GRAPHCORE

Poplar and PopLibs User Guide

Version latest

Graphcore Ltd

Aug 25, 2021

CONTENTS

1 Introduction **1**

2 Programming with Poplar **2**
- 2.1 Poplar programming model . . . 2
- 2.2 The structure of a Poplar program . . . 4
  - 2.2.1 Program flow control . . . 5
  - 2.2.2 What happens at run time . . . 5
- 2.3 Supported types . . . 6
  - 2.3.1 Scalar types . . . 6
  - 2.3.2 Floating point types . . . 6
  - 2.3.3 Vector types . . . 7
  - 2.3.4 Structure types . . . 8
  - 2.3.5 Bit fields . . . 8
- 2.4 Virtual graphs . . . 9
- 2.5 Replicated graphs . . . 10
  - 2.5.1 Replicating an existing graph . . . 10
  - 2.5.2 Creating a replicated graph . . . 11
- 2.6 Data streams and remote buffers . . . 12
  - 2.6.1 Data streams . . . 12
  - 2.6.2 Remote memory buffers . . . 13
  - 2.6.3 Stream buffer size limit . . . 13
  - 2.6.4 Optimising host data transfers . . . 13
- 2.7 IPU-Link and sync configuration . . . 14
  - 2.7.1 Link topologies . . . 14
  - 2.7.2 Sync groups . . . 14
  - 2.7.3 Software sync . . . 15
- 2.8 Device code . . . 16
  - 2.8.1 Stack allocation . . . 16
  - 2.8.2 Pre-compiling codelets . . . 17

3 Understanding vertices **18**
- 3.1 The Vertex class . . . 18
- 3.2 Vertex state . . . 18
  - 3.2.1 Vector and VectorList types . . . 18
  - 3.2.2 Specifying memory constraints . . . 18
- 3.3 MultiVertex worker threads . . . 19
  - 3.3.1 Thread safety . . . 20
- 3.4 Calling conventions . . . 20
  - 3.4.1 External codelets . . . 21
  - 3.4.2 Recursion and function pointers . . . 21
- 3.5 Vertex name mangling . . . 21

4 Vector types **23**
- 4.1 Parameters . . . 23

4.1.1 Types . . . 23
4.1.2 Layout . . . 23
4.1.3 Minimum alignment . . . 24
4.1.4 Interleaved memory . . . 25
4.2 Memory layout for vectors . . . 25
4.2.1 Pointer compression . . . 25
4.2.2 Vector<T> layout . . . 26
4.2.3 Vector<Input<Vector<T>>> layout . . . 26
4.2.4 VectorList layout . . . 28

5 Using the Poplar library 33

6 The PopLibs libraries 34
6.1 Using PopLibs . . . 34

7 Writing vertices in assembly 35
7.1 Instruction set overview . . . 37
7.1.1 Supervisors and workers . . . 37
7.1.2 Execution pipelines . . . 38
7.2 Memory architecture . . . 39
7.2.1 Getting information about the memory . . . 40
7.2.2 Mk1 Colossus (GC2) . . . 40
7.2.3 Mk2 Colossus (GC200) . . . 41
7.2.4 Load and store instructions . . . 42
7.3 Worker stack and scratch space . . . 43
7.3.1 Specifying stack size . . . 44
7.3.2 Examples . . . 44
7.4 Vertex pipelines . . . 45
7.4.1 Memory conflicts . . . 46
7.4.2 Modified pipeline . . . 47
7.4.3 Fill and drain . . . 48
7.5 Assembly hints & tips . . . 50
7.5.1 Using the assembler . . . 50
7.5.2 Architectural tips . . . 51
7.5.3 Division by 6 . . . 53
7.5.4 General . . . 54

8 Profiling 55
8.1 Generating profiling information . . . 55
8.1.1 Profiling options . . . 56
8.2 Profile summary . . . 56
8.2.1 Printing from a Poplar program . . . 57
8.2.2 Command line conversion . . . 57
8.2.3 Summary report format . . . 57

9 Environment variables 63
9.1 Logging . . . 63
9.1.1 Logging level . . . 63
9.1.2 Logging destination . . . 64
9.2 Profiling output . . . 64
9.3 Setting options . . . 64

10 Application binary interface (ABI) 65
10.1 Types . . . 65
10.1.1 Floating point types . . . 66
10.1.2 Structure types . . . 66
10.1.3 Bit fields . . . 66
10.2 Vertex calling convention . . . 67
10.3 Function calling convention . . . 67

10.3.1 Function parameters . . . 67
10.3.2 Return values . . . 67
10.3.3 Entry and exit . . . 67
10.3.4 Register assignments . . . 68
10.4 Stack frame . . . 68

11 Trademarks & copyright **69**

CHAPTER

ONE

INTRODUCTION

Poplar™ is a graph-programming framework for the Graphcore Intelligence Processing Unit (IPU), a new type of processor aimed at artificial intelligence and machine learning applications. An overview of the IPU architecture and programming model can be found in the IPU Programmer's Guide. You should familiarize yourself with this document before reading this guide.

The Poplar SDK includes tools and libraries to support programming the IPU. The Poplar SDK libraries provide a C++ interface. Poplar also supports industry-standard machine learning frameworks such as TensorFlow, MXNET, ONNX, Keras, and PyTorch which can be accessed from Python.

There are a number of example programs included with the SDK in the `examples` directory of the Poplar installation. Further examples, tutorials and benchmarks are available on the Graphcore GitHub.

The Poplar library provides classes and functions to implement and deploy parallel programs on the IPU. It uses a graph-based method to describe programs and, although it can be used to describe arbitrary accelerated computation, it has been designed specifically to suit the needs of artificial intelligence and machine learning applications.

The PopLibs libraries are a set of application libraries that implement operations commonly required by machine learning applications, such as linear algebra operations, elementwise tensor operations, non-linearities and reductions. These provide a fast and easy way to create programs that run efficiently using the parallelism of the IPU.

There are several command line tools to manage the IPU hardware. These are described in the "Getting Started" guide for your IPU system and the IPU Command Line Tools document.

CHAPTER

TWO

PROGRAMMING WITH POPLAR

You can use Poplar library functions to define graph operations and control the execution and profiling of code on the IPU.

Code can be compiled to run on IPU hardware, a simulated *IPU Model* or the host CPU. Running on an IPU Model or the CPU may be useful for functional testing of simple code when you do not have access to IPU hardware.

The IPU Model is a simulation of the *behaviour* of the IPU hardware. It does not completely implement every aspect of a real IPU. For example:

- The IPU Model does not fully support replicated graphs (see *Replicated graphs*).
- The arithmetic results may differ from what would be obtained by using the IPU hardware.
- Random number generation in the IPU Model is not the same as the hardware. In particular, every simulated tile has the same hard-coded seed (the `setSeed()` function is a no op). This means all IPU Model codelets will produce the same results every time they are run. Therefore, the IPU Model should **not** be used to verify any training or accuracy if the graph includes any random number generation.

If you encounter an out of memory error, it may be useful to run on the IPU Model device to debug the problem.

Consider the situation in which the event trace is being used to investigate a graph that creates a tile memory imbalance. In this case, running on the IPU will lead to an out of memory exception before the report is generated. Running on the IPU Model instead of actual hardware will still run out of memory, but the code will run to completion so the report can be generated.

Code running on a CPU device will be faster than the IPU Model, because it does not have the overhead of modelling the IPU hardware. CPU code runs with a single worker thread as if on a single tile on a single IPU. This means you do not need to think about tile allocation or the limited tile memory when initially developing vertex code. Running on a CPU device may also be useful for unit testing of vertices.

Interrogating a CPU device by calling `Engine` functions such as `getBytesPerTile()` or `getTileClockFrequency()` may not return accurate or meaningful results.

If you want to profile your code, you will need to run on either IPU hardware or the IPU Model.

2.1 Poplar programming model

For a more detailed introduction to the IPU architecture and programming model, see the IPU Programmer's Guide.

A Poplar computation graph defines the input/output relationship between variables and operations. Each variable is a multi-dimensional tensor of typed values and can be distributed across multiple tiles.

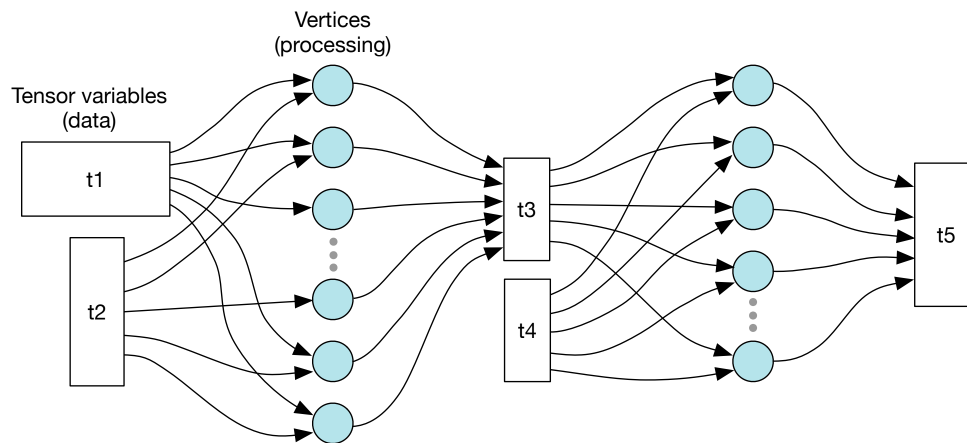


Fig. 2.1: Graph representation of variables and processing

The vertices of the graph are the code executed in parallel by the tiles. Each tile executes a sequence of steps, which form a *compute set* containing one or more vertices.

The edges of the graph define the data that is read and written by the vertices. Each tile only has direct access to the tensor elements that are stored locally.

Each vertex always reads and writes the same tensor elements. In other words, the connections defined by the execution graph are static and cannot be changed at run time. However, the host program can calculate the mapping and graph connectivity at run time when it constructs the execution graph. See Poplar tutorial 7 on the Graphcore GitHub for an example.

The placement of vertices and tensor elements onto tiles is known as the `tile mapping`.

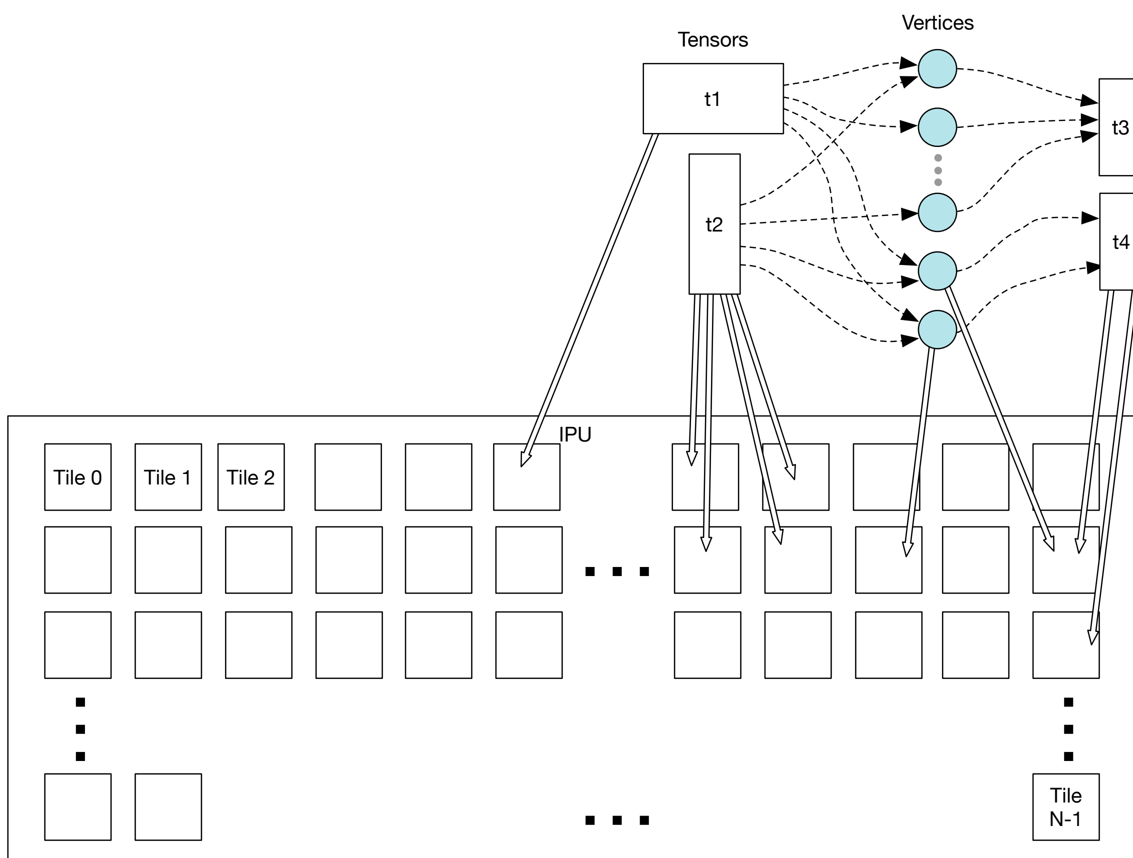


Fig. 2.2: Mapping tensors and vertices to tiles

2.2 The structure of a Poplar program

A Poplar program performs the following tasks:

- Find or create the target device type as a `Device` representing physical IPU hardware or a simulated `IPUModel`.
- Create a `Graph` object which will define the connections between computation operations and data, and how they are mapped onto the IPUs.
- Create one or more `Program` objects which will control the execution of the graph operations.
- Define the computations to be performed and add them to the `Graph` and `Program` objects. You can use the functions defined in Poplar and PopLibs, or you can write your own device code.
- Create an `Engine` object, which represents a session on the target device, using the `Graph` and `Program` objects.
- Connect input and output streams to the `Engine` object, to allow data to be transferred to and from the host.
- Execute the computation with the `Engine` object. This will compile your graph code and load it onto the IPU, along with any library functions required, and start execution.

The Poplar and PopLibs libraries also include programs for a wide range of operations on tensor data.

For more detailed descriptions and examples of each of these steps, see the tutorials in the Graphcore GitHub tutorials repository.

2.2.1 Program flow control

A program object can be constructed by combining other program objects in various ways. There are several sub-classes of `Program` that provide flow control.

The simplest of these is `Sequence`, which executes a number of sub-programs sequentially. There are also `Program` classes for executing loops, and for conditional execution.

Looping

Looping is supported by the `Repeat*` classes and the PopLibs counted loop functions, which provide a more flexible interface.

There are two types of repeat programs:

- Counted: this iterates a fixed number of times (a compile-time constant)
- While: there are two repeat programs (`RepeatWhileTrue` and `RepeatWhileFalse`) which will iterate while the condition is met. Any non-zero value of the predicate is treated as true.

The counted repeat program iterates a fixed number of times. PopLibs provides a more flexible interface with the loop functions in the `popops` library.

Each of the functions in popops/Loop.hpp returns a program object that implements a "for" loop. There are several versions of these functions that provide varying amounts of control over the loop variables; for example specifying the start, step and end values of the loop counter. The number of iterations can be defined at run time.

The loop count variable can be made available to the program in the body of the loop (unlike loops created with `Repeat`).

A basic outline for creating a `countedForLoop()` is shown below:

```
count = graph.addVariable(poplar::UNSIGNED_INT, {1});
limit = graph.addVariable(poplar::UNSIGNED_INT, {1});
loopBodyProg = Sequence();
popops::mapInPlace(graph, Add(_1, _2), {bodyVar,count}, loopBodyProg,
                   "/bodyFunction");
prog.add(poputil::countedForLoop(graph, count, 0, limit, 1, loopBodyProg,
                                  "/countedForLoop"));
```

Conditional execution

The `If` program is equivalent to an if-then-else: it runs one of two programs depending on the value of a scalar tensor. Any non-zero value of the predicate is treated as true. You can use an empty `Sequence` for the "else" branch if you just want a simple "if" conditional.

The `Switch` program runs one of a number of programs depending on the value of a tensor. You can also define a default case for when the value of the tensor does not match one of the switch values.

2.2.2 What happens at run time

When you run your program on the host, the Poplar run-time will compile your graph to create object code for each tile. The code may come from Poplar or PopLibs library functions, or from vertex code you write yourself (see *Device code*), and will be linked with any required libraries.

This object will contain:

1. The control-program code from your graph
2. Code to manage exchange sequences
3. Initialised vertex data
4. The tensor data mapped to that tile

The host program will load the object code onto the target device, which is then ready to execute the program.

2.3 Supported types

2.3.1 Scalar types

The scalar types supported by Poplar are shown in `colossus_abi_scalar_types`. In addition:

- By default the `char` type is signed.
- `long` is the same as `int`.
- `long double` is the same as `double`.
- The underlying type of an enumerated type is `int`.
- Function pointers are the same as data pointers.
- Only a subset of types are supported as fields of classes derived from the `Vertex` or `MultiVertex` classes. See `Type` for the supported types.

Table 2.1: Scalar data types

Type	Size (bits)	Align (bits)	Meaning
`char`	8	8	Character type
`short`	16	16	Short integer
`int`	32	32	Integer
`long`	32	32	Integer
`long long`	64	64	Integer
`half`	16	16	16-bit IEEE float
`float`	32	32	32-bit IEEE float
`double`	64	64	64-bit IEEE float
`long double`	64	64	64-bit IEEE float
`void *`	32	32	Data pointer

2.3.2 Floating point types

The IPU has hardware support for `float` (32 bit) and `half` (16 bit) operations.

Although the compiler supports `double` and `long double`, there is no hardware support for 64-bit floating-point operations and so any calculations will be implemented in software. You should be careful to avoid default promotions to double.

Half on the IPU

The IPU instruction set does not support operations on scalar half values, only vectors.

If operations on half values are not vectorised, either explicitly by the user (see `vector_types`) or the compiler, then they may be promoted to float. The compiler will use vector operations, wherever possible for half types. If not, it will either promote the values to float, or broadcast a single half value to a `half2` so it can use vector operations, then discard one half of the vector.

Half on the CPU

For CPU targets, `half` is, by default, an alias for float and `sizeof(half)` will be 4.

The parameter `accurateHalf` can be set to true when creating a CPU or IPU Model target, in which case `half` will be correctly implemented as 16-bit IEEE floating point. This will be slower, but will produce the same results as the IPU.

Codelets should be written to be generic to the size of `half` so that changing this setting requires no code changes.

2.3.3 Vector types

Poplar provides a number of vector types for representing short vectors of most scalar types. The supported vector types are shown in `colossus_abi_vector_types`. Only the floating point vectors have direct support in the IPU instruction set.

Table 2.2: Vector data types

Type	Size (bits)	Align (bits)	Meaning	iset
`char2`	16	16	Vector of 2 `char` values	N
`uchar2`	16	16	Vector of 2 `unsigned char` values	N
`char4`	32	32	Vector of 4 `char` values	N
`uchar4`	32	32	Vector of 4 `unsigned char` values	N
`int2`	64	64	Vector of 2 `int` values	N
`uint2`	64	64	Vector of 2 `unsigned int` values	N
`int4`	128	128	Vector of 4 `int` values	N
`uint4`	128	128	Vector of 4 `unsigned int` values	N
`long2`	64	64	Vector of 2 `long` values	N
`long4`	128	128	Vector of 4 `long` values	N
`longlong2`	128	128	Vector of 2 `long long` values	N
`longlong4`	256	256	Vector of 4 `long long` values	N
`short2`	32	32	Vector of 2 `short` values	N
`ushort2`	32	32	Vector of 2 `unsigned short` values	N
`short4`	64	64	Vector of 4 `short` values	N
`ushort4`	64	64	Vector of 4 `unsigned short` values	N
`float2`	32	32	Vector of 2 `float` values	Y
`float4`	64	64	Vector of 4 `float` values	Y
`half2`	32	32	Vector of 2 `half` values	Y
`half4`	64	64	Vector of 4 `half` values	Y

The vector types are defined with the vector extensions defined by GCC and Clang, using the vector_size variable attribute.

