IPUs

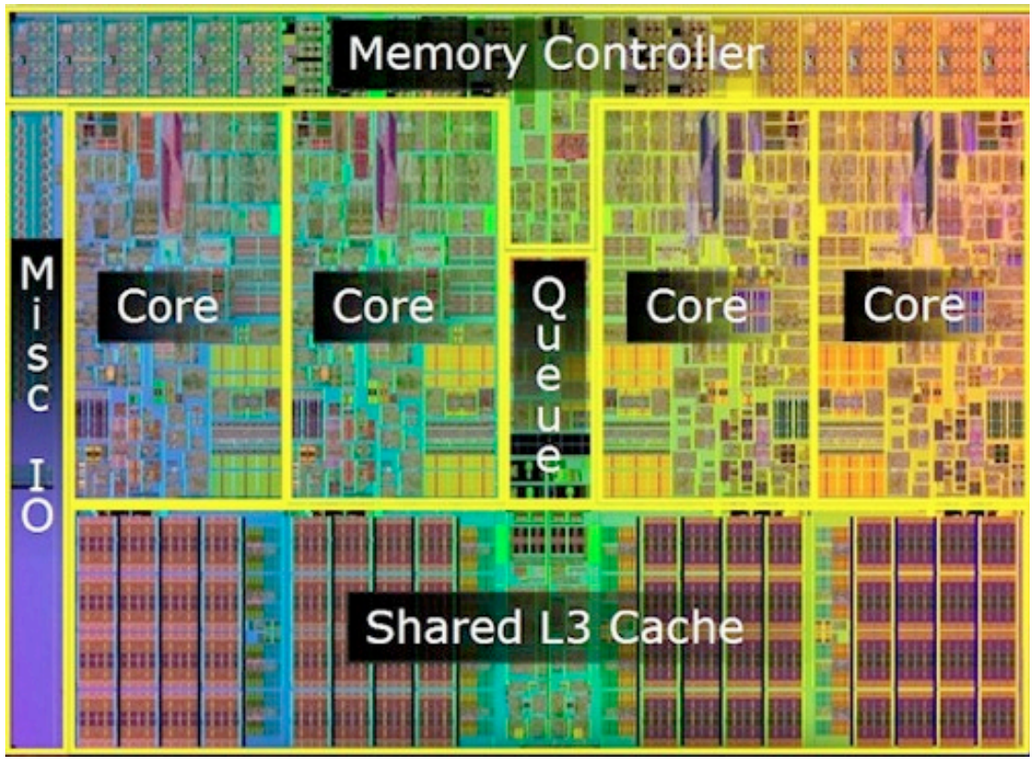
for analysis

Daniel O’Hanlon

University of BRISTOL

CPU

Low-latency, low throughput



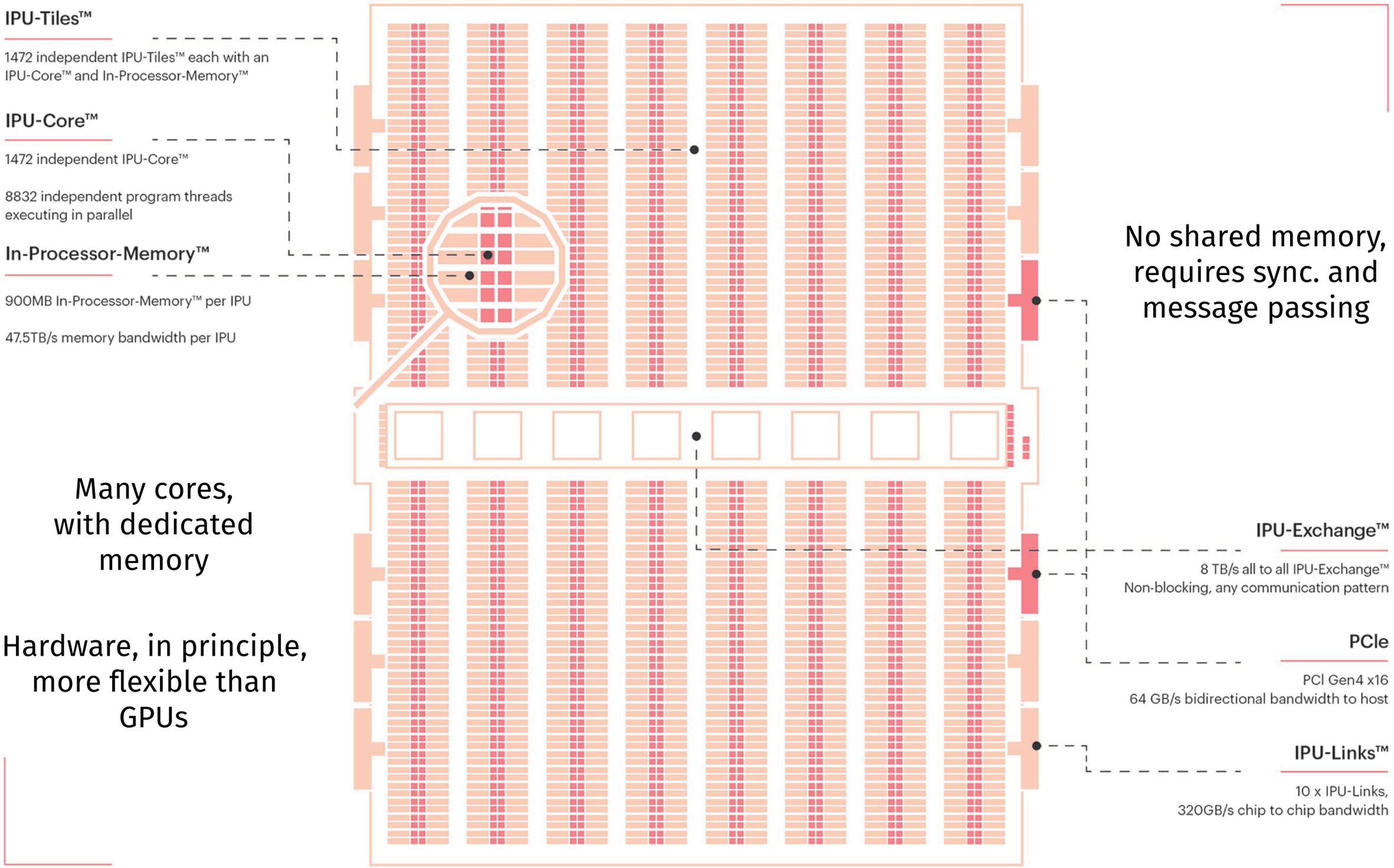
- Few complex cores
- Superscalar
- Few (+ SIMD) registers
- Cache hierarchy

GPU

High-latency, high throughput

- Many simple cores
- Lots of registers
- Memory hierarchy

The IPU



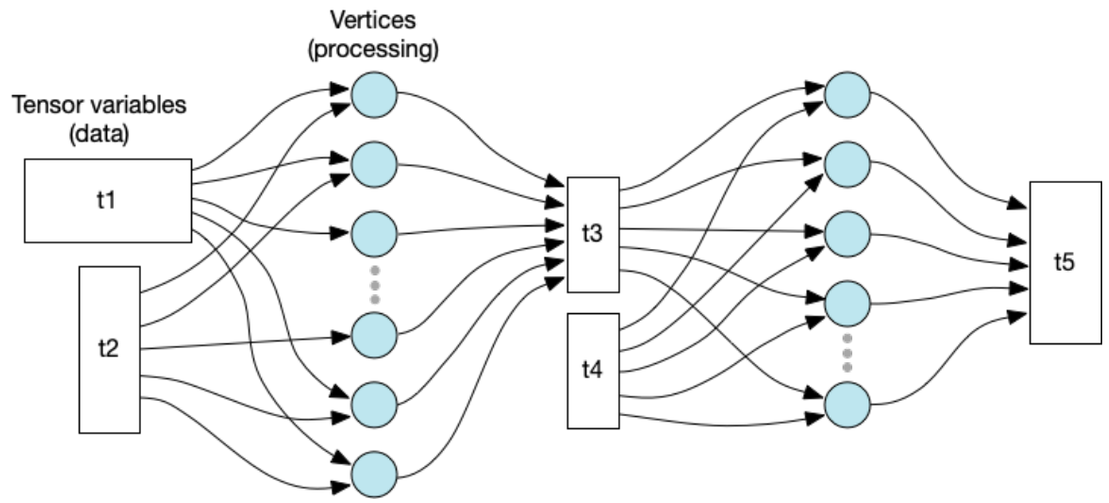
No shared memory, requires sync. and message passing

Many cores, with dedicated memory

Hardware, in principle, more flexible than GPUs

How?

