



INTRODUCTION TO THE IPU GRAPH COMPILER AND THE USE OF LLVM

DAVID BOZIER

THOMAS PREUD'HOMME

THE BOW IPU

WORLD'S FIRST 3D WAFER-ON-WAFER PROCESSOR



3D silicon wafer stacked processor

350 TeraFLOPS AI compute

Optimized silicon power delivery

0.9 GigaByte In-Processor-Memory @ **65TB/s**

1,472 independent processor cores (tiles)



8,832 independent parallel programs

10x IPU-Links™ delivering 320GB/s

IPU – ARCHITECTURED FOR AI

Massive parallelism with ultrafast memory access

Parallelism

Processors 
Memory 

Memory Access

CPU

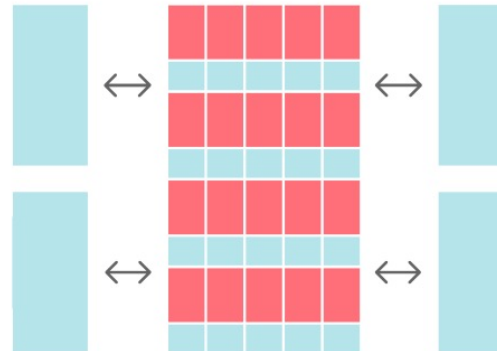
Designed for scalar processes



Off-chip memory

GPU

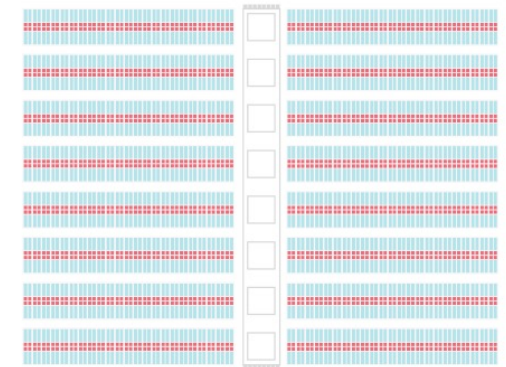
SIMD/SIMT architecture. Designed for large blocks of dense contiguous data



