND COMPILES GRAPH TO IPUs



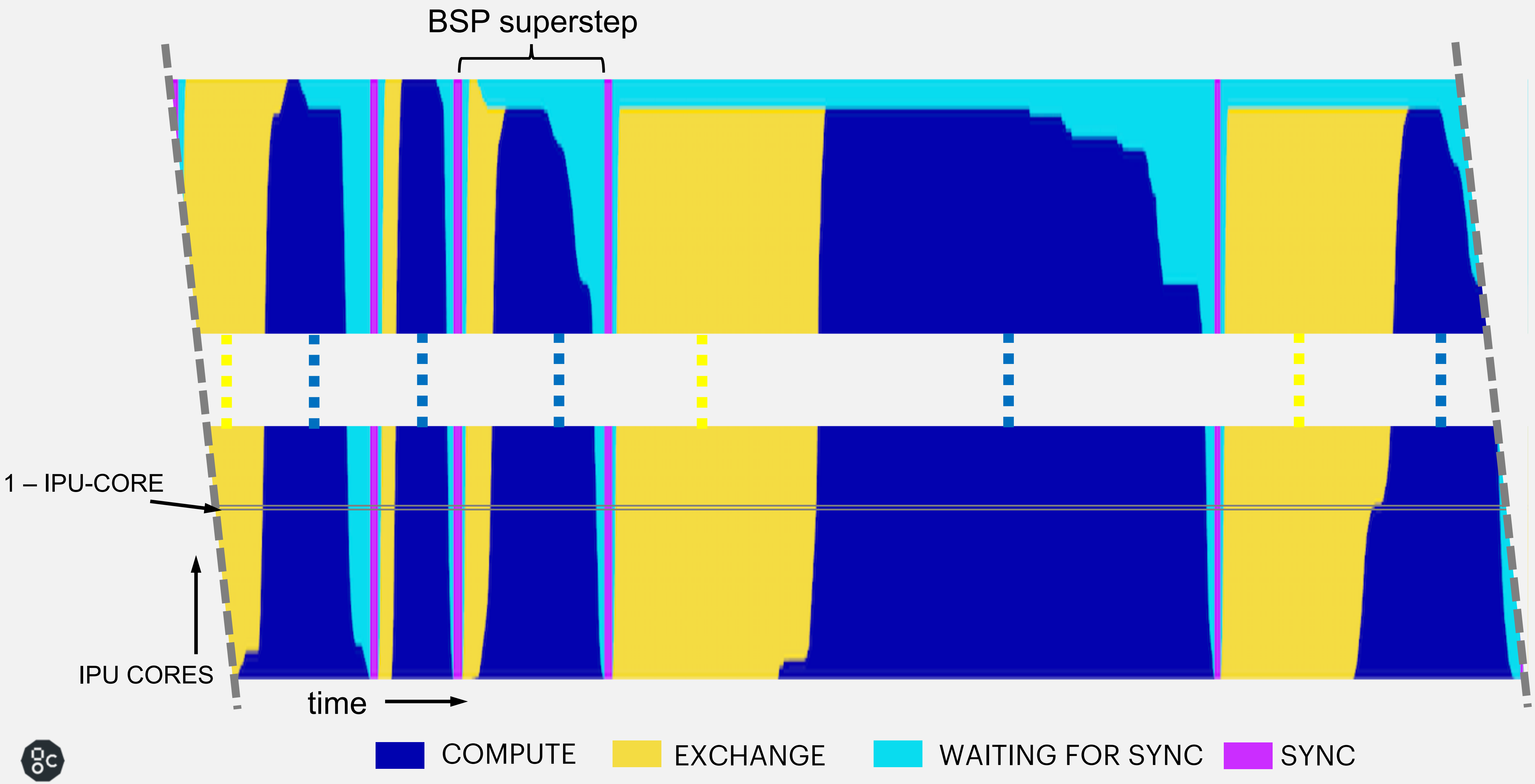
POPLAR® GRAPH COMPILER:

- Load balances code across IPU-CORES
- Allocates data to IN-PROCESSOR-MEMORY
- Orchestrates data exchanges

POPLAR® GRAPH ENGINE:

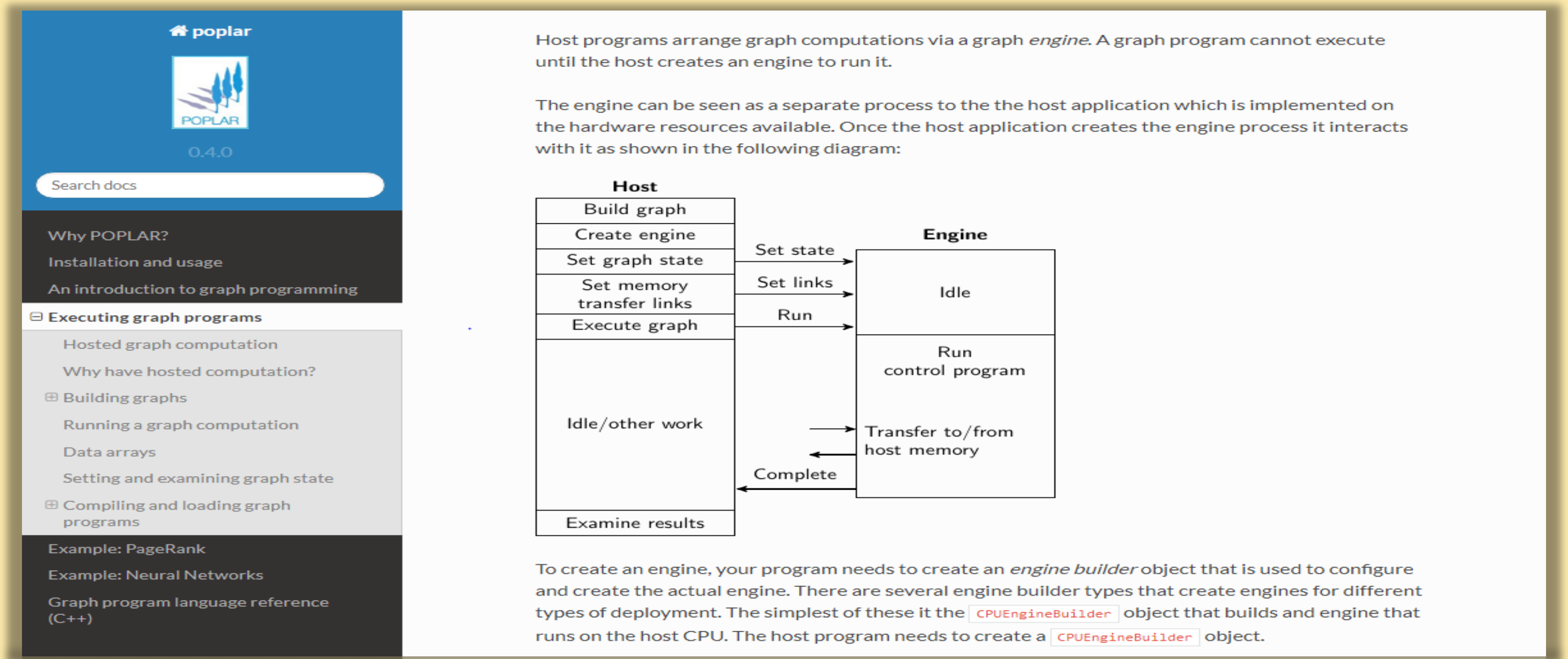
- Executes graph under BSP on IPU or multiple IPUs