- IPU runs a static compute graph, which describes all data flow, conditional execution, synchronisation, etc



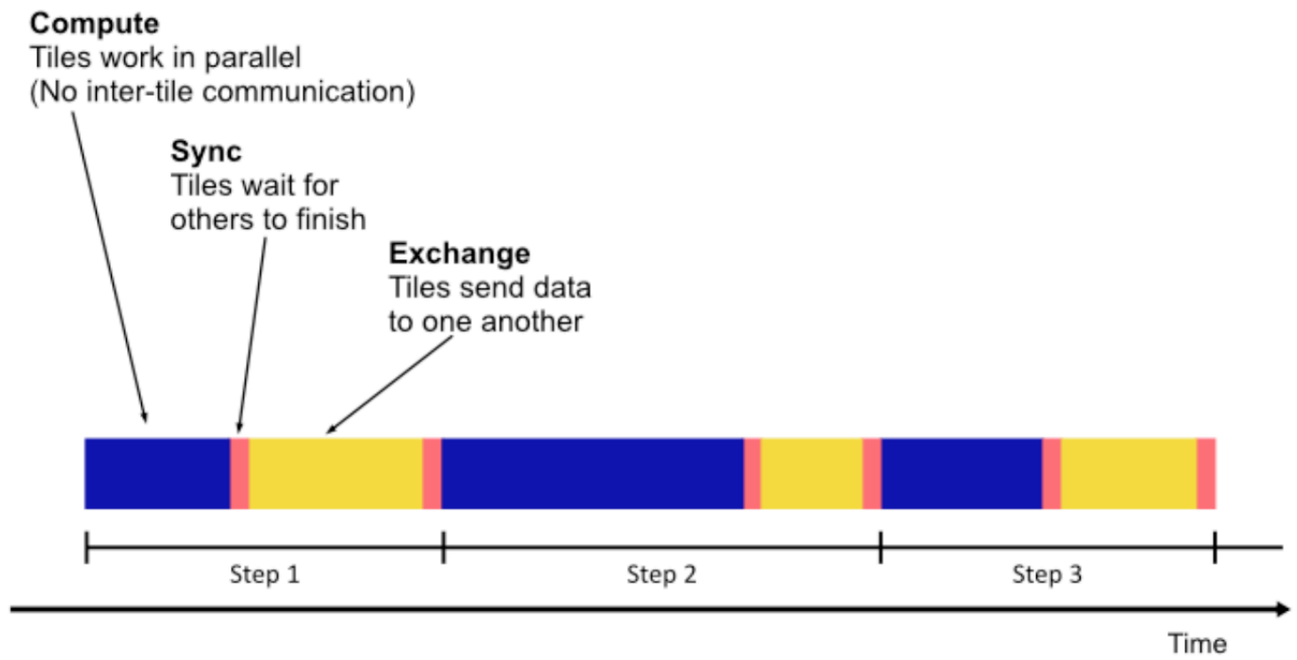
Defined in a declarative way and compiled, much like (pre-eager) TensorFlow

A few ways to build the graph:

- Easiest is with TensorFlow - supports most operations with explicit device preparation or compute ‘strategy’ (some optimised-for-IPU neural network layers)
- C++ ‘Poplar’ interface with full manual control over data placement and execution
- Execution of imported ONNX models in ‘PopART’
- PyTorch

How?

- Benefit from explicit mapping of graph to individual hardware cores, and fast all-to-all core connections is that it enables bulk-synchronous parallel execution

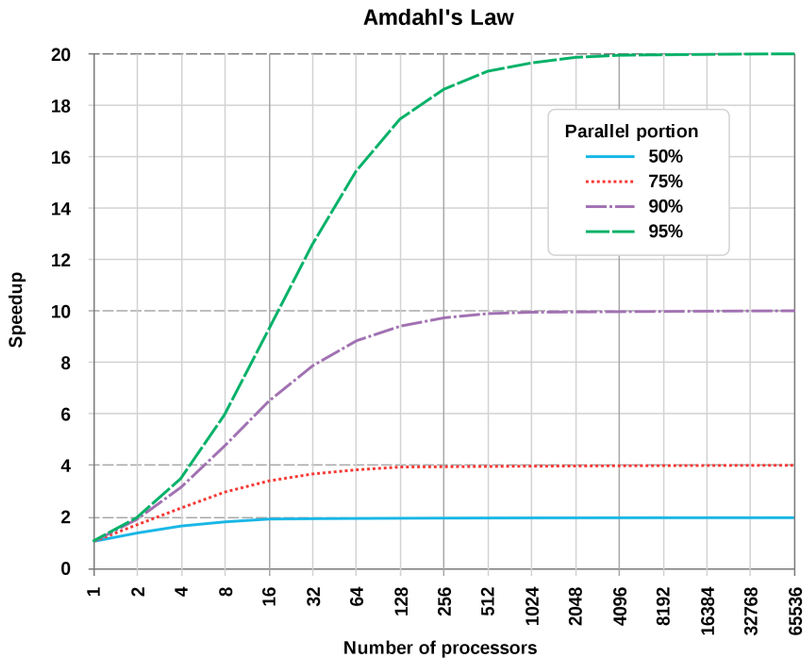


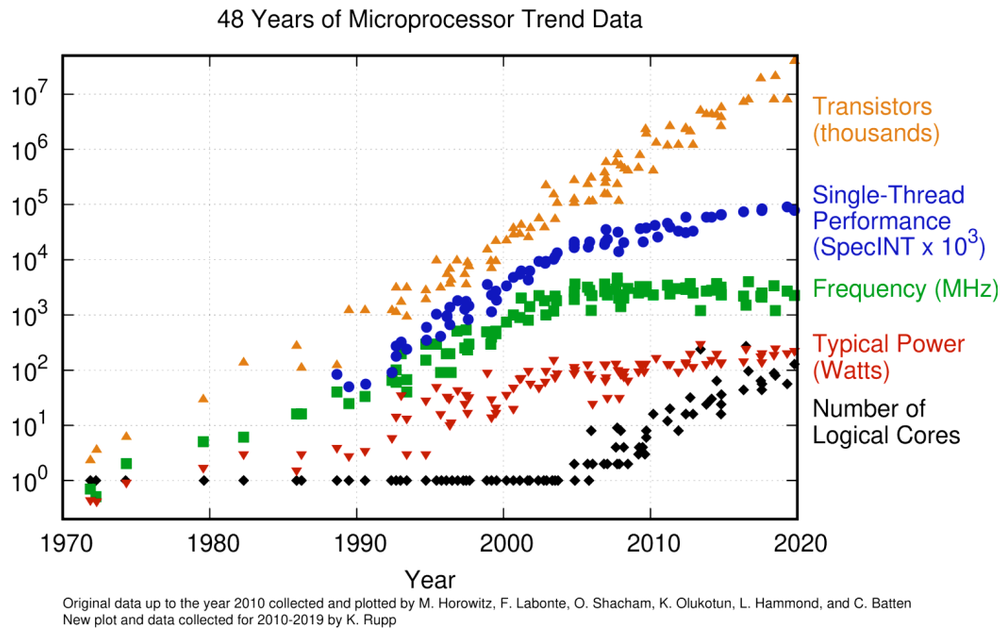
- Upshot: No mutexes, no blocking, no race conditions, no warp divergence, no bank conflicts….
- All you need is the graph, and less understanding of the underlying architecture
- Each core is a real, independent core that can run independent code on independent data (MIMD), modulo any synchronisation requirements

Why?

