GRAPHCORE

PopART C++ API Reference

Version latest

Graphcore Ltd

Oct 21, 2021

CONTENTS

1 Introduction **1**

2 PopART C++ API **2**
2.1 Sessions . . . 2
2.1.1 Training session . . . 5
2.1.2 Inference session . . . 6
2.1.3 Data input and output (IStepIO) . . . 7
2.1.4 Session options . . . 11
2.2 Optimizers . . . 25
2.2.1 Stochastic Gradient Descent (SGD) . . . 28
2.2.2 Adam, AdaMax & Lamb . . . 34
2.2.3 AdaDelta, RMSProp & AdaGrad . . . 39
2.3 Builder . . . 44
2.4 Data flow . . . 67
2.5 Device manager . . . 70
2.6 Op creation . . . 75
2.6.1 Op definition for PopART IR . . . 75
2.6.2 Op definition for Poplar implementation . . . 86
2.7 Utility classes . . . 87
2.7.1 Tensor information . . . 87
2.7.2 Tensor location . . . 89
2.7.3 Region . . . 91
2.7.4 Error handling . . . 93
2.7.5 Debug context . . . 94
2.7.6 Attributes . . . 94
2.7.7 Void data . . . 96
2.7.8 Input shape information . . . 97
2.7.9 Patterns . . . 97
2.7.10 Type definitions . . . 100

3 Index **102**

4 Trademarks & copyright **103**

Index **104**

CHAPTER

ONE

INTRODUCTION

The Poplar Advanced Run Time (PopART) is part of the Poplar SDK for implementing and running algorithms on networks of Graphcore IPU processors.

This document describes the PopART C++ API. For more information about PopART, please refer to the PopART User Guide.

CHAPTER

TWO

POPART C++ API

2.1 Sessions

```
#include <popart/session.hpp>
```

class popart::**Session**

Session is a runtime instance that provides an interface for executing ONNX graphs on IPU hardware.

Subclassed by *popart::InferenceSession*, *popart::TrainingSession*

Public Functions

~Session() = 0

std::vector<uint32_t> **getRNGState**()

void **setRNGState**(**const** std::vector<uint32_t>)

void **setRandomSeed**(uint64_t *seedValue*)

Sets the random number generator seed on all tiles of the device.

This ensures deterministic behaviour of random operations in the graph.

Parameters

- The: seed value.

void **compileAndExport**(std::string *filename*)

Compiles the graph and exports it to the specified path.

This will create a snap::Graph and compile the poplar::Executable before exporting the executable and metadata.

Parameters

- filename: Name of the file where the compiled executable and associated metadata will be saved.

void **compileAndExport**(std::ostream &*out*)

Compiles the graph and exports it to the specified stream.

This will create a snap::Graph and compile the poplar::Executable before exporting the executable and metadata.

Parameters

- out: Stream where the compiled executable and associated metadata will be written to.

void **loadExecutableFromFile**(std::string *filename*)
Load the `poplar::Executable` and the PopART metadata from the given file.

The file must have been created with *compileAndExport()*

Parameters

- `filename`: Name of the file to load the executable from.

void **loadExecutableFromStream**(std::istream &*in*)
Load the `poplar::Executable` and the PopART metadata from the given stream.

The stream must have been created with *compileAndExport()*

Parameters

- `in`: Stream to load the executable from.

void **prepareDevice**(bool *loadEngine* = true)
Prepare the network for execution.

This will create the `snap::Graph` and `poplar::Engine`.

Parameters

- `loadEngine`: Load the engine and connect the streams once the device is ready.

void **loadEngineAndConnectStreams**()
Load the engine on the device and connect the streams.

This will set up the `poplar::Streams`.

Note: This call is optional. The engine will implicitly be loaded on the device when required.

void **weightsFromHost**()
Write weights from host to the device.

void **weightsToHost**()
Copy the weights to host from the device.

uint64_t **getCycleCount**(std::string *id* = "")
Copy the cycle count tensor to host from the device.

void **connectStreamToCallback**(**const** std::string &*streamHandle*, std::function<void)void*
> *callback*, unsigned *index* = 0Connect Poplar stream callbacks.

In conjunction with `getGradAndVarStreamIds` the streams can be used to copy gradients to the host to perform collective operations after which the variables can be streamed back after they have been updated to the device. `index` referes to the replica index when using replicated graphs.

void **connectStream**(**const** std::string &*streamHandle*, void **buffer*)
Connect a given poplar stream handle with a buffer to copy the memory to or from IPU.

void **run**(*IStepIO* &*stepIO*, std::string *debugName* = "")
Perform one step.

Read input data from address in `stepIO.in`. Write the output data to addresses in `stepIO.out`.

Parameters

- `stepIO`: Input and output data.
- `debugName`: Debug string to identify this run in logs.

void **updateExternallySavedTensorLocations**(**const** std::string &*fromLocation*, **const** std::string &*toLocation*)
Update the tensor locations of the tensors in the *Session*'s ONNX model.

The new file will be created at this point, and written to when the ONNX model is saved with a subsequent call to modelToHost.

Parameters

- `fromLocation`: All externally saved tensors with location fromLocation will have their location updated to toLocation.
- `toLocation`: The updated location. Must not already exist.

void **modelToHost**(**const** std::string &*fn*)
Write current model to ONNX file.

Parameters

- `fn`: Path to file. Can be absolute or relative. If you plan to run your program in multiple processes simultaneously, you should avoid possible race conditions by writing to different files, for example by using temporary files.

TensorInfo **getInfo**(*TensorId*) **const**
Get the TensorInfo on a Tensor.

bool **hasInfo**(*TensorId*) **const**
Returns whether or not a the tensor for the specified ID has info.

If the return value is false, you will be unable to obtain an instance of TensorInfo using getInfo.

std::string **getSummaryReport**(bool *resetProfile* = true) **const**
Retrieve the summary from from the `poplar::Engine`.

The options which were given to the constructor will influence the information in the report.

This may only be called after the *prepareDevice()* call has been made.

Return A string containing the report.

Parameters

- `resetProfile`: Resets the execution profile.

std::string **getSerializedGraph**() **const**
Retrieve the serialized graph from the `poplar::Engine`.

A JSON format report is produced.

This may only be called after the *prepareDevice()* call has been made.

Return A string containing the serialized graph.

pva::Report **getReport**() **const**
Retrieve the graph report from the `poplar::Engine`.

The options which were given to the constructor will influence the information in the report.

This may only be called after the *prepareDevice()* call has been made.

Return The pva report object

void **resetHostWeights**(**const** std::string &*model*, **const** bool *ignoreWeightsInModelWithoutCorrespondingHostWeight* = false)
Reset the weights with the weights in an ONNX model that differs from the current model only in weights.

This only updates the weights on the host; the user still needs to call *weightsFromHost()* after this to update the weights on the device.

Parameters

- `model`: Either an ONNX model protobuf, or the name of a file containing an ONNX model protobuf.
- `ignoreWeightsInModelWithoutCorrespondingHostWeight`: If true, do not error if there are initializers in the ONNX model with no corresponding initializer tensor in the session's IR.

void **`readWeights`**(**`const`** IWeightsIO &*weightsIo*)
Read the weights.

Must have called *weightsToHost()* first.

The weight data is written to the addresses in `weightsIo.out`.

void **`writeWeights`**(**`const`** IWeightsIO &*weightsIo*)
Write the weights.

Must call *weightsFromHost()* after this.

The weight data is written to the addresses in `weightsIo.out`.

std::string **`serializeIr`**(*IrSerializationFormat format*)
Serizalise the IR graph to a string.

Parameters

- `format`: The format to use for serializing.

`const` Ir &**`getIr`**() **`const`**

`const` popx::*Devicex* &**`getDevice`**() **`const`**

popx::*Devicex* &**`getDevice`**()

`const` popx::IrLowering &**`getIrLowering`**() **`const`**

`const` popx::Executablex &**`getExecutable`**() **`const`**

2.1.1 Training session

```
#include <popart/session.hpp>
```

`class` popart::**`TrainingSession`** : **`public`** *popart*::*Session*
TrainingSession is a runtime instance that provides an interface for executing ONNX graphs on IPU hardware with training provided by optimizing the specified loss tensor using the specified optimizer and automatic differentiation (backpropagation)

Public Functions

`~TrainingSession`() **`override`**

void **`updateOptimizerFromHost`**(**`const`** *Optimizer* **optimizer*)
Update the optimizer and the associated hyperparameters but not the optimizer state tensors.

NOTE: The optimizer parameter has to be compatible with the optimizer passed to the constructor. For example, you cannot call this function with an SDG1 optimizer if you created the session with an SDG0 optimizer. The reason for this is that it is not possible to change the IR after it has been constructed.

Parameters

- `optimizer`: A pointer to a *popart::Optimizer*.

void **copyFromRemoteBuffer**(**const** std::string &*buffer*, void **w*, int *repeat_index*, unsigned *replication_index* = 0)
Read from a RemoteBuffer object into a user space pointer `w`.

This can be useful when we run larger models with host side reductions since HEXOPT is currently limited to 128 MB.

void **copyToRemoteBuffer**(void **w*, **const** std::string &*buffer*, int *repeat_index*, unsigned *replication_index* = 0)
Write to a RemoteBuffer object from a user space pointer `w`.

This can be useful when we run larger models with host side reductions since HEXOPT is currently limited to 128 MB.

Public Static Functions

std::unique_ptr<*TrainingSession*> **createFromIr**(std::unique_ptr<Ir> *ir*, std::shared_ptr<*DeviceInfo*> *deviceInfo*, **const** std::string *name* = DefaultTrainingSessionName)

std::unique_ptr<*TrainingSession*> **createFromOnnxModel**(**const** std::string &*model*, **const** *DataFlow* &*dataFlow*, **const** *TensorId* &*loss*, **const** *Optimizer* &*optimizer*, std::shared_ptr<*DeviceInfo*> *deviceInfo*, **const** *InputShapeInfo* &*inputShapeInfo* = *InputShapeInfo*(), **const** *SessionOptions* &*userOptions* = *SessionOptions*(), **const** *Patterns* &*patterns* = *Patterns*(), **const** std::string *name* = DefaultTrainingSessionName)
Create a runtime class for executing an ONNX graph on a set of IPU hardware for training.

Parameters

- `model`: Either an ONNX model protobuf, or the name of a file containing an ONNX model protobuf.
- `inputShapeInfo`: Information about the shapes of input and output tensors.
- `dataFlow`: Configuration for the data feeds and fetches.
- `loss`: The TensorId of the final scalar loss tensor for training.
- `optimizer`: The name of an optimizer to use when training.
- `userOptions`: String to configure session options.
- `patterns`: Optimization patterns to apply.

2.1.2 Inference session

```
#include <popart/session.hpp>
```

class popart::**InferenceSession** : **public** *popart*::*Session*
InferenceSession is a runtime instance that provides an interface for executing ONNX graphs on IPU hardware, without any automatic differentiation (backpropagation) or optimization.

Public Functions

~InferenceSession() override

Public Static Functions

std::unique_ptr<*InferenceSession*> **createFromIr**(std::shared_ptr<Ir> *ir*, std::shared_ptr<*DeviceInfo*> *deviceInfo*, **const** std::string *name* = DefaultInferenceSessionName)

std::unique_ptr<*InferenceSession*> **createFromOnnxModel**(**const** std::string &*model*, **const** *DataFlow* &*dataFlow*, std::shared_ptr<*DeviceInfo*> *deviceInfo*, **const** *InputShapeInfo* &*inputShapeInfo* = *InputShapeInfo*(), **const** *SessionOptions* &*userOptions* = *SessionOptions*(), **const** *Patterns* &*patterns* = *Patterns*(), **const** std::string *name* = DefaultInferenceSessionName)

Create a runtime class for executing an ONNX graph on a set of IPU hardware for inference.

Parameters

- `model`: Either an ONNX model protobuf, or the name of a file containing an ONNX model protobuf.
- `inputShapeInfo`: Information about the shapes of input and output tensors.
- `dataFlow`: Configuration for the data feeds and fetches.
- `userOptions`: String to configure session options.
- `patterns`: Optimization patterns to apply.

2.1.3 Data input and output (IStepIO)

```
#include <popart/istepio.hpp>
```

class popart::**IStepIO**

An abstract base class through which input and output data is passed to a *Session* (see *Session::run*).

Data is passed via buffers. In the case of buffers returned by *IStepIO::in* PopART *reads* from those buffers and in the case of *IStepIO::out* PopART *writes* to these buffers. The *IStepIO::inComplete* and *IStepIO::outComplete* functions are called by PopART to signal it is done with an input or output buffer.

An *IStepIO* implementation should conceptually implement a rolling queue of active buffers for each input and output tensor. Every successful call to *IStepIO::in* should yield a new data buffer for PopART to read from and add it to the head of the conceptual queue. Conversely, every call to *IStepIO::inComplete* should be taken to mean that the buffer at the tail-end of the queue is no longer being used by PopART. This buffer is removed from the conceptual queue.

Note that a *IStepIO::in* call with the `prefetch` flag set is only considered successful when it returns data.

Output works analogously to input.

The expected total number of input (resp. output) buffers that are 'completed' for a tensor in one *Session::run* call is bps $\times$ *SessionOptions::accumulationFactor* $\times$ *SessionOptions::replicatedGraphCount*, where bps is the number of batches per call to *Session::run* (this is a value captured by the *DataFlow* instance passed to *Session*).

Note, however, that there may be additional 'uncompleted' calls to *IStepIO::in* (resp. *IStepIO::out*).

Further more, the number of number of input (resp. output) buffers that may be 'incomplete' *at a given time for a given tensor* should not normally be higher than SessionOptions::bufferingDepth $\times$ *SessionOptions::replicatedGraphCount*, but this bound is not guaranteed.

EXAMPLE: Suppose a session is configured such that the total expected number of input buffers is 6 and these are input buffers for a tensor with ID `"t"` with 100 elements. The associated input calls in *IStepIO* may look like this if SessionOptions::bufferingDepth is 3:

```
in("t", 100, false) -> Give buffer[0] to PopART.
in("t", 100, true) -> Give buffer[1] to PopART.
in("t", 100, true) -> Give buffer[2] to PopART.
inComplete("t", 100) -> buffer[0] is no longer required and can be reused.
in("t", 100, true) -> Give buffer[3] to PopART.
inComplete("t", 100) -> buffer[1] is no longer required and can be reused.
in("t", 100, true) -> Give buffer[4] to PopART.
inComplete("t", 100) -> buffer[2] is no longer required and can be reused.
in("t", 100, true) -> Give buffer[5] to PopART.
inComplete("t", 100) -> buffer[3] is no longer required and can be reused.
in("t", 100, true) -> No data available, return nullptr.
inComplete("t", 100) -> buffer[4] is no longer required and can be reused.
inComplete("t", 100) -> buffer[5] is no longer required and can be reused.
```

Subclassed by *popart::StepIOCallback*, *popart::StepIOGeneric< ARRAY_TYPE, ACCESSOR_TYPE, ArrayInfoT >*, *popart::StepIOGeneric< IArray, StepIONS::IArrayAccessor, IArray &>*

Public Functions

~IStepIO() = default

ConstVoidData **in**(*TensorId* *id*, int64_t *numElements*, bool *prefetch*) = 0

Called to request a new input data buffer.