2.3.4 Structure types

Structure types pack according to the standard rules:

- Field offsets are aligned according to the field's type.
- A structure is aligned according to the maximum alignment of its members.
- Tail padding is added to make the structure's size a multiple of its alignment.

2.3.5 Bit fields

The following types may be specified in a bit-field's declaration: `char`, `short`, `int`, `long` , `long long` and `enum`.

If an `enum` type has negative values, `enum` bit-fields are signed. Otherwise, if a signed integer type of the specified width is not able to represent all `enum` values then enum bit-fields are unsigned. Otherwise, `enum` bit-fields are signed. All other bit-field types are signed unless explicitly unsigned.

Bit-fields pack from the least significant end of the allocation unit. Each non-zero bit field is allocated at the first available bit offset that allows the bit field to be placed in a properly aligned container of the declared type. Non bit-field members are allocated at the first available offset satisfying their declared type's size and alignment constraints.

A zero-width bit-field forces padding until the next bit-offset aligned with the bit field's declared type.

Unnamed bit-fields are allocated space in the same manner as named bit-fields.

A structure is aligned according to each of the bit field's declared types in addition to the types of any other members. Both zero-width and unnamed bit fields are taken into account when calculating a structure's alignment.

2.4 Virtual graphs

A graph is created for a target device with a specific number of tiles. It is possible to create a new graph from that, which is a *virtual* graph for a subset of the tiles. This is effectively a new view onto the parent graph for a virtual target, which has a subset of the real target's tiles and can be treated like a new graph. You can add vertices and tensors to the virtual sub-graphs. These will also appear in the parent graph.

Any change made to the *parent* graph, such as adding variables or vertices, may also affect the virtual sub-graph. For example, a variable added to the parent graph will appear in the sub-graph if it is mapped to tiles that are within the subset of tiles in the virtual target.

Virtual graphs can be used to manage the assignment of operations to a subset of the available tiles. This can be used, for example, to implement a pipeline of operations by creating a virtual graph for each stage of the pipeline and adding the operations to be performed on those tiles.

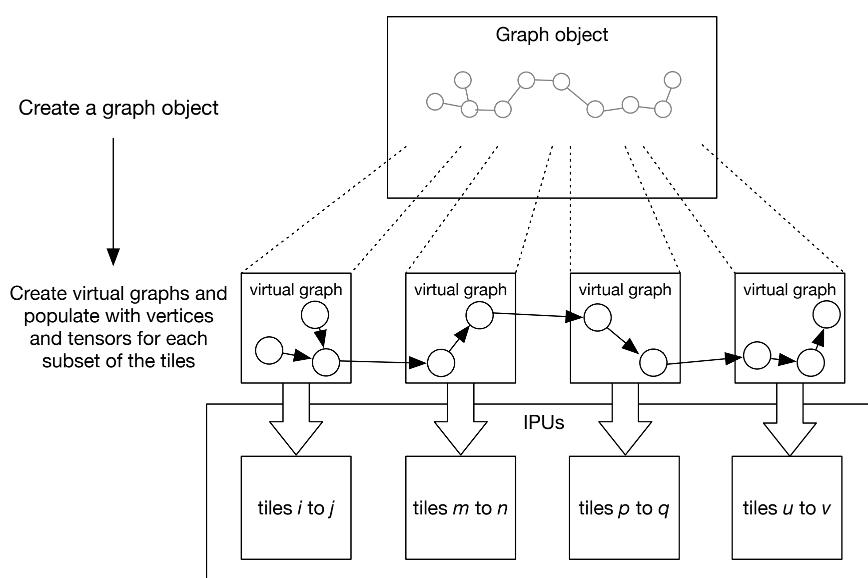


Fig. 2.3: Mapping a pipeline of operations to tiles using virtual graphs

There are several versions of the `createVirtualGraph` function, which provide different ways of selecting the subset of tiles to include in the virtual target.

2.5 Replicated graphs

You can also create a *replicated* graph. This effectively creates a number of identical copies, or replicas, of the same graph. Each replica targets a different subset of the available tiles (all subsets are the same size). This may be useful, for example, where the target consists of multiple IPUs and you want to create a replica to run on each IPU (or group of IPUs) in parallel.

Any change made to the replicated graph, such as adding variables or vertices, will affect all the replicas. A variable mapped to tile 0, for example, will have an instance on tile 0 in each of the replicas.

Replicated graphs can be created in two ways:

- Splitting an existing graph into a number replicas with the `createReplicatedGraph` function (see *Replicating an existing graph*).
- Creating a new replicated top-level graph by passing a replication factor to the `Graph` constructor (see *Creating a replicated graph*).

  Note: Replicated graphs created in this way are not supported when running on an IPU Model.

As an example, imagine you have a graph which targets two IPUs. You can run four copies of it, in parallel, on eight of the IPUs in your system by creating the two-IPU graph and replicating it four times. This can be done using either of the techniques above, each of which has advantages and disadvantages, summarised in the following descriptions.

2.5.1 Replicating an existing graph

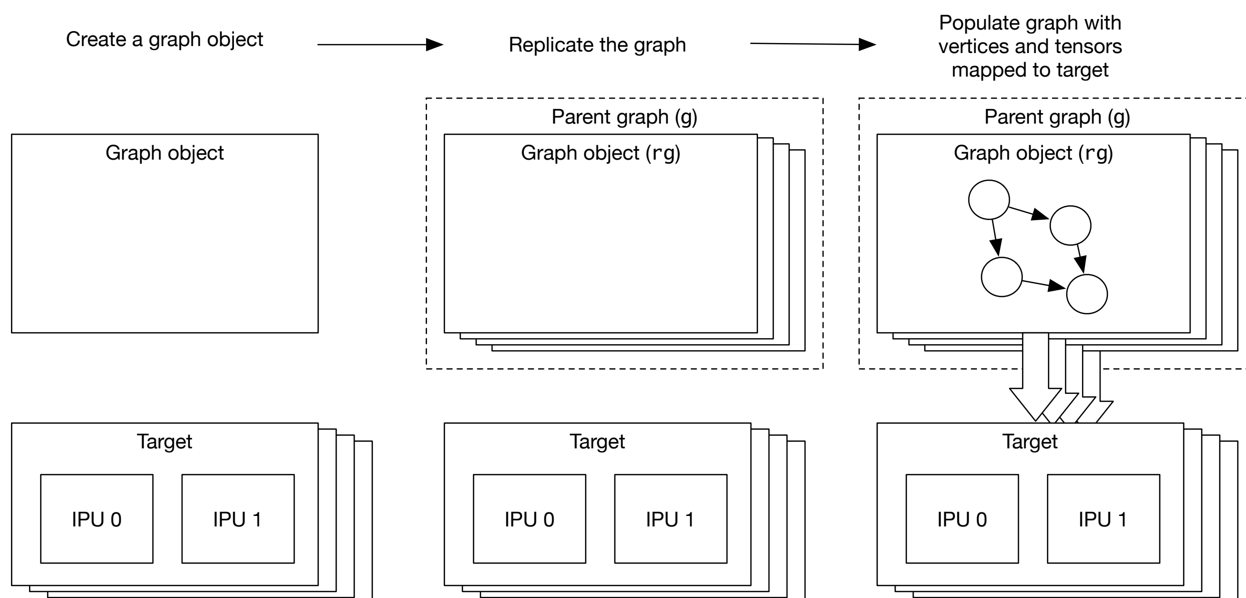


Fig. 2.4: Replicating an existing graph

We can start by creating a graph for eight IPUs, and then creating a replicated graph from that:

```
// Create a graph for 'target' which has 8 IPUs
Graph g  = Graph(target);
// Create 4 replicas each of which targets 2 IPUs
Graph rg = g.createReplicatedGraph(4);
```

Any changes, such as adding code or variables, made to the replica `rg` will be duplicated over all four replicas.

However, you can still do things with the original "parent" graph `g` that do not affect all the replicas. For example, a variable or an operation can be added to the parent graph and mapped to only one IPU. This will only be present on the replica that targets that IPU. It is also possible to access a variable that exists on all the replicas as a single tensor, using the `getNonReplicatedTensor` function. This adds an extra dimension to the variable to represent the mapping across the replicas.

This approach provides more flexibility but means that the graph of each replica needs to be compiled separately. This can make it slower to build the program.

2.5.2 Creating a replicated graph

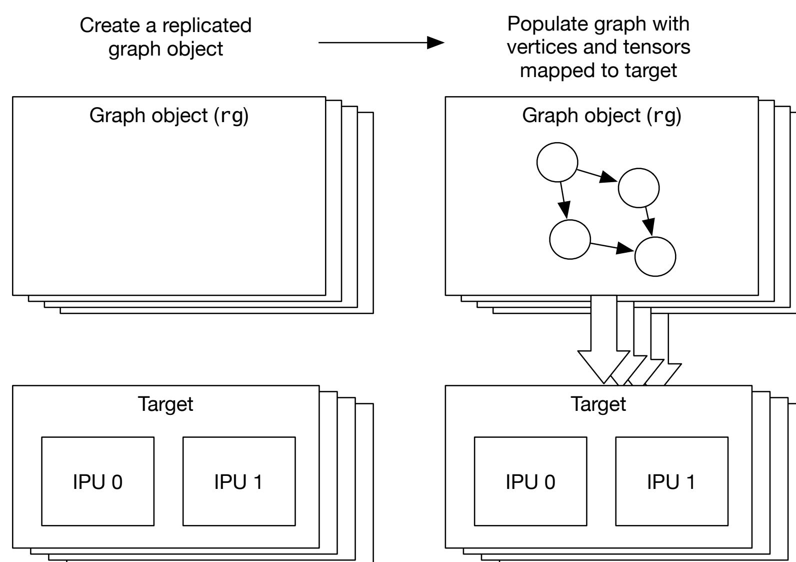


Fig. 2.5: Creating a replicated graph

In this case, we start by creating a replicated graph using the graph constructor:

```
// Create a graph with 4 replicas for each 2 IPUs
Graph rg = Graph(target, replication_factor(4));
```

We can add variables and vertices to this graph as usual. These additions will be applied to every replica. This graph *only* exists as a replica, with no parent graph that can be used to make modifications differently to each replica. Therefore, as all the replicas are guaranteed to be identical, the graph only needs to be compiled once. Copies of the object code are then loaded onto each of the pairs of IPUs when the program runs. Each instance of the replica is given a unique ID at load time; this can be used to identify it in functions such as `crossReplicaCopy`.

Any functions that rely on the existence of a parent, such as `getTopLevelGraph` or `getNonReplicatedTensor`, will fail.

2.6 Data streams and remote buffers

Memory external to the IPU can be accessed in two ways. Data streams enable the IPU to transfer data to and from host memory. Remote buffers enable the IPU to store data in external (off-chip) memory.

2.6.1 Data streams

Data streams are used for communication between the host and the IPU device. The data transfers are controlled by the IPU.

Each stream is a unidirectional communication from the host to the device, or from the device to the host. A stream is defined to transfer a specific number of elements of a given type. This means the buffer storage required by the stream is known (the size of the data elements times the number of elements).

The Poplar graph compiler will merge multiple stream transfers into a single transfer (up to the limits described in *Stream buffer size limit*).

Device-side streams

A stream object, represented by the `DataStream` class, is created and added to a graph using the `addHostToDeviceFIFO` or `addDeviceToHostFIFO` functions. The stream is defined to have:

- A name for the stream
- The type of data to be transferred
- The number of elements to be transferred

A host-to-device stream can also have a *replication mode*, if it is connected to a replicated graph. This defines whether a single stream will send the same data to all the replicated graphs (broadcast mode) or there will be a stream per replica.

Stream data transfer is done with a `Copy` program which copies data from the stream to a tensor, or from a tensor to the stream.

Host-side stream access

On the host side, a data stream is connected to a buffer allocated in memory. The buffer is connected to the stream using the `connectStream` function of an `Engine` object. This can, optionally, be implemented as a circular buffer to support more flexible transfers.

In order to synchronise with the data transfers from the IPU, a callback is connected to the stream using the `Engine::connectStreamToCallback` function. Callback implementations are derived from the `StreamCallback` interface and have a pointer to the stream buffer as an argument.

- For a device-to-host transfer, the callback function will be called when the transfer is complete so that the host can read the data from the buffer.
- For a host-to-device stream, by default the callback function will be called immediately before the IPU transfers the buffer contents to device memory. The host-side code should populate the stream buffer and then return.

2.6.2 Remote memory buffers

The IPU can also access off-chip memory as a *remote buffer*. This may be host memory or memory associated with the IPU system. This is not used for transferring data to the host, but just for data storage by the IPU program.

A `RemoteBuffer` object is created and added to the graph with the `addRemoteBuffer` function of the graph object. Data transfers to and from the remote buffer are performed using a `Copy` program which copies data from the buffer to a tensor, or from a tensor to the buffer.

The data type and size of the remote buffer are defined when it is created. The definition of the buffer and the parameters to the `Copy` program allow for very flexible addressing.

You can think of the buffer containing a number of data transfer "rows" of data. (These rows do not need to correspond to the structure the tensor being transferred or the organisation of the data in the buffer, but are just a way of managing data transfers.)

The size of each row and the number of rows are parameters to `addRemoteBuffer` when the buffer is created. Each row contains `numElements` data items and the entire buffer contains `repeat` rows.

Each transfer to or from the remote buffer can copy one or more rows of data. The rows to be copied are specified by the `offset` parameter to the `Copy` program. The number of offsets specifies the number of rows to copy.

2.6.3 Stream buffer size limit

The IPU has a memory address translation table which defines the external memory address range it can access. As a result, there is a maximum buffer size for data transferred by a stream. This limit is currently 128 MBytes per stream copy operation. More data can be transferred by a sequence of copies, separated by sync operations, so that the buffer memory can be reused for each transfer.

Each IPU has its own translation table. So, if there are multiple IPUs, this limit applies to each IPU individually.

2.6.4 Optimising host data transfers

There are several things you can do to optimise the use of data streams to and from the host. These are described below.

Prefetch

You can specify that the the IPU should call the callback function as early as possible (for example, immediately after it releases the stream buffer from a previous transfer). The host is then able to fill the buffer in advance of the transfer, meaning the IPU spends less time waiting for the host.

This mode of operation, known as prefetch, is enabled by setting the `exchange.enablePrefetch` option to "true" when the engine object is created.

Prefetch is only possible if the address range of the stream's data buffer does not overlap with another stream's buffer (this may be done to optimise memory use).

This means that the engine option `exchange.streamBufferOverlap` must be set to either "HostRearrangeOnly" or "None". The first of these is most useful as the performance of streams that are being rearranged is often less important. Setting the option to "None" may use too much memory.

The callback function returns a value that indicates if the buffer was filled.

If there is data available to fill the buffer, the callback function should return `Result::Success`. The device code will then call the `complete` callback when it has transferred the data.

Otherwise, if data is not available (either because it is the end of the stream, or the data is not ready yet), then the callback returns `Result::NotAvailable`.

2.7 IPU-Link and sync configuration

Multiple IPUs can be connected with IPU-Links to share data. There are also synchronisation (sync) signals that are used to indicate when IPUs are ready to exchange data and that data exchange is complete. These sync signals are also used to synchronise host transfers and access to remote buffers.

2.7.1 Link topologies

There are two ways of connecting IPU-Links and sync signals: in a mesh or as a torus. The mesh structure is similar to a ladder, where pairs of IPUs form each rung. In a torus, the ends of the "ladder" loop round to form a closed loop.

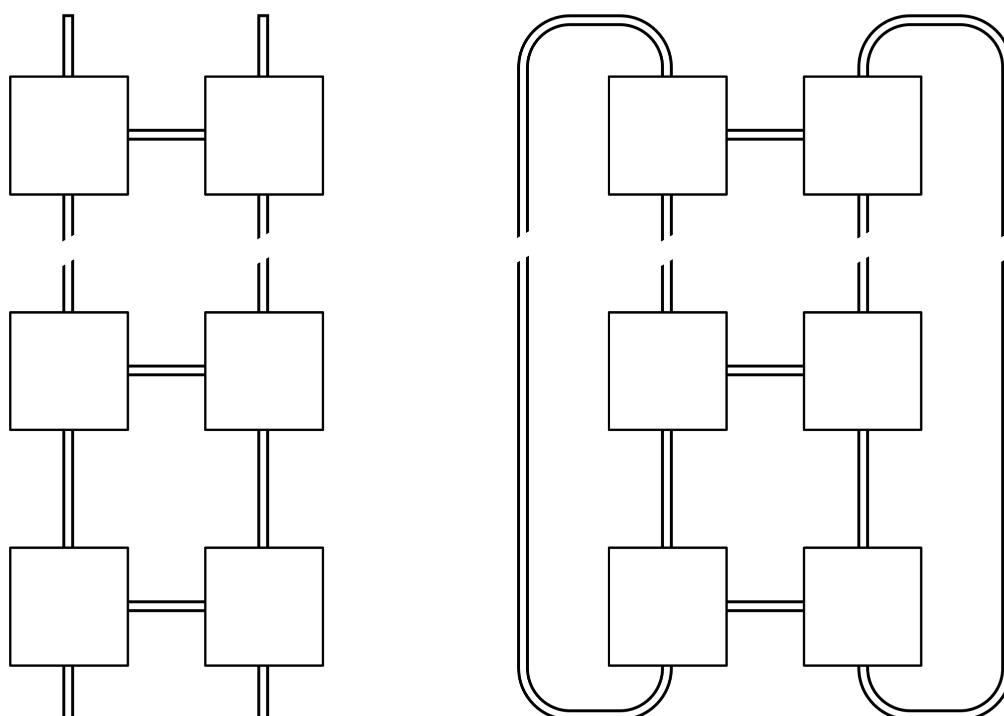


Fig. 2.6: Mesh and torus link topologies

When a target device is created in a Poplar program, the topology is defined by the `ipuLinkTopology` option to the `Target` object.

2.7.2 Sync groups

Each IPU can be allocated to one or more "sync groups". At a synchronization point, all the IPUs in a sync group will wait until all the other IPUs in the group are ready.

Sync groups can be used to to allow subsets of IPUs to overlap their operations. For example, one sync group can be performing data transfers to or from the host, while another group is processing a previous batch of data.

You can configure the sync groups as appropriate for your application. The allocation of IPUs to the sync groups (GS1 and GS2) can be configured using the `syncConfiguration` option when creating a target.

The options are:

- **intraReplicaAndAll:**
  - GS1 is used for synchronisation between the IPUs in each replica of a replicated graph (or all IPUs if there is no replication).

  - GS2 is used for synchronisation between all IPUs.
- **ipuAndAll:**
  - GS1 is used for synchronisation of each IPU individually.
  - GS2 is used for synchronisation between all IPUs.
- **intraReplicaAndLadder:**
  - GS1 is used for synchronisation between the IPUs in each replica of a replicated graph (or all IPUs if there is no replication).
  - GS2 is used by two independent subsets of IPUs. These can then synchronise independently of one another, so that they can alternate between one set doing host I/O, for example, while the other is computing.