Model and data spread across off-chip and small on-chip cache, and shared memory

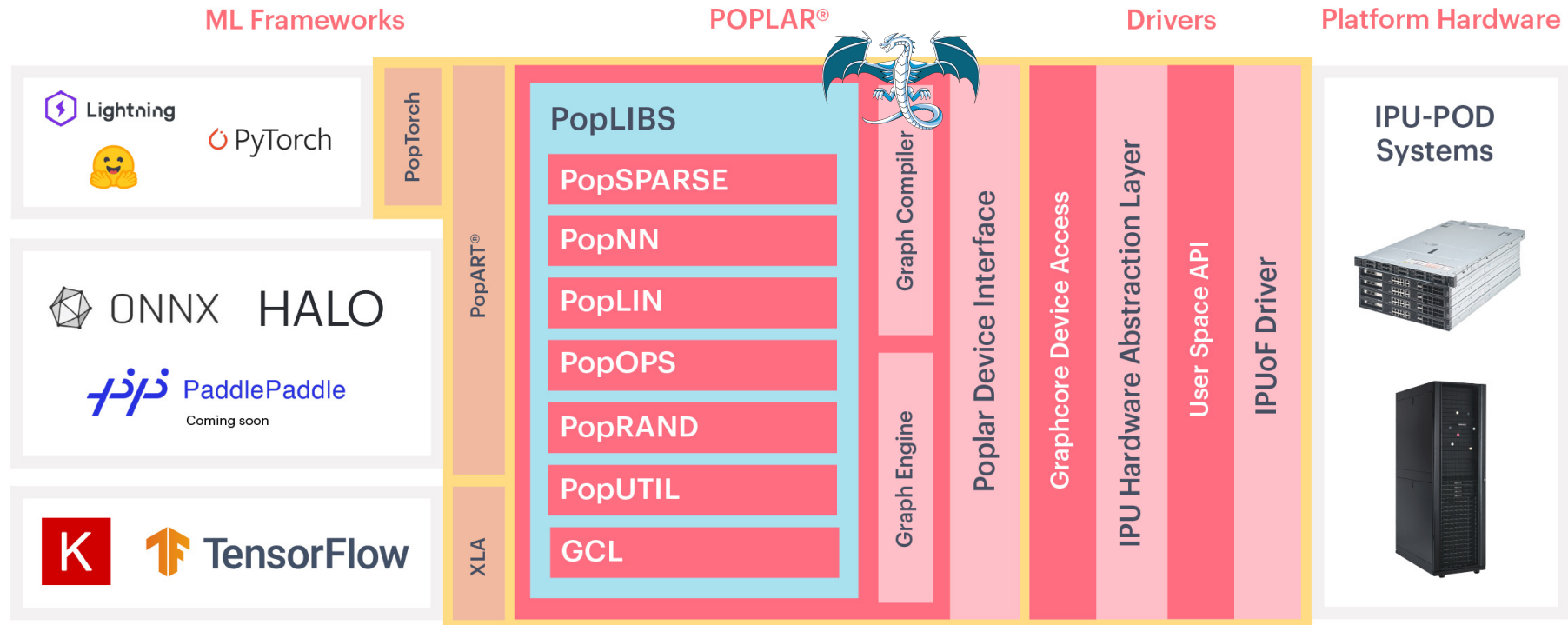
IPU

Massively parallel MIMD. Designed for fine-grained, high-performance computing



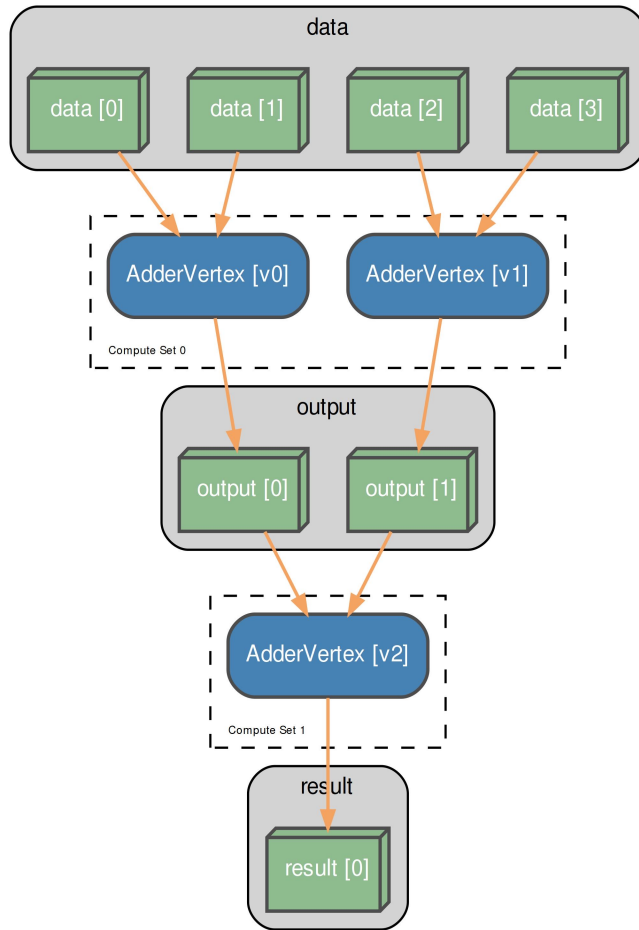
Model and data tightly coupled, and large locally distributed SRAM