- Single CPU cores are proportionally slower for the tasks we want them to do

- Many-core hardware accelerators are fast, but limited by:





- 1. Serial component of any parallel algorithm dominates as parallelism increases (Amdahl's law), (applies also to memory accesses)

- 2. PCI-e busses are slow compare to how fast PCI-e devices can crunch through data

- GPUs do many 100s of times more FLOPS than CPUs, but can end up being only a few times faster

- Pure FLOPS not the only answer, given 1 and 2 - want hardware that has flexibility to do more per unit of input data transferred, and hardware that allows developers to write code with more parallelism

Poplar, C++

Graphcore APIs

Lower-level interface ‘vertex code’, written in vanilla C++, which is compiled to the IPU instruction set and executes on a *single* tile:

E.g, from the Kalman filter,

a 2x2 matrix inverse:

(Lots of high level functionality isn’t present in poplar yet, and inverting should be okay for these matrices)

github.com/dpohanlon/IPU4HEP

```cpp
#include <poplar/Vertex.hpp>

class MatrixInverse2 : public poplar::Vertex {
public:

  // Fields
  poplar::Vector<poplar::Input<poplar::Vector<float>>> in;
  poplar::Vector<poplar::Output<poplar::Vector<float>>> out;

  // Compute function
  bool compute() {

    float det = in[0][0] * in[1][1] - in[0][1] * in[1][0];

    if (det == 0) return false; // Matrix is singular

    out[0][0] =  in[1][1] * (1. / det);
    out[0][1] = -in[0][1] * (1. / det);
    out[1][0] = -in[1][0] * (1. / det);
    out[1][1] =  in[0][0] * (1. / det);

    return true;
  }
};
```

Poplar, C++

Graphcore APIs

Possible also to copy (simple, POD…) objects to the IPU via reinterpret_cast

(thanks to Bristol HPC for this trick!)

```
class SomeVertex : public Vertex {
public:
    Input <Vector<char, VectorLayout::ONE_PTR>> kinematics;

    // Breaking the law
    inline auto asTileData(void *ref) -> Kinematics *const {
        return reinterpret_cast<Kinematics *const>(ref);
    }

    bool compute() {
        auto tileKinematics = asTileData(&kinematics[0]);

        tileKinematics->mass = 1.4f;
        // ...
    }

}
```

Kinematics data container, and simple numerical functions

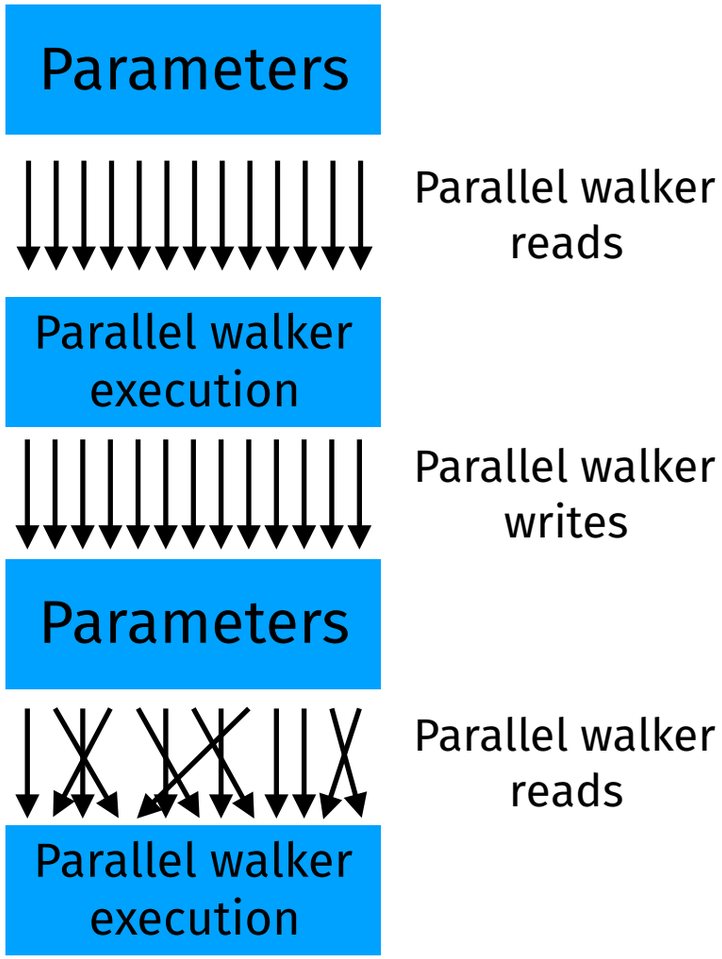
However not possible to (dynamically) allocate memory in the tile code, which makes some algorithms a little fiddly

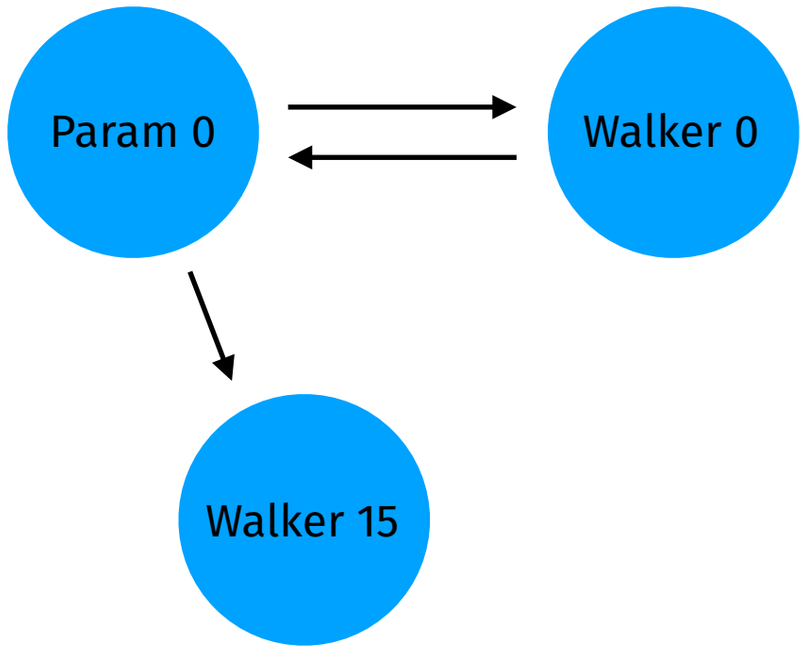
Application to MCMC sampling

- Goodman and Weare sampler, made famous by Emcee (emcee.readthedocs.io),

  paper here:

  - Generate some random initial points for n *walkers*
  - Use the position of other walkers and their posterior values to propose new values for each walker
  - Repeat until convergence