The way in which Poplar uses these sync groups is summarised in the following table:

syncConfiguration		**syncReplicasIndependently**	
		false (default)	**true**
intraReplicaAndAll (default) and **intraReplicaAndLadder**	GS1	Communication between IPUs within each replica (or all IPUs if the graph is not replicated). Remote buffer access.	Communication betwe within each replica (or if the graph is not rep Remote buffer access communication.
	GS2	Communication between replicas (all IPUs). Host communication.	Communication betwe cas (all IPUs).
ipuAndAll	GS1	Remote buffer access.	Remote buffer access communication.
	GS2	Communication between all IPUs. Host communication.	Communication betw IPUs.

2.7.3 Software sync

Software sync provides a third synchronisation mechanism that can replace the hardware sync that happens after a host exchange. Software sync is disabled by default. You can enable it by setting the option `opt.enableSwSyncs` to true when creating the engine object.

With software sync enabled, each IPU synchronises with the host independently. This means that each IPU can move onto the next operation as soon as its host data transfer is complete, instead of having to wait for all the other IPUs to finish.

If two IPUs don't need to synchronise then they can operate in parallel, completely independently. For example, this allows one to do I/O while the other is computing. But this applies more generally: each IPU can do an arbitrary sequence of compute and I/O operations without needing to synchronise with the other IPU until they need to communicate with one another.

Note: If you use software sync then the default sync configuration (`intraReplicaAndAll`) must be used and the `target.syncReplicasIndependently` option must not be set.

2.8 Device code

Each vertex of the graph is associated with some device code. This can come from a library function or you can write your own as a *codelet*. Codelets are specified as a class that inherits from the `poplar::Vertex` type. For example:

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

using namespace poplar;

class AdderVertex : public Vertex {
public:
  Input<float> x;
  Input<float> y;
  Output<float> sum;

  bool compute() {
    *sum = x + y;
    return true;
  }
};
```

The `Input` and `Output` fields connect the vertex to the tensor data that it reads and writes. An `Input` field should not be written and an `Output` field should not be read; the results are undefined. If you need a field that is read and written, then it should be defined as `InOut`.

These fields have `begin`, `end`, `operator[]` and `operator*` methods so they can be iterated over and accessed like other C++ containers. For `Input` fields all of these methods are `const`.

The `Output` field can be successfully updated even if the corresponding tensor is on another tile. This is because the data is not transferred to the destination tile until the compute is complete. However, *reading* an `Output` field is not guaranteed to return the expected value. If you need to both write to and read from a field, then it should be declared as an `InOut` type.

The types used in vertex code are described in the runtime API section of the Poplar and PopLibs API Reference.

You can add a codelet to your graph by using the `Graph::addCodelets` function. This will load the source file and compile the codelet when the host program runs. See the `adder` example provided with the Poplar distribution.

You can also pass compilation options (for example "-O3"). The code is compiled for both the host and for the IPU so the program can be run on IPU hardware or on the host.

There are a couple of predefined macros that may be useful when writing vertex code. `__POPC__` is defined when code is compiled by the codelet compiler. The macro `__IPU__` is defined when code is being compiled for the IPU (rather than the host).

You can also write codelets in assembly language for the IPU. See Section 7, Writing vertices in assembly for more information.

2.8.1 Stack allocation

When C++ functions are compiled, the compiler is usually able to determine the stack required. This is not possible if you use recursion, function calls via pointers or variable-length arrays (array variables that have a size that is not a compile time constant).

If you must use these techniques, then you must explicitly specify the stack used by your functions. Macros are provided for this purpose. These are defined in the Poplar header file `StackSizeDefs.hpp`. See the runtime API section of the Poplar API Reference for more information.

- `DEF_STACK_USAGE size function`

  This defines the total stack usage (in bytes) for the function specified *and any functions that it calls*. This means that Poplar will not traverse the call graph of the function to determine the total stack usage of the function.

If you use recursion, this macro must be used to specify the total stack usage of the recursive function itself, taking into account the maximum depth of the recursion and any other functions that can be called.

- DEF_FUNC_CALL_PTRS

  This defines a list of other functions that may be called via pointers. Note that this creates a maintainability problem as the macro use must be updated every time the code changes its use of function pointers.

2.8.2 Pre-compiling codelets

There is a command line tool to pre-compile codelets. This reduces loading time, and allows you to check for errors before running the host program.

The codelet compiler, popc, takes your source code as input and creates a graph program object file (conventionally, with a .gp file extension). For example:

```
$ popc codelets.cpp -o codelets.gp
```

This object file can be added to your graph in the same way as source codelets, using the same Graph::addCodelets function. See the adder_popc example provided with the Poplar distribution.

The general form of the popc command is:

```
$ popc [options] <input file> -o <output file>
```

The command takes several command line options. Most are similar to any other C compiler. For example:

-D<macro>	Add a macro definition
-I<path>	Add a directory to the include search path
-g	Enable debugging
-On	Set the optimization level (n = 0 to 3)

For a full list of options, use the --help option.

CHAPTER

THREE

UNDERSTANDING VERTICES

This chapter describes the `Vertex` class, including how vertices are run, parameters are passed and data types are stored.

3.1 The Vertex class

Vertices in Poplar are subclasses of the `Vertex` or `MultiVertex` base classes. They each have a `compute()` method that is run on the tile and returns a `bool` value. The `Vertex::compute()` method runs in a single worker thread. The `MultiVertex::compute(unsigned)` method runs in multiple worker threads.

In order to run the vertex's `compute()` method, you need to add the vertex to a compute set. When compiled, the compute set is reduced to a single function that calls the `compute()` functions of all the vertices it contains.

3.2 Vertex state

Any vertex state required is provided as member fields inside the vertex class.

3.2.1 Vector and VectorList types

There are two vector types used: `Vector` and `VectorList`, representing one and two dimensional blocks of data respectively. `VectorList` is a "jagged" 2D list, in other words the sub-lists need not be the same length.

Each of these vector types can be represented in memory in different ways. See *Vector types* for details.

3.2.2 Specifying memory constraints

The `poplar::constraint` attribute can be applied to vertices to restrict where vertex state is placed in memory. This takes one or more string parameters.

The parameters available are described in the table below, where `src` and `dst` are names of vectors in your vertex state.

Table 3.1: Poplar constraints

Type	Description
"elem(*src)!=elem(*dst)"	This constraint means that the vertex field src will be placed in a different memory element to the vertex field dst. This means you can do load from the src pointer and store to the dst pointer in the same cycle without causing a memory clash. If you find that this constraint doesn't give much performance benefit it should be removed as it can be costly in terms of total memory use.
"region(*src)!=region(*dst)"	This constraint means that two fields will be placed in different regions. This implies that one of them will be placed in interleaved memory, although it doesn't matter which one. As above, this means you can use load-store instructions to do a simultaneous load from src and store to dst.

3.3 MultiVertex worker threads

Vertices with the Vertex base class run in a single worker thread and can access all the information they need to run from their vertex state alone. The compute() method of vertices with the MultiVertex base class is run multiple times in different worker threads. The Poplar compiler generates code to run the compute method of a multi-vertex multiple times and passes a single argument to the multi-vertex compute method which is the thread ID of the running worker.

You can obtain the total number of invocations of the multi-vertex compute method for a given vertex in vertex code using the MultiVertex::numWorkers() method and in host code using the poplar::Target::getNumWorkerContexts() method.

The worker thread IDs given to the multi-vertex compute method will always be in the range [0, MultiVertex::numWorkers()) and the same ID will never be given twice in the same compute set for the same multi-vertex.

The worker thread ID allows a multi-vertex to have precise control over the split of work to be performed in parallel for a single vertex. For example you might split the work of adding 2 vectors of numbers together using a multi-vertex like so:

```
class AddTwoVectors : public MultiVertex {
public:
  Input<Vector<unsigned>> a;
  Input<Vector<unsigned>> b;
  Output<Vector<unsigned>> c;

  bool compute(unsigned workerId) {
    const auto numElements = a.size();
    for (std::size_t i = workerId;
         i < numElements;
         i += MultiVertex::numWorkers(); ++i) {
      c[i] = a[i] + b[i];
    }
    return true;
  }
};
```

3.3.1 Thread safety

An ordinary vertex provides compile-time thread safety checking because the regions of memory that are read and written by each worker in the compute set is defined. You control the read and written regions of memory by each worker thread in a multi-vertex and consequently no compile-time thread safety checking is available and you must take care to avoid any such issues yourself.

You may safely read from the same region of memory from multiple threads.

You may safely write to the same region of memory from multiple threads but the order of those writes is undefined.

However, you must also consider the atomic write size of the target in use. The atomic write size in bytes is available in host code as `poplar::Target::getAtomicStoreGranularity()`. This value gives the smallest alignment and size in bytes that can be written to memory atomically. When writing data where the number of bytes or the address is not a multiple of the atomic write size multiple instructions are required to perform the write. This is because the existing memory contents need to be read, partially modified with the new data, and then re-written to memory. This means that the write is not atomic. Consequently two threads writing to the same atom could overwrite the other's data.

3.4 Calling conventions

There is a vertex calling convention (see *Application binary interface (ABI)*) that is used by vertices. However, the `compute()` method itself does not use this calling convention. Because of this, when Poplar compiles a vertex it will create a new function that *does* use the calling convention, which then calls the `compute()` method and propagates the return value.

The name of this wrapper function is `__runCodelet_XXXX` where `XXXX` is the mangled name of the class that contains the compute method (see *Vertex name mangling*). The wrapper for a `Vertex::compute()` method looks like this:

```
int __runCodelet_MyVertex() {

  void *vertexPtr = __builtin_colossus_get_vertex_base();
  auto v = static_cast<MyVertex*>(vertexPtr);
  return v->compute();
}
```

The wrapper for a `MultiVertex::compute(unsigned)` method looks like this:

```
int __runCodelet_MyMultiVertex() {
  void *vertexPtr = __builtin_colossus_get_vertex_base();
  auto v = static_cast<MyVertex*>(vertexPtr);
  auto w = __builtin_ipu_get(CSR_W_WSR__INDEX) & CSR_W_WSR__CTXTID_M1__MASK;
  return v->compute(w);
}
```

Vertices with base class `Vertex` have no parameters. Vertices with base class `MultiVertex` have a single parameter to the `MultiVertex::compute(unsigned)` method which is the thread ID of the worker running the method.

3.4.1 External codelets

When you write an assembly, or external, implementation of a vertex you need to inform Poplar that you are providing the __runCodelet_XXXX function so it does not generate the wrapper itself. You do this by adding a static bool isExternalCodelet to the Vertex or MultiVertex class. When this exists and is set to true, Poplar will assume that the __runCodelet_XXXX function is defined, and will call that, ignoring the compute() method.

You can use this to define, at compile time, whether you have provided assembly code definitions for some or all possible template instantiations of a vertex. For example, consider a vertex like this:

```
template <typename FPType>
template class Foo : public Vertex {
  static bool isExternalCodelet = std::is_same<FPType, float>();
  bool compute() { return true; }
};

template class Foo<half>;
template class Foo<float>;
```

This states that Poplar should only use the compiled compute method for the Foo<half> vertex and that we will provide an assembly implementation of the Foo<float> vertex.

3.4.2 Recursion and function pointers

You should avoid the use of recursion and function calls via pointers. Using these will prevent the Poplar runtime from correctly computing the stack usage.

If you must use recursion, the DEF_STACK_USAGE macro (see Section 7.3.1, Specifying stack size) must be used to specify the total stack usage of the recursive function itself, taking into account the maximum depth of the recursion and any other functions that can be called.

If you need to call via function pointers, you can use the DEF_FUNC_CALL_PTRS macro to specify a a list of other functions that may be called via pointers. Note that this creates a maintainability problem as the macro use must be updated every time the code changes its use of function pointers. See the API documentation for details.

3.5 Vertex name mangling

The Poplar name mangling is designed to be easy to write. All mangled vertices begin with __runCodelet_ followed by the full class name, including namespace, with the following changes made:

- Replace __ (two underscores) with _Z
- Replace :: with __ (two underscores)
- Replace < with ___ (three underscores)
- Replace , in the template argument list with _ (one underscore)
- Discard >

Some examples are shown in the table below.

Table 3.2: Name mangling

Before	After
`popnn::NonLinearity2D<float, 2>`	`__runCodelet_popnn__NonLinearity2D___float_2`
`popops::UnaryOp<popops::expr::ABSOLUTE, half>`	`__runCodelet_popops__UnaryOp___popops__expr__ABSOLUTE_half`
`popconv::ConvPartial1x1<float, half, true>`	`__runCodelet_popconv__ConvPartial1x1___float_half_true`

CHAPTER

FOUR

VECTOR TYPES

The fields of a `Vertex` can include `Vector<T>` or `VectorList<T>` types, or a combination of those such as `Vector<Input<Vector<T>>>`, in its state fields. These are similar to `std::vector` but can have different layouts in memory, optimised for the tile architecture.

These types are documented in the runtime API section of the Poplar and PopLibs API Reference.

4.1 Parameters

As well as the data type, the `Vector` and `VectorList` templates also have parameters to specify minimum alignment of elements, and whether or not they need to be stored in interleaved memory, for example:

```
template <typename T, VectorLayout L, unsigned MinAlign, bool Interleaved>
class Input<Vector<T, L, MinAlign, Interleaved>>
  ...
```

4.1.1 Types

The vector data type (`T`) can be any of the supported Poplar types defined in Types.h.

4.1.2 Layout

The template parameter `L` defines the type of memory layout to use. The valid layouts for a `Vector` are shown in Table 4.1. Some of these layouts use compressed pointer formats. These are not supported on all platforms.

See *Pointer compression* for more information.

Table 4.1: Vector memory layouts

Name	Description	Platform support
SPAN (default)	A pointer to the start of the vector, and a count of the number of elements (not bytes) the vector contains. This means that the .size() member and iterators are available. The count is a 32-bit integer.	All
SHORT_SPAN	A pointer to the start of the vector, and a count of the number of elements (not bytes) the vector contains. The count is limited to 11 bits. This means that the .size() member and iterators are available.	Mk1, Mk2
ONE_PTR	The same as SPAN but the count is not stored, so this is a single pointer to the start of the vector. The vector does not know its size, which must be found by some other means. The .size() member and iterators are **not** available.	All
SCALED_PTR32	The same as ONE_PTR, but using a compressed 16-bit pointer containing bits 2-17 of a full 32-bit pointer. Since the lower two bits are not stored it can only point to 32-bit aligned data.	Mk1 only
SCALED_PTR64	The same as ONE_PTR, but using a compressed 16-bit pointer containing bits 3-18 of a full 32-bit pointer. Since the lower three bits are not stored it can only point to 64-bit aligned data.	Mk1 only
SCALED_PTR128	The same as ONE_PTR, but using a compressed 16-bit pointer containing bits 4-19 of a full 32-bit pointer. Since the lower three bits are not stored it can only point to 128-bit aligned data.	Mk1, Mk2
COMPACT_PTR	This pointer type will resolve into the most suitable pointer type, given the size of the address space and the alignment of the data.	All

These layouts are described in more detail in *Memory layout for vectors*.

Only SPAN and SHORT_SPAN provide a .size() method.

Some examples of how COMPACT_PTR resolves on Mk1 and Mk2 based on the required alignment are shown in Table 4.2.

Table 4.2: Compact pointer resolution

Alignment	Example of declaration	On MK1	On MK2
1,2	Input<Vector<T, COMPACT_PTR, 1>>	ONE_PTR	ONE_PTR
4	Input<Vector<T, COMPACT_PTR, 4>>	SCALED_PTR32	ONE_PTR
8	Input<Vector<T, COMPACT_PTR, 8>>	SCALED_PTR64	ONE_PTR
>= 16	Input<Vector<T, COMPACT_PTR, 16>>	SCALED_PTR128	SCALED_PTR128

4.1.3 Minimum alignment

The MinAlign template parameter specifies the required alignment, in bytes, of the data in the Vector or VectorList.

- The default value for this is 1 byte for SPAN, SHORT_SPAN or ONE_PTR layouts.
- For SCALED_PTR32, the default alignment is 4.
- For SCALED_PTR64, the default alignment is 8.
- For SCALED_PTR128, the default alignment is 16.

However, the alignment is never less than the size of the data type. Values are always naturally aligned.

4.1.4 Interleaved memory

The final template parameter, `Interleaved`, tells the compiler that the data must be placed in interleaved memory (see *Memory architecture*).

4.2 Memory layout for vectors

This section describes the ways in which `Vector` types can be arranged in memory.

4.2.1 Pointer compression

In order to reduce memory usage, the size of pointers to the vector data can be compressed, based on the tile memory size.

Note: Future implementations of the IPU may have memory with different sizes and base addresses. You should not hard-code any assumptions about the memory system. The Poplar library includes functions that provide information about the memory system that the code is running on (see *Memory architecture* for more information).

Not all of these compressed pointer formats are available on all platforms. The header file `AvailableVTypes.h` provides macros that define which formats are supported. For example:

```
#include "poplar/AvailableVTypes.h"

#if defined(VECTOR_AVAIL_SCALED_PTR32)
  Input<Vector<char, VectorLayout::SCALED_PTR32, 4>> desc;
#else
  Input<Vector<char, VectorLayout::ONE_PTR, 4>> desc;
#endif
```

SCALED_PTR32

A 4-byte aligned, 32-bit pointer can be compressed to 16 bits by taking advantage of the fact that the valid memory range is from 0x40000 to 0x80000. Therefore, bits [31:19] are always 0 and bit 18 is always 1. Bits [1:0] are also 0. So only bits [17:2] need to be represented.

Note that this means `SCALED_PTR32` pointers are effectively offsets from 0x40000.

This encoding can be represented as:

```
scaled_ptr = (address & ~TMEM_REGION0_BASE_ADDR) >> 2
```

And to decode it:

```
address = (scaled_ptr << 2) | TMEM_REGION0_BASE_ADDR
```

SCALED_PTR64

A 32-bit pointer can be compressed to 16 bits by enforcing 64-bit data alignment. In this case, bits [2:0] are always 0 and the compressed pointer contains bits [18:3] of the address.

SCALED_PTR128

A 32-bit pointer can be compressed to 16 bits by enforcing 128-bit data alignment. In this case, bits [3:0] are always 0 and the compressed pointer contains bits [19:4] of the address.

4.2.2 Vector<T> layout

`Vector` is the simplest array type. It always stores a pointer to the start of the data array, and can optionally store the number of elements. If the number of elements is present, a `.size()` method is available.

The supported memory layouts are shown in Table 4.1.

Fig. 4.1 shows the memory layout for `ONE_PTR` and `SPAN`. `SCALED_PTR32`, `SCALED_PTR_64` and `SCALED_PTR_128` are similar to `ONE_PTR` but their `begin` pointers are 16 bits instead of 32.

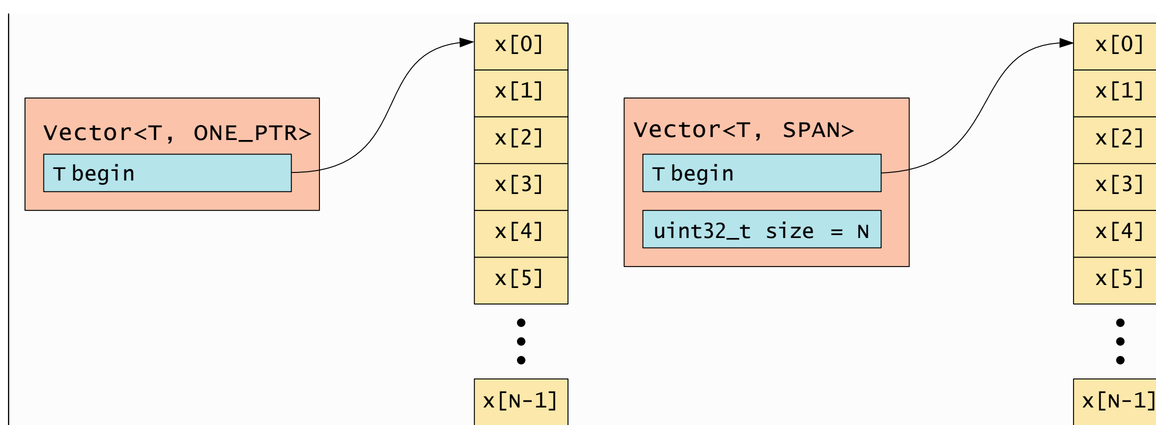


Fig. 4.1: Vector<T> memory layout

The `SPAN` layout can be represented as:

```
T* begin; // 32-bit pointer
uint32_t size;
```

Whereas `SHORT_SPAN` has a layout like this:

```
T* begin; // Truncated 20-bit pointer
// 1 bit reserved for the future
uint11_t size;
```

Which means it can only store up to 2,047 elements.

4.2.3 Vector<Input<Vector<T>>> layout

It is possible to nest `Vectors`, and at each level the memory layout can be different. We use `Vector<Input<Vector<T>>>` to illustrate how these are implemented, but `Input` could also be `Output` or `InOut`.

For example, if both levels use `ONE_PTR` you would have the layout shown in Fig. 4.2.

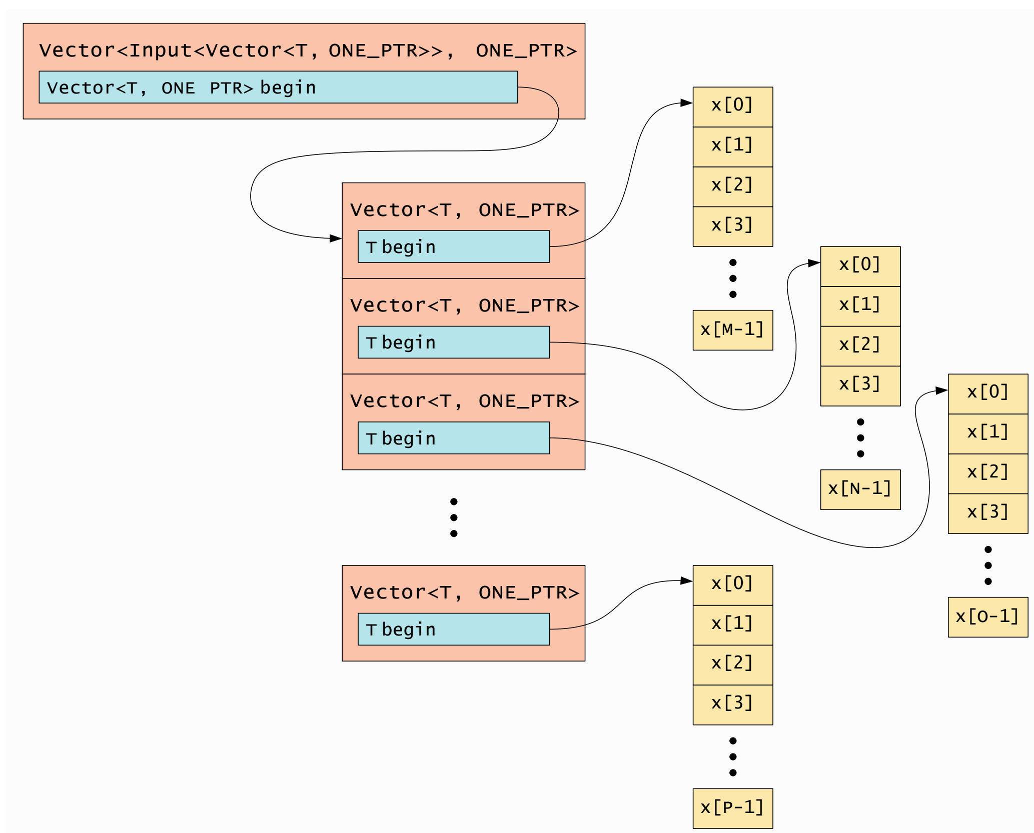


Fig. 4.2: Vector<Input<Vector<T>>> memory layout using ONEPTR

Or if both levels used `SPAN` the layout would be as shown in Fig. 4.3.

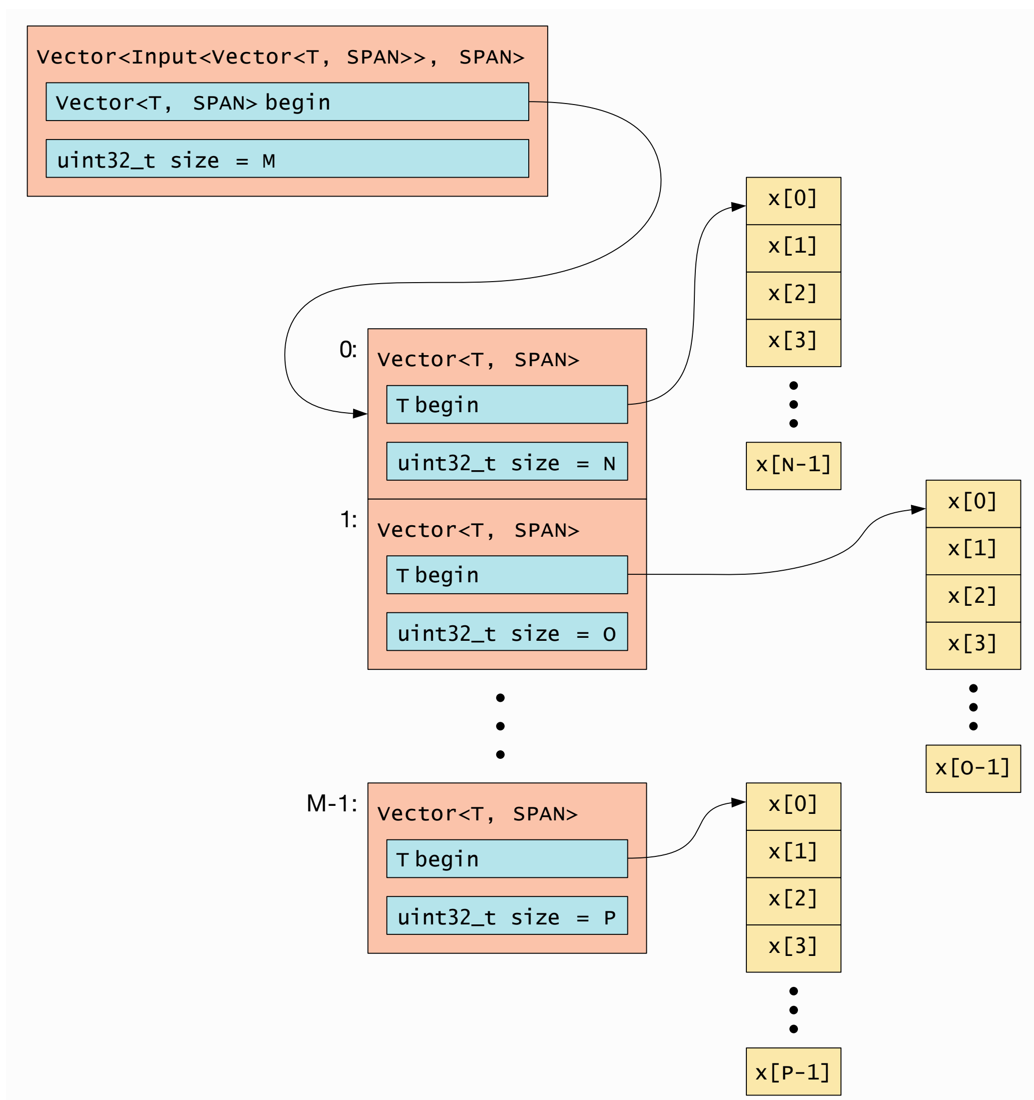


Fig. 4.3: Vector<Input<Vector<T>>> memory layout with SPAN

Note that this produces a "jagged" 2D vector. In other words, the length of each sub-vector is not guaranteed to be identical (although it might be).

You can use different layouts for each level, for example: `Vector<Input<Vector<T, ONE_PTR>>, SPAN>`.

4.2.4 VectorList layout

Because nested vectors such as `Vector<Input<Vector<T>>>` can use a lot of memory, Poplar provides a more memory-efficient 2D vector type called `VectorList`. The available layouts for a `VectorList` are shown in Table 4.3 and described in detail in the following sections.

Table 4.3: VectorList layouts

Layout	Platform support
DELTANELEMENTS	Mk2
DELTAN	Mk1
COMPACT_DELTAN	All

These have a *base structure* which contains the base address of the data, the size of the vector (that is, the number of sub-vectors) and a pointer to an array of structures describing the sub-vectors.

Each of the sub-vector structures contain a pointer to its data (as an offset from the base address) and the number of data elements. Each sub-vector can be a different size. The base address points to the start of the vector data and so one of the offsets is always zero.

The implementation of these memory layouts on the IPU is described below.

DELTANELEMENTS layout

The `DELTANELEMENTS` layout is always supported. The implementation is shown in Fig. 4.4 (using C-like types to represent the pointer and count sizes).

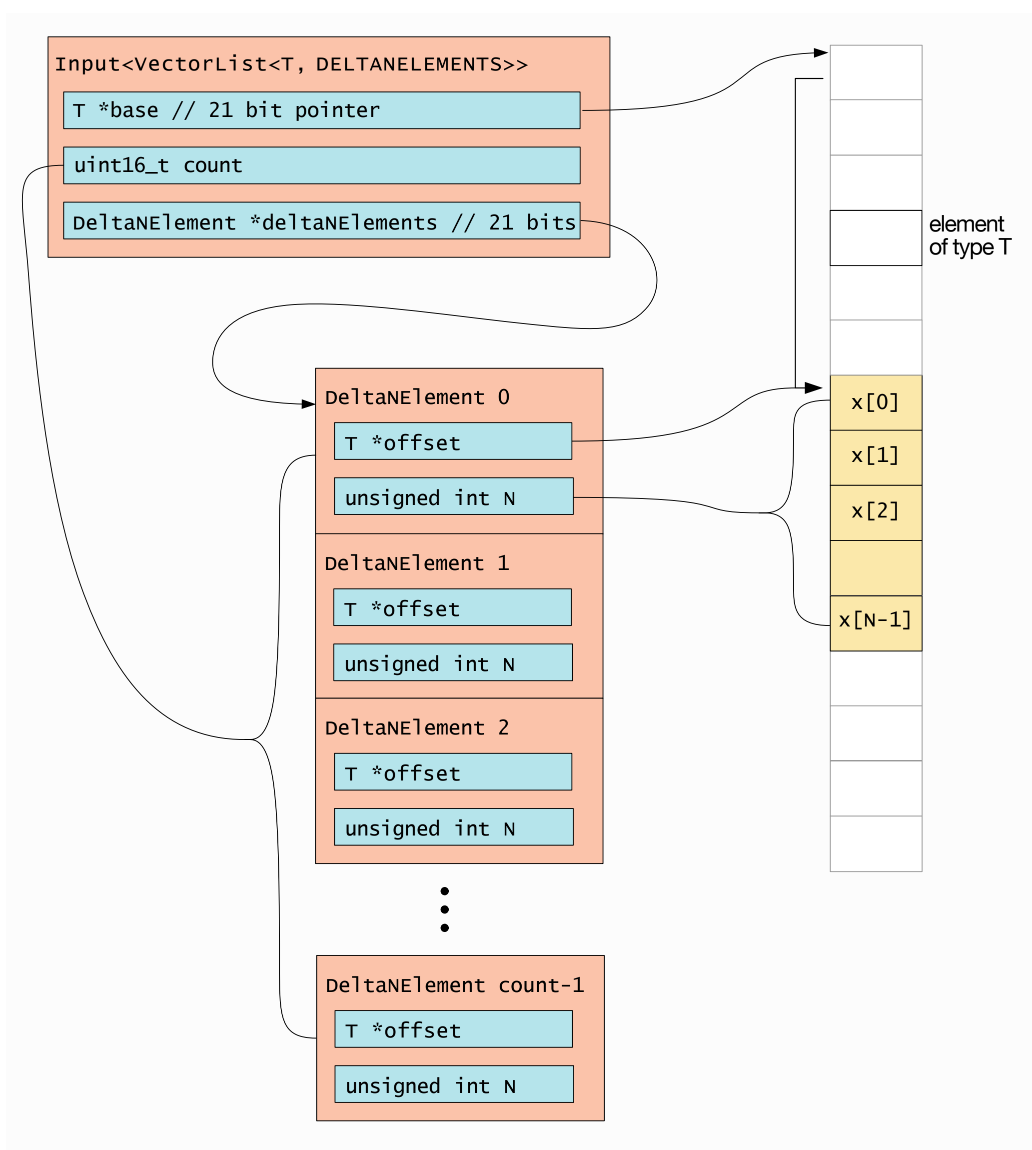
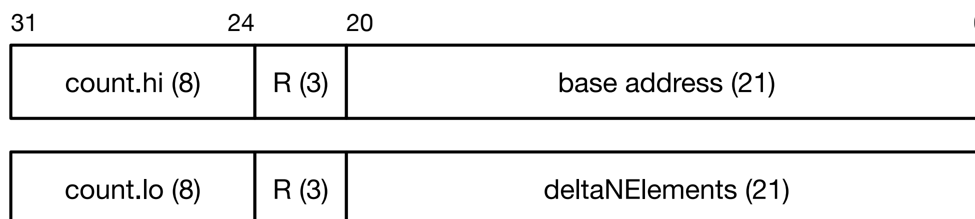


Fig. 4.4: DELTANELEMENTS memory layout

The top-level structure contains a pointer to the base of the vector data, a count of the number of sub-vectors and a pointer to an array of DeltaNElement structures for the sub-vectors. Both pointers are 21 bits so that the full architectural memory space of the tile can be addressed.

These values are packed into two 32-bit words, as shown in Fig. 4.5. The reserved bits should not be assumed to be zero and should be masked off when extracting the address fields.



count = count.hi << 8 + count.lo

R = reserved

Fig. 4.5: DELTANELEMENTS base structure bit packing

Each `DeltaNElement` structure represents a sub-vector. It has a pointer to the data, which is an element-sized offset from the base address (so, for example, for naturally-aligned `float` data, this will be an offset in 32-bit words). It also has a count of the number of data elements in this sub-vector.

The offset and count are packed into a 32-bit word. The number of bits for each depends on the data alignment. For byte-aligned data, the offset is 21 bits and the count is 11 bits. For larger alignments, fewer bits are required for the offset and more bits are available for the count. For example, for 32-bit aligned data, only 19 bits are required for the offset, so 13 bits are available for the sub-vector size.

The number of bits available for the offset and the count for various data types are summarised in Table 4.4 and illustrated in Fig. 4.6.

Table 4.4: Offset and count sizes for DELTANELEMENTS

Type	Offset size	Count size	offset unpacking
`uint8_t`	`uint21_t`	`unit11_t`	<< 0
`uint16_t`	`uint20_t`	`unit12_t`	<< 1
`uint32_t`	`uint19_t`	`unit13_t`	<< 2
`uint64_t`	`uint18_t`	`unit14_t`	<< 3
`uint128_t`	`uint17_t`	`unit15_t`	<< 4

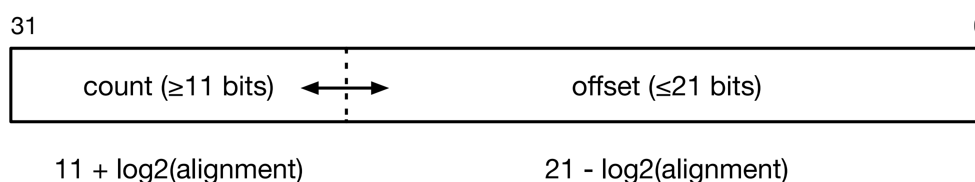


Fig. 4.6: DELTANELEMENTS sub-vector structure bit packing

Note that the alignment must be a multiple of the data size and must be a power of 2.

- The number of address bits required can defined as 21 - $\log_2$(alignment).
- The number of bits available to represent the sub-vector size is: 11 + $\log_2$(alignment).

The maximum size of the sub-vectors therefore depends on the data type. For byte-aligned data, for example, the maximum size is 2^{11}-1, while for 16-bit alignment (for example, `half` data) it is 2^{12}-1.

DELTAN layout

The `DELTAN` layout is used for smaller memory systems where `SCALED_PTR32` pointer compression is supported. This is only available on Mk1 platforms. The macro `VECTORLIST_AVAIL_DELTAN` (defined in `AvailableVTypes.h`) can be used to check if it is supported.

The top-level structure contains a pointer to the base of the vector data. This is truncated to 20 bits. The remaining 12 bits of the word are used to store the number of sub-vectors. This means the outer dimension of the `VectorList` has a maximum size of 4,095. Finally, there is a `SCALED_PTR32` pointer to an array of `DeltaN` structures.

This is shown in Fig. 4.7.

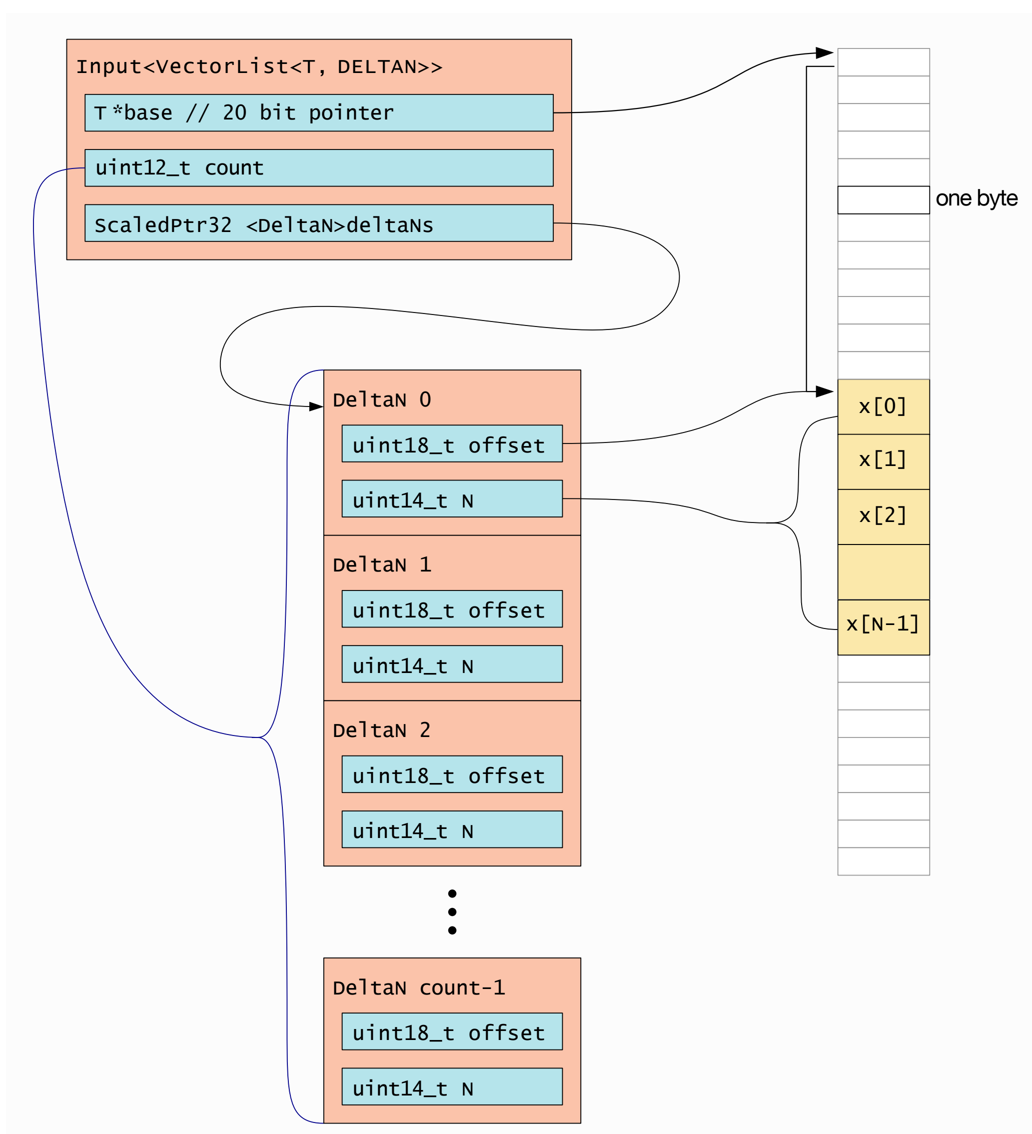


Fig. 4.7: DELTAN memory layout

The base pointer and count are packed into a 32-bit word, as shown in Fig. 4.8.

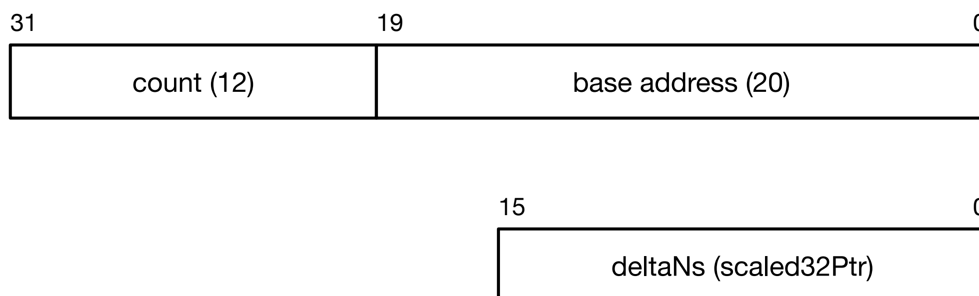


Fig. 4.8: DELTAN base structure bit packing

Each `DeltaN` represents a sub-vector. Its data pointer is stored as an 18-bit offset, in bytes, from the `base` address. The size is stored in the remaining 14 bits, as illustrated in Fig. 4.9.

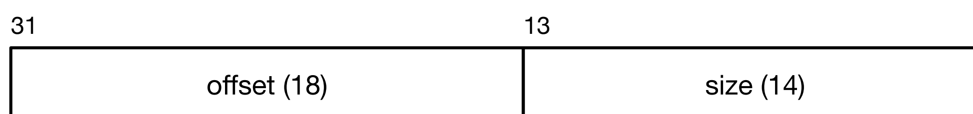


Fig. 4.9: DELTAN sub-vector structure bit packing

This means that the sub-vectors each have a maximum of size of 16,383 (2^{14}-1) elements.

COMPACT_DELTAN layout

This layout will resolve into the most suitable inner pointer type, depending on the available address space.

For example on Mk1 it is equivalent to `DELTAN`, as that can point to everything in the available memory. On Mk2, it is equivalent to `DELTANELEMENTS`.

CHAPTER

FIVE

USING THE POPLAR LIBRARY

The Poplar library provides classes and functions to implement graph programs. These can be accessed by including the appropriate header files. For example:

```
#include <poplar/Graph.hpp>
#include <poplar/Program.hpp>
#include <poplar/Engine.hpp>

using namespace poplar;
```

You do not need any special command line tools to compile Poplar programs. You just use the standard host C++ compiler and link with the Poplar library, as shown below:

```
$ g++ -std=c++11 my-program.cpp -lpoplar
```

The header files are in the `include/poplar` directory of the Poplar installation. The library file is the `lib` directory.

The main classes defined in the Poplar library are summarised below.

- **Graph:** The class used to create a graph structure defining the connections between tensors and operations.
- **Device:** Represents a physical device or simulation that will execute a graph.
- **Tensor:** A class for representing and operating on tensors.
- **Type:** Represents data types on the target device (to distinguish them from types on the host). These include:
  - **INT:** 32-bit integer
  - **SHORT:** 16-bit integer
  - **CHAR:** 8-bit integer (signed by default)
  - **FLOAT:** IEEE 32-bit floating point
  - **HALF:** IEEE 16-bit floating point
  - **BOOL:** Boolean value (stored as one byte)
- **Program:** The base class for creating control programs that define how vertices will be executed. Complex control programs can be built up by combining sub-programs in various ways. The sub-classes for creating and combining programs include:
  - **Execute:** The basic class for creating a program from a compute set
  - **Sequence:** Executes a sequence of sub-programs sequentially
  - **Repeat:** Execute a sub-program a fixed number of times
  - **If:** Conditionally execute a sub-program
- **Engine:** From a graph and one or more control programs, creates an object that can be used to execute the graph on a device.

For full details of all the classes and functions in the Poplar library, see the Poplar and PopLibs API Reference.

CHAPTER

SIX

THE POPLIBS LIBRARIES

The PopLibs libraries provide application-level functions that can be used in programs for the IPU. The available libraries are listed in the table below.

Library	Description
poplin	Linear algebra functions (matrix multiplications, convolutions)
popnn	Functions used in neural networks (for example, non-linearities, pooling and loss functions)
popops	Functions for operations on tensors in control programs (elementwise functions and reductions)
poprand	Functions for populating tensors with random numbers
popsparse	Functions for operating on sparse tensors
poputil	General utility functions for building graphs

Examples using the library functions can be found in the Graphcore GitHub tutorials repository.

For details of all the functions in the PopLibs libraries, see the Poplar and PopLibs API Reference.

6.1 Using PopLibs

The PopLibs libraries are in the `lib` directory of the Poplar installation. Each library has its own include directory and library object file. For example, the include files for the `popops` library are in the `include/popops` directory:

```
#include <include/popops/ElementWise.hpp>
```

You will need to link the relevant PopLibs libraries with your program, in addition to the Poplar library. For example:

```
$ g++ -std=c++11 my-program.cpp -lpoplar -lpopops
```

Some libraries are dependent on other libraries, which you will also need to link with your program. See the Poplar and PopLibs API Reference for details.

CHAPTER SEVEN

WRITING VERTICES IN ASSEMBLY

This chapter introduces the general concepts required for writing vertices in assembly code. For other Graphcore-specific terminology, see the Glossary.

Table 7.1: Terminology used in this document

Notation	Meaning
ARF	Auxiliary register file; the registers associated with the auxiliary execution pipeline.
Aux	The auxiliary execution pipeline.
Bundle	A group of instructions executed in parallel.
Context	The complete and distinct environment for a single thread of execution. Each tile supports a single supervisor context alongside a number of worker contexts. State, such as the registers, is replicated for each context.
CSR	Control and/or status register
LIW	Long instruction word. An architecture that executes multiple instructions in parallel.
Main	The main execution pipeline.
MRF	Main register file; the registers associated with the main execution pipeline.
Naturally aligned	Natural alignment is where the alignment of a data object in memory is equal to the data size.
PC	Program counter
SP	Stack pointer
Supervisor	A single execution context. The supervisor execution thread is responsible for: • Initiating worker threads • Performing the exchange and barrier synchronisation phases of a superstep. A single tile supports just one supervisor context. Supervisor code cannot perform some operations, most notably floating point. See the *Tile Worker ISA* document for more information. This is available from Graphcore support on request.
Worker	A single execution context. A worker execution thread is responsible for performing the computation phase of a superstep. A tile has hardware support for multiple worker contexts.

Table 7.2: Notation used in this document

Notation	Meaning
`0xvalue`	A value written using hexadecimal notation
`0bvalue`	A value written using binary notation
`value`	A value written using decimal notation
`value[i]`	Bit `i` of *value*
`value[j:i]`	Bits `i` through `j` of *value*
`$SOME_STATE`	Refers to tile architectural state, such as a register
`$an`	Register *n* in the arithmetic register file (ARF)
`$mn`	Register *n* in the main register file (MRF)

7.1 Instruction set overview

The instruction set architecture (ISA) of the tile processor, including the execution pipeline and registers, is described in detail in the *Tile Worker ISA* document, available on request.

This section introduces some of the concepts that will be referred to in later chapters. For a high-level introduction to the IPU, please refer to the IPU Programmer's Guide

The tile is a highly-deterministic, asymmetric, dual pipeline, long instruction word (LIW) processor. It supports multiple hardware resident execution contexts. These contexts are time multiplexed onto shared hardware resources to achieve high utilisation by hiding local instruction latencies, including memory access and branch latencies.

Each tile includes tightly-coupled local memory which is used to store all code and data required by the tile.

7.1.1 Supervisors and workers

An IPU tile has two types of hardware execution contexts: a supervisor context and six worker contexts. There are six execution slots that can run these contexts. A round-robin schedule is used to time multiplex the execution slots (and therefore active contexts) onto the shared hardware resources.

Initially, there is only a single supervisor thread that runs in all execution slots. When worker threads run they occupy a single execution slot. When six workers are running, the supervisor is suspended until an execution slot is made available by the termination of a worker.

The supervisor can only perform certain operations. For example, it cannot execute floating-point instructions.

Supervisor code is used for overall control of execution, synchronisation and exchanges, but all floating-point processing must be done in a worker context.

Workers can execute instructions individually or in parallel with another instruction as part of an execution *bundle*. A bundle consists of one instruction for the main pipeline and one for the aux pipeline (see Section 7.1.2, Execution pipelines).

You can access the *Vertex state* from assembly code. In the same way that you would expect the `this` variable in C++ to point to the first field inside a class, in a vertex function the equivalent pointer is available in the `$mvertex_base` register.

Vertices with base class `Vertex` have no parameters (all state is accessible through the register `$mvertex_base`). On termination they must provide an exit status using one of the exit instructions. They are not required to preserve any register state.

Vertices with base class `MultiVertex` have a single parameter to the `MultiVertex::compute(unsigned)` method which is the thread ID of the worker running the method. This value is available in the `$WSR` register. It must be masked out of the register which contains several bitfields packed together.

```
get $m0, $WSR
and $m0, $m0, CSR_W_WSR__CTXTID_M1__MASK
```

The total number of invocations of the multi-vertex compute method is provided by the constant `CTXT_WORKERS` defined in `poplar/TileConstants.hpp`.

7.1.2 Execution pipelines

The tile has a pair of asymmetric execution pipelines, *main* and *aux*:

- Main is designed primarily to perform control flow, address manipulation, integer arithmetic and load/store operations
- Aux is designed primarily to perform floating-point based compute

A supervisor thread cannot use the aux pipeline and its associated state.

Each pipeline has an associated register file. The main execution pipeline is associated (and tightly coupled) with the main register file (MRF) and the aux pipeline with the auxiliary register file (ARF).

These register files, as well as control and status registers and some internal state, are replicated for each context.

The are 16 32-bit registers in the main and arithmetic register files. These are referenced by the names `$mn` and `$an`. Some of these registers have predefined functions and some are read only, as shown in Table 7.3. See the *Tile Worker ISA* document for full details.

Table 7.3: Special registers

Register	Alias	Function	Read only
`$m9`	`$fp`	Frame pointer	N
`$m10`	`$lr`	Link register (return address)	N
`$m11`	`$sp`	Stack pointer	N
`$m12`	`$mworker_base`	*Worker stack base*	Y
`$m13`	`$mvertex_base`	*Vertex stack base*	Y
`$m15`	`$mzero`	Returns zero	Y
`$a15`	`$azero`	Returns zero	Y
`$a14:15`	`$azeros`	Returns 64-bit zero	Y

Note: The registers used as `$fp`, `$lr` and `$sp` are normal general-purpose registers and could be used for any purpose. Their special function is just a convention defined by the ABI. The other registers in the table have hardware-defined functions.

For full details of all the registers, see the ISA reference manual.

7.2 Memory architecture

The architectural size of the tile memory is limited to 21 address bits (2 MB). The tile memory is the only memory directly accessible by tile instructions. It contains both the code and data used by that tile. There is no shared memory access between tiles.

The tile uses a contiguous unsigned 21-bit address space, beginning at address 0x00000. Every context, both worker and supervisor, has visibility of the entire address space. In practice, only a part of this memory space is populated with memory. The physical memory has a non-zero start address as a simple way to prevent invalid zero-valued addresses from being accessed. Attempting to access an unpopulated memory address will cause an exception.

The memory is organised as two regions, each made up of a number of 64-bit wide banks. Concurrent accesses can be made to addresses in different banks. This allows, for example, a 64-bit instruction fetch and two 64-bit data accesses to occur simultaneously (one may be a write).

Accesses to the banks in region 1 are *interleaved*, with bit 3 of the address selecting 64-bit words from alternating odd and even banks. A pair of banks containing interleaved addresses form a single 128-bit wide memory *element*.

Fig. 7.1 shows the layout of non-interleaved and interleaved memory on Mk1. The organisation on the Mk2 IPU is similar, but the number and addressing of the memory banks is different. See *Mk2 Colossus (GC200)*.

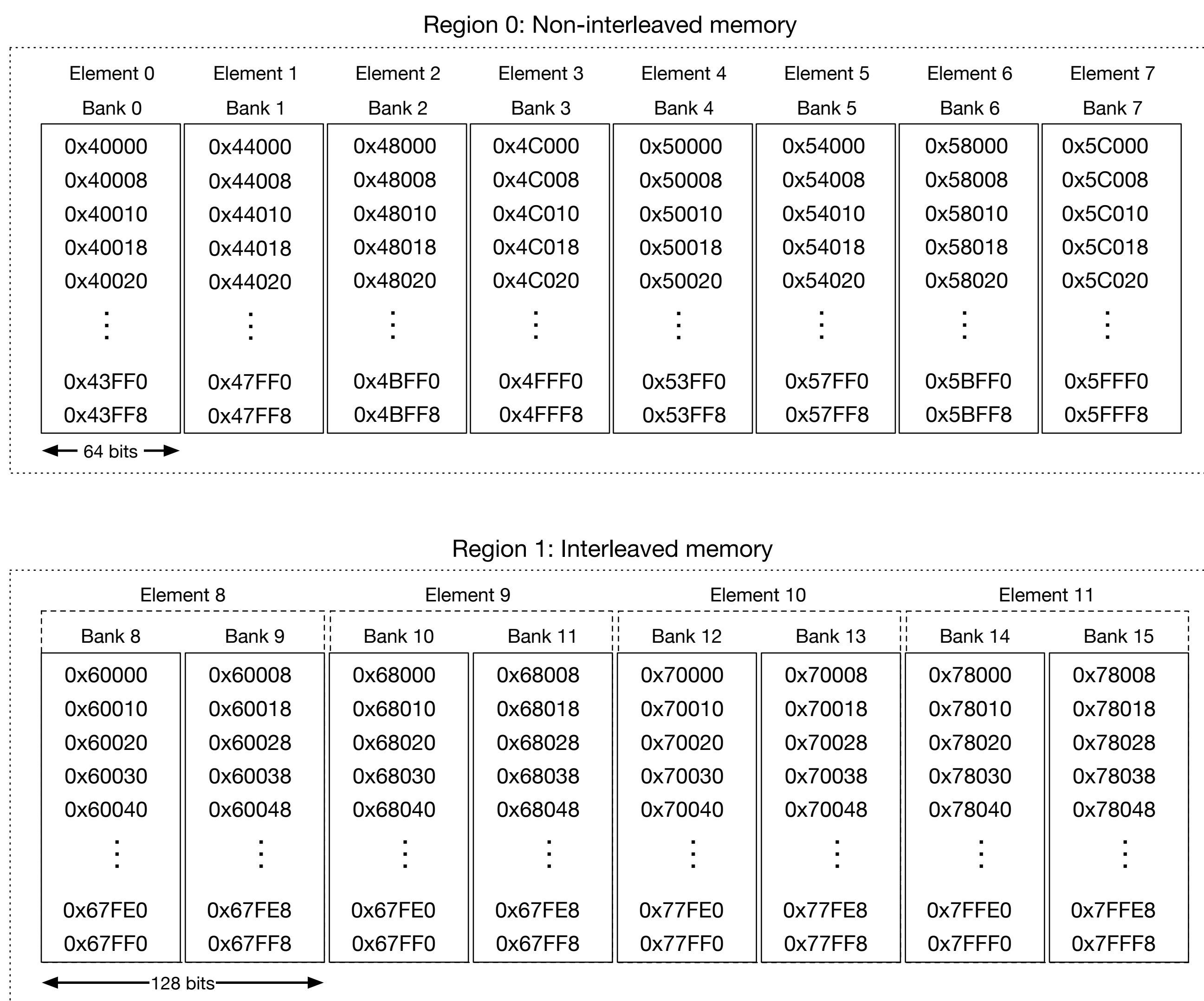


Fig. 7.1: Interleaved and non-interleaved memory regions on Mk1 Colossus

Interleaving allows for two 64-bit aligned addresses, **a** and **a+8**, to be accessed simultaneously. This enables, for example, a 128-bit load and a simultaneous 64-bit load or store. Such simultaneous accesses would cause a *memory clash* exception in the non-interleaved memory region (unless the two addresses happened to straddle the boundary between two banks).

Instructions can only be fetched from region 0. Attempting to execute code from interleaved memory will cause an exception.

7.2.1 Getting information about the memory

The Poplar API provides details of the hardware that the software is actually executing on.

- `Target::getBytesPerTile()` function returns the size of tile memory in bytes.
- `getMemoryElementOffsets()` returns an array containing the offsets, from the start of memory, of each of the *elements* (not banks). For the Mk1 Colossus, for example, this will return 12 values, corresponding to the eight 16 KB elements and the four 32 KB elements.
- `getInterleavedMemoryElementIndex()` returns the index of the first element in interleaved memory (for Mk1 Colossus, for example, this will return 8).

Details of these and other related functions can be found in the Poplar and PopLibs API Reference.

7.2.2 Mk1 Colossus (GC2)

In the Mk1 Colossus, each tile has 256 kilobytes of SRAM, made up of two regions each of 128 KB, as shown in Fig. 7.2. This means that an IPU with 1,216 tiles has about 300 MB of memory in total.

The available memory starts at address 0x40000 and ends at 0x7FFFF.

Table 7.4: Memory organization for Mk1

Region	Size	Interleaved	Banks (16 KB)	Elements (size)
0	128 KB	N	8	8 (16 KB)
1	128 KB	Y	8	4 (32 KB)

Region 0 is selected when bit 17 of the address is 0, and is addressed with bits [16:3]. Bits [16:14] select the bank, or memory element, and bits [13:3] select a 64-bit word from that bank.

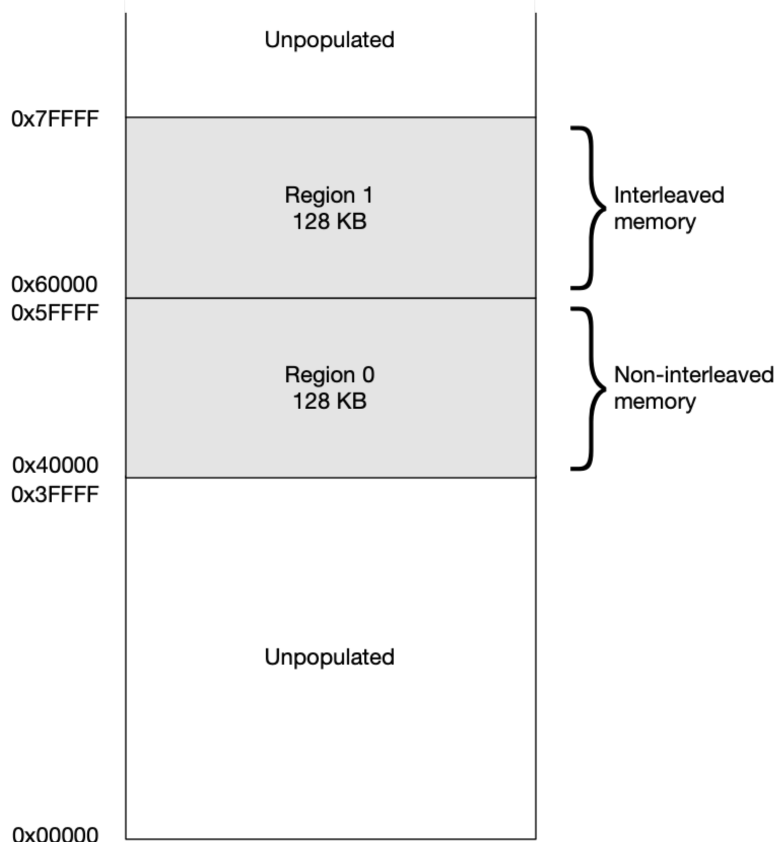


Fig. 7.2: Memory architecture for Mk1 Colossus

7.2.3 Mk2 Colossus (GC200)

In the Mk2 Colossus, each tile has 624 kilobytes of SRAM (see Fig. 7.3). This means that an IPU with 1,472 tiles has just under 900 MB of memory in total.

The available memory starts at address 0x4C000 and ends at 0xE7FFF.

Table 7.5: Memory organization for Mk2

Region	Size	Interleaved	Banks (16 KB)	Elements (size)
0	208 KB	N	13	13 (16 KB)
1	406 KB	Y	26	13 (32 KB)

Region 0 is selected when bit 19 of the address is 0, and is addressed with bits [18:3]. Bits [18:14] select the bank, or memory element, and bits [13:3] select a 64-bit word from that bank.

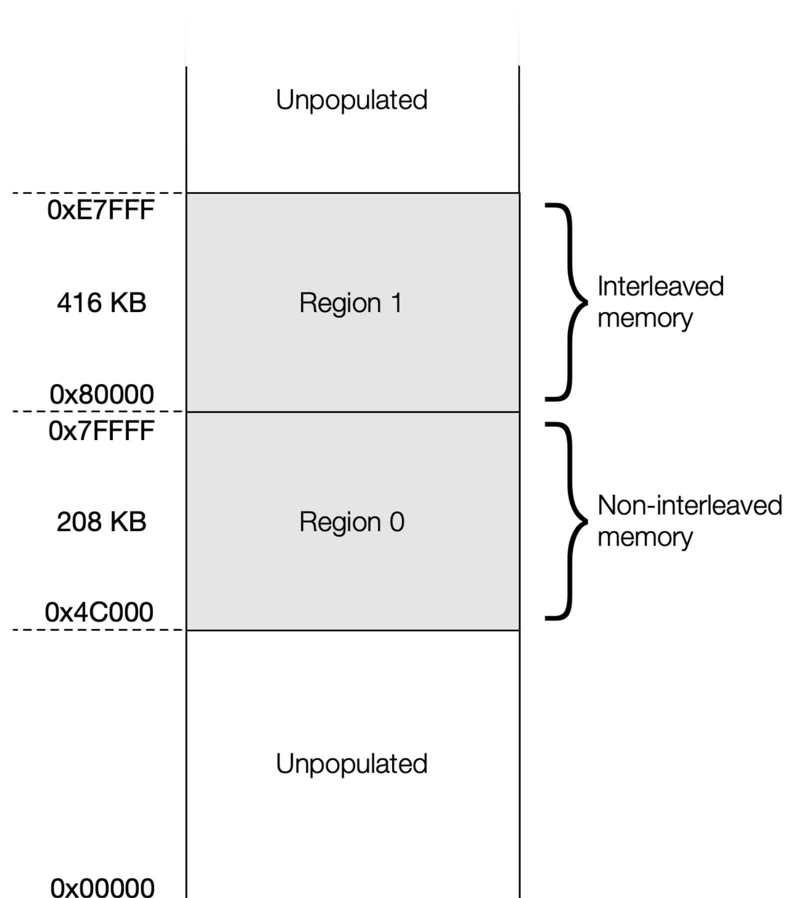


Fig. 7.3: Memory architecture for Mk2 Colossus

7.2.4 Load and store instructions

There are load instructions for the following data sizes:

- 8 bit
- 16 bit
- 32 bit
- 64 bit
- 128 bit (only from region 1)

And store instructions for the following data sizes:

- 32 bit
- 64 bit

There are instructions that can perform multiple simultaneous loads, as well as instructions that do a simultaneous load and store.

If you try to make more than one access to a memory bank in one cycle you will get a memory conflict. Interleaved memory places sequential 64-bit words in alternating banks, allowing you to use more efficient load-store instructions like `ld128`, `ld2xst64pace`, etc. See *Vertex pipelines* for an example of their use.

All loads (including instruction fetches) and stores must be naturally aligned. Misaligned accesses will result in an exception.

7.3 Worker stack and scratch space

The codelet is allocated a stack and a small "scratch space", which can be used for temporary storage.

The stack and scratch space are arranged as shown below.

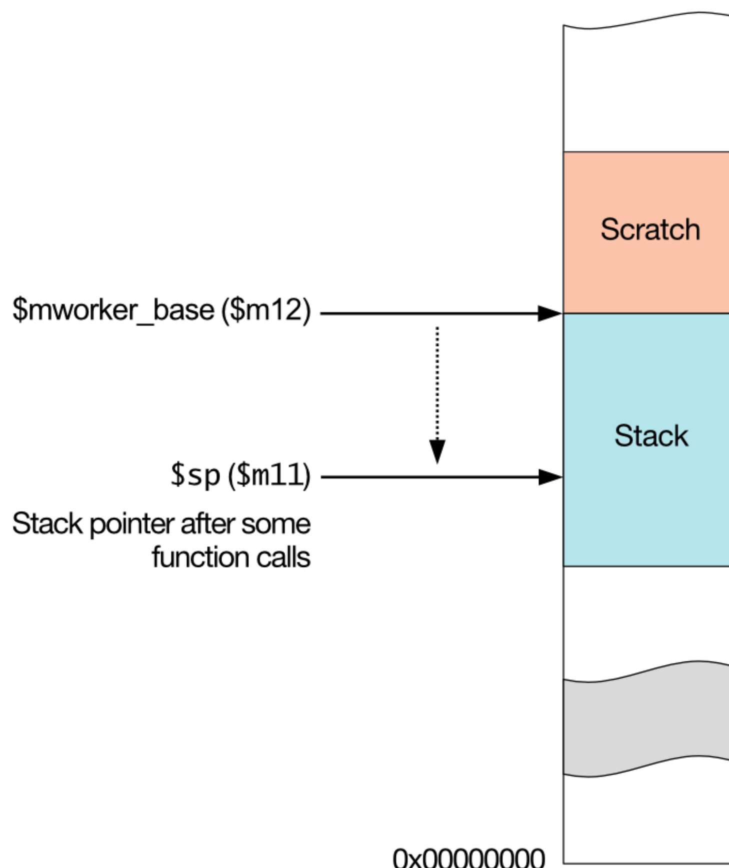


Fig. 7.4: Stack memory layout

The register `$mworker_base` (an alias for `$m12`) is a read-only register that is initialised to the top of the stack, which is also the bottom of the scratch space. The stack pointer, `$sp` (an alias for `$m11`), is not initialised. If you want to use the stack you must initialise `$sp` to (`$mworker_base - stack_allocation_size`) before storing objects on the stack.

Any functions that are called will need to allocate stack by adjusting the stack pointer, for example to allocate 32 bytes of stack for a function:

```
add $sp, $sp, -32
```

The stack space will need to be deallocated before the function returns.

If your code does not need any stack space, you can directly store items in the scratch space using `$mworker_base` and use `$m11/$sp` as a general purpose register instead.

The size of the scratch space is defined by `poplar::WORKER_SCRATCH_SIZE` in `Engine.hpp`. This cannot be changed.

The size of the stack needs to be defined for each function as described below.

7.3.1 Specifying stack size

When C++ functions are compiled, the compiler is usually able to determine the stack required. This is not possible for assembly code so you must explicitly specify the stack used by your functions. This enables the Poplar runtime software to determine the total stack usage when it builds the graph.

Note that C++ versions of these macros are also provided for those cases where the compiler is not able to determine the stack space required

Note: If the stack usage for an assembly function is *not* specified (or is specified more than once) then graph compilation will fail with an exception.

If the stack usage is *wrongly* specified (that is, smaller than required) then this can result in a stack overflow at run time. This will not be detected (stack overflow detection works only for C++ codelets) and may cause errors that are hard to debug.

A couple of macros are provided for specifying the stack size of functions. These are defined in the Poplar header file `StackSizeDefs.hpp` which you will need to include in your source file. See the runtime API section of the Poplar API Reference for more information.

- `DEF_STACK_SIZE_OWN size function`

  This defines the stack size (in bytes) used *only* by the specified function. The Poplar runtime will traverse the call graph (the tree of all functions called, with the root being the specified function) to calculate the total stack required. You will also need to explicitly define the stack size of all functions called (or jumped to) by `function`.

  Note that if the assembly function calls library or other C++ functions then you must use this macro and not `DEF_STACK_USAGE`.

- `DEF_STACK_USAGE size function`

  This defines the total stack usage (in bytes) for the function specified *and any functions that it calls*. In this case, Poplar will *not* traverse the call graph of the function and you do not need to specify the stack usage of the functions called by `function`.

Most assembly vertices don't use a stack and have a simple code structure so the `DEF_STACK_USAGE` macro is often all you need.

The top-level entry point in the assembly vertex (the `__runCodelet_XXXXX` function) *always* needs to be marked with one of the two macros, even if the stack size is zero.

In these macros, you can specify `function` as either a function name or a section name.

If each function is contained in its own section (as it should be, so that the Poplar runtime can remove unused code) then it makes no difference whether you specify the function name or the section name.

If you specify a section name and that section contains multiple functions then `size` is the maximum stack space used by any of the functions in the section.

7.3.2 Examples

1. A codelet that uses no stack space at all:

   ```
   DEF_STACK_USAGE  0  __runCodelet_XXXX
   ```

   This specifies that the Poplar runtime does not need to look at the call graph of the function, as no stack used. This may be the most common case.

2. A codelet using `N` bytes of stack space in total:

   ```
   DEF_STACK_USAGE  N  __runCodelet_XXXX
   ```

Again, the Poplar runtime does not need to look at the call graph of the function, as the stack usage is fully specified.

3. A codelet function that uses N bytes of stack space itself, but which calls other functions. The assembler will need to traverse the call graph to compute the total stack usage:

```
DEF_STACK_SIZE_OWN  N  __runCodelet_XXXX
```

7.4 Vertex pipelines

A tile can simultaneously execute memory and arithmetic operations in a single cycle using an instruction bundle. You can double the performance of some loops if you can arrange the instructions to take advantage of this.

For example, consider a function that adds a value to all the elements of an array. The code for this, using vectors of two-floats, is:

```
float2 addend = ...;
float2* acts = ...;
for (unsigned i = 0; i < N; ++i)
    acts[i] += addend;
```

In other words:

1. Load 64 bits of acts (two floats).
2. Add the two addend floats to them.
3. Store them back and increment the `acts` pointer by 64 bits.
4. Loop.

A simple implementation using a `rpt` loop is shown below. The `rpt` instruction takes two operands: a loop count (which can be an immediate or a register) and the length of the loop body, which follows the instruction.

```
  rpt $N, (2f - 1f)/8 - 1
1:
  {
    ld64 $tmp0, $mzero, $acts, 0
    fnop
  }
  {
    nop
    f32v2add $tmp0, $tmp0, $addend
  }
  {
    st64step $tmp0, $mzero, $acts+=, 1
    fnop
  }
2:
```

This works, but look at all those `nop` instructions. We can do better. First, there are instructions that do both a load and a store in one cycle. This means we can load the *next* 64 bits of `addend` at the same time as storing the previous 64 bits by using the `ldst64pace` instruction. That also provides control over how the pointers are incremented.

Second, we can put the `f32v2add` in the same bundle as the `ldst64pace`. So we want our unrolled loop to be something like this:

```
...
{
  ldst64pace .... // tmp = acts[i]
  f32v2add   ....
}
{
  ldst64pace ....
```

(continues on next page)

(continued from previous page)

```
  f32v2add   .... // tmp += addend
}
{
  ldst64pace .... // acts[i] = tmp
  f32v2add   ....
}
...
```

Here, the load-store instruction in each bundle stores the result of the add from the previous bundle, and loads the data for the add in the following bundle. In other words, the load instruction is fetching data *two* words ahead of the store instruction.

We can implement this as a loop:

```
  rpt ..., (2f - 1f)/8 - 1
1:
  {
    ldst64pace $tmp0, $tmp0, $acts_packed+=, $mzero, 0
    f32v2add $tmp1, $tmp1, $addend
  }
  {
    ldst64pace $tmp1, $tmp1, $acts_packed+=, $mzero, 0
    f32v2add $tmp0, $tmp0, $addend
  }
2:
```

The register `$acts_packed` contains both the load and store addresses.

Unrolled, ths loop would effectively execute this:

```
{
  acts[0] = tmp0, tmp0 = acts[2]
  tmp1 += addend   // acts[1]
}
{
  acts[1] = tmp1, tmp1 = acts[3]
  tmp0 += addend   // acts[2]
}
{
  acts[2] = tmp0, tmp0 = acts[4]
  tmp1 += addend   // acts[3]
}
{
  acts[3] = tmp1, tmp1 = acts[5]
  tmp0 += addend   // acts[4]
}
...
```

If you look at `acts[2]`, it is loaded in the first bundle, added in the second, and then stored in the third. We aren't handling the beginning or end of the loop correctly yet, we'll come back to that. Just think about the loop body for now.

7.4.1 Memory conflicts

This code looks like it would work, and it *almost* does. But if you try to execute it you will get a memory conflict exception. This is because concurrent loads or stores must be done to different memory banks.

Using interleaved memory (see *Memory architecture*) can help in many cases, because alternate 64-bit words are in different memory banks. That means a 128-bit load can be done in one cycle, or a load-store instruction can store to one word and simultaneously read from the following word.

However, in our loop the loads and stores are two words apart. We store to the 64-bit word `i` at the same time as loading from `i+2`, which is 16 bytes offset from it. This means that these are in the same memory bank, even in interleaved memory.

7.4.2 Modified pipeline

To make use of the interleaved memory banks, the store needs to be delayed by an extra cycle. This means we need need to add a stage to the pipeline.

So, instead of this pipeline:

Cycle	Load	Add	Store
0	0		
1	1	0	
2	2	1	0
3	3	2	1
4	4	3	2
5	5	4	3
6	6	5	4
7		6	5
8			6

We do this:

Cycle	Load	Add	Store
0	0		
1	1	0	
2	2	1	
3	3	2	0
4	4	3	1
5	5	4	2
6	6	5	3
7		6	4
8			5
9			6

This requires additional temporary values. We can implement this as shown below in pseudo-code:

```
rpt {
  {
    acts[0] = tmp0, tmp0 = acts[3]
    tmp2 += addend
  }
  {
    acts[1] = tmp1, tmp1 = acts[4]
    tmp0 += addend
  }
  {
    acts[2] = tmp2, tmp2 = acts[5]
    tmp1 += addend
  }
}
```

Or in assembly:

```
  rpt ..., (2f - 1f)/8 - 1
1:
  {
    ldst64pace $tmp0, $tmp0, $acts_packed+=, $mzero, 0
    f32v2add $tmp2, $tmp2, $addend
  }
  {
    ldst64pace $tmp1, $tmp1, $acts_packed+=, $mzero, 0
    f32v2add $tmp0, $tmp0, $addend
  }
  {
    ldst64pace $tmp2, $tmp2, $acts_packed+=, $mzero, 0
    f32v2add $tmp1, $tmp1, $addend
  }
2:
```

And now it works. In some cases you may not need to use explicit temporary values. For example, instead of using `f32v2add` we could use `f32v2axpy` which uses an internal accumulator as a temporary value.

The instruction `f32v2axpy $dst, $src0, $src1` performs the following two operations simultaneously (it effectively has an internal 1-stage pipeline):

- `$dst = $internal_accumulator`
- `$internal_accumulator = $TAS * $src0 + $src1`

`$TAS` is a scale register. You can set it to 1.0 if you just want addition.

We can use this to simplify our loop to the following:

```
  rpt ..., (2f - 1f)/8 - 1
1:
  {
    ldst64pace $tmp1, $tmp1, $acts_packed+=, $mzero, 0
    f32v2axpy $tmp0, $tmp0, $addend
  }
  {
    ldst64pace $tmp0, $tmp0, $acts_packed+=, $mzero, 0
    f32v2axpy $tmp1, $tmp1, $addend
  }
2:
```

This saves eight bytes, and more importantly frees up an ARF register.

7.4.3 Fill and drain

Until now we have ignored the start and end of the loop. In practice we must take special care with filling and draining the pipeline because:

1. We don't want to write junk before the start of our output array.
2. We don't want to process junk after the end of our input array.

Note that it is ok to *read* junk after the end of our input array. However processing it (for example, with `f32v2axpy`) may raise a floating point exception. Also, although we can read past the end of our data, we can't read past the end of the tile's memory space, so depending on your alignment, stride, etc. you may need to avoid any over-reading.

Therefore, we essentially need to duplicate the loop body a few times before and after the loop, but changing the `ldst64pace` as appropriate to avoid extra reads and writes. For example:

```
  // Copy the address of the start of the acts as the store address, and
  // the load address for the pipeline fill stage.
  mov $mscratch0, $acts

  // Cycle 0:      Load 0
```

(continues on next page)

(continued from previous page)

```
  ld64step $tmp0, $mzero, $mscratch0+=, 1

  // Cycle 1:      Load 1    Add 0
  {
    ld64step $tmp1, $mzero, $mscratch0+=, 1
    f32v2axpy $azeros, $current_addend0, $tmp0
  }

  // Cycle 2:      Load 2    Add 1
  {
    ld64step $tmp0, $mzero, $mscratch0+=, 1
    f32v2axpy $tmp1, $current_addend0, $tmp1
  }

  // First address is the load pointer. Second is ignored. Third is the
  // store pointer.
  // By using ld64step above we have set $mscratch0 to $acts+24
  tapack $acts_packed, $mscratch0, $mzero, $acts

  // We do two of the C loop bodies, in each rpt body, so divide N by 2.
  shr $rpt_count, $N, 1

  rpt $rpt_count, (2f - 1f)/8 - 1
1:
  {
    ldst64pace $tmp1, $tmp1, $acts_packed+=, $mzero, 0
    f32v2axpy $tmp0, $tmp0, $addend
  }
  {
    ldst64pace $tmp0, $tmp0, $acts_packed+=, $mzero, 0
    f32v2axpy $tmp1, $tmp1, $addend
  }
2:

  // We may need 1 or 2 more stores depending on whether or not $N is odd.
  and $mscratch0, $N, 0x01
  brnz $mscratch0, 1f
  // If it's even, do this:
  {
    st64pace $tmp1, $acts_packed+=, $mzero, 0
    f32v2axpy $tmp0, $addend, $azeros
  }
  st64pace $tmp0, $acts_packed+=, $mzero, 0
  bri 2f
1:
  // If it's odd, do this:
  {
    ldst64pace $tmp1, $tmp1, $acts_packed+=, $mzero, 0
    f32v2axpy $tmp0, $addend, $tmp0
  }
  {
    st64pace $tmp0, $acts_packed+=, $mzero, 0
    f32v2axpy $tmp1, $addend, $azeros
  }
  st64pace $tmp1, $acts_packed+=, $mzero, 0
2:
```

This example is even more complex because the loop body actually processes *four* floats per loop. So, if our array is 10 floats (for example), we need to add in another half of the `rpt` body at the end. Handling the ends of arrays can get tedious!

7.5 Assembly hints & tips

This chapter introduces some tips that may be useful when writing assembly code for the IPU.

This covers ideas related to assembly language, the IPU architecture and some more general comments.

7.5.1 Using the assembler

For full details of the assembler syntax and features please see the GNU Assembler Manual.

Assembler macros

The assembler supports macros which can be useful for avoiding code repetition, especially when creating versions of a vertex that use different types. The basic syntax is:

```
// Define a simple example macro
.macro GET_PARAMS VOFF_IN_PTR VOFF_OUT_START_PTR VOFF_OUT_END_PTR

  brz      $m0, dummy_label@
  ld32     IN_PTR, $mzero, $mvertex_base,VOFF_IN_PTR
  ld32     OUT_START_PTR, $mzero, $mvertex_base,VOFF_OUT_START_PTR
  ld32     OUT_END_PTR, $mzero, $mvertex_base,VOFF_OUT_END_PTR

dummy_label@: .endm

// Use the simple example macro

GET_PARAMS 0 1 2
```

This substitutes the parameters and inserts the generated code. The `@` syntax simply inserts a number which increments each time any macro is used so the assembler produces unique labels which can be referenced within the macro. There are other useful assembler macro features, see the GNU assembler manual for details.

Labels

In assembly code you can define a label like this:

```
my_label:
```

Any use of the label refers to its address. So, for example, you can branch to a label like this:

```
bri my_label
```

`my_label` will become a symbol in the final ELF file. You can also create "local" symbols that can be jumped to within the file, but are not stored in the ELF symbol table. This is done by prefixing the label with `.L`

```
.Linfinite_loop:
  bri .Linfinite_loop
```

Here `.Linfinite_loop` will not appear in the symbol table.

Another useful trick is the labels `0:` to `9:`. These are also local labels: they do not appear in the symbol table.