POPLAR[®] SDK

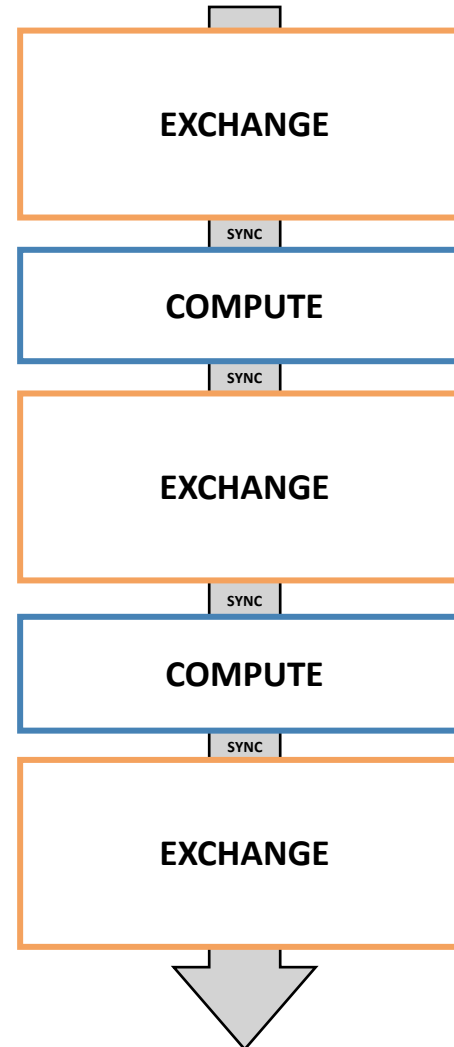


COMPUTATION GRAPH AND EXECUTION MODEL

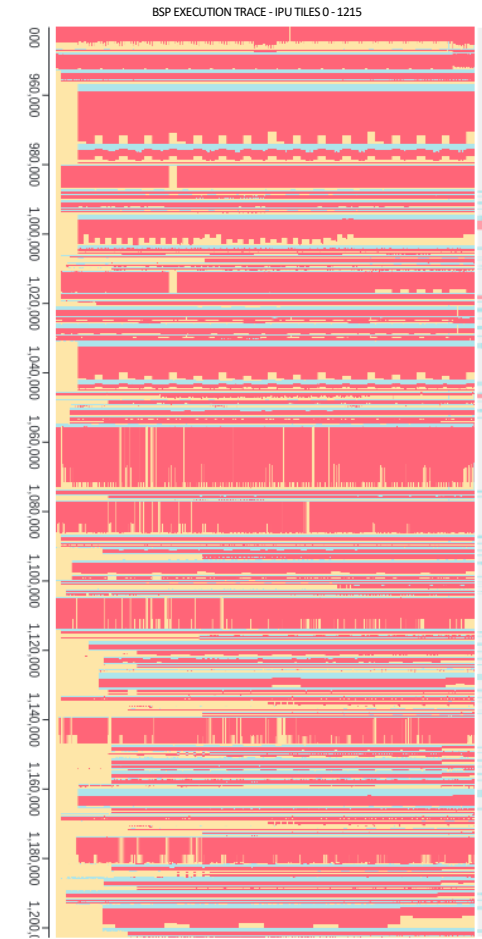
COMPUTATIONAL GRAPH



BSP SCHEDULE



OPTIMIZED IPU EXECUTION



OUTPUT FROM POPVISION GRAPH ANALYSER

POPLAR® HOST PROGRAM

- Constructs Poplar® compute graph, tensors, compute sets, Poplar® engine
- Controls tile-mapping of tensors, vertices
- Acquires/pre-processes data on the CPU
- Creates stream copies to send data to/from the IPU
- Poplar® engine sends instructions to the IPU

```
Graph graph(target);

graph.addCodelets("SumVertex.cpp");

Tensor v1 = graph.addVariable(FLOAT, {4}, "v1");
Tensor v2 = graph.addVariable(FLOAT, {4}, "v2");

Sequence prog;

Tensor c1 = graph.addConstant<float>(FLOAT, {4}, {1.0, 1.5, 2.0, 2.5});
prog.add(Copy(c1, v1));

ComputeSet computeSet = graph.addComputeSet("computeSet");
for (unsigned i = 0; i < 4; ++i) {
    VertexRef vtx = graph.addVertex(computeSet, "SumVertex");
    graph.connect(vtx["in"], v1.slice(i, 4));
    graph.connect(vtx["out"], v2[i]);
    graph.setTileMapping(vtx, i);
}

prog.add(Execute(computeSet));
prog.add(PrintTensor("v2", v2));

Engine engine(graph, prog);
engine.load(device);
engine.run(0);
```



CODELET DEFINITIONS

- Codelets written in C++ 17
- Compute functions single threaded
- Vertices are written as a C++ class that inherit from the **Vertex** class
- The fields of the vertex specify the inputs, output and internal data
- The **compute** method specifies the vertex execution behaviour
- Compute functions can also be written in assembly by adding **isExternalCodelet** field
- Many Vertices provided Poplar® Libraries