Walker 15 could depend on walker 0,

but in reality doesn't

Application to MCMC sampling

- Construct compute graph over nTiles = nWalkers

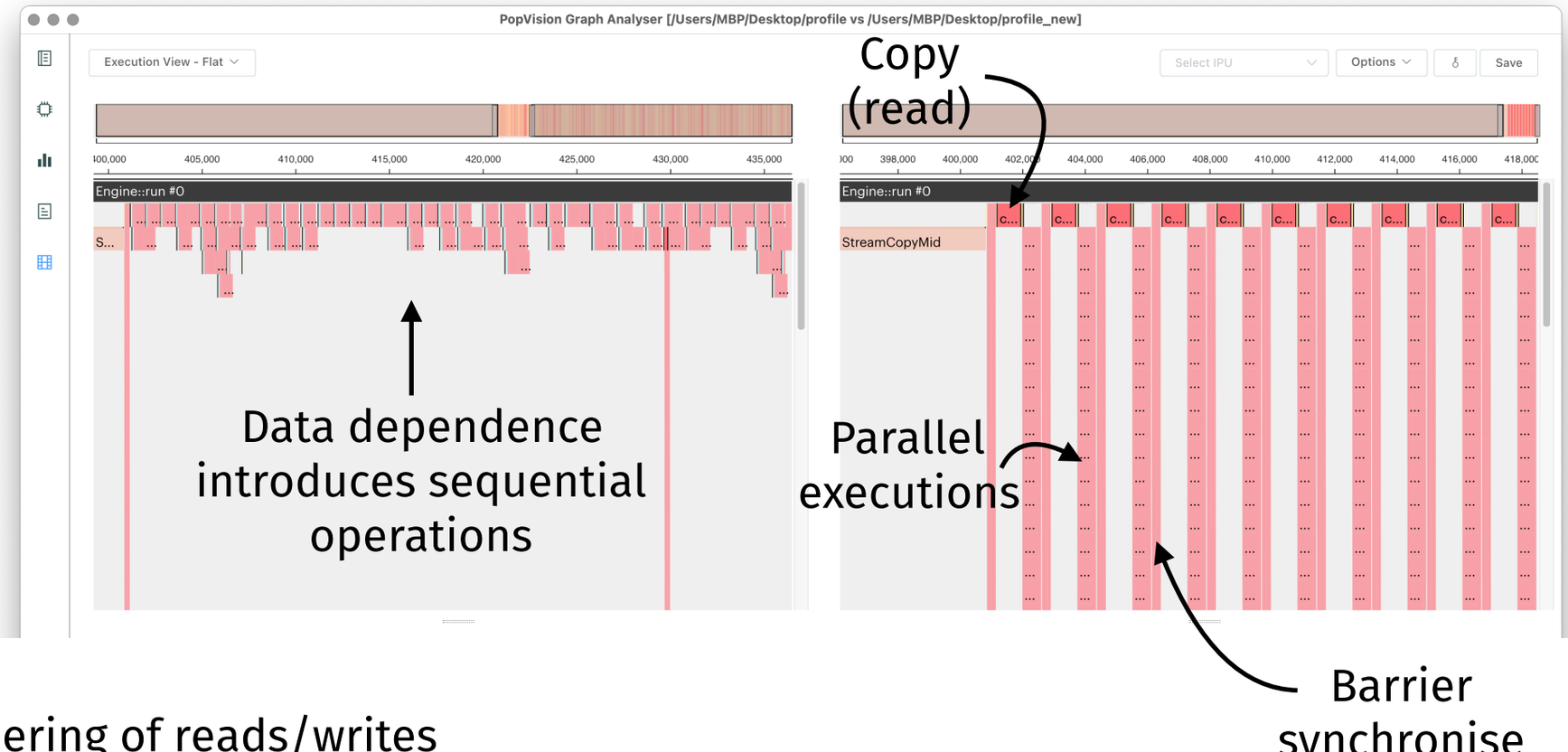
```
for (int tile = 0; tile < nWalkers; tile++) {

    // Separate loop, in case out[rand] location doesn't exist yet
    // Before: generate random uniform
    inSeq.add(Copy(out[tile], thisTheta[tile]));

    // Copy selected 'other' tile parameters to slice of the input array for this tile
    for (int otherIdx = 0; otherIdx < nExchange ; otherIdx++) {
        int fromIndex = exchangeIndices[tile][otherIdx];
        inSeq.add(Copy(out[fromIndex], otherTheta[tile].slice(otherIdx, otherIdx + 1, 0)));
    }

    outSeq.add(Execute(gwCS[tile]));
}
```

Inter-tile copies (indices) pre-specified

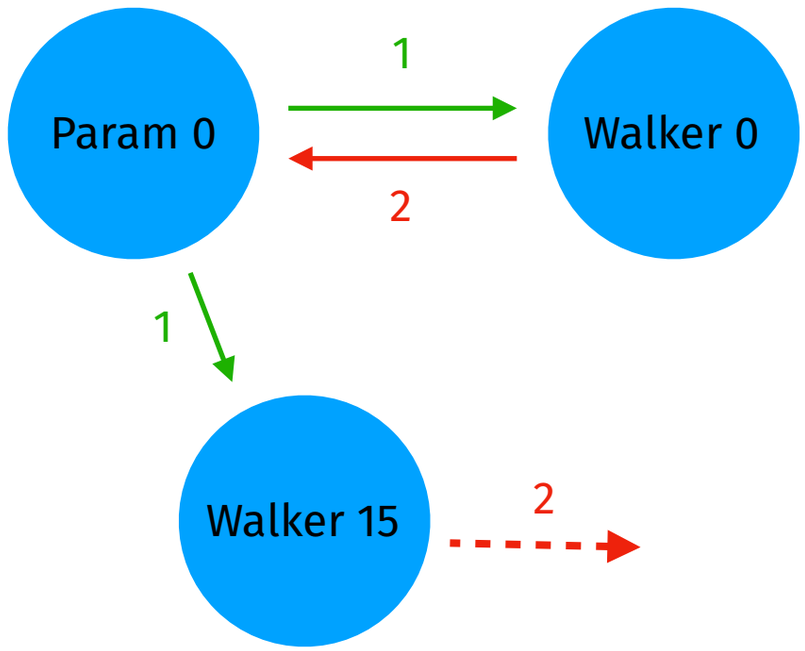
Synchronise *all* tiles after ‘read’ step (probably other more elegant ways to solve this problem)