```
0:
  bri 0b
```

However, you can use them multiple times. In order to specify one of these labels you append `b` or `f`. `0b` means "the nearest `0:` searching backwards" and `0f` means "the nearest `0:` searching forwards". This is quite useful for short conditional blocks:

```
  // Set $b to 5 if $a is not 0.
  brz $a, 1f
  setzi $b, 5
1:
```

It is also useful for calculating the size of `rpt` bodies:

```
  rpt $n, (2f - 1f)/8 - 1
1:
  {
    ...
  }
  {
    ...
  }
  {
    ...
  }
2:
```

Recording the code size of the vertex

We recommended that you include a `.size`size directive at the end of a function definition. This allows tools like ``objdump`` to report the space used by the function. The information may be used by Poplar to choose between a faster specialisation or a smaller, general-purpose vertex.

This can be done as shown below:

```
myfunc:

  ...

.size myfunc, .-myfunc
```

Place each vertex in a unique section

With compiled code, you can use the linker option `--function-sections` to place each function in its own section and then remove unused sections for dead code elimination.

For functions written in assembly, the section needs to be specified explicitly. Each function should be in a different section with a name of the form `.section .text.VERTEX_NAME`, where `VERTEX_NAME` is the symbol of the function.

The same thing applies to data objects, they should be placed inside sections with names similar to `.section .data.SYMBOL_NAME`.

NOTE: new sections have a default alignment of 1 so you need to add a `.align` directive at the start of every section (either 4 or 8 depending on whether you have a `rpt` in your function or not).

7.5.2 Architectural tips

Aligning repeat bodies

A `rpt` body consists of a sequence of instruction bundles, and these must be 8-byte aligned. This is the only case where instructions must be 8-byte aligned (instruction bundles in general only need to be 4-byte aligned).

The `rpt` instruction itself only needs to be 4-byte aligned.

During development of a vertex you can ensure this alignment automatically like this:

```
.align 8
{
  rpt $n, (2f - 1f)/8 - 1
  nop
}
1:
  {
    ...
  }
  {
    ...
  }
2:
```

The `.align 8` will ensure that the `rpt` bundle is 8-byte aligned (the assembler will insert a `nop`, if necessary to achieve the alignment). Then, since the `rpt` bundle is 8 bytes, the loop body will also be 8-byte aligned.

However, this can result in code like this:

```
nop // inserted by the .align 8
{
  rpt $n, (2f - 1f)/8 - 1
  nop
}
1:
  {
    ...
  }
  {
    ...
  }
2:
```

Instead, it would be better to have:

```
rpt $n, (2f - 1f)/8 - 1
1:
  {
    ...
  }
  {
    ...
  }
2:
```

Therefore, in the final code you should remove the `.align 8` directives and then manually remove the `nop` instructions from the `rpt` bundles as necessary to align them.

The assembler will report an error if the body is not aligned. So the easiest way to check the alignment is to assemble the code and see if there is an error.

For a (very small) improvement in power consumption, it is better to place a `nop` in a place that will never be executed (such as above the entry point or after an unconditional branch) rather than bundling an instruction with a `nop` to correct the alignment. If this isn't easy to do then it isn't worth worrying about too much, as the effect is small.

Over-reading and over-processing

In pipelined loops (see *Vertex pipelines*) you might do a load of the data required in the next iteration. This can result in *over-reading*; reading beyond the memory occupied by data structure being processed.

Over-reading memory does not cause a problem. Poplar will always arrange vertex data so that it is safe to over-read up to eight 32-bit words without causing a memory exception.

However, it is important that any data read from outside the data structure is not used in any computation. Any such *over-processing* could use uninitialised data which could, for example, cause a floating point exception.

Scratch space

Workers get a stack space, which includes a small scratch area that can be used without having to set up the stack pointer. This is useful if you run out of registers. See *Worker stack and scratch space* for details.

Loading constants

You can use `f16v2exp $dst, $azero` to load `[1.0h, 1.0h]` into a register in a single instruction. This is special-cased so it only takes one cycle and produces an exact result.

Similarly, you can use `f16v2sigm $dst, $azero` to load `[0.5h, 0.5h]`. This is also special-cased and only takes one cycle and provides an exact result.

7.5.3 Division by 6

It is common to need to split work between the six workers. Six is an awkward number to divide by so a couple of methods are suggested here.

Division on the IPU

The fastest way to approximate a division by six on the IPU is to multiply by 0xaaab and shift right by 18.

```
count_per_worker = (n * 0xaaab) >> 18;
```

Note: this only works for numbers less than 98,304 ($n < 6 \times 2^{14}$).

Division on the host

If you also want the remainder, an alternative method is to do the division on the host. The result and the remainder can be stored a single `uint16_t` by using some simple bit shifting and masking.

For example, in the Poplar C++ code:

```
uint16_t count = ((n / 6) << 3) | (n % 6);
```

In the assembly codelet:

```
// Load the count
ldz16 $count, $mvertex_base, $mzero, OFFSET

// Get the worker ID.
get $worker_id, $WSR
and $worker_id, $worker_id, CSR_W_WSR__CTXTID_M1__MASK

// Get blocks per worker and remainder.
shr $count_per_worker, $count, 3
and $remainder, $count, 0x7
```

(continues on next page)

(continued from previous page)

```
// Work out where this worker should begin begin, accounting for remainders
mul $begin, $count_per_worker, $worker_id
min $mscratch0, $worker_id, $remainder
add $begin, $begin, $mscratch0

// Add remainder to workers with IDs less than the remainder.
cmpult $mscratch0, $worker_id, $remainder
add $count_per_worker, $count_per_worker, $mscratch0
```

7.5.4 General

Focus on optimising the vectorised case

A lot of vertices will need to handle unaligned (or left-over) elements individually before processing as many as possible at the widest vectorised width (4 for `float`, 8 for `half`).

In the initial implementation, don't worry about writing the edge cases as optimally as possible, processing one element at a time in a loop is fine.

Making these unaligned cases faster can be done in the future, if profiling shows it to be necessary.

Testing

When testing a handwritten vertex it is important to check all of the edge cases and branches inside the assembly code. A recommended way to do this is by creating a single graph that has multiple compute sets, with each compute set containing the vertex to be tested and the data for a different test case.

If you need to check that your code works for any alignment you may need to load a whole tensor and execute multiple test cases that slice the tensor at different offsets from the beginning to check each possible alignment.

Bit twiddling

A large number of code examples for bit manipulation can be found on the website Bit Twiddling Hacks.

CHAPTER

EIGHT

PROFILING

You can use Poplar to instrument your code to produce detailed compile-time information about the graph program and run-time information about execution, including how memory is used, and where memory and processor cycles are consumed.

This information can be output to files and viewed with the PopVision Graph Analyser. This provides a graphical view of the profile data and includes full documentation as context-sensitive help. See the document Poplar Profile Data for a detailed description of the contents of these files.

You can also generate a summary of the profiling information for quick review. See *Profile summary*.

The profiling information available depends on the target:

Target	Memory Profiling	Execution Profiling
IPU	Exact	Hardware measurement with limitations
IPU Model	Exact (optional)	Detailed but based on estimates
CPU	None	None

The `IPUModel::compileIPUCode` option, described below, can be used to generate exact memory profiling information for an IPU Model.

Because profiling adds code and extra variables to extract the profiling information, it can change the performance and memory usage of your program.

8.1 Generating profiling information

After you have loaded your `Graph` into an `Engine`, you can get static profile information about the graph and the resources required. This includes cycle estimates (for an IPU Model) and memory information.

```
ProfileValue graphProfile = engine.getGraphProfile();
```

After you have run the program one or more times you can get dynamic profiling information (what code was run, cycle counts, and so on).

```
ProfileValue executionProfile = engine.getExecutionProfile();
```

You can save the profiling information to files for use by the Graph Analyser. For example:

```
poplar::serializeToJSON(graphFile, graphProfile, true);
poplar::serializeToJSON(executionFile, executionProfile, true);
```

The last parameter of `serializeToJSON()` is optional and controls whether or not to pretty print the data.

8.1.1 Profiling options

There are some options for controlling profiling on IPU targets (hardware or simulator).

By default, profiling is disabled. The instrumentation of compute cycles and external exchange cycles can be enabled with the following options:

- `debug.instrument` Set to "true" enable instrumentation.
- `debug.instrumentCompute` Set to "true" or "false" to enable or disable instrumentation of compute cycles.
- `debug.instrumentExternalExchange` Set to "true" or "false" to enable or disable instrumentation of cycles used for exchanges between IPUs, or between the IPU and the host.

Note that there is no option to instrument *internal* exchanges because all internal exchange is statically scheduled.

If the instrumentation of compute is enabled, then the compute cycles counted can be specified with the `debug.computeInstrumentationLevel` option. This can have the following values:

- **"tile"** Store the cycle count for the last execution of each compute set on every tile (default).
- **"vertex"** Store the cycle count for the last execution of each vertex on every tile.
- **"ipu"** Store the cycle count for the last execution of each compute set on a single tile per IPU. This measures the execution time of the longest-running tile in the compute set. This saves memory compared to "tile" but loses all the per-tile cycle information.

These can be specified when the `Engine` constructor is called. For example:

```
Engine engine(graph, prog,
              OptionFlags({{"debug.instrumentCompute", "true"},
                           {"debug.instrumentExternalExchange", "false"},
                           {"debug.computeInstrumentationLevel", "tile"}}));
```

These options can also be defined in the environment variable `POPLAR_ENGINE_OPTIONS`. For example:

```
export POPLAR_ENGINE_OPTIONS='{"debug.instrument": "true",
                               "debug.computeInstrumentationLevel": "vertex"}'
```

For complete information about the available options, see the Poplar and PopLibs API Reference.

For IPU Model targets you can optionally tell Poplar to compile code for the IPU (in addition to the CPU code that is actually executed by the model). If this is not done, then the reported memory usage will not include memory used for code. If this is enabled then the memory profile should give the same results as an IPU target. This option is specified by setting the `compileIPUCode` members of the model, for example:

```
// Create the IPU Model device
IPUModel ipuModel;
ipuModel.compileIPUCode = true;
```

8.2 Profile summary

You can generate a readable summary of the information, which may be useful for a quick overview of the resource usage and performance of your program. This summary report can be printed by the program or generated from the profile data.

8.2.1 Printing from a Poplar program

To print a summary of both the graph and execution profiling information you can call the function:

```
poplar::printProfileSummary(std::cout, graphProfile,
                            executionProfile, options);
```

You can also print the graph and execution summaries separately by calling `printGraphSummary` or `printExecutionSummary` (see the Poplar and PopLibs API Reference for more information).

The following options are available:

- `colours`: Control the use of colours in the summary output
- `showVarStorage`: Show liveness information for each program
- `showOptimizations`: Show compile optimisation details
- `showExecutionSteps`: Show the execution steps
- `showPerIpuMemoryUsage`: Show memory usage for each IPU

Colours can be used to highlight different sections of the output, to make it easier to understand. By default, colour is enabled when output is to a supported terminal. By setting the `colours` option to `"true"`, colour output will be generated even if outputting to a file or piping the output to another program.

8.2.2 Command line conversion

A program is included with the Poplar SDK to convert the generated profile data into a readable summary. This has options `--graph-profile` and `--execution-profile` to specify the graph and execution profile files to use. For example:

```
$ poplar_profile_summary --graph-profile graph_profile.json
```

The following command line options can be used to control the summary output:

- `--show-execution-steps`: Show execution steps
- `--show-optimizations`: Show optimisation information
- `--show-per-ipu-memory-usage`: Report the memory usage for each IPU
- `--show-var-storage`: Show liveness information for each program

8.2.3 Summary report format

There are two environment variables that can also be used to control colour output. These are:

- `CLICOLOR`: If set to 0, then colour output will not be generated. This overrides the `colours` option value.
- `CLICOLOR_FORCE`: If set to 1, then colour output will always be generated, even if output is not to the terminal. This overrides the `colours` option value and the `CLICOLOR` environment variable.

If there are replicated graphs then the memory usage will only be shown for one replica, as it will be the same for all of them.

If any tiles are out of memory, the memory used a few tiles with the largest usage will be shown.

The output with `showVarStorage` and `showExecutionSteps` will be similar to that shown below.

If any tiles are out of memory, the memory used by a few tiles with the largest usage will be shown.

```
Target:
  Number of IPUs:         1
  Tiles per IPU:          1,216
  Total Tiles:            1,216
  Memory Per-Tile:        256.0 kB
  Total Memory:           304.0 MB
  Clock Speed (approx):   1,600.0 MHz
  Number of Replicas:     1
  IPUs per Replica:       1
  Tiles per Replica:      1,216
  Memory per Replica:     304.0 MB

Graph:
  Number of vertices:            5,564
  Number of edges:              18,564
  Number of variables:          47,973
  Number of compute sets:           43

Memory on all IPUs:
Memory Usage:
  Total:
    Including Gaps:         67,725,024 B
    Excluding Gaps:
      By Memory Region:
        Non-interleaved:     6,522,641 B
        Interleaved:           197,696 B
        Overflowed:                  0 B
      Total:                 6,720,337 B
      By Data Type:
        Not Overlapped
            Variables:                              254,280 B
            Internal Exchange Message Buffers:      109,788 B
            Data Rearrangement Buffers:                  96 B
            Host Exchange Packet Headers:            10,720 B
            Stack:                                3,852,288 B
            Vertex Instances:                        65,768 B
            Copy Descriptors:                        16,161 B
            VectorList Descriptors:                     528 B
            Vertex Field Data:                       24,052 B
            Control Code:                           467,696 B
            Vertex Code:                          1,097,868 B
            Internal Exchange Code:                 215,364 B
            Host Exchange Code:                     211,132 B
          Total:                                  6,325,741 B
        Overlapped
            Variables:                              340,056 B
            Internal Exchange Message Buffers:      697,128 B
            Data Rearrangement Buffers:              15,424 B
          Total:                                  1,052,608 B
          Total After Overlapping:                  394,596 B
      Vertex Data (106,509B):
        By Category:
          Internal vertex state:         34,114 B
          Edge pointers:                 38,876 B
          Copy pointers:                 16,668 B
          Padding:                          346 B
          Descriptors:                   16,505 B
        By Type:
          poplin::ConvPartial1x1Out<float,float,true,false>                              185,792 B
          poplin::ReduceAdd<float,float>                                                 117,728 B
          poprand::SetSeedSupervisor                                                      34,048 B
          poplar_rt::Memcpy64BitSupervisor                                                31,468 B
          poplar_rt::DstStridedCopyDA32                                                   30,242 B
          poplin::Transpose2d<float>                                                      23,784 B
          popops::ScaledAddSupervisor<float,float,true>                                   18,336 B
          popnn::NonLinearityGradSupervisor<float,popnn::NonLinearityType::SIGMOID>       13,024 B
          popnn::NonLinearitySupervisor<float,popnn::NonLinearityType::SIGMOID>           12,512 B
          poplar_rt::StridedCopyDA32                                                       9,043 B
          popops::EncodeOneHot<unsigned int,float>                                         3,120 B
          poplar_rt::DstStridedCopy64BitMultiAccess                                        2,677 B
          popops::Reduce<popops::ReduceAdd,float,float,false,2>                            1,632 B
```

(continues on next page)

(continued from previous page)

```
        poplar_rt::ShortMemcpy                                                        1,324 B
        popops::ScaledAdd2D<float,true>                                               1,164 B
        popops::ScaledReduce<popops::ReduceAdd,float,float,true,1>                      888 B
        popnn::LossSumSquaredTransform<float>                                           816 B
        poplar_rt::StridedCopy64BitMultiAccess                                          719 B
        poplar_rt::DstStridedMemsetZero64Bit                                            416 B
        popnn::ReduceMaxClassGather<float,unsigned int>                                 384 B
        popops::Reduce<popops::ReduceAdd,float,float,false,1>                           368 B
        poplar_rt::MemsetZero                                                           172 B
        popops::Reduce<popops::ReduceAdd,float,float,false,3>                           152 B
        popnn::CalcAccuracy<unsigned int>                                                72 B
        poplar_rt::MemsetZero64Bit                                                       68 B
        popops::ScaledReduce<popops::ReduceAdd,float,float,true,0>                       56 B


  By Tile (Excluding Gaps):
    Range (KB) Histogram (Excluding Gaps)               Count (tiles)
        3 -  4 ****************************************    824
        4 -  5                                              0
        5 -  6                                              0
        6 -  7                                              0
        7 -  8 ********                                   168
        8 -  9 ***                                         44
        9 - 10 ******                                     126
      10 - 11 **                                          30
      11 - 12 *                                           16
      12 - 13                                             0
      13 - 14                                             0
      14 - 15 *                                           2
      15 - 16 *                                           2
      16 - 17 *                                           3
      17 - 18                                             0
      18 - 19                                             0
      19 - 20 *                                           1