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

class SumVertex : public poplar::Vertex {
public:
    // Fields
    poplar::Input<poplar::Vector<float>> in;
    poplar::Output<float> out;
    float bias;
    // Compute function
    bool compute() {
        *out = bias;
        for (const auto &v : in) {
            *out += v;
        }
        return true;
    }
};
```

ADDING A CODELET TO THE GRAPH

Codelets are processed in 2 stages:

First we analyze the codelet using **libclang** where we:

- Verify vertices are well formed
- Create metadata of the vertex which is used by poplar for connecting tensor data to the vertex fields
- Add a vertex wrapper in source as an entry point

Then we create a **CompilerInstance** to compile the codelet to generate optimized IPU machine code

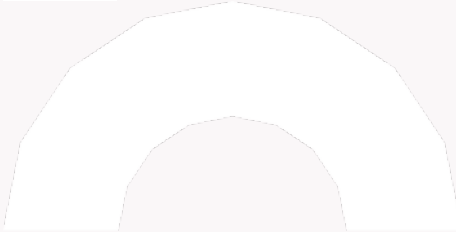
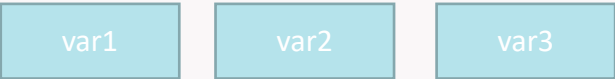
```
graph.addCodelets("SumVertex.cpp");
```