Before synchronisation

After synchronisation



Enforce ordering of reads/writes

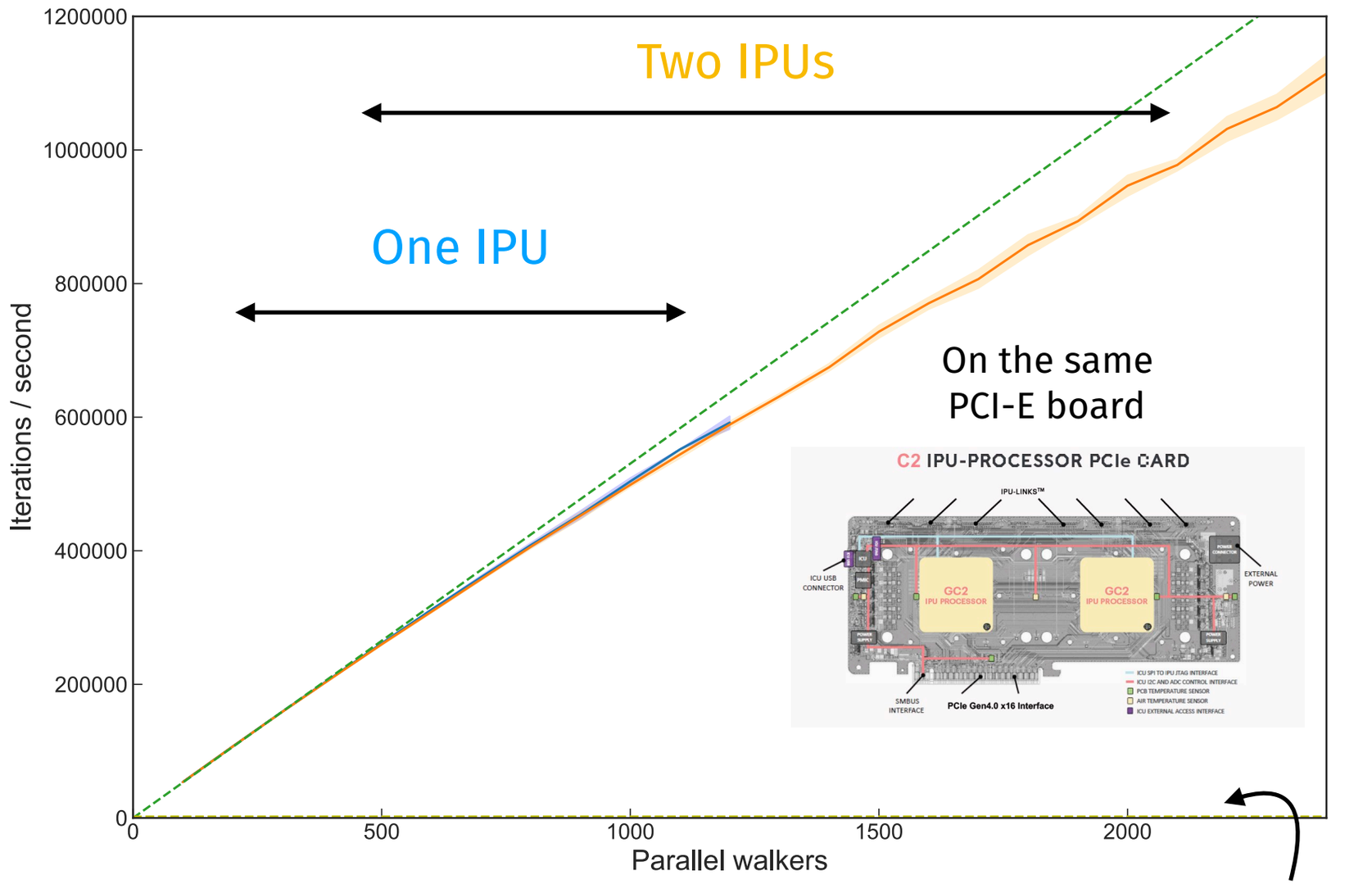



```
294     for (int i = 0; i < nIterations; i++) {
295         progMain.add(inSeq);
296         progMain.add(Sync(poplar::SyncType::INTERNAL));
297         progMain.add(outSeq);
298     }
```

github.com/dpohanlon/congenial-octo-fiesta

Multi-IPU execution

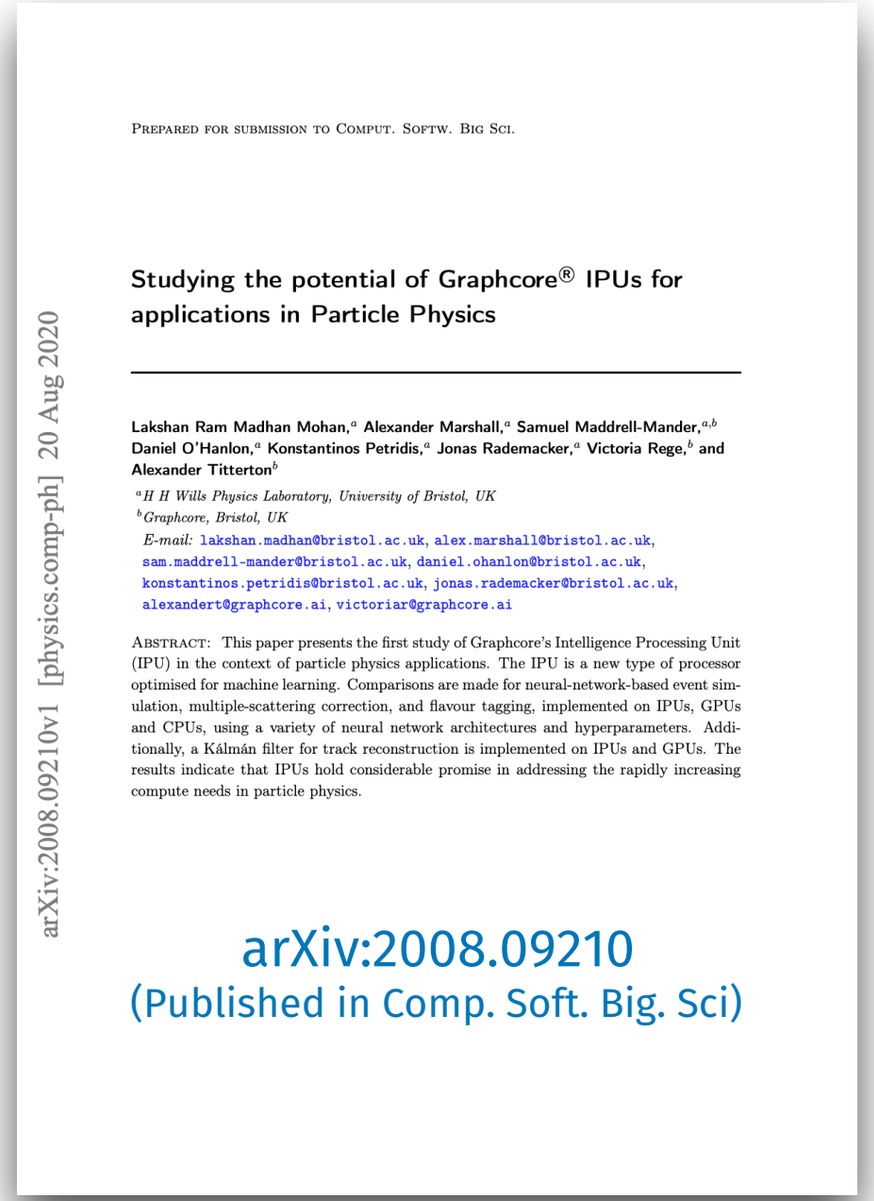
- For code where the graph can be scaled according to the number of tiles, e.g., if ‘embarrassingly parallel’, can seamlessly construct a ‘virtual IPU’ of all those on the same physical device



1 x CPU: ~2000/s
(different implementation)

Other benchmarks

- Studies of HEP related workloads (old Mk1 IPUs)
- Some conventional **ML** applications in **TensorFlow** and **PyTorch** (GANs, RNNs/CNNs for classification)
- Also a basic Kálmán filter implemented in the more flexible C++ interface
- Studies indicate that these outperform P100 GPUs, whilst being more flexible due to MIMD architecture
- Future work will be on HEP workflows for HLT and additional ML applications (*e.g.*, graph nets)



Prepared for submission to Comput. Softw. Big Sci.

Studying the potential of Graphcore® IPUs for applications in Particle Physics

Lakshan Ram Madhan Mohan,[a] Alexander Marshall,[a] Samuel Maddrell-Mander,[a,b] Daniel O'Hanlon,[a] Konstantinos Petridis,[a] Jonas Rademacker,[a] Victoria Rege,[b] and Alexander Titterton[b]

[a] *H H Wills Physics Laboratory, University of Bristol, UK*
[b] *Graphcore, Bristol, UK*
E-mail: lakshan.madhan@bristol.ac.uk, alex.marshall@bristol.ac.uk, sam.maddrell-mander@bristol.ac.uk, daniel.ohanlon@bristol.ac.uk, konstantinos.petridis@bristol.ac.uk, jonas.rademacker@bristol.ac.uk, alexandert@graphcore.ai, victoriar@graphcore.ai