    Maximum (Including Gaps): 131,608 (128.5 K) on tile 0
    Maximum (Excluding Gaps): 20,294 (19.8 K) on tile 0
    0 tile(s) out of memory



  Variable Storage Liveness: All tiles

    Always-live bytes: 5,976,425
    Always-live variables:
      <anon>                                                                                           48
      <const>                                                                                         120
      Layer1/Fwd/Conv_1/worklists                                                                  14,112
      Layer3/Bwd/Conv_1/worklists                                                                      72
      Layer3/Fwd/Conv_1/worklists                                                                     144
      Layer3/Wu/ReduceFinalStage/IntermediateToOutput/numPartials                                      60
      Layer5/Bwd/LossSumSquared/offset                                                                112
      Layer5/Bwd/LossSumSquared/reduce_loss/ReduceOnTile/InToIntermediateNoExchange/numPartials        32
      Layer5/Bwd/LossSumSquared/sliceLen                                                              128
      ValuePadder/padding                                                                              16
      controlCode                                                                                 467,696
      copyDescriptor                                                                               16,161
      hostExchangeCode                                                                            211,132
      hostExchangePacketHeader                                                                     10,720
      internalExchangeCode                                                                        215,364
      numCorrect                                                                                        4
      stack                                                                                     3,852,288
      vectorListDescriptor                                                                            528
      vertexCode                                                                                1,097,868
      vertexFieldData                                                                              24,052
      vertexInstanceState                                                                          65,768

    Maximum live bytes (including always-live): 6,556,737

    Sequence
```

(continues on next page)

(continued from previous page)

```
      StreamCopy
        Live Bytes (excluding always-live): 95,452
        Live Vars (excluding always-live):
          <anon>                        12
          Layer1/Fwd/biases            120
          Layer1/Fwd/weights        94,080
          Layer3/Fwd/biases             40
          Layer3/Fwd/weights         1,200
      StreamCopy
        Live Bytes (excluding always-live): 95,456
        Live Vars (excluding always-live):
          <anon>                        12
          Layer1/Fwd/biases            120
          Layer1/Fwd/weights        94,080
          Layer3/Fwd/biases             40
          Layer3/Fwd/weights         1,200
          programId                      4
      DoExchange: switchControlBroadcast13/ExchangePre
        Live Bytes (excluding always-live): 100,312
        Live Vars (excluding always-live):
          <anon>                         8
          Layer1/Fwd/biases            120
          Layer1/Fwd/weights        94,080
          Layer3/Fwd/biases             40
          Layer3/Fwd/weights         1,200
          broadcastProgramId         4,860
          programId                      4
      Switch
        Live Bytes (excluding always-live): 100,312
        Live Vars (excluding always-live):
          <anon>                         8
          Layer1/Fwd/biases            120
          Layer1/Fwd/weights        94,080
          Layer3/Fwd/biases             40
          Layer3/Fwd/weights         1,200
          broadcastProgramId         4,860
          programId                      4
        Sequence
          DoExchange: init/setMasterSeed/ExchangePre
            Live Bytes (excluding always-live): 105,168
            Live Vars (excluding always-live):
              <anon>                         8
              <message:anon>             9,720
              Layer1/Fwd/biases            120
              Layer1/Fwd/weights        94,080
              Layer3/Fwd/biases             40
              Layer3/Fwd/weights         1,200
          OnTileExecute: init/setMasterSeed
            Live Bytes (excluding always-live): 105,168
            Live Vars (excluding always-live):
              <anon>                         8
              <message:anon>             9,720
              Layer1/Fwd/biases            120
              Layer1/Fwd/weights        94,080
              Layer3/Fwd/biases             40
              Layer3/Fwd/weights         1,200
...

Execution:

  Total cycles:                                  54,202 (approx 33.9 microseconds)
  Total compute cycles (including idle threads): 1,933,324
  Total compute cycles (excluding idle threads): 1,769,242
  Total IPU exchange cycles:                     482,221
  Total global exchange cycles:                  0
  Total host exchange cycles:                    4,194,423
  Total shared structure copy cycles:            0
  Total sync cycles:                             58,852,900
  Total tile balance:                            2.9%
  Total thread balance:                          91.5%
```

(continues on next page)

(continued from previous page)

```
  Cycles by vertex type:
    poplin::ConvPartial1x1Out<float,float,true,false>                         (808 instances):      764,028
    poplar_rt::DstStridedCopyDA32                                             (1393 instances):      509,173
    poprand::SetSeedSupervisor                                                (1216 instances):      177,536
    poplin::ReduceAdd<float,float>                                            (376 instances):      154,116
    popops::ScaledAddSupervisor<float,float,true>                             (460 instances):      139,812
    poplar_rt::StridedCopyDA32                                                (455 instances):       83,995
    poplar_rt::Memcpy64BitSupervisor                                          (203 instances):       59,048
    popnn::NonLinearitySupervisor<float,popnn::NonLinearityType::SIGMOID>      (68 instances):       15,780
    popnn::NonLinearityGradSupervisor<float,popnn::NonLinearityType::SIGMOID>  (68 instances):       14,628
    poplar_rt::ShortMemcpy                                                     (73 instances):        5,256
    poplin::Transpose2d<float>                                                 (99 instances):        4,770
    popops::EncodeOneHot<unsigned int,float>                                   (20 instances):        4,408
    popops::Reduce<popops::ReduceAdd,float,float,false,2>                     (112 instances):        3,584
    poplar_rt::DstStridedCopy64BitMultiAccess                                  (41 instances):        2,437
    popops::ScaledReduce<popops::ReduceAdd,float,float,true,1>                 (32 instances):        1,600
    poplar_rt::StridedCopy64BitMultiAccess                                     (33 instances):        1,388
    popops::ScaledAdd2D<float,true>                                            (39 instances):        1,371
    popnn::ReduceMaxClassGather<float,unsigned int>                            (16 instances):        1,056
    popops::Reduce<popops::ReduceAdd,float,float,false,1>                      (16 instances):          912
    popnn::LossSumSquaredTransform<float>                                       (8 instances):          792
    poplar_rt::DstStridedMemsetZero64Bit                                       (18 instances):          740
    popops::Reduce<popops::ReduceAdd,float,float,false,3>                       (4 instances):          144
    popops::ScaledReduce<popops::ReduceAdd,float,float,true,0>                  (2 instances):          100
    popnn::CalcAccuracy<unsigned int>                                           (1 instances):           78
    poplar_rt::MemsetZero64Bit                                                  (2 instances):           26
    poplar_rt::MemsetZero                                                       (1 instances):           15
Steps
  --- External Sync ---
  StreamCopy (cycles 118 - 118)
    Cycles:                         7
    Active Tiles:                   1 / 1,216
    Tile Balance:                   0.1%
    Active Tile Balance:            100.0%
    Total Data:                     2
    Data Balance (mean / max data per tile): 0.1%
  --- Internal Sync ---
  --- Internal Sync ---
  --- External Sync ---
  StreamCopy (cycles 472 - 472)
    Cycles:                         9
    Active Tiles:                   1 / 1,216
    Tile Balance:                   0.1%
    Active Tile Balance:            100.0%
    Total Data:                     4
    Data Balance (mean / max data per tile): 0.1%
  --- Internal Sync ---
  --- Internal Sync ---
  --- Internal Sync ---
  DoExchange (cycles 826 - 946): switchControlBroadcast13/ExchangePre
    Cycles:                         120 (0 overlapped with previous)
    Active Tiles:                   1,216 / 1,216
    Tile Balance:                   86.6%
    Active Tile Balance:            86.6%
    Total Data:                     2,432
    Data Balance (mean / max data per tile): 100.0%
  --- External Sync ---
  StreamCopy (cycles 1,064 - 1,064)
    Cycles:                         7
    Active Tiles:                   1 / 1,216
    Tile Balance:                   0.1%
    Active Tile Balance:            100.0%
    Total Data:                     2
    Data Balance (mean / max data per tile): 0.1%
  --- Internal Sync ---
  --- Internal Sync ---
  --- External Sync ---
  StreamCopy (cycles 1,418 - 12,098)
    Cycles:                         10,700
```

(continues on next page)

(continued from previous page)

```
    Active Tiles:                    392 / 1,216
    Tile Balance:                    32.2%
    Active Tile Balance:             100.0%
    Total Data:                      25,122
    Data Balance (mean / max data per tile): 21.1%
  --- Internal Sync ---
  --- Internal Sync ---
  --- Internal Sync ---
  DoExchange (cycles 12,432 - 12,581): Layer1/Fwd/Conv_1/Convolve/ExchangePre
    Cycles:                          169 (20 overlapped with previous)
    Active Tiles:                    392 / 1,216
    Tile Balance:                    27.9%
    Active Tile Balance:             86.6%
    Total Data:                      125,440
    Data Balance (mean / max data per tile): 32.2%
  OnTileExecute (cycles 12,581 - 13,580): Layer1/Fwd/Conv_1/Convolve
    Cycles:                          999 (0 overlapped with previous)
    Active Tiles:                    392 / 1,216
    Thread Balance:                  100.0%
    Tile Balance:                    31.7%
    Active Tile Balance:             98.3%
    By vertex type:
      poplin::ConvPartial1x1Out<float,float,true,false>        (392 instances):      384,944
....
```

CHAPTER

NINE

ENVIRONMENT VARIABLES

There are several environment variables which you can use to control the behaviour of the Poplar SDK.

9.1 Logging

Logging messages can be generated when your program runs. This is controlled by the environment variables described below. For more detailed information about the execution of your program, see *Profiling output*.

- `POPLAR_LOG_LEVEL`: Enable logging for Poplar
- `POPLAR_MODULE_LOG_LEVEL`: Enable logging for one Poplar module
- `POPLAR_LOG_DEST`: Specify the destination for Poplar logging
- `POPLIBS_LOG_LEVEL`: Enable logging for PopLibs
- `POPLIBS_MODULE_LOG_LEVEL`: Enable logging for one PopLibs module
- `POPLIBS_LOG_DEST`: Specify the destination for PopLibs logging

9.1.1 Logging level

The Poplar modules are:

- `CODELETS`
- `ENGINE`
- `GRAPH`
- `PROFILER`
- `PROGRAM`
- `TARGET`
- `TENSOR`

The PopLibs modules are:

- `POPFLOAT`
- `POPLIN`
- `POPNN`
- `POPOPS`
- `POPRAND`
- `POPSOLVER`
- `POPSPARSE`

- POPUTIL

If not specified, each module uses the logging level specified by POPLAR_LOG_LEVEL or POPLIBS_LOG_LEVEL.

The following example runs a program with Engine logging messages at log level "TRACE", all other Poplar modules at log level "INFO" and no PopLibs logging.

```
$ POPLAR_LOG_LEVEL=INFO POPLAR_ENGINE_LOG_LEVEL=TRACE ./my_poplar_program
```

The supported logging levels are shown in the table below.

Table 9.1: Poplar logging levels

"OFF"	No logging information. The default.
"ERR"	Only error conditions will be reported.
"WARN"	Warnings when, for example, the software cannot achieve what was requested (for example, if the convolution planner can't keep to the memory budget, or Poplar has determined that the model won't fit in memory but the debug.allowOutOfMemory option is enabled).
"INFO"	Very high level information, such as PopLibs function calls.
"DEBUG"	Useful per-graph information.
"TRACE"	The most verbose level. All useful per-tile information.

All Poplar log messages are prefixed with "PO". Messages from PopLibs are prefixed with "PL".

9.1.2 Logging destination

The logging information is sent to standard error (stderr) by default. This can be changed by setting the POPLAR_LOG_DEST or POPLIBS_LOG_DEST environment variables. The value can be "stdout", "stderr" or a filename.

9.2 Profiling output

- POPLAR_ENGINE_OPTIONS: Provide options to generate profiling data for use by the PopVision Graph Analyser.

In order to capture the reports needed for the PopVision Graph Analyser you only need to set POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true"}' before you run a program.

For more information, and other options, see *Profiling* and the PopVision User Guide.

9.3 Setting options

The following environment variables can be used to override the option values specified in the program:

- POPLAR_ENGINE_OPTIONS
- POPLAR_SIMULATOR_OPTIONS

For more information about the available options, see the Poplar and PopLibs API Reference.

CHAPTER

TEN

APPLICATION BINARY INTERFACE (ABI)

This chapter describes the Colossus IPU 32-bit application binary interface (ABI).

The Executable and Linkable Format (ELF) defines a linking interface for compiled programs. This document is the processor-specific supplement for use with ELF on the 32-bit Colossus IPU. It is intended for linking objects compiled in C, C++ and assembly code.

The ELF specification can be found in: System V Application Binary Interface, Edition 4.1.

10.1 Types

The data types used by Poplar are described in the tables colossus_abi_scalar_types and colossus_abi_vector_types. In addition:

- By default the `char` type is signed.
- `long` is the same as `int`.
- `long double` is the same as `double`.
- The underlying type of an enumerated type is `int`.
- Function pointers are the same as data pointers.
- `double` and `long double` are not supported by Poplar targets.
- `long`, `long long`, `unsigned long`, `unsigned long long` are not supported on the IPU.

Table 10.1: Scalar data types

Type	Size (bits)	Align (bits)	Meaning
`char`	8	8	Character type
`short`	16	16	Short integer
`int`	32	32	Integer
`long`	32	32	Integer
`long long`	64	64	Integer
`half`	16	16	16-bit IEEE float
`float`	32	32	32-bit IEEE float
`double`	64	64	64-bit IEEE float
`long double`	64	64	64-bit IEEE float
`void *`	32	32	Data pointer

Table 10.2: Vector data types

Size (bits)	Align (bits)
32	32
64	64
128	64

10.1.1 Floating point types

The IPU has hardware support for `float` and `half`. For CPU targets, `half` is, by default, an alias for float (and `sizeof(half)` will be 4).

The parameter `accurateHalf` can be set to true when creating a CPU target, in which case `half` will be correctly implemented as 16-bit IEEE floating point. This will be slower, but will produce the same results as the IPU.

Codelets should be written to be generic to the size of `half` so that changing this setting requires no code changes.

10.1.2 Structure types

Structure types pack according to the standard rules:

- Field offsets are aligned according to the field's type.
- A structure is aligned according to the maximum alignment of its members.
- Tail padding is added to make the structure's size a multiple of its alignment.

10.1.3 Bit fields

The following types may be specified in a bit-field's declaration: `char`, `short`, `int`, `long` , `long long` and `enum`.

If an `enum` type has negative values, `enum` bit-fields are signed. Otherwise, if a signed integer type of the specified width is not able to represent all `enum` values then enum bit-fields are unsigned. Otherwise, `enum` bit-fields are signed. All other bit-field types are signed unless explicitly unsigned.

Bit-fields pack from the least significant end of the allocation unit. Each non-zero bit field is allocated at the first available bit offset that allows the bit field to be placed in a properly aligned container of the declared type. Non bit-field members are allocated at the first available offset satisfying their declared type's size and alignment constraints.

A zero-width bit-field forces padding until the next bit-offset aligned with the bit field's declared type.

Unnamed bit-fields are allocated space in the same manner as named bit-fields.

A structure is aligned according to each of the bit field's declared types in addition to the types of any other members. Both zero-width and unnamed bit fields are taken into account when calculating a structure's alignment.

10.2 Vertex calling convention

Vertex functions (codelets) are only called by the supervisor thread and have no arguments.

On entry to a vertex codelet, the `$mvertex_base` and `$mworker_base` registers hold the address of the vertex state structure and the address of the worker thread's scratch space. A vertex codelet is not required to preserve callee save registers, therefore all `$m` and `$a` registers can be used as scratch storage.

Worker vertices have no parameters (all state is accessible through the register `$mvertex_base`). On termination they must provide a boolean exit status to the supervisor thread, using one of the exit instructions.

10.3 Function calling convention

10.3.1 Function parameters

Function calling uses `$m0` to `$m3` to pass integer parameters and `$a0` to `$a5` to pass floating point parameters. Additional parameters are passed on the stack.

Parameters of 64 or 128 bits must be passed in aligned register pairs or quads respectively. An aligned register pair is numbered even then odd, for example `$m0:1` or `$m2:3`. An aligned register quad starts with a register whose number modulo four is zero, for example `$a0:3`. Proceeding arguments can only be passed in the remaining consecutive registers.

All variadic function parameters are passed via the stack.

Scalar types smaller than 32 bits are passed as zero- or sign-extended 32-bit values.

An aggregate containing a single member is passed as if it was an argument of that member type. Otherwise aggregates are passed by passing a pointer to the aggregate. The callee is allowed to write to the pointed to aggregate.

10.3.2 Return values

Integer return values are passed in `$m0` to `$m3` and floating-point return values are passed in `$a0` to `$a3`.

Scalar types smaller than 32 bits are returned as zero- or sign-extended 32-bit values.

An aggregate containing a single member is returned as if it was an argument of that member type. Otherwise, the caller passes as an implicit parameter the address of the return destination. This must be a valid address to which the return value can be written. The return destination must not alias any other memory visible to the callee.

10.3.3 Entry and exit

On entry to a function, the caller ensures the link register, `$m10`, holds the address of the instruction to execute after the function returns, and the stack pointer, `$sp`, holds a 64-bit aligned address at the top of the stack.

On exiting a function, the callee resets the stack pointer to its value on entry if it was modified, and copies the on-entry value of the link register register to the program counter, `$pc`.

10.3.4 Register assignments

The register assignments are described in Table 10.3 and Table 10.4.

Table 10.3: MRF register assignments for the function calling convention

Registers	Type	Usage
`$m0` - `$m3`	Caller save	Arguments and return values
`$m4` - `$m6`	Caller save	Scratch
`$m7`	Callee save	Scratch
`$m8`	Callee save	Scratch or base pointer
`$m9`	Callee save	Scratch or frame pointer
`$m10`	Caller save	Link register
`$m11`	Callee save	Stack pointer

- `$m10` register holds the address to return to when a function completes (see *Entry and exit*).
- `$m11` holds the base address of the stack of the current function (see *Stack frame*).

Table 10.4: ARF register assignments for the function calling convention

Registers	Type	Usage
`$a0` - `$a5`	Caller save	Arguments and return values
`$a6`, `$a7`	Callee save	Scratch

10.4 Stack frame

Table 10.5 and Table 10.6 illustrates the organisation of the two stack frames when a function is called. The stack pointer grows downwards towards address 0x0. The outgoing arguments are written so that earlier arguments have smaller offsets from the stack pointer, so are at lower addresses.

See *Worker stack and scratch space* for information on allocating stack memory.

Table 10.5: Stack frame layout (callee)

Callee's frame
outgoing arguments (caller writes)
Local objects and spills
Incoming arguments (callee writes)

Table 10.6: Stack frame layout (caller)

Caller's frame
Incoming arguments (caller writes)
Local objects and spills
Incoming arguments (callee writes)

CHAPTER

ELEVEN

TRADEMARKS & COPYRIGHT

Graphcore® and Poplar® are registered trademarks of Graphcore Ltd.

AI-Float™, Colossus™, Exchange Memory™, Graphcloud™, In-Processor-Memory™, IPU-Core™, IPU-Exchange™, IPU-Fabric™, IPU-Link™, IPU-M2000™, IPU-Machine™, IPU-POD™, IPU-Tile™, PopART™, PopLibs™, PopVision™, PopTorch™, Streaming Memory™ and Virtual-IPU™ are trademarks of Graphcore Ltd.

All other trademarks are the property of their respective owners.

Copyright © 2016-2020 Graphcore Ltd. All rights reserved.