```
__attribute__((target("worker")))
__attribute__((colossus_vertex))
int << __runCodelet_SumVertex() {
    void *vertexPtr = __builtin_ipu_get_vertex_base();
    auto v = static_cast<SumVertex*>(vertexPtr);
    return v->compute();
}
```

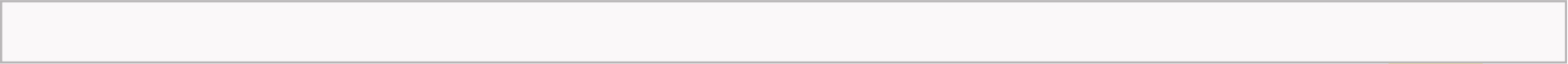

Compiled Vertices



Variables



Tile 0



Compiled Vertices



Variables



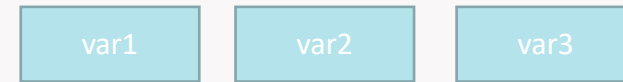
Tile 0



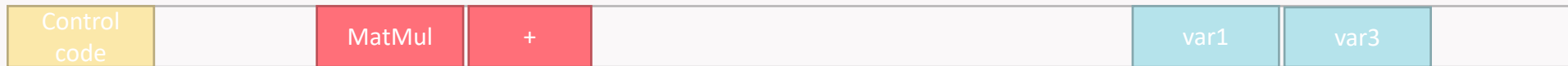
Compiled Vertices



Variables



Tile 0

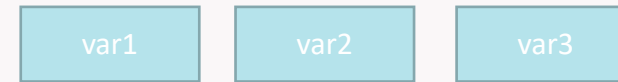


- Control code added that spawns and runs thread on the tile

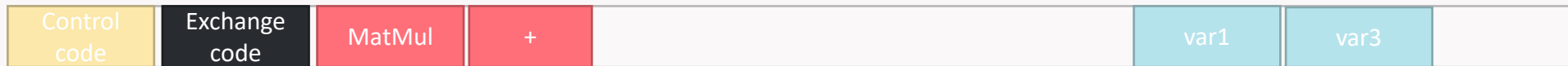
Compiled Vertices



Variables

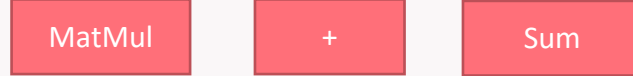


Tile 0



- Control code added that spawns and runs thread on the tile
- Exchange code for communication between tiles

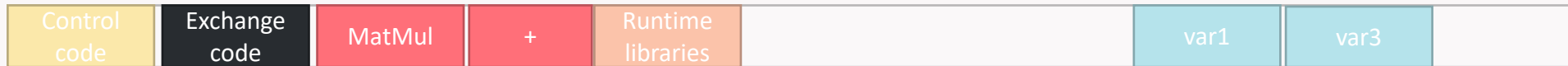
Compiled Vertices



Variables



Tile 0

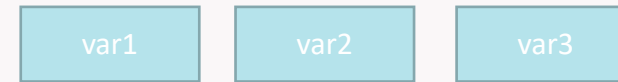


- Control code added that spawns and runs thread on the tile