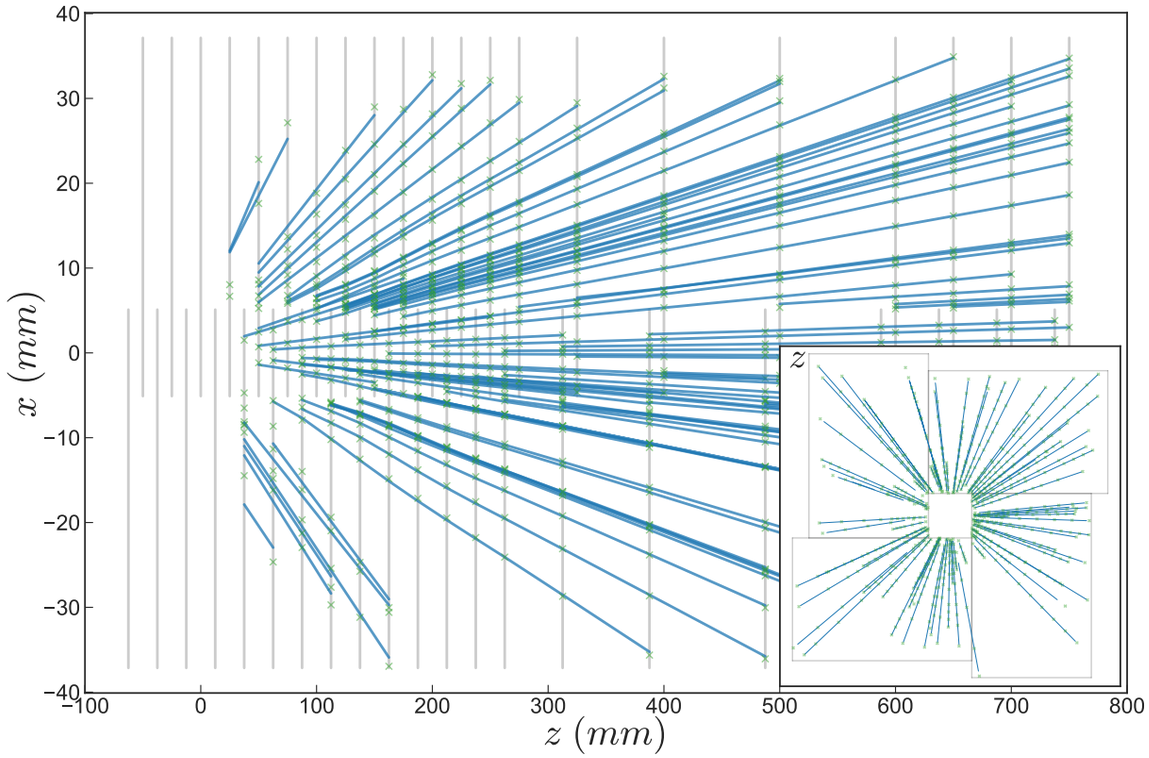
Abstract: This paper presents the first study of Graphcore's Intelligence Processing Unit (IPU) in the context of particle physics applications. The IPU is a new type of processor optimised for machine learning. Comparisons are made for neural-network-based event simulation, multiple-scattering correction, and flavour tagging, implemented on IPUs, GPUs and CPUs, using a variety of neural network architectures and hyperparameters. Additionally, a Kálmán filter for track reconstruction is implemented on IPUs and GPUs. The results indicate that IPUs hold considerable promise in addressing the rapidly increasing compute needs in particle physics.

arXiv:2008.09210
(Published in Comp. Soft. Big. Sci)

Summary

- IPUs billed as an ‘AI accelerator’ but can be used for conventional HEP computing workloads

- Starting to understand the hardware and SDK via ‘toy’ implementations, lots of useful information provided by Bristol HPC group



- Promising applications in particular for amplitude fits, features complementary to GPUs and provide an avenue to increase performance by being more flexible

- Also preparing some documentation and tutorials to widen access and understanding of the hardware within HEP!

Backup

Graphcore APIs

Now public:
docs.graphcore.ai

- API is an important driver for how many FLOPs of useful work can be done
- Current still being worked on very rapidly (with opportunity for us to contribute suggestions)

- 1.x and 2.x supported, with different interfaces

1.x , 2.x

```
# Configure the IPU system
cfg = ipu.utils.create_ipu_config(profiling=True, use_poplar_text_report=True)
cfg = ipu.utils.set_ipu_model_options(cfg, compile_ipu_code=False)
cfg = ipu.utils.auto_select_ipus(cfg, NUM_IPUS)
ipu.utils.configure_ipu_system(cfg)
```

1.x

```
# Create the IPU section of the graph
with ipu_scope("/device:IPU:0"):
  result = ipu.ipu_compiler.compile(sharded_graph, [pa, pb, pc])
```

- Build the model - custom NN operations optimised for IPU execution - CNN, RNN, etc

Graphcore APIs

1.x

```
with tf.Session() as sess:
  sess.run(infeed_queue.initializer)

  sess.run(run_loop)
  result = sess.run(dequeue_outfeed)
  print(result)
```

Queues for data ingestion

1.x

```
def create_ipu_estimator(args):
  ipu_options = ipu.utils.create_ipu_config(
      profiling=False,
      use_poplar_text_report=False,
  )
  ipu.utils.auto_select_ipus(ipu_options, num_ipus=args.replicas)

  ipu_run_config = ipu.ipu_run_config.IPURunConfig(
      iterations_per_loop=args.iterations_per_loop,
      num_replicas=args.replicas,
      ipu_options=ipu_options,
  )

  config = ipu.ipu_run_config.RunConfig(
      ipu_run_config=ipu_run_config,
      log_step_count_steps=args.log_interval,
      save_summary_steps=args.summary_interval,
      model_dir=args.model_dir,
  )

  return ipu.ipu_estimator.IPUEstimator(
      config=config,
      model_fn=model_fn,
      params={
          "learning_rate": args.learning_rate,
          "replicas": args.replicas
      },
  )
```

Wrapper for model, data, training config, etc

Graphcore APIs

2.x easier than 1.x - can use IPUStrategy:

```python
from tensorflow.python import ipu

# Create an IPU distribution strategy
strategy = ipu.ipu_strategy.IPUStrategy()

with strategy.scope():
    ...
```

...and (IPU) Keras out of the box:

```python
#
# Create the model using the IPU-specific Sequential class
#
def create_model():
  m = ipu.keras.Sequential([
      keras.layers.Flatten(),
      keras.layers.Dense(128, activation='relu'),
      keras.layers.Dense(10, activation='softmax')
  ])
  return m
```

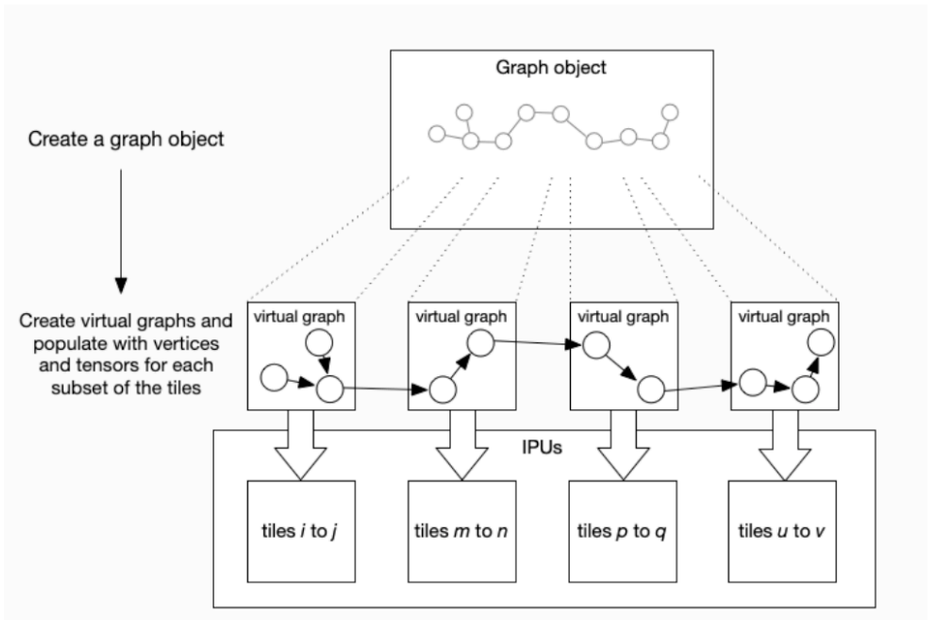
Custom functions require ‘compiled’ XLA operations with tf.function dectorator, not all TensorFlow ops (well) supported

Also guides on porting TensorFlow models, with graph ‘sharding’ over multiple IPUs

Poplar, C++

Graphcore APIs

High-level interface (matmul, scatter, gather, reduce, etc), on Tensors, control of tile mapping is handled by a ‘virtual’ graph (subset of the total program graph):



e.g,

```
poplar::Tensor reduce(poplar::Graph &graph, const  poplar::Tensor &in, const  poplar::Type
&outType, const  std::vector<std::size_t> &dims, ReduceParams params,
std::vector<poplar::ComputeSet> &css, const  std::string &debugPrefix = "", const
poplar::OptionFlags &options = {})
```

Apply a reduction operation to a tensor.