The memory in this buffer be available for use in PopART until the corresponding inComplete call.

NOTE: Failing to provide a valid data buffer will result in a runtime failure if prefetch is set to `false`.

Return The input buffer for this tensor (or nullptr on failure) wrapped in a *ConstVoidData* object.

Parameters

- `id`: The tensor ID to return data for.
- `numElements`: The number of elements in the tensor.
- `prefetch`: If set to `true` the inability to provide data is not considered an error.

void **inComplete**(*TensorId* *id*, int64_t *numElements*) = 0

Called to notify the user that a previously retrieved input data buffer is no longer used by PopART and it's memory can be reused.

Parameters

- `id`: The tensor ID to return data for.
- `numElements`: The number of elements in the tensor.

MutableVoidData **out**(*TensorId* *id*, int64_t *numElements*) = 0

Called to request a new output data buffer.

The memory in this buffer be available for use in PopART until the corresponding inComplete call and will be modified in-place.

NOTE: Failing to provide a valid data buffer will result in a runtime failure.

Return The output buffer for this tensor wrapped in a *MutableVoidData* object.

Parameters

- `id`: The tensor ID to return data for.
- `numElements`: The number of elements in the tensor.

void **outComplete**(*TensorId*)
Called to notify the user that a previously retrieved output data buffer is no longer used by PopART and it's memory can be reused.

Parameters

- `id`: The tensor ID to return data for.
- `numElements`: The number of elements in the tensor.

void **enableRuntimeAsserts**(bool *b*)

bool **runtimeAssertsEnabled**() **const**

void **assertNumElements**(**const** popx::Executablex&) **const** = 0

```
#include <popart/stepio_generic.hpp>
```

template<typename **ARRAY_TYPE**, typename **ACCESSOR_TYPE**, typename **ArrayInfoT**>
class popart::**StepIOGeneric** : **public** *popart*::*IStepIO*
Subclassed by popart::StepIO

Public Functions

void **assertNumElements**(**const** popx::Executablex &*exe*) **const** **final**

TensorInfo **getTensorInfo**(*ARRAY_TYPE* &*array*) **const**

template<typename **T**>
T **get**(*TensorId* *id*, std::map<*TensorId*, ArrayInfo> &*M*, int64_t *numElements*, bool *advance_*, std::string *mapName*)

template<typename **T**>
void **advance**(*TensorId* *id*, std::map<*TensorId*, ArrayInfo> &*M*, int64_t *numElements*, std::string *mapName*)

ConstVoidData **in**(*TensorId* *id*, int64_t *numElements*, bool) **final**

void **inComplete**(*TensorId* *id*, int64_t *numElements*) **final**

MutableVoidData **out**(*TensorId* *id*, int64_t *numElements*) **final**

..doxygentypedef:: popart::StepIOCallback::InputCallback ..doxygentypedef:: popart::StepIOCallback::InputCallbackComplete
..doxygentypedef:: popart::StepIOCallback::OutputCallback ..doxygentypedef:: popart::StepIOCallback::OutputCallbackComp

class popart::**StepIOCallback** : **public** *popart*::*IStepIO*
Class that implements the *IStepIO* interface using user-provided callback functions.

The *IStepIO* interface contains a number of pure virtual member functions through which PopART receives buffers to read data from and buffers to write data to. This class inherits from *IStepIO* and implements those member functions by delegating the logic to the callback functions passed in the constructor. This gives the user full control as to how data buffers are provisioned.

See *IStepIO* for more details on the expected behaviour of the callbacks.

Public Types

using InputCallback = std::function<*ConstVoidData*(*TensorId*, bool)>
Callable object that implements *IStepIO::in()*.

using InputCompleteCallback = std::function<void(*TensorId*)>
Callable object that implements *IStepIO::inComplete()*.

using OutputCallback = std::function<*MutableVoidData*(*TensorId*)>
Callable object that implements *IStepIO::out()*.

using OutputCompleteCallback = std::function<void(*TensorId*)>
Callable object that implements *IStepIO::outComplete()*.

Public Functions

StepIOCallback(*InputCallback* *inputCallback*, *InputCompleteCallback* *inputCompleteCallback*, *OutputCallback* *outputCallback*, *OutputCompleteCallback* *outputCompleteCallback*)
Construct a new *StepIOCallback* object.

Parameters

- `inputCallback`: The callback function the constructed *StepIOCallback* instance will use when *IStepIO::in()* is called. See *IStepIO* for details on how to implement this method.
- `inputCompleteCallback`: The callback function the constructed *StepIOCallback* instance will use when *IStepIO::inComplete()* is called. See *IStepIO* for details on how to implement this method.
- `outputCallback`: The callback function the constructed *StepIOCallback* instance will use when *IStepIO::out()* is called. See *IStepIO* for details on how to implement this method.
- `outputCompleteCallback`: The callback function the constructed *StepIOCallback* instance will use when *IStepIO::outComplete()* is called. See *IStepIO* for details on how to implement this method.

void **assertNumElements**(**const** popx::Executablex&) **const**

ConstVoidData **in**(*TensorId* *id*, int64_t *numElements*, bool *prefetch*) **final**
This function is called by PopART when a *StepIOCallback* instance is passed to *Session::run()* and will internally call the `inputCallback` parameter passed to the constructor.

You should not call this function directly.

void **inComplete**(*TensorId* *id*, int64_t *numElements*) **final**
This function is called by PopART when a *StepIOCallback* instance is passed to *Session::run()* and will internally call the `inputCompleteCallback` parameter passed to the constructor.

You should not call this function directly.

MutableVoidData **out**(*TensorId* *id*, int64_t *numElements*) **final**
This function is called by PopART when a *StepIOCallback* instance is passed to *Session::run()* and will internally call the `outputCallback` parameter passed to the constructor.

You should not call this function directly.

void **outComplete**(*TensorId* *id*) **final**
This function is called by PopART when a *StepIOCallback* instance is passed to *Session::run()* and will internally call the `outputCompleteCallback` parameter passed to the constructor.

You should not call this function directly.

2.1.4 Session options

```
#include <popart/sessionoptions.hpp>
```

struct popart::**SessionOptions**
A structure containing user configuration options for the *Session* class.

Public Functions

SessionOptions &**operator=**(**const** *SessionOptions* &*rhs*) = default

bool **explicitPipeliningEnabled**() **const**

bool **implicitPipeliningEnabled**() **const**

bool **shouldDelayVarUpdates**() **const**

int64_t **getGlobalReplicationFactor**() **const**
Helper method to handle the different replication options.

If enableDistributedReplicatedGraphs is true return globalReplicationFactor if enableReplicatedGraphs return replicatedGraphCount otherwise return 1

unsigned **getAccumulationFactor**() **const**
Helper method to check the accumulation factor settings for consistency if gradient accumulation is not enabled and the factor is set to >1.

Returns the accumulation factor otherwise.

unsigned **getPrefetchBufferingDepth**(**const** *TensorId* &*id*, unsigned *defaultValue*) **const**
Get the buffering depth for a TensorId.

Will return 1 unless prefetching is enabled and the buffering depth is overwritten in the `prefetchBufferingDepthMap` variable.

Not part of public API

bool **autoRecomputationEnabled**() **const**
Returns true if auto-recomputation is enabled.

Public Members

std::string **logDir**
A directory for log traces to be written into.

std::set<*DotCheck*> **dotChecks** = {}
When to write `.dot` files during Ir construction.

int **firstDotOp** = 0
The ops to write to the `.dot` file will be a continuous interval of the schedule, controlled by firstDotOp and finalDotOp.

In particular, it will be [min(0, firstDotOp), max(N ops in Ir, finalDotOp)).

int **finalDotOp** = 10000
See *firstDotOp*.

bool **dotOpNames** = false
Include the Op name in the `.dot` file (the Op type is always exported).

bool **exportPoplarComputationGraph** = false
Export Poplar computation graph.

bool **exportPoplarVertexGraph** = false
Export Poplar vertex graph.

bool **separateCallOpPdfs** = true
: When generating PDFs of IR graphs, create separate PDFs for each subgraph.

bool **enableOutlining** = true
: Identify and extract repeated parts of computational graph into subgraphs.

bool **enableOutliningCopyCostPruning** = true
: When `true` the cost of copying of cached sections should be included in the outlining cost model.

float **outlineThreshold** = 1.0f
: The incremental value that a sub-graph requires, relative to its nested sub-graphs (if any), to be eligible for outlining.

  A high threshold results in fewer sub-graphs being outlined, a negative value results in all being outlined. The gross value of a sub-graph is the sum of its constituent Ops' Op::getSubgraphValue() values. To disable outlining, it is better to set enableOutlining to false than to set this value to infinity. The default value of 1.0f results in all high value operations such as convolution being cached, but standalone low Value operations such as Relu will not be.

float **outlineSequenceBreakCost** = 10000.0f
: The penalty applied to outlining potential sub-graphs if the sub-graph to be created breaks up a sequence of operations that are more efficient (for example for overlapping compute and exchange) when outlined together.

  Default value is set to ~10 * Op::getHighSubgraphValue().

SubgraphCopyingStrategy **subgraphCopyingStrategy** = *SubgraphCopyingStrategy*::*OnEnterAndExit*
: This setting determines how copies for inputs and outputs for subgraphs are lowered.

  By setting this value to JustInTime you may save memory at the cost of fragmenting subgraphs into multiple Poplar functions. This may be particularly useful when a number of weight updates are outlined in one subgraph, as it may prevent multiple weight tensors from being live at the same time inside the subgraph.

RecomputationType **autoRecomputation** = *RecomputationType*::*None*
: Enable recomputation of operations in the graph in the backwards pass to reduce model size at the cost of computation cycles.

MergeVarUpdateType **mergeVarUpdate** = *MergeVarUpdateType*::*None*
: Enable merging of VarUpdates into groups of VarUpdates, by flattening and concatenating variable tensors and updating tensors.

int64_t **mergeVarUpdateMemThreshold** = 1000000
: The MergeVarUpdateType::AutoLoose and MergeVarUpdateType::AutoTight *VarUpdateOp* merging algorithms have a threshold on the total memory of variable tensors to merge for updating.

  Defined as total memory in bytes.

int64_t **looseThresholdAtPeak** = 8000
: The MergeVarUpdateType::AutoLoose *VarUpdateOp* merging algorithm has an absolute threshold defined by:

  `min`(*mergeVarUpdateMemThreshold*, `liveAtPeak` - `liveCurrently` + *looseThresholdAtPeak*)

  where:

  - `liveAtPeak` is an estimate of the maximum live memory of the computation; and
  - `liveCurrently` is an estimate of the live memory where the threshold is being used to determine whether to schedule or postpone a *VarUpdateOp*.

bool **rearrangeAnchorsOnHost** = true
: Before anchor tensors are streamed from device to host, they are not necessarily arranged in memory as required when they are to be copied from host stream to host.

  This can be done on the device or on the host. Done on host by default to save memory, but often at the expense of cycles, especially for larger anchor tensors.

bool **rearrangeStreamsOnHost** = false
Before stream tensors are streamed from host to device, they are not necessarily arranged in memory as required when they are to be copied from host stream to device.

This can be done on the device or on the host. Done on device by default.

bool **enablePrefetchDatastreams** = true
By default, we will use prefetching for input data streams.

Poplar will speculatively read data for a stream before is is required to allow the 'preparation' of the data to occur in parallel with compute.

unsigned **defaultPrefetchBufferingDepth** = 1
When *enablePrefetchDatastreams* is set this is the default buffering depth value used for input streams that are not re-arranged on the host.

This value can be overridden via *prefetchBufferingDepthMap*.

std::map<*TensorId*, unsigned> **prefetchBufferingDepthMap**
When *enablePrefetchDatastreams* is set this mapping can be used to set tensor-specific buffering depths for tensors that are streamed to the host (typically input tensors).

This buffering depth could be envisaged as being the size of a circular buffer that feeds data to Poplar. A buffering depth greater than 1 may improve the performance due to increased parallelisation but comes at the cost of increasing the memory footprint. Streams for tensors that have no entry in this map default to a buffering depth of 1.

bool **enableNonStableSoftmax** = false
By default, we use the stable softmax Poplar function.

The input tensor to softmax, x, is preprocessed by subtracting max(x) from each element before computing the exponentials, ensuring numerical stability. If you are sure the inputs to your softmax operations are small enough to not cause overflow when computing the exponential, you can enable the non-stable version instead, to increase the speed.

bool **enableReplicatedGraphs** = false
Enable replication of graphs.

bool **enableGradientAccumulation** = false
Enable gradient accumulation.

ReductionType **accumulationAndReplicationReductionType** = ReductionType::Sum
Specify how gradients are reduced when using gradient accumulation and graph replication.

MeanReductionStrategy **meanAccumulationAndReplicationReductionStrategy** = MeanReductionStrategy::PostAndLoss
Specify when to divide by a mean reduction factor when accumulationAndReplicationReductionType is set to ReductionType::Mean.

int64_t **replicatedGraphCount** = 1
If enableReplicatedGraphs is true, `replicatedGraphCount` will set the number of model replications.

For example, if your model uses 1 IPU, a `replicatedGraphCount` of 2 will use 2 IPUs. If your model is pipelined across 4 IPUs, a `replicatedGraphCount` of 4 will use 16 IPUs total. Therefore, the number of IPUs you request must be a multiple of `replicatedGraphCount`. If the training is done across multiple instances then the `replicatedGraphCount` is the number of replicas for this instance.

int64_t **accumulationFactor** = 1
Specify the number of micro-batches to accumulate before applying the varUpdate.

VirtualGraphMode **virtualGraphMode** = *VirtualGraphMode*::*Off*
This option allows you to place ops on virtual graphs to achieve model parallelism - either manually using model annotations, or automatically.

bool **enablePipelining** = false
Enable pipelining of virtual graphs.

SyntheticDataMode **syntheticDataMode** = *SyntheticDataMode*::*Off*
Use synthetic data: disable data transfer to/from the host.

Set to SyntheticDataMode::Off to use real data.

bool **instrumentWithHardwareCycleCounter** = false
Add instrumentation to your program to count the number of device cycles (of a single tile, on a single IPU) that your main program takes to execute.

Expect this to have a small detrimental impact on performance.

std::set<*Instrumentation*> **hardwareInstrumentations** = {*Instrumentation*::*Outer*}

bool **disableGradAccumulationTensorStreams** = false
If true, the weight gradient tensors are not saved off the device when `devicex.weightsFromHost()` is called.