- Exchange code for communication between tiles
- Some additional runtime library functions may also be required

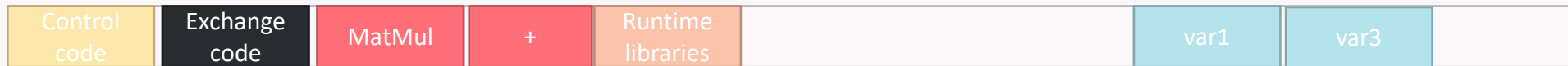
Compiled Vertices



Variables



Tile 0



Tile 1



...

Tile 1471



Repeat for all tiles

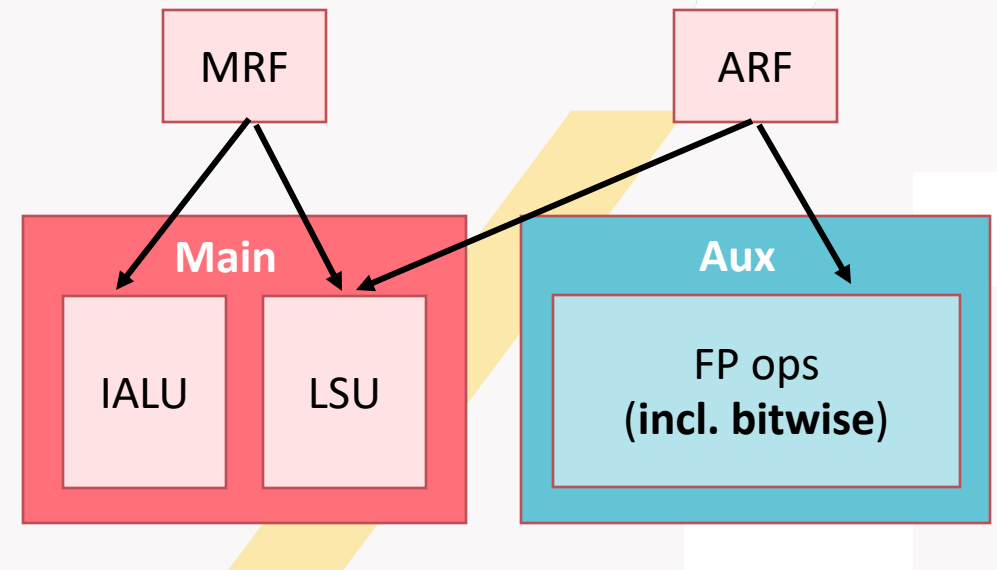


LLVM specifics

Bitwise operations

IPU tiles are superscalar:

- 2 instruction pipelines (main & aux)
- Pipeline strongly associated with register file
- ARF -> MRF via atom, MRF -> ARF via spill
- Bitwise ops available on both pipeline



Challenge: how to reduce register file changes using bitwise operations on ARF

Bitwise operation placement

Key ideas of algorithm:

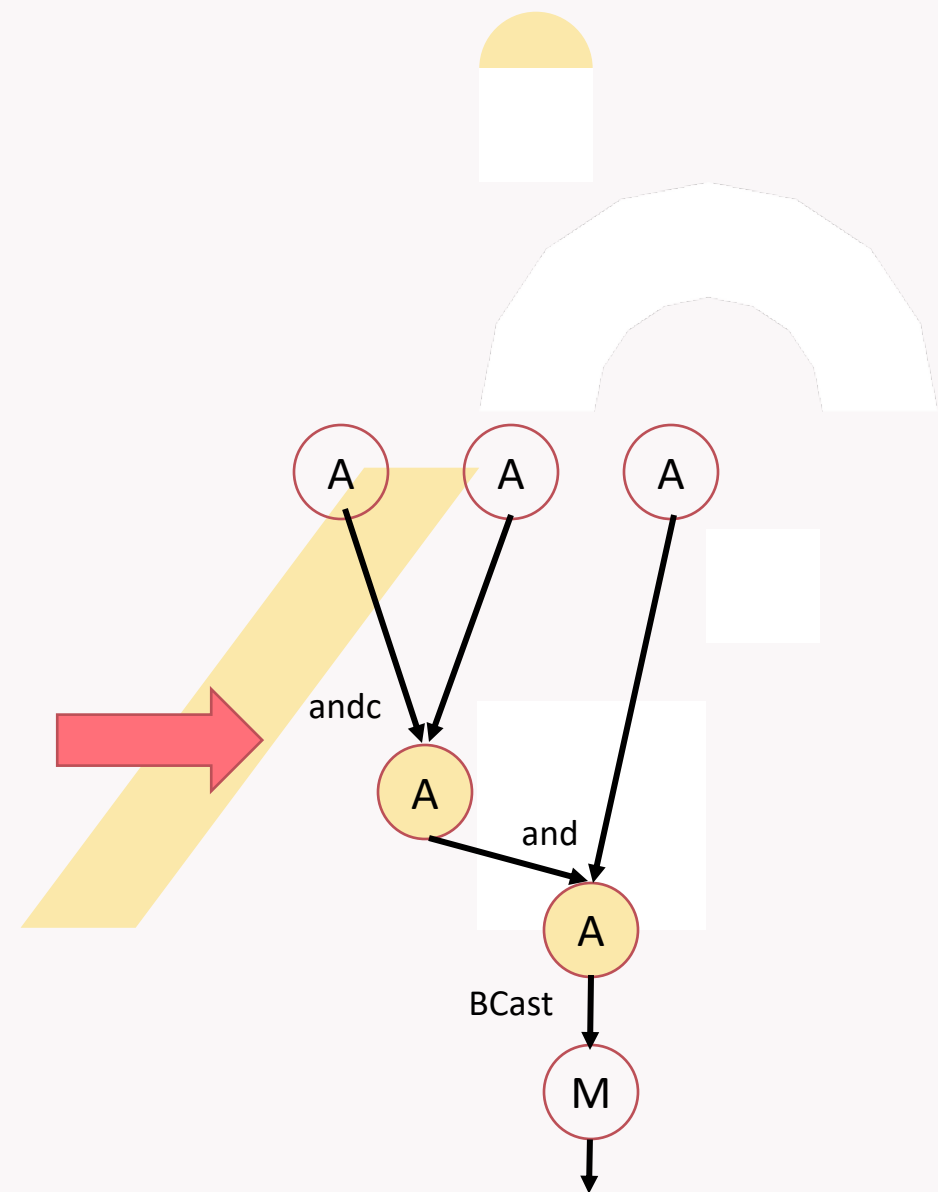
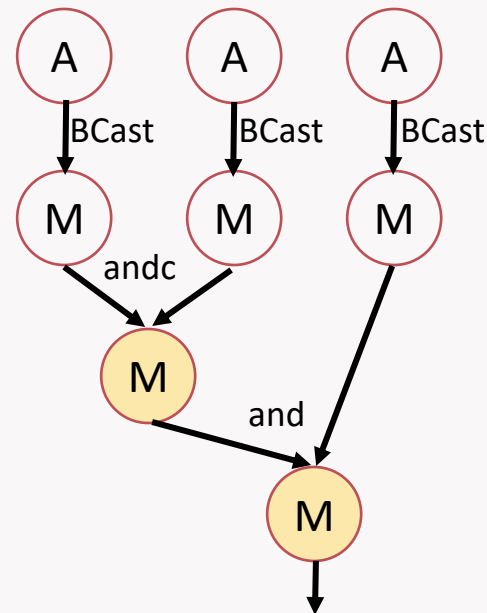
- SDAG nodes have pipeline preference (main, aux or either)
- Preference is also function of operands
- For a binop, compare its current pipeline with preference of its operands

A Aux

M Main

Single pipeline