Graphcore APIs

Poplar, C++

Lower-level interface ‘vertex code’, written in vanilla C++, which is compiled to the IPU instruction set and executes on a *single* tile:

E.g, from the Kalman filter,

a 2x2 matrix inverse:

(Lots of high level functionality isn’t present in poplar yet, and inverting should be okay for these matrices)

github.com/dpohanlon/IPU4HEP

```cpp
#include <poplar/Vertex.hpp>

class MatrixInverse2 : public poplar::Vertex {
public:

  // Fields
  poplar::Vector<poplar::Input<poplar::Vector<float>>> in;
  poplar::Vector<poplar::Output<poplar::Vector<float>>> out;

  // Compute function
  bool compute() {

    float det = in[0][0] * in[1][1] - in[0][1] * in[1][0];

    if (det == 0) return false; // Matrix is singular

    out[0][0] =  in[1][1] * (1. / det);
    out[0][1] = -in[0][1] * (1. / det);
    out[1][0] = -in[1][0] * (1. / det);
    out[1][1] =  in[0][0] * (1. / det);

    return true;
  }
};
```

Poplar, C++

Graphcore APIs

Corresponding interface code, to connect the vertex to some inputs and outputs:

```
128  std::tuple<ComputeSet, Tensor> KalmanFilter::inverse(Graph & graph, const Tensor & in, uint tile, uint dim)
129  {
130    ComputeSet computeSet = graph.addComputeSet("inverse" + std::to_string(tile));
131    VertexRef vtx = graph.addVertex(computeSet, "MatrixInverse" + std::to_string(dim));
132    graph.setCycleEstimate(vtx, 100);
133
134    Tensor out = graph.addVariable(FLOAT, {dim, dim}, "invOut" + std::to_string(tile));
135
136    graph.connect(vtx["in"], in);
137    graph.connect(vtx["out"], out);
138    graph.setTileMapping(vtx, tile);
139    graph.setTileMapping(out, tile);
140
141    return std::make_pair(computeSet, out);
142  }
```

Poplar, C++

Graphcore APIs

And then call the ‘vertex’ when building the graph:

Computation

Output

```
239     // inv_C_proj + HG @ H
240     auto [computeHGH, HGH] = product(graph, HG, H[i], i);
241     op.add(Execute(computeHGH));
242
243     auto [computeInverseC, invOutC] = inverse(graph, C[i], i, 4);
244     op.add(Execute(computeInverseC));
245
246     auto [computeAddCHGH, CHGH_HGH] = scaledAdd(graph, invOutC, HGH, i);
247     op.add(Execute(computeAddCHGH));
248
249     auto [computeInverseCHGH, invOutCHGH] = inverse(graph, CHGH_HGH, i, 4);
250     op.add(Execute(computeInverseCHGH));
```

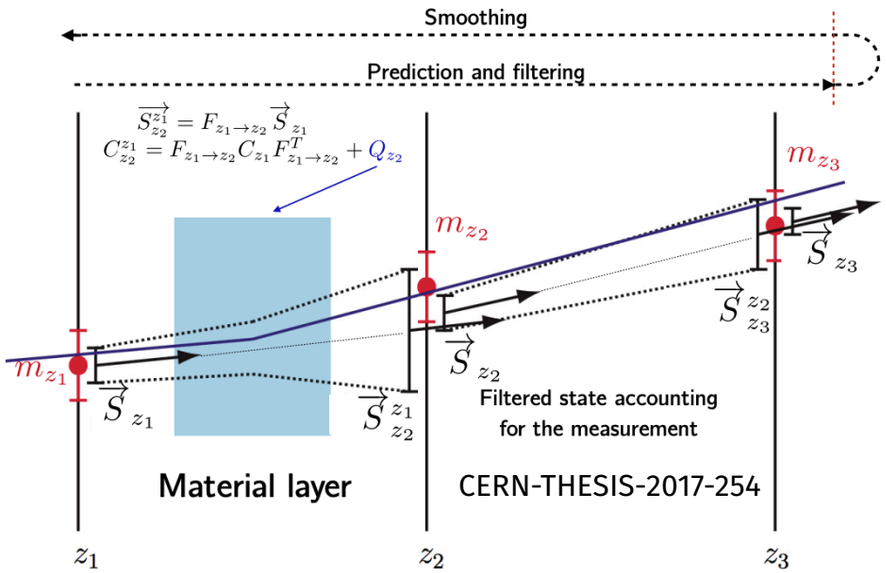
Create a node that executes the computation, puts the result in the output (added to a graph that is then added to a large program graph)

NB: Declarative style - you describe the *structure* of the graph, but nothing gets evaluated yet!

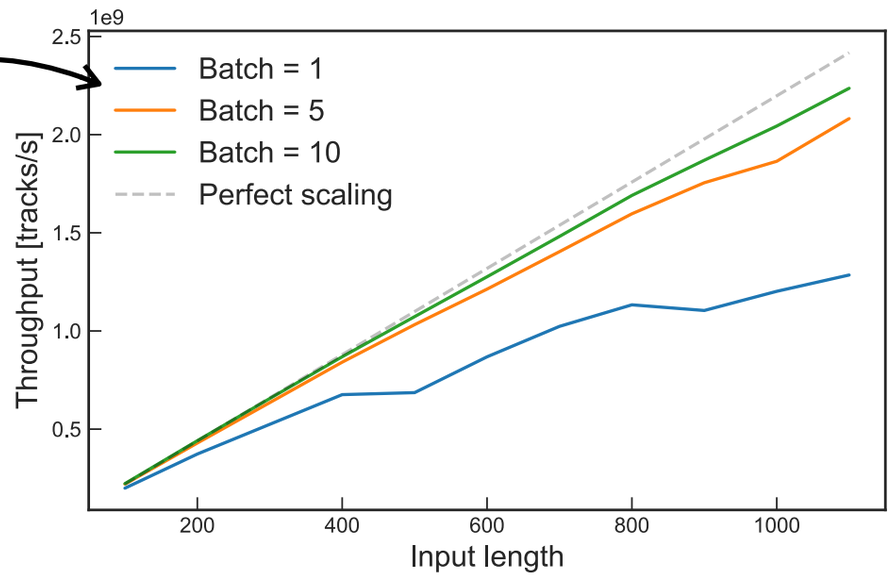
Gives precise control over what executes on which tiles, potentially beneficial over TF interface for fixed parallel structures (the Kalman filter, graph neural networks, etc)