Note: this option is overridden if *syntheticDataMode* is not SyntheticDataMode::Off.

bool **compileEngine** = true
If false, the backend will build the Poplar graph but not compile it into an Engine.

In this case, no execution can be performed, and nothing can be transferred to the device. API calls which retrieve information from the graph building stage, such as tile mapping introspection, can still be used.

bool **constantWeights** = true
An optimization for an inference session to have constant weights, true by default.

Set this option to false if you are going to want to change the weights with a call to *Session::resetHostWeights* after the session has been prepared. This option has no effect on a training session

bool **enableEngineCaching** = false
Enable Poplar executable caching.

std::string **cachePath** = "session_cache"
Folder to save the `poplar::Executable` to.

bool **enableFloatingPointChecks** = false
Throw an exception when floating point errors occur.

bool **enableStochasticRounding** = false
Enable stochastic rounding.

ExecutionPhaseSettings **executionPhaseSettings**
Configuration settings for execution phases.

AccumulateOuterFragmentSettings **accumulateOuterFragmentSettings**
Configuration setting for operations in the accumulate outer fragment.

bool **explicitRecomputation** = false
Enable explicit recomputation.

NumIOTiles **numIOTiles**
Number of IPU tiles dedicated to IO.

bool **aliasZeroCopy** = false
Enable zero-copy for subgraphs.

BatchSerializationSettings **batchSerializationSettings**
Configuration setting for batch serialization.

AutodiffSettings **autodiffSettings**
Configuration settings for the autodiff transform.

bool **delayVarUpdates** = true
Options to delay variable updates as much as possible.

TODO: Remove with T19212

bool **scheduleNonWeightUpdateGradientConsumersEarly** = false
When #shouldDelayVarUpdates is true, the other ops in the proximity of the delayed var updates may inherit the -inf schedule priority used to delay the var updates.

This is undesirable for some ops that consume gradients, as we would like to consume (and thus be able to recycle the memory of) those gradients as soon as possible. Two examples are HistogramOps when doing automatic loss scaling, and the AccumulateOps that accumulate the gradients when doing gradient accumulation.

If true, if #shouldDelayVarUpdates is true, this option will cause the schedule priority of the above described ops to be re-overriden to +inf. TODO: Remove with T19212.

bool **enableFullyConnectedPass** = true
Enable the global #fullyConnectedPass option for matmuls.

bool **enableSerializedMatmuls** = true
Enable/disable the serializing of matmuls.

std::string **partialsTypeMatMuls**
For partialsTypeMatMuls, possible values are defined by `fromString` in op/matmul.cpp.

As of last check, those are: “float”, “half” in any letter case. Set the partials type globally for matmuls. Can be overridden individually with *Builder.setPartialsType()*. Valid values are `"float"` and `"half"`. By default, this is not set, so no global partials type is imposed.

bool **enableStableNorm** = false
If true, computes the mean first and subtracts the activations from it before computing the variance.

The implementation with this flag set to true is slower than when set to false. The stable version requires the first order moment to be estimated and applied to the sample set before the second order central moment is calculated.

std::map<std::string, std::string> **engineOptions**
Poplar engine options.

std::map<std::string, std::string> **convolutionOptions**
Poplar convolution options.

std::map<std::string, std::string> **lstmOptions**
Poplar LSTM options.

std::map<std::string, std::string> **matmulOptions**

std::map<std::string, std::string> **reportOptions**
Poplar reporting options.

std::map<std::string, std::string> **gclOptions**
GCL options.

std::vector<std::string> **customCodelets**
List of codelets (with filetype) to be added to the Poplar graph.

See the Poplar documentation for more information.

std::string **customCodeletCompileFlags**
Compile flags for the custom codelets.

For example `-g` to generate debug info.

double **timeLimitScheduler** = 1e9
The maximum allowed time that can be spent searching for a good graph schedule before a solution must be returned.

int64_t **swapLimitScheduler** = static_cast<int64_t>(1e9)
The maximum number of improving steps allowed by the scheduling algorithm before a solution must be returned.

std::string **serializedPoprithmsShiftGraphsDir** = {}
PopART uses Poprithms for scheduling PopART graphs.

The Poprithms graphs created for scheduling can be optionally serialised (written to file). The string below specified the directory to serialize Poprithms graphs to. If it is empty, then the graphs will not be serialised. The names of serialization files will be `poprithms_shift_graph_i.json` for the lowest non-existing values of `i`. The directory must already exist, PopART will not create it.

std::string **kahnTieBreaker** = "greedy"
The initial scheduling is done with Kahn's algorithm.

When several Ops are free to be scheduled, this controls which method is used.

size_t **transitiveClosureOptimizationThreshold** = {100000}
The transitive closure optimization pass can significantly accelerate the scheduler.

It does not in general affect the final schedule returned. It is run between initialization with Kahn's algorithms and the shifting swaps. The transitive closure optimization pass is O(nOps^2) and so should not be used for extremely large Graphs. If a Graph is above the following threshold, the transitive closure optimization pass is not run.

bool **decomposeGradSum** = false
Replaces single sums of partial gradients with a tree of additions.

This can reduce max liveness at the cost of extra cycles. A typical use case for this would be if a large weight tensor is used as an input to many operations.

bool **enableDistributedReplicatedGraphs** = false
Enable training with Poplar replicated graphs across multiple PopART instances.

int64_t **globalReplicationFactor** = 1
The total number of replicas in a multi instance replicated graph training session (this should be left as the default value (1) if distributed replicated graphs are disabled).

This value includes local replication.

int64_t **globalReplicaOffset** = 0
The first replica index that this PopART instance is running.

bool **groupHostSync** = false
Allows to group the streams from host at the beginning and the streams to host at the end, this trades off sum-liveness efficiency for cycle efficiency.

bool **strictOpVersions** = true
Strict op version checks will throw an error if the exact version of an op required for the models opset is not supported.

Turning this check off will cause PopART to fall back to the latest implementation of the op that is supported. Warning, turning off these checks may cause undefined behaviour.

bool **opxAliasChecking** = false
Run Opx checks to verify IR tensor aliasing information corresponds to lowered Poplar tensor aliasing.

bool **opxModifyChecking** = false
Run Opx checks to verify IR tensor modification information corresponds to lowered Poplar tensor modifications.

bool **useHostCopyOps** = false
Uses IR graph operations for data and anchor streams.

bool **enableLoadAndOffloadRNGState** = false
Allows to load/offload device RNG state from host.

TensorLocationSettings **activationTensorLocationSettings** = *TensorLocationSettings*{*TensorLocation*(), 2, 8192}
Tensor location settings for activation/gradient tensors.

TensorLocationSettings **weightTensorLocationSettings** = *TensorLocationSettings*{*TensorLocation*(), 2, 8192}
Tensor location for weight tensors.

TensorLocationSettings **optimizerStateTensorLocationSettings** = *TensorLocationSettings*{*TensorLocation*(), 2, 8192}
Tensor location for optimizer state tensors.

TensorLocationSettings **accumulatorTensorLocationSettings** = *TensorLocationSettings*{*TensorLocation*(), 2, 8192}
Tensor location for gradient accumulator tensors.

std::map<*TensorId*, *TensorLocation*> **tensorLocationSettingsOverride**
Override tensor location for specific tensors by setting a *TensorLocation* for specific TensorId values.

AutomaticLossScalingSettings **automaticLossScalingSettings**
Settings to enable and configure the automatic loss scaling behaviour when training.

Note: Automatic loss scaling is currently experimental and under active development. We recommend that the user sets the loss scale manually.

DeveloperSettings **developerSettings**

bool **enableSupportedDataTypeCasting** = true
If enabled, casts any tensor of unsupported data types to supported data types when lowering to Poplar Currently, this implies casting: INT64 -> INT32 UINT64 -> UINT32 The cast will error for incompatible data types and over/underflows, and inform on narrowing casts.

bool **enableExplicitMainLoops** = false
Enables explicit main loop transformation, and disables implicit training loops.

This will become deprecated and enabled by default.

bool **groupNormStridedChannelGrouping** = false
Group norms have a fast math mode /which changes the implementation to run faster on IPU but as a consequence/ is incompatable with other implementations (i.e running trained weights on host).

We default to correct and slightly slower but a user can opt into fast but incorrect.

std::function<void(int, int)> **compilationProgressLogger**
Callback function used to to indicate PopART compilation progress.

The function is passed two integers. The first is the progress value and the second is the maximum value for the progress.

The function should not block. All calls to the callback function will be made from the main thread so blocking in the callback will block compilation from progressing.

If this logger is not set then compilation progress will be printed on the info channel.

int **compilationProgressTotal** = 100
Total progress ticks until compilation complete.

bool **enableMergeExchange** = true
Enables merging remote and host IO operations to facilitate IO overlap.

class NumIOTiles
A wrapper class for the *numIOTiles* option that permits any int value and has an 'unassigned' state.

Public Functions

NumIOTiles()
Constructor.

NumIOTiles(int *numIOTiles*)
Constructor.

bool **operator==**(**const** int &*rhs*) **const**
Compare with int.

operator int() **const**
Auto convert to int.

NumIOTiles &**operator=**(**const** int &*x*)
Assign value using int.

enum popart::**AccumulateOuterFragmentSchedule**
Enum type that determines how the operations in the accumulate outer fragment will be scheduled accross virtual graphs (only relevant to pipelined modes).

Values:

enumerator Scheduler = 0
Don't add additional constraints and let the scheduler work it out.

enumerator Serial
Add constraints that ensure ops are executed in virtual graph ID order.

enumerator OverlapCycleOptimized
Try and parallelise ops with different virtual graph IDs as much as possible.

enumerator OverlapMemoryOptimized
Try and parallelise ops with different virtual graph IDs but avoid certain steps that are costly in terms of memory usage.

struct popart::**AccumulateOuterFragmentSettings**
A structure containing accumulate outer fragment settings.

Public Functions

AccumulateOuterFragmentSettings() = default

AccumulateOuterFragmentSettings(*AccumulateOuterFragmentSchedule* *schedule_*, **const** std::vector<int> &*excludedVirtualGraphs_*)

AccumulateOuterFragmentSettings &**operator=**(**const** *AccumulateOuterFragmentSettings* &*rhs*) = default

Public Members

AccumulateOuterFragmentSchedule **schedule** = *AccumulateOuterFragmentSchedule*::*Serial*
Tell PopART how you would like to schedule the accumulate outer fragment.

This setting is experimental and may change.

std::vector<int> **excludedVirtualGraphs** = {}
A setting to explicitly tell PopART to avoid to try and parallelise the given virtual graph ids.

This setting is experimental and may change.

struct popart::**AutomaticLossScalingSettings**
A structure containing user configuration for automatic loss scaling settings.

Note: Automatic loss scaling is currently experimental and under active development. We recommend that the user sets the loss scale manually.

Public Functions

AutomaticLossScalingSettings() = default

AutomaticLossScalingSettings(bool *enabled_*, **const** nonstd::optional<std::vector<*TensorId*>> &*toTrackTensors_*, float *binEdgeLocation_* = 1.0, float *thresholdUpperCountProportion_* = 1e-7)

AutomaticLossScalingSettings &**operator=**(**const** *AutomaticLossScalingSettings* &*rhs*) = default

std::size_t **hash**() **const**

Public Members

bool **enabled** = false
If true, keep track of the distribution of gradient tensor elements over the floating point range.

Adjust the value loss scaling tensor accordingly, with the aim of preventing underflow or overflow.

float **binEdgeLocation** = 1.0
The location of the bin edge as a proportion of the absolute numerical range of the tracked gradient tensor elements, in the range [0, 1].

`0` represents the smallest representable value, and `1` the maximum. This is the single bin edge of the histogram that is an input to the loss scale updater algorithm.

float **thresholdUpperCountProportion** = 1e-7
The proportion of the elements in the upper bin above which the loss scale is increased, and below which the loss scale is decreased.

Should be in the range [0, 1].

nonstd::optional<std::vector<*TensorId*>> **toTrackTensors**
An optional list of model tensor names, for which gradient statistics will be collected.

If unset, the gradients of all tensors produced by a default operations (matmul, conv) will be used.

enum popart::**BatchSerializationBatchSchedule**
Enum type that describes how to change the batch serialisation subgraph schedule before outlining.

NOTE: This setting is experimental and may change.

Values:

enumerator Scheduler = 0
Don't encourage any particular scheduling for ops within batch subgraphs (leave it to the scheduler) but tell the scheduler to schedule subgraphs in sequence.

enumerator Isomorphic
Encourage all ops within batch subgraphs to be scheduled identically and for each subgraph to be scheduled in sequence (good for outlineability).

enumerator OverlapOnIo
Attempt to put the RemoteLoad for batch N+1 right after the compute phase of batch N.

enumerator OverlapOnCompute
Attempt to put the RemoteLoad for batch N+1 right before the compute phase of batch N.

enumerator N
The number of `BatchSerializationBatchSchedule` values.

enum popart::**BatchSerializationMethod**
Enum type that describes how to apply the batch serialization.

NOTE: This setting is experimental and may change.

Values:

enumerator UnrollDynamic = 0
Unroll the batch with dynamic slicing.

enumerator UnrollStatic
Unroll the batch with static slicing.

enumerator Loop
Loop over the batch dimension.

enumerator N
The number of `BatchSerializationMethod` values.

struct popart::**BatchSerializationSettings**
A structure containing batch serialization settings.

Public Functions

BatchSerializationSettings() = default

BatchSerializationSettings(int *factor_*, bool *concatOnVirtualGraphChange_*, bool *concatOnExecutionPhaseChange_*, bool *concatOnPipelineStageChange_*, *BatchSerializationTransformContext* *transformContext_* = *BatchSerializationTransformContext*::*Fwd*, *BatchSerializationMethod* *method_* = *BatchSerializationMethod*::*UnrollDynamic*, *BatchSerializationBatchSchedule* *batchSchedule_* = *BatchSerializationBatchSchedule*::*Isomorphic*)

BatchSerializationSettings &**operator=**(**const** *BatchSerializationSettings* &*rhs*) = default

Public Members

int **factor** = 0
The number of compute batches to split operations into.

bool **concatOnVirtualGraphChange** = true
Break batch serialization chains when the virtual graph changes (by concatenating the compute batches to the local batch).

bool **concatOnExecutionPhaseChange** = true
Break batch serialization chains when the execution phase changes (by concatenating the compute batches to the local batch).

bool **concatOnPipelineStageChange** = true
Break batch serialization chains when the pipeline stage changes (by concatenating the compute batches to the local batch).

BatchSerializationTransformContext **transformContext** = *BatchSerializationTransformContext*::*Fwd*
Experimental value to control when batch serialization is applied.