ADVANCED VISUALIZATION AND DEBUG TOOLS



SUPPORT@GRAPHCORE.AI

COMPREHENSIVE USER DOCUMENTATION, EXAMPLES, FAQS, APPLICATION NOTES, TUTORIALS AND ONLINE SUPPORT



poplar

POPLAR

0.4.0

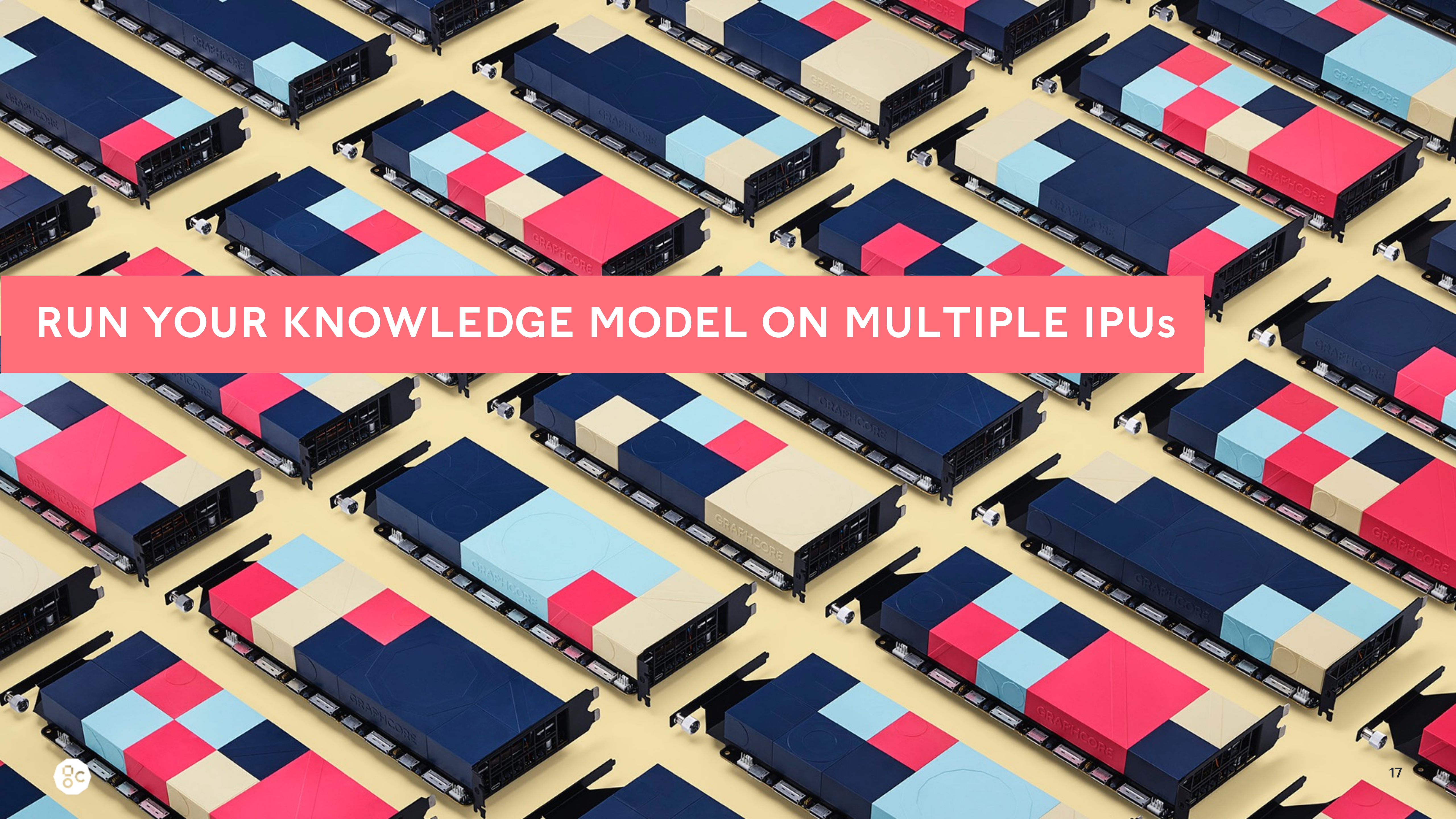
Search docs

- Why POPLAR?
- Installation and usage
- An introduction to graph programming
- Executing graph programs
  - Hosted graph computation
  - Why have hosted computation?
  - Building graphs
  - Running a graph computation
  - Data arrays
  - Setting and examining graph state
  - Compiling and loading graph programs
- Example: PageRank
- Example: Neural Networks
- Graph program language reference (C++)