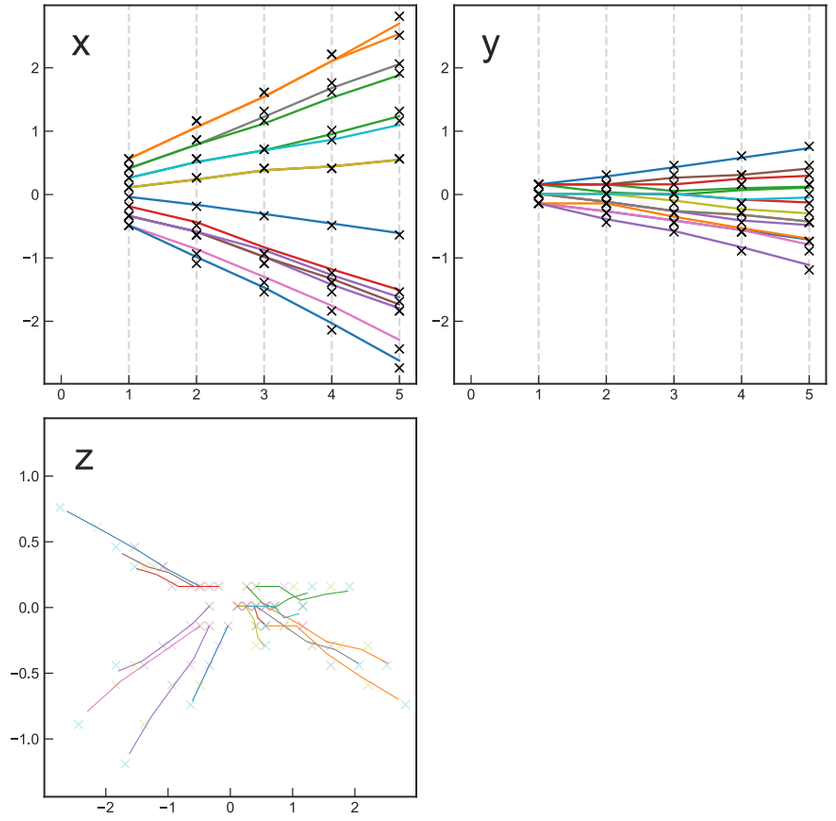
Benchmarks - Kalman filtering

- Simulate simple set of tracking stations, no magnetic field, parameterised homogeneous multiple scattering
- Kalman filtering: Progressive fit of track state to hits at each tracker plane, refined using detector uncertainty and kinematics projection (Bayesian updates)
- Inherently parallel between tracks
- Assign each track to a tile, constrain operations to the respective tile - fit 1216 tracks in parallel
- Poplar IPU implementation:



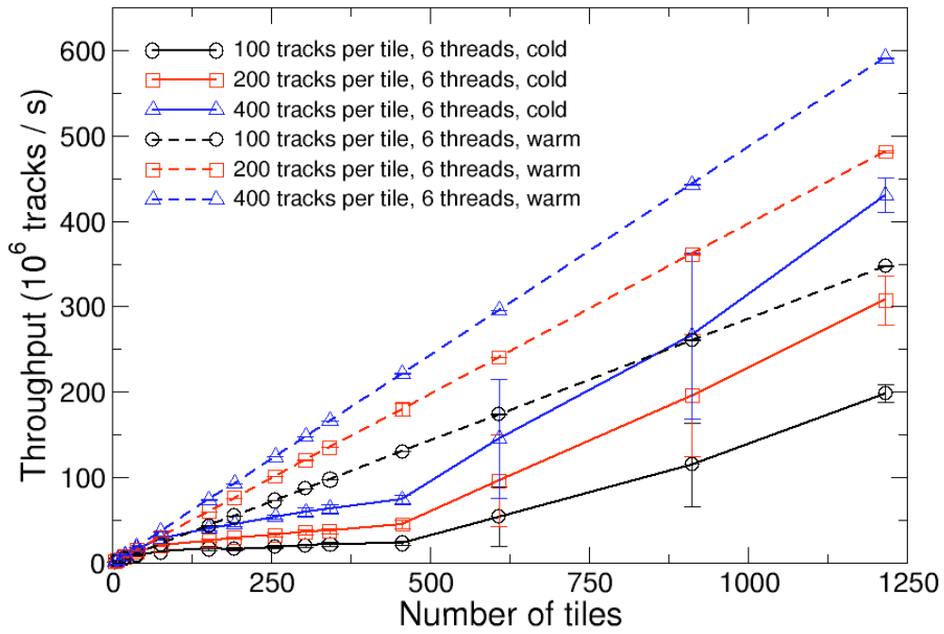
Reduce overhead by copying n batches each time

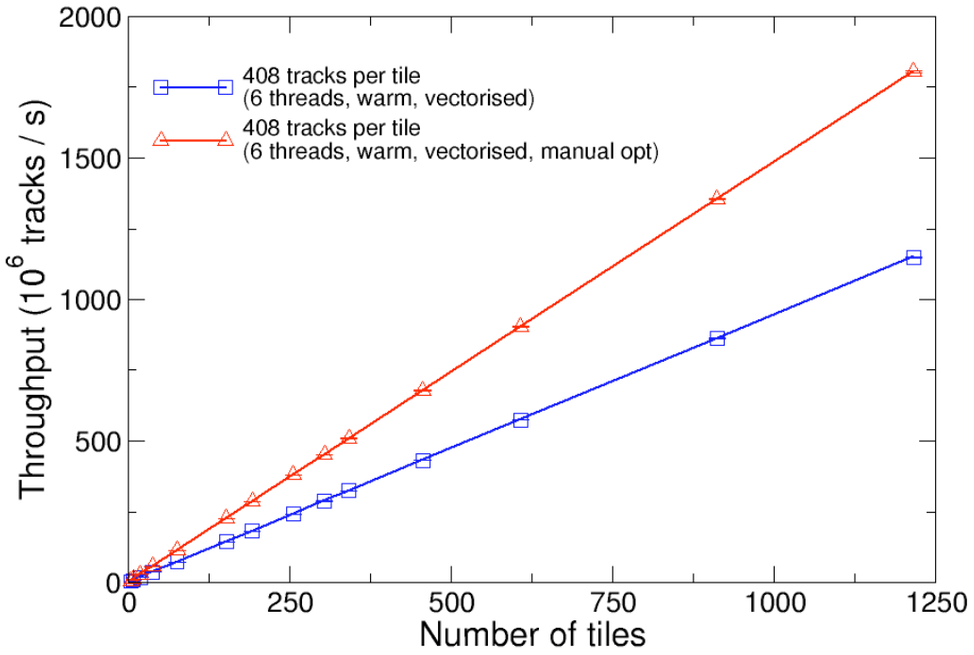




Kalman filtering redux

- Re-implemented by Lester to execute the full Kalman filter (per event) on a single tile: github.com/lohedges/trackr





- A number of ‘easy win’ optimisations:
  - Task based parallelism by oversubscribing compute vertices to tiles (6 HW threads)
  - ‘Vectorisation’ by packing 32 bit floats into 64 bit operands
  - Task specific optimisations (loop unrolling, etc)
- Additional tips from Bristol HPC group:

  github.com/UoB-HPC/ipu-hpc-cookbook

You all know what an RBW looks like, but here it is in vertex code:

```
bool compute() {

    // Breaking the law
    auto dp = reinterpret_cast<::DP *>(&dpInput[0]);

    int size = mass.size();

    for (int i = 0; i < size; i++) {

        float mass = sqrt(mass2);

        auto [cosTheta, p, q] = cosThetaPQ(*(this->dp), mass2, otherMass2);
        auto [_, p0, q0] = cosThetaPQ(*(this->dp), this->resMass2, otherMass2);

        float spinTerm = this->calcSpinTerm(cosTheta);

        float qTerm = this->calcQTerm(q, q0);

        auto [ffRatioP, ffRatioR] = calcBarrierTerms(p, q, p0, q0);

        float totWidth = this->resWidth * qTerm;
        totWidth *= this->resMass / mass;
        totWidth *= ffRatioP * ffRatioR;

        float m2Term = this->resMass2 - mass2;

        float scale = spinTerm;
        scale /= m2Term * m2Term + this->resMass2 * totWidth * totWidth;
        scale *= ffRatioP * ffRatioR;

        ampRe[i] = m2Term * scale;
        ampIm[i] = resMass * totWidth * scale;

    }
```