BatchSerializationMethod **method** = *BatchSerializationMethod*::*UnrollDynamic*
Experimental value to control how batch serialization is applied.

BatchSerializationBatchSchedule **batchSchedule** = *BatchSerializationBatchSchedule*::*Isomorphic*
Experimental value that changes how operations are scheduled.

enum popart::**BatchSerializationTransformContext**
Enum type that describes when to apply the batch serialization.

NOTE: This setting is experimental and may change.

Values:

enumerator Fwd = 0
Apply before growing the backward pass.

enumerator Bwd
Apply after growing the backward pass.

enumerator N
The number of `BatchSerializationTransformContext` values.

enum popart::**DotCheck**
Enum type used to identify at which stages of IR construction to export `.dot` files.

Values:

enumerator Fwd0 = 0
Generate graph after construction of the forward pass.

enumerator Fwd1
Generate graph after running pre-aliasing patterns.

enumerator Bwd0
Generate graph after backwards construction.

enumerator PreAlias
Generate graph after all transformations, patterns, except the aliasing.

enumerator Final
Generate graph after running aliasing patterns (the final IR).

enumerator All
Generate all graphs.

enumerator N
The number of `DotCheck` values.

enum popart::**ExecutionPhaseIOSchedule**
Enum type to specify when to load tensors.

Values:

enumerator Preload = 0
Preload tensors in previous phase for use in current phase.

enumerator OnDemand
Load tensors just before they are required.

enumerator N
The number of `ExecutionPhaseIOSchedule` values.

struct popart::**ExecutionPhaseSettings**
A structure containing ExecutionPhase settings.

Public Functions

ExecutionPhaseSettings() = default

ExecutionPhaseSettings(int *phases_*, bool *stages_*, *ExecutionPhaseIOSchedule* *weightIOSchedule_*, *ExecutionPhaseIOSchedule* *activationIOSchedule_*, *ExecutionPhaseIOSchedule* *optimizerStateIOSchedule_*, *ExecutionPhaseIOSchedule* *accumulatorIOSchedule_*, *ExecutionPhaseSchedule* *schedule_*)

ExecutionPhaseSettings &**operator=**(**const** *ExecutionPhaseSettings* &*rhs*) = default

Public Members

int **phases** = 0
Number of ExecutionPhases for the whole model.

int **stages** = 2
Number of overlapping stages 1: Parallel streaming memory, default for 1 IPU / replica 2: PingPong between 2 IPUs, default for >= 2 IPUs / replica.

ExecutionPhaseIOSchedule **weightIOSchedule** = *ExecutionPhaseIOSchedule*::*Preload*
The execution phase IO schedule for weight tensors.

ExecutionPhaseIOSchedule **activationIOSchedule** = *ExecutionPhaseIOSchedule*::*Preload*
The execution phase IO schedule for activation and gradient tensors.

ExecutionPhaseIOSchedule **optimizerStateIOSchedule** = *ExecutionPhaseIOSchedule*::*OnDemand*

ExecutionPhaseIOSchedule **accumulatorIOSchedule** = *ExecutionPhaseIOSchedule*::*Preload*

ExecutionPhaseSchedule **schedule** = *ExecutionPhaseSchedule*::*Interleaving*

enum popart::**ExecutionPhaseSchedule**
Enum type to specify the order of processing optimizer operations for different weights of the same execution phase.

The steps for phased execution consists of:

- Copy to IO tiles if necessary (1)
- Run collective operations if necessary (2)
- Load optimizer state (3)
- Update optimizer state (4)
- Apply optimizer (5)
- Store updated tensor if necessary (6)

Values:

enumerator Interleaving = 0
Process above steps for one weight at a time (for example: 123456, 123456, 123456).

The scheduler may interleave these steps.

enumerator Batch
Process above steps for all weights together, in a way that maximises overlap potential between compute and exchange (for example: 333, 111, 222, 444, 555, 666).

enumerator BatchClusteredIO
Process above steps for all weights together, in a way that maximises overlap potential between compute and exchange, and maximise stream copy merges by keeping RemoteLoad/RemoteStore operations clustered (for example: 333, 111, 222, 444, 555, 666).

enumerator N
The number of `ExecutionPhaseSchedule` values.

enum popart::**Instrumentation**
Enum type used to specify an instrumentation type.

Values:

enumerator Outer = 0
Outer loop instrumentation, graph over all IPUs.

enumerator Inner
Inner loop instrumentation, graph per IPU.

enumerator N
The number of `Instrumentation` values.

enum popart::**IrSerializationFormat**
Enum type used to specify a serialization format.

Values:

enumerator JSON
JavaScript Object Notation (JSON).

enum popart::**MergeVarUpdateType**
Enum type used to specify which `VarUpdateOp` ops to merge.

Values:

enumerator None = 0
Do not merge *VarUpdateOp* ops.

enumerator All
Merge all *VarUpdateOp* ops into as few groups as possible.

This is a good choice when memory is not a constraint.

enumerator AutoLoose
Merge into groups while attempting not to increase maximum variable liveness, and also not slice tensor variables so they they will need to be processed by different *VarUpdateOp* ops.

enumerator AutoTight
Merge into groups, so that *VarUpdateOp* ops process tensors of exactly `mergeVarUpdateMemThreshold` in size.

enumerator N
The number of `MergeVarUpdateTypes` values.

enum popart::**RecomputationType**
Enum type to specify which ops to recompute in the backwards pass when doing auto-recomputation.

Values:

enumerator None = 0
No ops are recomputed.

enumerator Standard
Algorithm to pick checkpoints to try and minimise max liveness.

enumerator NormOnly
Only Norm ops (+ non-linearities, if following) are recomputed.

enumerator Pipeline
Recompute all forward pipeline stages.

enumerator RecomputeAll
Recompute all ops.

enumerator N
The number of `RecomputationTypes` values.

enum popart::**SubgraphCopyingStrategy**
Enum type that describes how copies for inputs and outputs for subgraphs are lowered.

Currently this only affects subgraphs associated with CallOps.

Values:

enumerator OnEnterAndExit = 0
Copy all inputs before the start of the subgraph, copy all outputs after all ops in the subgraph.

With this strategy subgraphs will always map to a single Poplar function.

enumerator JustInTime
Copy inputs just before they are consumed and copy outputs as soon as they are produced.

With this strategy subgraphs may be lowered into multiple Poplar functions.

enumerator N
The number of `SubgraphCopyingStrategy` values.

enum popart::**SyntheticDataMode**
Enum type used to specify the data source for input tensors.

Values:

enumerator Off = 0
Use real data.

enumerator Zeros
Input tensors are initialised to all zeros.

enumerator RandomNormal
Input tensors are initialised with distribution ~N(0,1).

enumerator N
The number of `SyntheticDataMode` values.

struct popart::**TensorLocationSettings**
A structure containing user configuration for cache/offloading settings.

Public Functions

TensorLocationSettings() = default

TensorLocationSettings(*TensorLocation* *location_*, int *minElementsForOffChip_* = 2, int *minElementsForReplicatedTensorSharding_* = 8192)

TensorLocationSettings(*TensorStorage* *storage_*, int *minElementsForOffChip_* = 2, int *minElementsForReplicatedTensorSharding_* = 8192)

TensorLocationSettings &**operator=**(**const** *TensorLocationSettings* &*rhs*) = default

Public Members

TensorLocation **location** = *TensorLocation*()
The default tensor location for this tensor type.

int **minElementsForOffChip** = 2
A minimum number of elements below which offloading won't be considered.

int **minElementsForReplicatedTensorSharding** = 8192
A minimum number of elements below which replicated tensor sharding (RTS) won't be considered.

enum popart::**VirtualGraphMode**
Enum type used to specify a virtual graph mode.

Values:

enumerator Off = 0
Virtual graphs are not enabled.

enumerator Manual
User must set the `virtualGraph` attribute on all ops.

enumerator Auto
Use `autoVirtualGraph` transform.

enumerator ExecutionPhases
Virtual graphs are tied to execution phases.

enumerator N
The number of `VirtualGraphModes` values.

2.2 Optimizers

```
#include <popart/optimizer.hpp>
```

class popart::**Optimizer**
Interface for describing an *Optimizer* and, internally, how to grow the optimiser step for each weight.

- The end-user facing interface constructed by the user to describe what kind of optimiser to use.
- Then also used internally by the Ir to grow the optimiser step for each weight.
- Stores OptimizerValues for optimizer parameters like learning rate, loss scaling, etc.

  See OptimiserValue.
- *Optimizer* stores the values for each weight - they can have different values. There is a "default" for all weights, then you can specify specific values for specific weights. This is encapsulated by an OptimizerValueMap, which is a sparse map from weight to value, with unspecified values implying the default.

  See OptimizerValueMap.
- At runtime, the user can dynamically update the *Optimizer*, e.g. by setting new OptimizerValues. validReplacement determines whether the new *Optimizer* is interchangable with the one the Ir was built for. For example, trying to replace an *SGD Optimizer* with an *Adam Optimizer* would throw.

Subclassed by *popart::Adam*, *popart::Adaptive*, *popart::SGD*

Public Functions

~Optimizer() = default

- *Optimizer* class has a two-part initialisation. The ctor, used by the end-user, and setFactorsFromOptions called by the Ir to finish initialisation once we have all the relevant information during Ir preparation.
- Some key methods used by the Ir to grow optimiser step for each weight are createOp, getInputIds, optimizerInputs.
- If the *OptimizerValue* is const, no Ir tensor for that value is created and the *VarUpdateOp* created for that weight will not have the optional input for that tensor. The Opx of the *VarUpdateOp* will emit poplar code that uses the provided value directly.

  If the *OptimizerValue* is not const, an Ir tensor for that value is created and the *VarUpdateOp* created for that weight will have the optional input for that tensor. The tensor will be a stream tensor, so that it can be updated later from host. The tensor will be streamed an initial value of the *OptimizerValue*'s value.
- It is common for *Optimizer* implementations to make use of "compound

  scalars". Take for example the SGD0 weight update equation: w <- w * (1 - lr * (1 - dm) * wd) - g * (lr * (1 - dm) / ls) w is the weights and g is the grads. lr, dm, wd, ls are all the "atomic scalars". These are the scalars/hyperparameters of the *Optimizer* that the user can set using OptimizerValues, as described above.

  Multiple atomic scalars appear in expressions together, and will be operated on together before being used by an Op that also consumes a tensor (in this case the weights or grads). For SGD0, they can be grouped as follows:

```
w <- w * {1 -  lr * (1 - dm) * wd} -  g * { lr * (1 - dm) / ls }
         ^^^^^^^^^^^^^^^^^^^^^^^^^         ~~~~~~~~~~~~~~~~~~~~~~
                    |                                 |
   weight decay scale factor 0                        |
                                              scaled learning rate 0
```

We call wdsf0 and slr0 the “compound scalars”.

We can statically precompute the OptimizerValues for these compound scalars using the OptimizerValues of the atomic scalars. This makes the Ir simpler, as we now have only:

```
w <- w * wdsf0 - g * slr0
```

The CompoundScalarHelpers are used to precompute the compound scalar values. If any of the composite atomic scalars are non-const, the compound scalar is non-const.

See compoundscalarhelper.hpp

Optimizer(*OptimizerValue lossScaling*, **const** std::vector<*ClipNormSettings*> &*clipNormSettings*)

Optimizer(**const** *Optimizer*&) = default

void **validReplacement**(**const** *Optimizer* &*other*) **const**

OptimizerType **type**() **const** = 0

std::string **type_s**() **const** = 0

std::unique_ptr<*Optimizer*> **clone**() **const** = 0

void **resetTensorData**(Tensor&) **const** = 0

void **setTensorData**(Tensor&) **const** = 0

std::unique_ptr<*Op*> **createOp**(**const** Tensor &*weight*, Graph&) **const** = 0

std::vector<*TensorId*> **getInputIds**(**const** Tensor &*weight*) **const** = 0

Returns the TensorIds of the input tensors to the *VarUpdateOp* this optimiser will create for the given `weight` .

Specifically, The TensorId at index i will be the id of the input tensor at InIndex i of the *VarUpdateOp*. If the input is an *OptimizerValue*, if it is const, then “” will be returned, else the relevant reservered prefix for that *OptimizerValue* will be used, followed by the weight id. The prefixes are defined in tensornames.hpp, for example `reservedDefaultWeightDecayScaleFactor0Prefix` or `reservedSpecificScaledLearningRate1Prefix` (note there are different prefixes depending on if the weight has a specific or default value for that *OptimizerValue*).

std::vector<std::tuple<*TensorId*, *TensorInfo*>> **getOptimizerInputs**(**const** Tensor &*weight*) **const** = 0

const *OptimizerValue* &**lossScaling**() **const**

float **getLossScalingVal**() **const**

float **getFinalLossScalingVal**() **const**

TensorId **getInverseLossScalingTensorId**(**const** Tensor &*weight*) **const** = 0

void **setFactorsFromOptions**(**const** *SessionOptions*&)

bool **gradientAccumulationEnabled**() **const**

bool **meanReductionEnabled**() **const**

bool **postMeanAccumulationEnabled**() **const**

bool **postMeanReplicationEnabled**() **const**

bool **lossMeanReplicationEnabled**() **const**

int64_t **getReplicatedGraphCount**() **const**

int64_t **getAccumulationFactor**() **const**

bool **meanGradientAccumulationEnabled**() **const**

const std::vector<*ClipNormSettings*> &**getClipNormSettings**() **const**

bool **hasSpecific**(**const** Tensor &*w*) **const** = 0

bool **hasSpecific**() **const** = 0

size_t **hash**() **const**

Public Static Functions

TensorId **getLossScalingTensorId**(*DataType*)

enum popart::**OptimizerType**
Types of optimizers.

Values:

enumerator SGD = 0

enumerator Adam

enumerator Adaptive

enumerator NTYPES

enum popart::**OptimizerReductionType**
Reduction mode when doing data-parallel training over replicated graphs.

Depending on the optimizer used and its configuration, this option describes how the reduction of gradients over replicas will occur. For example, directly on the gradient, on the gradient accumulator, or on the momentum. See the documentation of individual optimizers for more information.

Values:

enumerator None = 0
No replicated graph reduction.

enumerator GradReduce
Gradient reduction (every iteration, after a weight's gradient is produced)

enumerator AcclReduce
Momentum reduction (SGD1, after the gradient accumulation loop, if applicable)

enumerator AccumReduce
Accumulator reduction (Adam/SGD2 + gradient accumulation, after the gradient accumulation loop)

enum popart::**WeightDecayMode**
Values:

enumerator Decay
Weight decay (e.g. AdamW)

enumerator L2Regularization
L2 regularization (e.g. PyTorch-like *Adam*)

```
#include <popart/optimizervalue.hpp>
```

class popart::**OptimizerValue**
A class used to represent values of hyper parameters.

Public Functions

OptimizerValue() = default
Equivalent to OptimizerValue(0, false).