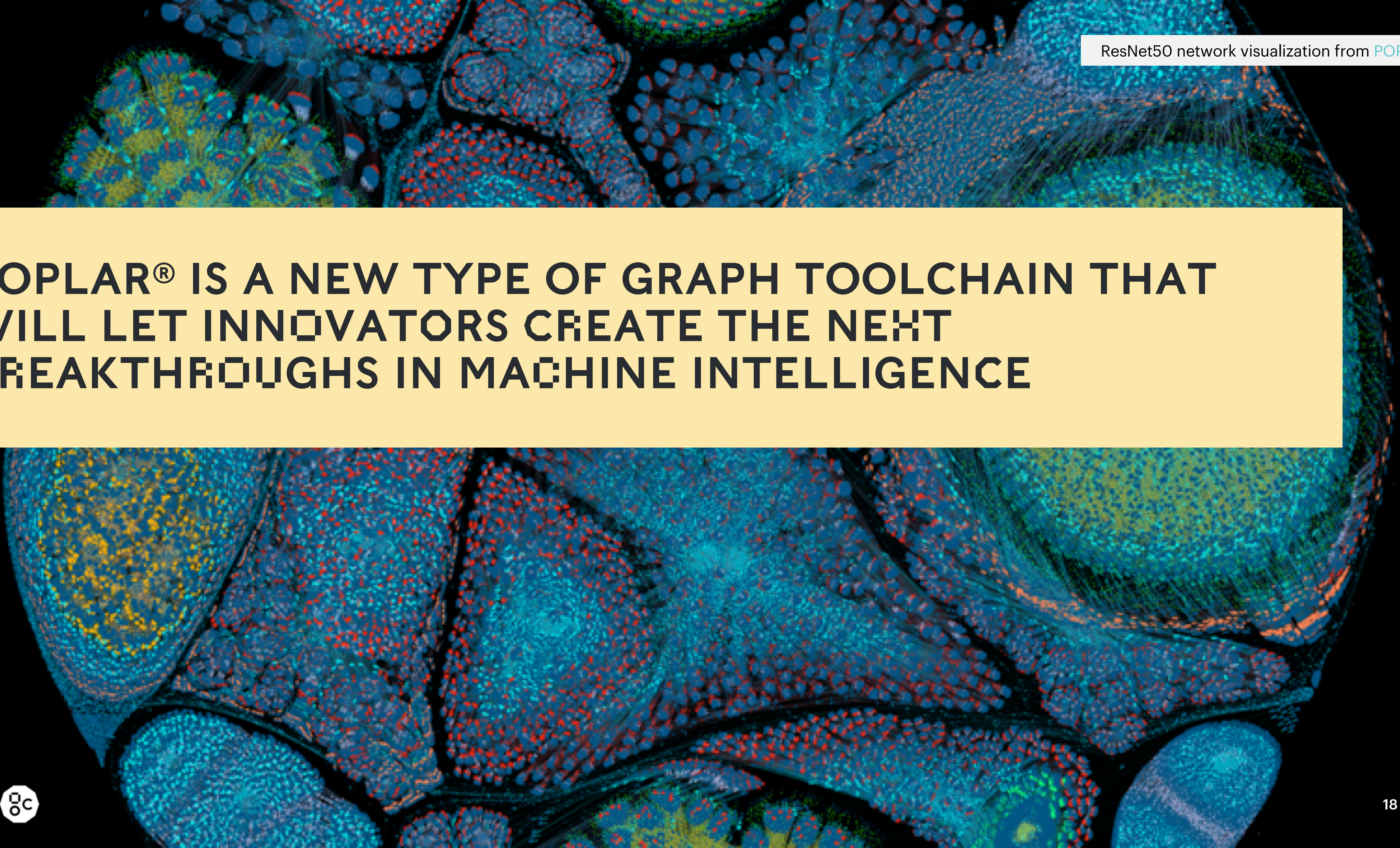
Host programs arrange graph computations via a graph *engine*. A graph program cannot execute until the host creates an engine to run it.

The engine can be seen as a separate process to the the host application which is implemented on the hardware resources available. Once the host application creates the engine process it interacts with it as shown in the following diagram:

To create an engine, your program needs to create an *engine builder* object that is used to configure and create the actual engine. There are several engine builder types that create engines for different types of deployment. The simplest of these it the `CPUEngineBuilder` object that builds and engine that runs on the host CPU. The host program needs to create a `CPUEngineBuilder` object.



RUN YOUR KNOWLEDGE MODEL ON MULTIPLE IPUs



ResNet50 network visualization from POPLAR®

POPLAR® IS A NEW TYPE OF GRAPH TOOLCHAIN THAT WILL LET INNOVATORS CREATE THE NEXT BREAKTHROUGHS IN MACHINE INTELLIGENCE

THANK YOU