Either pipeline



Runtime environment overview

Compared to typical desktop systems, IPU runtime environment has some important differences:

- No operating system
- No dynamic allocation
- 624KiB memory

C++ freestanding proposal addresses the first aspect, but more work needed for the rest.



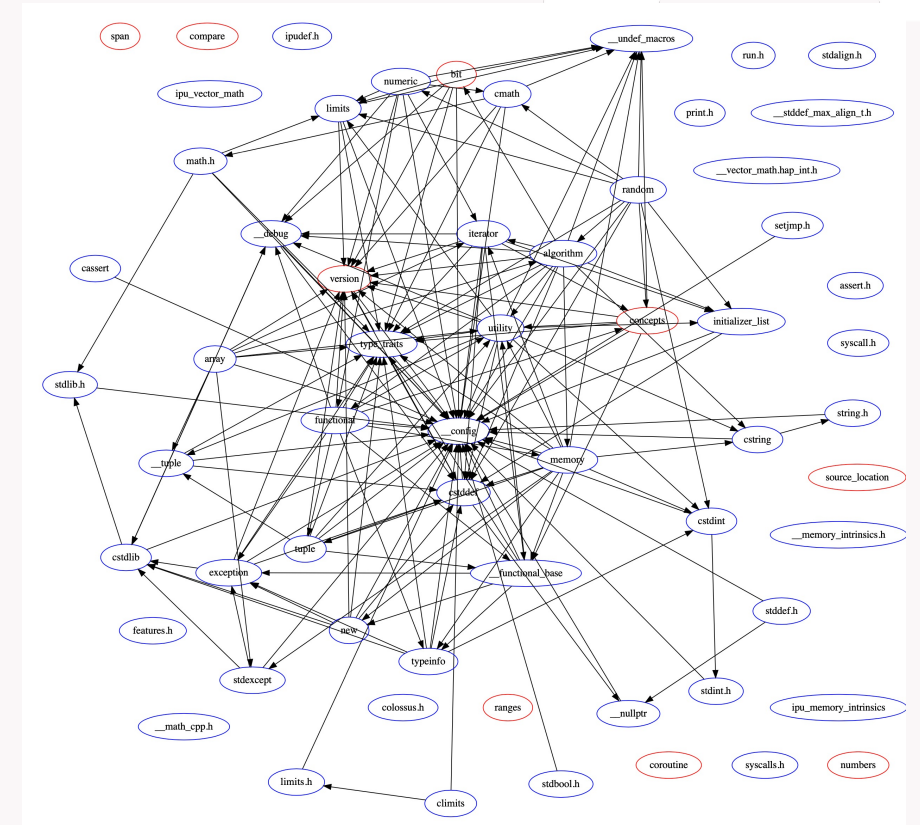
Libcxx changes

<random>:

- Reduce memory usage of components to fit in tile memory
- Disable PRNGs using dynamic memory allocation
- Adapt some of the predefined values to increase quality of numbers with reasonable size requirements

Misc:

- Replace `std::vector` by `std::array` in tests
- Remove bits to reduce include dependencies
- Copy `libcxx` in separate repo to reduce merge frequency



<ap_int.h>

IPU supports hardware loop: [see Janek van Oirschot's talk](#)

How to hint trip count bounds to help hardware loop generation?

Answer 1: `__builtin_assume()` but does not always work

Answer 2: `_BitInt` (C23 proposal) and lots of template magic