OptimizerValue(float *v*)
Equivalent to OptimizerValue(v, true).

OptimizerValue(float *v*, bool *c*)
Constructor.

Parameters

- v: The current value of the hyper parameter.
- c: A boolean flag to indicate whether the parameter will remain at this value forever (`true`) or may change over time (`false`).

OptimizerValue(std::pair<float, bool> *x*)

float **val**() **const**

bool **isConst**() **const**

void **validReplacement**(**const** *OptimizerValue* &*rhs*) **const**

bool **operator==**(**const** *OptimizerValue* &*rhs*) **const**

2.2.1 Stochastic Gradient Descent (SGD)

```
#include <popart/optimizer.hpp>
```

class popart::**ClipNormSettings**
A data structure used to represent a maximum value constraint on one or more weights.

This is passed to the optimizer on construction.

Public Types

enum Mode
Values:

enumerator ClipSpecifiedWeights

enumerator ClipAllWeights

Public Functions

ClipNormSettings(**const** std::vector<*TensorId*> &*weightIds_*, float *maxNorm_*)
DEPRECATED This will be removed from a future release.

Constructor.

Parameters

- `weightIds_`: The weight tensor IDs that this constraint applies to.
- `maxNorm_`: The maximum permissible value.

const std::vector<*TensorId*> &**getWeightIds**() **const**

float **getMaxNorm**() **const**

Mode **getMode**() **const**

bool **operator==**(**const** *ClipNormSettings*&) **const**

bool **operator!=**(**const** *ClipNormSettings* &*other*) **const**

Public Members

std::vector<*TensorId*> **weightIds**

float **maxNorm**

Public Static Functions

ClipNormSettings **clipWeights**(**const** std::vector<*TensorId*> &*weightIds_*, float *maxNorm_*)

ClipNormSettings **clipAllWeights**(float *maxNorm_*)

class popart::**SGD** : **public** *popart*::*Optimizer*

Stochastic Gradient Descent (SGD) optimizer.

Akin to any optimizer implementation, this class is responsible for updating each weight tensor (w) in the model using the gradient (g) of the loss function with respect to the weight as calculated during the backwards pass.

The SGD optimizer has the following **state** for each weight:

- *velocity* (v)

The SGD optimizer has the following **hyper parameters**:

- *learning rate* (lr)
- *momentum* (mm)
- *weight decay* (wd)
- *dampening* (dm)
- *velocity scaling* (vs)
- *loss scaling* (ls)
- *clip norm settings*

The values of these parameters can be shared between all weights but some can be overridden with weight-specific values (see *SGD::insertSpecific*). Hyper parameters are captured using *OptimizerValue* objects and therefore can be either a constant value or a non-constant value that can be adjusted by the user.

In the following we will describe how this optimizer updates a weight using a gradient. In the context of this description the gradient is is the value of the gradient *after* any gradient accumulation has been performed and *after* the application of a loss scaling factor to the gradient has been corrected for.

When the optimizer needs to update a weight, w, using a gradient, g, it first updates the optimizer state as follows:

$$v' := v * \mathsf{mm} + (1 - \mathsf{dm}) * (g + \mathsf{wd} * w) \ .$$

Following the update of the optimizer state the optimizer uses said state to update the weight:

$$w' := w - \mathsf{lr} * v' \ .$$

In addition to the above, the *velocity scaling* hyper parameter is a scaling factor that can provide improved numerical stability by ensuring the values stored in the optimizer state, v, are scaled by this value. When using this parameter PopART will automatically deal with the artificially scaled velocity value during the weight update and other hyper parameters do not need to be adjusted).

In addition, the *loss scaling* hyper parameter is similar in nature to the velocity scaling parameter. It is a scaling value that is applied to the loss gradient at the start of the the backwards pass and, at the end of the backwards pass, this scaling is reversed by multiplying the gradients for each weight with the inverse of the loss scaling value prior to updating the optimizer state. Using loss scaling can also improve numerical stability in some cases.

Finally, it is possible to add clip norm settings for this optimizer. These clip norms compute the L2 norm for a group of weights and adds a scalar term to the weight update that effectively divides it by the norm (or a constant value that is provided as part of the clip norm, which ever is greater).

See the *SGD* notes in optimizer.hpp for a more detailed and comprehensive derivation of the *SGD* optimizer step in PopART.

Subclassed by *popart::ConstSGD*

Public Functions

SGD(*OptimizerValue defaultLearningRate*, *OptimizerValue defaultWeightDecay*, *OptimizerValue defaultMomentum*, *OptimizerValue defaultDampening*, *OptimizerValue defaultVelocityScaling*, *OptimizerValue lossScaling*, **const** std::vector<*ClipNormSettings*> &*clipNormSettings* = {}, SGDAccumulatorAndMomentum *sgdAccMm* = SGDAccumulatorAndMomentum::Combined, *DataType accumType* = *DataType*::*UNDEFINED*, *DataType accl1Type* = *DataType*::*UNDEFINED*)

Constructor.

See SGDAccumulatorAndMomentum. Defaults to SGDAccumulatorAndMomentum::Combined.

Parameters

- `defaultLearningRate`: The learning rate value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultWeightDecay`: The weight decay value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultMomentum`: The momentum value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultDampening`: The dampening value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultVelocityScaling`: The velocity scaling value to use for weights for which no weight-specific hyper parameter have been inserted.
- `lossScaling`: The loss scaling value to use.
- `clipNormSettings`: A vector of *ClipNormSettings* (this can be used to set maximum values for weights).
- `sgdAccMm`: The implementation strategy to use when gradient accumulation and/or momentum are used, otherwise ignored.

Parameters

- `accumType`: The DataType of the accum tensor, when gradient accumulation is used and sgdAccMm = SGDAccumulatorAndMomentum::Separate, otherwise ignored. Only FLOAT, FLOAT16 and UNDEFINED are supported. Defaults to UNDEFINED. If UNDEFINED, the same type as the weights will be used. If accumType is FLOAT16 and accl1Type is FLOAT, this parameter causes accum to be upcasted before being passed to the op that updates accl1.

- `accl1Type`: The DataType of the accl1 tensor, when gradient accumulation is used and sgdAccMm = SGDAccumulatorAndMomentum::Separate, otherwise ignored. Only FLOAT, FLOAT16 and UNDEFINED are supported. Defaults to UNDEFINED. If UNDEFINED, the same type as the weights will be used. If accumType is FLOAT16 and accl1Type is FLOAT, this parameter causes accum to be upcasted before being passed to the op that updates accl1.

SGD(**const** std::map<std::string, std::pair<float, bool>> &*params*, **const** std::vector<*ClipNormSettings*> &*clipNormSettings* = {}, SGDAccumulatorAndMomentum *sgdAccMm* = SGDAccumulatorAndMomentum::Combined, *DataType* *accumType* = *DataType*::*UNDEFINED*, *DataType* *accl1Type* = *DataType*::*UNDEFINED*)

Constructor.

EXAMPLE:

```
SGD({{"defaultLearningRate", {0.02, False}},
    {"defaultMomentum":{0.6, True}}});
```

This will create an *SGD Optimizer* which has a constant momentum of 0.6 and a changeable learning rate initially of 0.02. All OptimizerValues not present in the map will take values from the `getUnset*` functions.

See SGDAccumulatorAndMomentum. Defaults to SGDAccumulatorAndMomentum::Combined.

Parameters

- `params`: A parameter map where the keys are one or more of `"defaultLearningRate"`, `"defaultWeightDecay"`, `"defaultMomentum"`, `"defaultDampening"`, `"defaultVelocityScaling"` or `"lossScaling"`. The map's values are pairs of floats and booleans representing *OptimizerValue* constructor arguments. The map does not have to specify each hyper parameter because default values will be used where parameters are missing.
- `clipNormSettings`: A vector of *ClipNormSettings* (this can be used to set maximum values for weights).
- `sgdAccMm`: The implementation strategy to use when gradient accumulation and/or momentum are used, otherwise ignored.

Parameters

- `accumType`: The DataType of the accum tensor, when gradient accumulation is used and sgdAccMm = SGDAccumulatorAndMomentum::Separate, otherwise ignored. Only FLOAT, FLOAT16 and UNDEFINED are supported. Defaults to UNDEFINED. If UNDEFINED, the same type as the weights will be used. If accumType is FLOAT16 and accl1Type is FLOAT, this parameter causes accum to be upcasted before being passed to the op that updates accl1.
- `accl1Type`: The DataType of the accl1 tensor, when gradient accumulation is used and sgdAccMm = SGDAccumulatorAndMomentum::Separate, otherwise ignored. Only FLOAT, FLOAT16 and UNDEFINED are supported. Defaults to UNDEFINED. If UNDEFINED, the same type as the weights will be used. If accumType is FLOAT16 and accl1Type is FLOAT, this parameter causes accum to be upcasted before being passed to the op that updates accl1.

SGD()

Default constructor Creates *SGD* with default scalars (equivalent to getUnset<scalar>() methods), and other default parameters of main constructor.

SGD(**const** *SGD*&) = default

Construct an *SGD* instance with default values.

~SGD() = default

OptimizerType **type**() **const final**

std::string **type_s**() **const final**

SGDAccumulatorAndMomentum **getSGDAccumulatorAndMomentum**() **const**

std::unique_ptr<*Optimizer*> **clone**() **const final**

std::unique_ptr<*Op*> **createOp**(**const** Tensor &*weight*, Graph&) **const final**
Returns the *VarUpdateOp* for the given `weight` .

If no gradient accumulation of momentum, this will be a SGD0VarUpdateOp. Else, if `getSGDAccumulatorAndMomentum() == Combined`, this will be an SGD1ComboOp, else if `getSGDAccumulatorAndMomentum() == Combined`SGD2ComboOp, an SGD2ComboOp. The required compound scalar OptimizerValues for the *VarUpdateOp* wil be computed and passed to the Op. See the *SGD* notes above this class for how they are derived. Recall that if non-const, the *VarUpdateOp* will take an input Tensor for the compound scalar.

See Optimizer::createOp

The OptimizerReductionType of the Op is derived as follows: No replication => None Replication, no grad acc => GradReduce Replication, grad acc, SGD1 => AcclReduce Replication, grad acc, SGD2 => AccumReduce See the *SGD* notes above this class for why this is.

If SGD2, the DataType of the accum and accl1 tensors passed to the SGD2ComboOp will be as set in the *SGD* constructor. Recall DataType::UNDEFINED means use the same as the weight.

An SGD1ComboOp will later be decomposed by SGD1Decompose pattern into a series of Ops and Tensors that implement the SGD1 optimiser step. An SGD12ComboOp will later be decomposed by SGD2Decompose pattern into a series of Ops and Tensors that implement the SGD2 optimiser step.

See SGD1Decompose

See SGD2Decompose

std::vector<*TensorId*> **getInputIds**(**const** Tensor &*weight*) **const final**

See *Optimizer::getInputIds*

std::vector<std::tuple<*TensorId*, *TensorInfo*>> **getOptimizerInputs**(**const** Tensor &*weight*) **const final**
smm1 and wdsf0 have the same data type as the `weight` . Everything else

void **validReplacement**(**const** *Optimizer* &*other*) **const final**

void **resetTensorData**(Tensor&) **const final**

void **setTensorData**(Tensor&) **const final**

float **getStoredValue**(**const** *TensorId* &*optId*) **const**
Tensor “opt” has an id, which it uses to match a compound scalar which this object can compute from the atomic scalars.

void **insertSpecific**(**const** *TensorId* &*weight*, *OptimizerValue* *learningRate*, *OptimizerValue* *weightDecay*, *OptimizerValue* *momentum*, *OptimizerValue* *dampening*, *OptimizerValue* *velocityScaling*)
Insert a weight-specific set of hyper parameters.

Parameters

- `weight`: The TensorId of the weight.
- `learningRate`: The learning rate value to use for this specific weight.
- `weightDecay`: The weight decay value to use for this specific weight.
- `momentum`: The momentum value to use for this specific weight.
- `dampening`: The dampening value to use for this specific weight.
- `velocityScaling`: The velocity scaling value to use for this specific weight.

void **insertSpecific**(**const** *TensorId* &*weight*, **const** std::map<std::string, std::pair<float, bool>> &*params*)
Insert a weight-specific set of hyper parameters.

Parameters

- `weight`: The TensorId of the weight.
- `params`: A parameter map where keys are one of `"learningRate"`, `"weightDecay"`, `"momentum"`, `"dampening"`, or `"velocityScaling"` and the map's values pairs of floats and booleans representing *OptimizerValue* constructor arguments. The map does not have to specify each hyper parameter as default values will be used where parameters are missing.

bool **hasSpecific**(**const** Tensor &*w*) **const final**

bool **hasSpecific**() **const final**

TensorId **getInverseLossScalingTensorId**(**const** Tensor &*weight*) **const**

const OptimizerValueMap &**learningRates**() **const**

const OptimizerValueMap &**weightDecays**() **const**

const OptimizerValueMap &**momentums**() **const**

const OptimizerValueMap &**dampenings**() **const**

const OptimizerValueMap &**velocityScalings**() **const**

size_t **hash**() **const**

Public Static Functions

OptimizerValue **getUnsetLearningRate**()
Default learning rate value.

OptimizerValue **getUnsetWeightDecay**()
Default weight decay value.

OptimizerValue **getUnsetMomentum**()
Default momentum value.

OptimizerValue **getUnsetDampening**()
Default dampening value.

OptimizerValue **getUnsetVelocityScaling**()
Default velocity scaling value.

OptimizerValue **getUnsetLossScaling**()
Default loss scaling value.

SGD **fromDefaultMap**(**const** std::map<std::string, *OptimizerValue*>&)

class popart::**ConstSGD** : **public** *popart*::*SGD*
Stochastic Gradient Descent (*SGD*) optimizer with constant learning rate, weight decay, loss scaling and clip norm settings (and default values for momentum, dampening or velocity scaling).

NOTE: See *SGD* for detailed meaning for these parameters.

NOTE: This class exists for backwards compatibility with the Python API and may be removed at some point in the future.

Public Functions

ConstSGD(float *learningRate*, float *weightDecay* = 0, float *lossScaling* = 1, **const** std::vector<*ClipNormSettings*> &*clipNormSettings* = {})
Constructor.

Parameters

- `learningRate`: A constant learning rate.
- `weightDecay`: A constant weight decay value.
- `lossScaling`: A constant loss scaling value.
- `clipNormSettings`: A vector of *ClipNormSettings* (this can be used to set maximum values for weights).

2.2.2 Adam, AdaMax & Lamb

```
#include <popart/adam.hpp>
```

enum popart::**AdamMode**
Enum type describing the mode of an *Adam* optimizer instance.

Values:

enumerator Adam = 0
Adam or AdamW mode, depending on weight decay setting (see Kingma & Ba, 2015 and Loshchilov & Hutter, 2018).

enumerator AdamNoBias
Like *Adam* but without bias correction.

enumerator AdaMax
Adamax mode.

enumerator Lamb
Lamb mode (see You et al., 2020).

enumerator LambNoBias
Like Lamb but without bias correction.

class popart::**Adam** : **public** *popart*::*Optimizer*
AdamW, Lamb and AdaMax optimizer implementation.

Akin to any optimizer implementation, this class is responsible for updating each weight tensor (w) in the model using the gradient (g) of the loss function with respect to the weight as calculated during the backwards pass.

The optimizer has the following **state** for each weight:

- *first-order momentum* (m)
- *second-order momentum* (v)
- *time step* (t)

The optimizer has the following **hyper parameters**:

- *learning rate* (lr)
- *weight decay* (wd)
- *beta1* (β_1)
- *beta2* (β_2)

- *epsilon* (ϵ)
- *loss scaling* (ls)
- *maximum weight norm* (mwn)

The values of these parameters can be shared between all weights but some can be overridden with weight-specific values (see *Adam::insertSpecific*). Hyper parameters are captured using *OptimizerValue* objects and therefore can be either a constant value or a non-constant value that can be adjusted by the user.

The values of #AdamMode and #WeightDecayMode passed to the constructor determines how weights are updated (see below).

In the following we will describe how this optimizer updates a weight using a gradient. In the context of this description the gradient is is the value of the gradient *after* any gradient accumulation has been performed and *after* the application of a loss scaling factor to the gradient has been corrected for.

When the optimizer needs to update a weight, w, using a gradient, g, it first computes a term g_{tmp}, which is effectively is g with L2 regularization applied if the #WeightDecayMode is set to WeightDecayMode::L2Regularization this, as follows:

$$g_{\text{tmp}} := \begin{cases} g & \text{(Decay)} \\ (g + \text{wd} * w) & \text{(L2Regularization)} \end{cases} .$$

Secondly, the optimizer updates the optimizer state as follows:

$$\begin{aligned} m' &:= \beta_1 * m + (1 - \beta_1) * g_{\text{tmp}} \\ v' &:= \begin{cases} \beta_2 * v + (1 - \beta_2) * g_{\text{tmp}}^2 & \text{(Adam/AdamNoBias)} \\ \beta_2 * v + (1 - \beta_2) * g_{\text{tmp}}^2 & \text{(Lamb/LambNoBias)} \\ \max(\beta_2 * v, |g_{\text{tmp}}|) & \text{(AdaMax)} \end{cases} \\ t' &:= t + 1 \end{aligned}$$

Next, it computes the following terms:

$$\begin{aligned} m_{\text{tmp}} &:= \begin{cases} m' & \text{(AdamNoBias/LambNoBias)} \\ \dfrac{m'}{(1 - \beta_1^{t'})} & \text{(Adam/Lamb/AdaMax)} \end{cases} \\ v_{\text{tmp}} &:= \begin{cases} v' & \text{(AdamNoBias/LambNoBias)} \\ \dfrac{v'}{(1 - \beta_2^{t'})} & \text{(Adam/Lamb/AdaMax)} \end{cases} \\ u_{\text{tmp}} &:= \begin{cases} \dfrac{m_{\text{tmp}}}{(\sqrt{v_{\text{tmp}}} + \epsilon)} + \text{wd} * w & \text{(Decay)} \\ \dfrac{m_{\text{tmp}}}{(\sqrt{v_{\text{tmp}}} + \epsilon)} & \text{(L2Regularization)} \end{cases} \end{aligned}$$

Finally, the optimizer updates the weight as follows:

$$w' := \begin{cases} w - \text{lr} * u_{\text{tmp}} & \text{(Adam/AdamNoBias/AdaMax)} \\ w - \left(\dfrac{\min(\|w\|, \text{mwn})}{\|u_{\text{tmp}}\|} \right) * \text{lr} * u_{\text{tmp}} & \text{(Lamb/LambNoBias)} \end{cases}$$

In addition to the above, the *loss scaling* hyper parameter is similar in nature to the velocity scaling parameter. It is a scaling value that is applied to the loss gradient at the start of the the backwards pass and, at the end of the backwards pass, this scaling is reversed by multiplying the gradients for each weight with the inverse of the loss scaling value prior to updating the optimizer state. Using loss scaling can also improve numerical stability of the gradient calculations. If scaledOptimizerState is enabled then the the lossScaling will not be removed before updating the optimizer state. This can improve the numerical stability when accl1_type is set to FLOAT16.

NOTE: The maximum weight norm is referred to as ϕ in You et al., 2020.

Public Functions

bool **hasSpecific**(**const** Tensor &*w*) **const final**

bool **hasSpecific**() **const final**

TensorId **getInverseLossScalingTensorId**(**const** Tensor &*weight*) **const final**

Adam(*OptimizerValue* *defaultLearningRate*, *OptimizerValue* *defaultWeightDecay*, *OptimizerValue* *defaultBeta1*, *OptimizerValue* *defaultBeta2*, *OptimizerValue* *defaultEps*, *OptimizerValue* *lossScaling*, *OptimizerValue* *maxWeightNorm*, *AdamMode* *adamMode*, *WeightDecayMode* *weightDecayMode*, *DataType* *accumType*, *DataType* *accl1Type*, *DataType* *accl2Type*, **const** std::vector<*ClipNormSettings*> &*clipNormSettings* = {}, bool *scaledOptimizerState* = false)
Constructor.

Parameters

- `defaultLearningRate`: The learning rate value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultWeightDecay`: The weight decay value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultBeta1`: The beta1 value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultBeta2`: The beta2 value value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultEps`: The epsilon value to use for weights for which no weight-specific hyper parameter have been inserted.
- `lossScaling`: The loss scaling value to use.
- `maxWeightNorm`: The maxWeightNorm value to use.
- `adamMode`: The AdamMode value to use.
- `weightDecayMode`: The WeightDecayMode value to use.
- `maxWeightNorm`: The maxWeightNorm value to use.
- `accumType`: Data type to use for gradient accumulation.
- `accl1Type`: Data type to use for tensor that stores first-order momentum optimizer state.
- `accl2Type`: Data type to use for tensor that stores second-order momentum optimizer state.
- `clipNormSettings`: A vector of *ClipNormSettings* (this can be used to set maximum values for weights).
- `scaledOptimizerState`: Experimental Option. Does not remove lossScaling before updating the optimizer state. This should have no effect on the update equation. However, it does ensure a more numerically stable implementation when accl1_type is set to DataType::FLOAT16. Note: When loading a model that includes initialised optimizer state, ensure that accl1 and accl2 are scaled by lossScaling and lossScaling^2 respectively.

Adam(*OptimizerValue* *defaultLearningRate*, *OptimizerValue* *defaultWeightDecay*, *OptimizerValue* *defaultBeta1*, *OptimizerValue* *defaultBeta2*, *OptimizerValue* *defaultEps*, *OptimizerValue* *lossScaling*, *AdamMode* *adamMode*, *WeightDecayMode* *weightDecayMode*, *DataType* *accumType*, *DataType* *accl1Type*, *DataType* *accl2Type*, **const** std::vector<*ClipNormSettings*> &*clipNormSettings* = {}, bool *scaledOptimizerState* = false)

Adam(*OptimizerValue* *defaultLearningRate*, *OptimizerValue* *defaultWeightDecay*, *OptimizerValue* *defaultBeta1*, *OptimizerValue* *defaultBeta2*, *OptimizerValue* *defaultEps*, *OptimizerValue* *lossScaling*, *OptimizerValue* *maxWeightNorm*, *AdamMode* *adamMode*, *DataType* *accumType*, *DataType* *accl1Type*, *DataType* *accl2Type*, **const** std::vector<*ClipNormSettings*> &*clipNormSettings* = {}, bool *scaledOptimizerState* = false)

Adam(*OptimizerValue* *defaultLearningRate*, *OptimizerValue* *defaultWeightDecay*, *OptimizerValue* *defaultBeta1*, *OptimizerValue* *defaultBeta2*, *OptimizerValue* *defaultEps*, *OptimizerValue* *lossScaling*, *AdamMode* *adamMode*, *DataType* *accumType*, *DataType* *accl1Type*, *DataType* *accl2Type*, **const** std::vector<*ClipNormSettings*> &*clipNormSettings* = {}, bool *scaledOptimizerState* = false)

Adam(**const** std::map<std::string, std::pair<float, bool>> &*params*, *AdamMode* *adamMode*, *WeightDecayMode* *weightDecayMode*, *DataType* *accumType*, *DataType* *accl1Type*, *DataType* *accl2Type*, **const** std::vector<*ClipNormSettings*> &*clipNormSettings* = {}, bool *scaledOptimizerState* = false)

Constructor.

EXAMPLE:

```
Adam({{"defaultLearningRate", {0.02, False}},
      {"defaultBeta1", {0.9, True}},
      {"defaultBeta2":{0.999, True}}},
      AdamMode::Adam,
      WeightDecayMode::Decay,
      DataType::FLOAT,
      DataType::FLOAT,
      DataType::FLOAT);
```

Parameters

- `params`: A parameter map where keys are one of `"defaultLearningRate"`, `"defaultWeightDecay"`, `"defaultBeta1"`, `"defaultBeta2"`, `"defaultEps"`, `"lossScaling"` or `"maxWeightNorm"`, and the map's values pairs of floats and booleans representing *OptimizerValue* constructor arguments. The map does not have to specify each hyper parameter as default values will be used where parameters are missing.
- `adamMode`: The AdamMode value to use.
- `weightDecayMode`: The WeightDecayMode value to use.
- `maxWeightNorm`: The maxWeightNorm value to use.
- `accumType`: Data type to use for gradient accumulation.
- `accl1Type`: Data type to use for tensor that stores first-order momentum optimizer state.
- `accl2Type`: Data type to use for tensor that stores second-order momentum optimizer state.
- `clipNormSettings`: A vector of *ClipNormSettings* (this can be used to set maximum values for weights).
- `scaledOptimizerState`: Experimental Option. Does not remove lossScaling before updating the optimizer state. This should have no effect on the update equation. However, it does ensure a more numerically stable implementation when accl1_type is set to DataType::FLOAT16. Note: When loading a model that includes initialised optimizer state, ensure that accl1 and accl2 are scaled by lossScaling and lossScaling^2 respectively.

Adam(**const** *Adam*&) = default

~Adam() = default

OptimizerType **type**() **const final**

std::string **type_s**() **const final**

std::unique_ptr<*Optimizer*> **clone**() **const final**

std::unique_ptr<*Op*> **createOp**(**const** Tensor &*weight*, Graph&) **const final**

std::vector<*TensorId*> **getInputIds**(**const** Tensor &*weight*) **const final**
The names of the inputs for the *VarUpdateOp* for the Variable Tensor "weight".

In the returned vector, an empty string ("") is used as a placeholder for constant inputs.

std::vector<std::tuple<*TensorId*, *TensorInfo*>> **getOptimizerInputs**(**const** Tensor &*weight*) **const final**
The names and infos of the optimizer tensors.

void **validReplacement**(**const** *Optimizer* &*other*) **const final**

void **resetTensorData**(Tensor&) **const final**

void **setTensorData**(Tensor&) **const final**

float **getStoredValue**(**const** *TensorId* &*optId*) **const**
Tensor "opt" has an id, based on which it matches a compound scalar which this object can compute from the atomic scalars.

void **insertSpecific**(**const** *TensorId* &*weight*, *OptimizerValue* *learningRate*, *OptimizerValue* *weightDecay*, *OptimizerValue* *beta1*, *OptimizerValue* *beta2*, *OptimizerValue* *eps*, *OptimizerValue* *mwn*)
Insert a weight-specific set of hyper parameters.

Parameters

- `weight`: The TensorId of the weight.
- `learningRate`: The learning rate value to use for this specific weight.
- `weightDecay`: The weight decay value to use for this specific weight.
- `beta1`: The beta1 value to use for this specific weight.
- `beta2`: The beta2 value to use for this specific weight.
- `eps`: The epsilon value to use for this specific weight.
- `mwn`: The max weight norm value to use for this specific weight.

void **setStep**(int64_t *step*)

void **setStep**(**const** *TensorId*&, int64_t *step*)

void **setStep**(std::map<*TensorId*, int64_t> *steps*)

void **insertSpecific**(**const** *TensorId* &*weight*, **const** std::map<std::string, std::pair<float, bool>> &*params*)
Insert a weight-specific set of hyper parameters.

Parameters

- `weight`: The TensorId of the weight.
- `params`: A parameter map where keys are one of `"defaultLearningRate"`, `"defaultWeightDecay"`, `"defaultBeta1"`, `"defaultBeta2"`, `"defaultEps"`, `"lossScaling"` or `"maxWeightNorm"` and the map's values pairs of floats and booleans representing *OptimizerValue* constructor arguments. The map does not have to specify each hyper parameter as default values will be used where parameters are missing.

const OptimizerValueMap &**learningRates**() **const**

const OptimizerValueMap &**weightDecays**() **const**

const OptimizerValueMap &**beta1s**() **const**

const OptimizerValueMap &**beta2s**() **const**

const OptimizerValueMap &**epss**() **const**

const OptimizerValueMap &**maxWeightNorms**() **const**

const *WeightDecayMode* &**getWeightDecayMode**() **const**

bool **useScaledOptimizerState**() **const**

size_t **hash**() **const final**

void **setFactorsFromOptions**(**const** *SessionOptions*&) **final**

Public Static Functions

OptimizerValue **getUnsetLearningRate**()
Default learning rate value.

OptimizerValue **getUnsetWeightDecay**()
Default weight decay value.

OptimizerValue **getUnsetBeta1**()
Default beta1 value.

OptimizerValue **getUnsetBeta2**()
Default beta2 value.

OptimizerValue **getUnsetEps**()
Default epsilon value.

OptimizerValue **getUnsetLossScaling**()
Default loss scaling value.

OptimizerValue **getUnsetMaxWeightNorm**()
Default maximum weight norm value.

Adam **fromDefaultMap**(**const** std::map<std::string, *OptimizerValue*>&, *AdamMode* *adamMode_*, *WeightDecayMode* *decayMode_*, *DataType* *accumType_*, *DataType* *accl1Type_*, *DataType* *accl2Type_*)

2.2.3 AdaDelta, RMSProp & AdaGrad

```
#include <popart/adaptive.hpp>
```

enum popart::**AdaptiveMode**
Enum class representing a type of adaptive optimizer.

Values:

enumerator AdaGrad = 0
AdaGrad optimizer.

enumerator RMSProp
RMSProp optimizer.

enumerator CenteredRMSProp
CenteredRMSProp optimizer.

enumerator AdaDelta
AdaDelta optimizer.

class popart::**Adaptive** : **public** *popart*::*Optimizer*
AdaDelta, RMSProp and AdaGrad optimizer implementation.