```
template<std::size_t N, bool is_signed>
struct ap_int {
public:
    using int_type = std::conditional_t<
        is_signed, _BitInt(num_bits), unsigned _BitInt(num_bits)>;
    // operator overloads

private:
    int_type value;
}
```



LLDB

Supports multi-tile applications as a multi-process application
=> 1 tile = 1 process

Augmented with IPU-specific commands:

- `app` -- Global commands (e.g., continue, interrupt)
- `log` -- Control IPU device logging
- `soc` -- Read specified SoC register(s)
- `thread` -- Tile threads commands (e.g., list, status)
- `tile` -- Tile commands (e.g., attach, select, list)



Positive experiences

A lot of features worked out of the box:

- Middle end optimisations & C++ support
- `_BitInt` (ex `_ExtInt`)
- Builtins
- Pragma and attribute (e.g., `nounroll` pragma and `noinline` attribute)

Also, good experience with community response to patches:

- Usually review in a timely manner
- Sympathetic to out-of-tree targets issues that can be solved without affecting in-tree targets and without maintenance burden

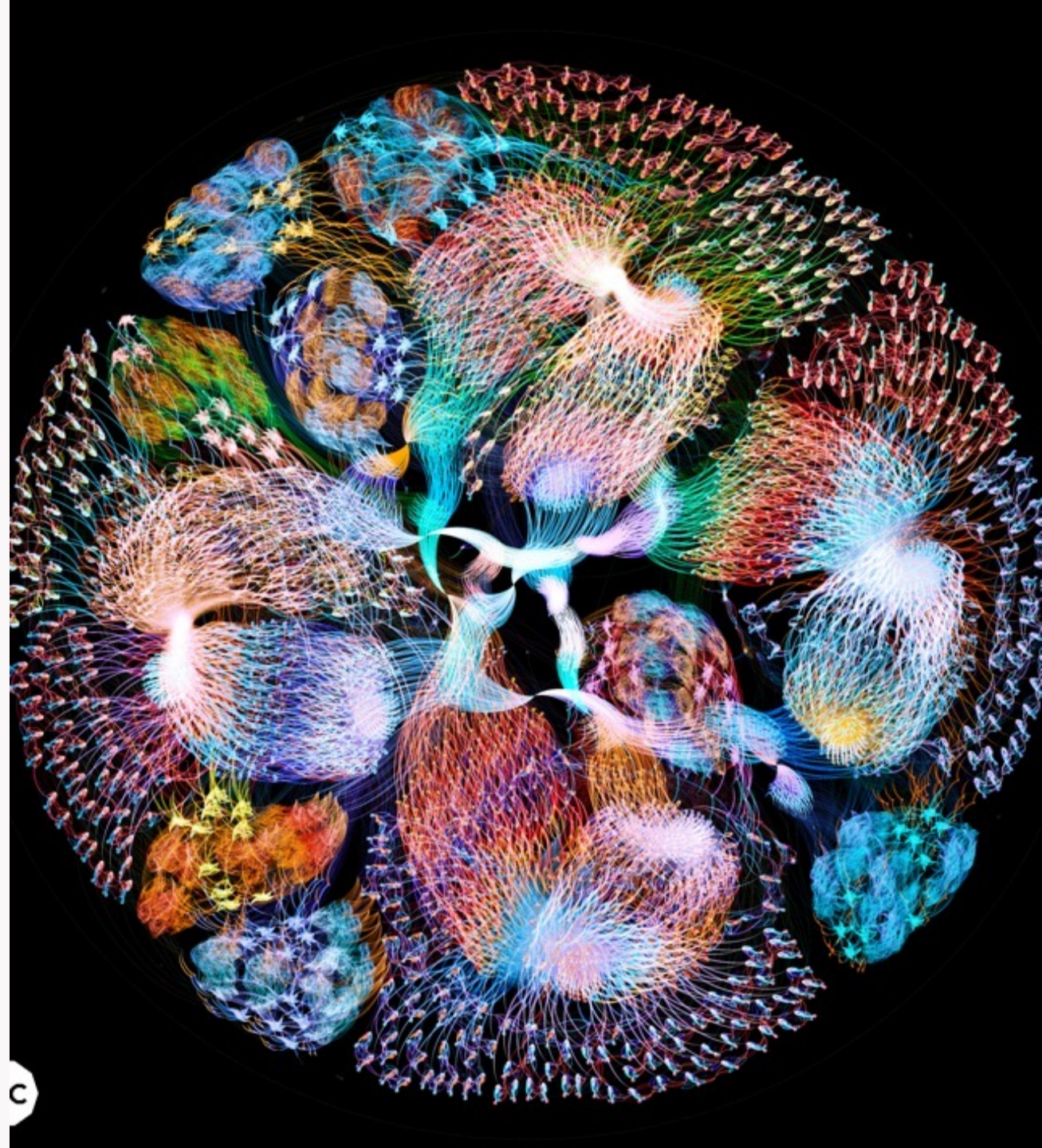
RESOURCES

Find out more about us including documentation and source code at:

www.graphcore.ai

<https://github.com/graphcore>

Open source fork of our LLVM backend will be available very soon



The slide features a light gray background with large, stylized red arches at the top and bottom. A white rectangular box is positioned in the center, containing the text "THANK YOU".

THANK YOU