Akin to any optimizer implementation, this class is responsible for updating each weight tensor (w) in the model using the gradient (g) of the loss function with respect to the weight as calculated during the backwards pass.

The optimizer has the following **state** for each weight:

- *first-order momentum* (v_1)
- *second-order momentum* (v_2) (only for AdaGrad/RMSProp)
- *third-order momentum* (v_3)

The optimizer has the following **hyper parameters**:

- *learning rate* (lr)
- *weight decay* (wd)
- *alpha* (α)
- *momentum* (m))
- *epsilon* (ϵ)
- *loss scaling* (ls)

The values of these parameters can be shared between all weights but some can be overridden with weight-specific values (see *Adaptive::insertSpecific*). Hyper parameters are captured using *OptimizerValue* objects and therefore can be either a constant value or a non-constant value that can be adjusted by the user.

The values of #AdaptiveMode and #WeightDecayMode passed to the constructor determines how weights are updated (see below).

In the following we will describe how this optimizer updates a weight using a gradient. In the context of this description the gradient is is the value of the gradient *after* any gradient accumulation has been performed and *after* the application of a loss scaling factor to the gradient has been corrected for.

When the optimizer needs to update a weight, w, using a gradient, g, it first computes a term g_{tmp}, which is effectively is g with L2 regularization applied if the #WeightDecayMode is set to WeightDecayMode::L2Regularization this, as follows:

$$g_{\text{tmp}} := \begin{cases} g & \text{(Decay)} \\ (g + \text{wd} * w) & \text{(L2Regularization)} \end{cases} .$$

Secondly, the optimizer updates v_1 the optimizer state as follows:

$$v_1' := \begin{cases} \alpha * m + (1 - \alpha) * g_{\text{tmp}}^2 & \text{(RMSProp/AdaDelta)} \\ \alpha * m + (1 - \alpha) * g_{\text{tmp}}^2 & \text{(CenteredRMSProp)} \\ v_1 + g_{\text{tmp}}^2 & \text{(AdaGrad)} \end{cases}$$

Next, v_2 is updated, but only for CenteredRMSProp:

$$v_2' := \alpha * v_2 + (1 - \alpha) * g_{\text{tmp}} \quad \text{(CenteredRMSProp)}$$

Next, it computes the update term u_{tmp}:

$$u_{\text{tmp}} := \begin{cases} \frac{g_{\text{tmp}}}{\sqrt{v_1'} + \epsilon} & \text{(AdaGrad/RMSProp)} \\ \frac{g_{\text{tmp}}}{\sqrt{v_1' - v_2'^2} + \epsilon} & \text{(CenteredRMSProp)} \\ \frac{g_{\text{tmp}} * \sqrt{v_2 + \epsilon}}{\sqrt{v_1' + \epsilon}} & \text{(AdaDelta)} \end{cases}$$

Next, v_2 is updated, but only for AdaDelta:

$$v_2' := \alpha * v_2 + (1 - \alpha) * u_{\text{tmp}}^2 \quad \text{(AdaDelta)}$$

Next the third momentum is updated for all modes:

$$v_3' := m * v_3 + u_{\text{tmp}}$$

Finally, the optimizer updates the weight as follows:

$$w' := \begin{cases} w - \text{lr} * (v_3' + \text{wd} * w) & \text{(Decay)} \\ w - \text{lr} * v_3' & \text{(L2Regularization)} \end{cases}$$

In addition to the above, the *loss scaling* hyper parameter is similar in nature to the velocity scaling parameter. It is a scaling value that is applied to the loss gradient at the start of the the backwards pass and, at the end of the backwards pass, this scaling is reversed by multiplying the gradients for each weight with the inverse of the loss scaling value prior to updating the optimizer state. Using loss scaling can also improve numerical stability in some cases.

Public Functions

bool **hasSpecific**(**const** Tensor &*w*) **const**

bool **hasSpecific**() **const**

TensorId **getInverseLossScalingTensorId**(**const** Tensor &*weight*) **const**

Adaptive(*OptimizerValue* *defaultLearningRate*, *OptimizerValue* *defaultWeightDecay*, *OptimizerValue* *defaultAlpha*, *OptimizerValue* *defaultMomentum*, *OptimizerValue* *defaultEps*, *OptimizerValue* *lossScaling*, *AdaptiveMode* *adaptiveMode*, *WeightDecayMode* *weightDecayMode*, *DataType* *accumType*, *DataType* *accl1Type*, *DataType* *accl2Type*, *DataType* *accl3Type*, bool *rmspropTFVariant* = false)

Constructor.

Parameters

- `defaultLearningRate`: The learning rate value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultWeightDecay`: The weight decay value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultAlpha`: The alpha value to use for weights for which no weight-specific hyper parameter have been inserted.
- `defaultMomentum`: The momentum value to use for weights for which no weight-specific hyper parameter have been inserted.

- `defaultEps`: The epsilon value to use for weights for which no weight-specific hyper parameter have been inserted.
- `lossScaling`: The loss scaling value to use.
- `adaptiveMode`: The AdaptiveMode value to use.
- `weightDecayMode`: The WeightDecayMode value to use.
- `accumType`: Data type to use for gradient accumulation.
- `accl1Type`: Data type to use for tensor that stores first-order momentum optimizer state.
- `accl2Type`: Data type to use for tensor that stores second-order momentum optimizer state.
- `accl2Type`: Data type to use for tensor that stores third-order momentum optimizer state.

Adaptive(**const** std::map<std::string, std::pair<float, bool>> &*params*, *AdaptiveMode* *adaptiveMode*, *WeightDecayMode* *weightDecayMode*, *DataType* *accumType*, *DataType* *accl1Type*, *DataType* *accl2Type*, *DataType* *accl3Type*, bool *rmspropTFVariant* = false)

Constructor.

EXAMPLE: ``` *Adaptive*({{"defaultLearningRate", {0.02, False}}, */ // {"defaultAlpha", {0.99, True}}}, /** AdaptiveMode::RMSProp, WeightDecayMode::Decay, DataType::FLOAT, DataType::FLOAT, DataType::FLOAT, DataType::FLOAT); ```

Parameters

- `params`: A parameter map where keys are one of `"defaultLearningRate"`, `"defaultWeightDecay"`, `"defaultAlpha"`, `"defaultMomentum"`, `"defaultEps"` or `"lossScaling"`, and the map's values pairs of floats and booleans representing *OptimizerValue* constructor arguments. The map does not have to specify each hyper parameter as default values will be used where parameters are missing.
- `adaptiveMode`: The AdaptiveMode value to use.
- `weightDecayMode`: The WeightDecayMode value to use.
- `accumType`: Data type to use for gradient accumulation.
- `accl1Type`: Data type to use for tensor that stores first-order momentum optimizer state.
- `accl2Type`: Data type to use for tensor that stores second-order momentum optimizer state.
- `accl2Type`: Data type to use for tensor that stores third-order momentum optimizer state.

Adaptive(**const** *Adaptive*&) = default

~Adaptive() = default

OptimizerType **type**() **const final**

std::string **type_s**() **const final**

std::unique_ptr<*Optimizer*> **clone**() **const final**

std::unique_ptr<*Op*> **createOp**(**const** Tensor &*weight*, Graph&) **const final**

std::vector<*TensorId*> **getInputIds**(**const** Tensor &*weight*) **const final**

The names of the inputs for the *VarUpdateOp* for the Variable Tensor "weight".

In the returned vector, an empty string ("") is used as a placeholder for constant inputs.

std::vector<std::tuple<*TensorId*, *TensorInfo*>> **getOptimizerInputs**(**const** Tensor &*weight*) **const final**

The names and infos of the optimizer tensors.

void **validReplacement**(**const** *Optimizer* &*other*) **const final**

void **resetTensorData**(Tensor&) **const final**

void **setTensorData**(Tensor&) **const final**

float **getStoredValue**(**const** *TensorId* &*optId*) **const**
Tensor "opt" has an id, based on which it matches a compound scalar which this object can compute from the atomic scalars.

void **insertSpecific**(**const** *TensorId* &*weight*, *OptimizerValue* *learningRate*, *OptimizerValue* *weightDecay*, *OptimizerValue* *alpha*, *OptimizerValue* *momentum*, *OptimizerValue* *eps*)
Insert a weight-specific set of hyper parameters.

Parameters

- `weight`: The TensorId of the weight.
- `learningRate`: The learning rate value to use for this specific weight.
- `weightDecay`: The weight decay value to use for this specific weight.
- `alpha`: The alpha value to use for this specific weight.
- `momentum`: The momentum value to use for this specific weight.
- `eps`: The epsilon value to use for this specific weight.

void **setStep**(int64_t *step*)

void **setStep**(**const** *TensorId*&, int64_t *step*)

void **setStep**(std::map<*TensorId*, int64_t> *steps*)

void **insertSpecific**(**const** *TensorId* &*weight*, **const** std::map<std::string, std::pair<float, bool>> &*params*)
Insert a weight-specific set of hyper parameters.

Parameters

- `weight`: The TensorId of the weight.
- `params`: A parameter map where keys are one of `"defaultLearningRate"`, `"defaultWeightDecay"`, `"defaultAlpha"`, `"defaultMomentum"`, `"defaultEps"` or `"lossScaling"` and the map's values pairs of floats and booleans representing *OptimizerValue* constructor arguments. The map does not have to specify each hyper parameter as default values will be used where parameters are missing.

const OptimizerValueMap &**learningRates**() **const**

const OptimizerValueMap &**weightDecays**() **const**

const OptimizerValueMap &**alphas**() **const**

const OptimizerValueMap &**momentums**() **const**

const OptimizerValueMap &**epss**() **const**

size_t **hash**() **const**

Public Static Functions

OptimizerValue **getUnsetLearningRate**()
Default learning rate value.

OptimizerValue **getUnsetWeightDecay**()
Default weight decay value.

OptimizerValue **getUnsetAlpha**()
Default alpha value.

OptimizerValue **getUnsetMomentum**()
Default momentum value.

OptimizerValue **getUnsetEps**()
Default epsilon value.

OptimizerValue **getUnsetLossScaling**()
Default loss scaling value.

Adaptive **fromDefaultMap**(**const** std::map<std::string, *OptimizerValue*>&, *AdaptiveMode* *adaptiveMode_*, *WeightDecayMode* *decayMode_*, *DataType* *accumType_*, *DataType* *accl1Type_*, *DataType* *accl2Type_*, *DataType* *accl3Type_*)

2.3 Builder

```
#include <popart/builder.hpp>
```

class popart::**Builder**
An interface for a *Builder*, used for creating ONNX graphs.

A builder interface for creating ONNX graphs.

ONNX defines a specification for describing graphs and serialising them as protobuf files. This class provides a builder interface for creating such a graph.

Note, in ONNX, all Ops belong to an "Opset". The *Builder* itself does not have methods for creating Ops in the ONNX graph, but instead has accessors to Opsets, like AiGraphcoreOpset1, which contain the methods for creating Ops in the graph.

Public Functions

Builder &**createSubgraphBuilder**()
Return a *Builder* for a graph which is nested inside this *Builder*'s graph.

~Builder()

TensorId **addInputTensor**(**const** *TensorInfo* &*tensorInfo*, **const** *popart*::*DebugContext* &*debugContext* = {})
Add a new input tensor to the model (with `TensorInfo`).

Return The unique name of the input tensor.

Parameters

- `tensorInfo`: The shape and type of the input tensor.
- `debugContext`: Optional debug information.

TensorId **addInputTensor**(**const** std::string &*dataType*, **const** *Shape* &*shape*, **const** *popart*::*DebugContext* &*debugContext* = {})
Add a new input tensor to the model (with data type and shape).

Return The unique name of the input tensor.

Parameters

- `dataType`: The type of the input tensor.
- `shape`: The shape of the input tensor.
- `debugContext`: Optional debug information.

TensorId **addInputTensor**(**const** *TensorInfo* &*tensorInfo*, **const** InputSettings &*settings*, **const** *popart*::*DebugContext* &*debugContext* = {})
Add a new input tensor to the model (with `TensorInfo` and additional settings for `TileSet` and `ExchangeStrategy`).

Return The unique name of the input tensor.

Parameters

- `tensorInfo`: The shape and type of the input tensor.
- `InputSettings`: Settings for `TileSet` and `ExchangeStrategy`
- `debugContext`: Optional debug information.

TensorId **addInputTensor**(**const** std::string &*dataType*, **const** *Shape* &*shape*, **const** InputSettings &*settings*, **const** *popart*::*DebugContext* &*debugContext* = {})
Add a new input tensor to the model (with data type and shape, with `TensorInfo` and additional settings for `TileSet` and `ExchangeStrategy`).

Return The unique name of the input tensor.

Parameters

- `dataType`: The type of the input tensor.
- `shape`: The shape of the input tensor.
- `InputSettings`: Settings for `TileSet` and `ExchangeStrategy`
- `debugContext`: Optional debug information.

TensorId **addUntypedInputTensor**(**const** *popart*::*DebugContext* &*debugContext* = {})
Add a new input tensor without a type or shape to the model.

Return The unique name of the input tensor.

Parameters

- `debugContext`: Optional debug information.

void **addInputTensorFromParentGraph**(**const** *TensorId* &*tensorId*)
Add a new named input tensor to the model.

Parameters

- `tensorId`: The identifier string of the input tensor. This identifier must already exist in the parent GraphProto's name scope and must appear topologically before this sub-graph.

TensorId **addInitializedInputTensor**(**const** *ConstVoidData* &*initData*, **const** *popart*::*DebugContext* &*debugContext* = {})
Add a new pre-initialized input tensor to the model.

Return The unique name of the input tensor.

Parameters

- `initData`: The initial data of the input tensor.
- `debugContext`: Optional debug information.

void **addOutputTensor**(**const** *TensorId* &*arg0*)
Adds one of the outputs from a node in the graph into the list of output tensors.

AiOnnxOpset6 **aiOnnxOpset6**()
Return the builder interface for ai.onnx opset 6.

AiOnnxOpset7 **aiOnnxOpset7**()
Return the builder interface for ai.onnx opset 7.

AiOnnxOpset8 **aiOnnxOpset8**()
Return the builder interface for ai.onnx opset 7.

AiOnnxOpset9 **aiOnnxOpset9**()
Return the builder interface for ai.onnx opset 9.

AiOnnxOpset10 **aiOnnxOpset10**()
Return the builder interface for ai.onnx opset 10.

AiOnnxOpset11 **aiOnnxOpset11**()
Return the builder interface for ai.onnx opset 11.

AiOnnxMlOpset1 **aiOnnxMlOpset1**()
Return the builder interface for ai.onnx.ml opset 1.

AiGraphcoreOpset1 **aiGraphcoreOpset1**()
Return the builder interface for ai.graphcore opset 1.

std::vector<*TensorId*> **customOp**(**const** OperatorIdentifier &*opid*, int *opsetVersion*, **const** std::vector<*TensorId*> &*inputs*, **const** unsigned *numOutputs*, **const** std::map<std::string, *popart*::any> &*attributes*, **const** *DebugContext* &*debugContext* = {})

void **customOp**(**const** OperatorIdentifier &*opid*, int *opsetVersion*, **const** std::vector<*TensorId*> &*inputs*, **const** std::vector<*TensorId*> &*outputs*, **const** std::map<std::string, *popart*::any> &*attributes*, **const** *DebugContext* &*debugContext* = {})

template<class **T**>
TensorId **reshape_const**(*T* &*t*, **const** std::vector<*TensorId*> &*args*, **const** std::vector<int64_t> &*shape*, **const** std::string &*name* = {})
This is a helper function that will add a constant and a reshape using the provided domain.

void **outputTensorLocation**(**const** *TensorId* &*nodeOutputName*, *TensorLocation* *value*)

void **recomputeOutput**(**const** *TensorId* &*nodeOutputName*, RecomputeType *value*)

void **recomputeOutputInBackwardPass**(**const** *TensorId* &*nodeOutputName*, RecomputeType *value* = RecomputeType::Recompute)
Enable/disable recomputation of the output of the node in the backward pass.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node.
- `value`: If the recompute is enabled/disabled.

void **recomputeOutputInBackwardPass**(**const** std::set<*TensorId*> &*nodeOutputNames*, RecomputeType *value* = RecomputeType::Recompute)
Enable/disable recomputation of the output of the node in the backward pass.

Parameters

- `nodeOutputNames`: Names of the output tensors of the ONNX node.
- `value`: If the recompute is enabled/disabled.

bool **getRecomputeOutputInBackwardPass**(**const** *TensorId* &*nodeOutputName*)
Return whether the given node will have its output recomputed in the backward pass.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node used to find the node in the ONNX model.

bool **getRecomputeOutputInBackwardPass**(**const** std::set<*TensorId*> &*nodeOutputNames*)
Return whether the given node will have its output recomputed in the backward pass.

Parameters

- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

std::vector<*TensorId*> **checkpointOutput**(**const** std::vector<*TensorId*> &*nodeOutputNames*)
Add checkpoint operations to the model.

This is the same as an identity but is recomputeType Checkpoint by default. Use this to checkpoint a subset of an operation's output tensors.

Return The checkpointed tensors.

Parameters

- `nodeOutputNames`: Tensors to checkpoint.

void **virtualGraph**(**const** *TensorId* &*nodeOutputName*, int64_t *value* = 0)
Set the virtual graph that computes the given node.

Applies when creating a graph for a multi-IPU configuration.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node.
- `value`: The index of the virtual graph that computes this node.

void **executionPhase**(**const** *TensorId* &*nodeOutputName*, int64_t *value* = 0)
Set the execution phase that computes the given node.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node.
- `value`: The index of the virtual graph that computes this node.

void **pipelineStage**(**const** *TensorId* &*nodeOutputName*, int64_t *value*)

void **pipelineStage**(**const** std::set<*TensorId*> &*nodeOutputNames*, int64_t *value*)

void **excludePatterns**(**const** *TensorId* &*nodeOutputName*, **const** std::vector<std::string> &*patternNames*)

void **excludePatterns**(**const** std::set<*TensorId*> &*nodeOutputNames*, **const** std::vector<std::string> &*patternNames*)

void **setSerializeMatMul**(**const** std::set<*TensorId*> &*nodeOutputNames*, std::string *mode*, int64_t *factor*, bool *keep_precision*)
Set the settings for matmuls that should be serialized.

This option will split a matmul into separate smaller matmuls that will be executed in series. This will also serialize the grad operations if training.

Parameters

- `nodeOutputNames`: Name of the output matmul tensors of the ONNX node.
- `mode`: Which dimension of the mat mul to serialize on (choose from 'input_channels', 'output_channels', 'reducing_dim', 'none').
- `factor`: The number of serialised matmuls, must be a factor of the dimensions to serialise on.

void **setPartialsType**(**const** *TensorId* &*nodeOutputName*, **const** std::string *partialsType*)
Set the partials type for the given node.

Used on the convolution op.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node.
- `partialsType`: The type for the partials. Can be either FLOAT or HALF.

void **setEnableConvDithering**(**const** *TensorId* &*nodeOutputName*, int64_t *value*)
Enable convolution dithering.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node.
- `value`: 1, if convolution dithering should be enabled. 0, otherwise.

std::string **getPartialsType**(**const** *TensorId* &*nodeOutputName*)
Get the partials type for the given node.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node.

void **setInplacePreferences**(**const** *TensorId* &*nodeOutputName*, **const** std::map<*OpType*, float> &*prefs*)

void **setAvailableMemoryProportion**(**const** *TensorId* &*nodeOutputName*, **const** float *availableMemoryProportion*)
Set the available memory for the given node.

Used on the convolution op.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node.
- `availableMemoryProportion`: The available memory proportion 0 < x <= 1.

void **setAttribute**(**const** std::string &*attribute*, *popart*::any *value*)
Set an attribute that will be set on all subsequent operations.

popart::any **getAttribute**(**const** std::string *attribute*) **const**
Get an attribute that has been set for all subsequent operations.

bool **hasAttribute**(**const** std::string &*attribute*) **const**

void **clearAttribute**(**const** std::string &*attribute*)
Unset an attribute that will be set on all subsequent operations.

bool **hasAttribute**(**const** std::string &*attribute*)
Check if an attribute is set.

popart::any **getAttribute**(**const** std::string &*attribute*)
Get the current attribute value.

int64_t **getPipelineStage**() **const**
A convenience function for getting the pipeline stage attribute.

int64_t **getExecutionPhase**() **const**
A convenience function for getting the execution phase attribute.

int64_t **getVirtualGraph**() **const**
A convenience function for getting the virtual graph attribute.

void **virtualGraph**(**const** std::set<*TensorId*> &*nodeOutputNames*, int64_t *value* = 0)
Set the virtual graph that computes the given node.

Applies when creating a graph for a multi-IPU configuration.

Parameters

- `nodeOutputNames`: Names of the output tensors of the ONNX node.
- `value`: The index of the virtual graph that computes this node.

void **executionPhase**(**const** std::set<*TensorId*> &*nodeOutputNames*, int64_t *value* = 0)

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** int64_t &*attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)

Add an attribute to the ONNX node which is uniquely identified by the outputs.

This functions will throw an exception if it can't find the unique node or the attribute already exists.

Parameters

- `attributeName`: The name of the attribute to add.
- `attributeValue`: An `int64_t` value of the attribute to add.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** std::vector<int64_t> &*attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)

Add an attribute to the ONNX node which is uniquely identified by the outputs.

This functions will throw an exception if it can't find the unique node or the attribute already exists.

Parameters

- `attributeName`: The name of the attribute to add.
- `attributeValue`: A `std::vector<int64_t>` value of the attribute to add.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** float &*attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)

Add an attribute to the ONNX node which is uniquely identified by the outputs.

This functions will throw an exception if it can't find the unique node or the attribute already exists.

Parameters

- `attributeName`: The name of the attribute to add.
- `attributeValue`: A `float` value of the attribute to add.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** std::vector<float> &*attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)

Add an attribute to the ONNX node which is uniquely identified by the outputs.

This functions will throw an exception if it can't find the unique node or the attribute already exists.

Parameters

- `attributeName`: The name of the attribute to add.
- `attributeValue`: The `std::vector<float>` value of the attribute to add.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** std::string &*attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)
Add an attribute to the ONNX node which is uniquely identified by the outputs.

This functions will throw an exception if it can't find the unique node or the attribute already exists.

Parameters

- `attributeName`: The name of the attribute to add.
- `attributeValue`: A `std::string` value of the attribute to add.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** char **attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** std::vector<std::string> &*attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)
Add an attribute to the ONNX node which is uniquely identified by the outputs.

This functions will throw an exception if it can't find the unique node or the attribute already exists.

Parameters

- `attributeName`: The name of the attribute to add.
- `attributeValue`: A `std::vector<std::string>` value of the attribute to add.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** bool *attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)
Add an attribute to the ONNX node which is uniquely identified by the outputs.

This functions will throw an exception if it can't find the unique node or the attribute already exists.

Parameters

- `attributeName`: The name of the attribute to add.
- `attributeValue`: A bool value of the attribute to add.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

void **addNodeAttribute**(**const** std::string &*attributeName*, **const** *ConstVoidData* &*attributeValue*, **const** std::set<*TensorId*> &*nodeOutputNames*)
Add an attribute to the ONNX node which is uniquely identified by the outputs.

This functions will throw an exception if it can't find the unique node or the attribute already exists.

Parameters

- `attributeName`: The name of the attribute to add.
- `attributeValue`: A constant tensor initializer.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

bool **nodeHasAttribute**(**const** std::string &*attributeName*, **const** std::set<*TensorId*> &*nodeOutputNames*)
Check whether the ONNX node has an attribute set.

This functions will throw an exception if it can't find the unique node.

Parameters

- `attributeName`: The name of the attribute to find.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

int64_t **`getInt64NodeAttribute`**(**`const`** std::string &*attributeName*, **`const`** std::set<*TensorId*> &*nodeOutputNames*)

Get the `int64_t` value of the attribute for the ONNX node.

This functions will throw an exception if it can't find the unique node or the attribute does not exist or it has not been set to the `int64_t` type.

Return Value of the attribute.

Parameters

- `attributeName`: The name of the attribute to find.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

std::vector<int64_t> **`getInt64VectorNodeAttribute`**(**`const`** std::string &*attributeName*, **`const`** std::set<*TensorId*> &*nodeOutputNames*)

Get the `std::vector<int64_t>` value of the attribute for the ONNX node.

This functions will throw an exception if it can't find the unique node or the attribute does not exist or it has not been set to the `std::vector<int64_t>` type.

Return Value of the attribute.

Parameters

- `attributeName`: The name of the attribute to find.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

float **`getFloatNodeAttribute`**(**`const`** std::string &*attributeName*, **`const`** std::set<*TensorId*> &*nodeOutputNames*)

Get the `float` value of the attribute for the ONNX node.

This functions will throw an exception if it can't find the unique node or the attribute does not exist or it has not been set to the `float` type.

Return Value of the attribute.

Parameters

- `attributeName`: The name of the attribute to find.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

std::vector<float> **`getFloatVectorNodeAttribute`**(**`const`** std::string &*attributeName*, **`const`** std::set<*TensorId*> &*nodeOutputNames*)

Get the `std::vector<float>` value of the attribute for the ONNX node.

This functions will throw an exception if it can't find the unique node or the attribute does not exist.

Return Value of the attribute.

Parameters

- `attributeName`: The name of the attribute to find.

- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

std::string **getStringNodeAttribute**(**const** std::string &*attributeName*, **const** std::set<*TensorId*> &*nodeOutputNames*)

Get the `std::string` value of the attribute for the ONNX node.

This functions will throw an exception if it can't find the unique node or the attribute does not exist or it has not been set to the `std::string` type.

Return Value of the attribute.

Parameters

- `attributeName`: The name of the attribute to find.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

std::vector<std::string> **getStringVectorNodeAttribute**(**const** std::string &*attributeName*, **const** std::set<*TensorId*> &*nodeOutputNames*)

Get the `std::vector<std::string>` value of the attribute for the ONNX node.

This functions will throw an exception if it can't find the unique node or the attribute does not exist.

Return Value of the attribute.

Parameters

- `attributeName`: The name of the attribute to find.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

bool **getBoolNodeAttribute**(**const** std::string &*attributeName*, **const** std::set<*TensorId*> &*nodeOutputNames*)

void **removeNodeAttribute**(**const** std::string &*attributeName*, **const** std::set<*TensorId*> &*nodeOutputNames*)

Remove an attribute from the ONNX node.

This functions will throw an exception if it can't find the unique node or the attribute does not exist.

Parameters

- `attributeName`: The name of the attribute to find.
- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

std::vector<std::string> **getAllNodeAttributeNames**(**const** std::set<*TensorId*> &*nodeOutputNames*)

Get all the attribute names from the ONNX node.

This functions will throw an exception if it can't find the unique node.

Parameters

- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

int64_t **getVirtualGraph**(**const** *TensorId* &*nodeOutputName*)

Get the index of the virtual graph that computes this node.

This applies in a multi IPU system.

Parameters

- `nodeOutputName`: Name of the output tensor of the ONNX node used to find the node in the ONNX model.

int64_t **`getVirtualGraph`**(**`const`** std::set<*TensorId*> &*nodeOutputNames*)
Get the index of the virtual graph that computes this node.

This applies in a multi IPU system.

Parameters

- `nodeOutputNames`: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

int64_t **`getExecutionPhase`**(**`const`** *TensorId* &*nodeOutputName*)

int64_t **`getExecutionPhase`**(**`const`** std::set<*TensorId*> &*nodeOutputNames*)

std::string **`getModelProto`**() **`const`**
Retrieve the ONNX serialized ModelProto.

Return A serialized ONNX ModelProto.

void **`saveModelProto`**(**`const`** std::string &*fn*)
Save the builder's ONNX ModelProto into the builder and validate it.

Parameters

- `fn`: The name of a file containing an ONNX model protobuf.

void **`saveInitializersExternally`**(**`const`** std::vector<*TensorId*> &*ids*, **`const`** std::string &*fn*)
Save tensor data externally.

The model data cannot exceed 2GB - the maximum size of a Protobuf message. To avoid this, for large models ONNX tensor data can be saved separately.

Parameters

- `ids`: The names of tensors whose data is to be saved externally.
- `fn`: The name of a file containing the binary tensor data. This can be an absolute or relative path. If a relative path, when the ONNX model is saved, external tensor data will be written to a path relative to your current working directory.

std::vector<*TensorId*> **`getInputTensorIds`**() **`const`**
Return a list of ONNX graph input tensor ids.

Return A vector of input tensor names.

std::vector<*TensorId*> **`getOutputTensorIds`**() **`const`**
Return a list of ONNX graph output tensor ids.

Return A vector of output tensor names.

std::vector<*TensorId*> **`getValueTensorIds`**() **`const`**
Return a list of ONNX graph value tensor ids.

These tensors are stored in the `value_info` section of the ONNX GraphProto structure.

Return A vector of output tensor names.

std::vector<*TensorId*> **getTrainableTensorIds**() **const**
Return a list of ONNX graph initialized tensor ids.

These tensors are stored in the `initialized` section of the ONNX GraphProto structure..

Return A vector of tensor names.

bool **hasValueInfo**(**const** *TensorId* &*id*) **const**
Return whether or not the specified tensor has value info.

A tensor may not have value info if has simply does not exist or if shape inference failed

Return A boolean.

std::vector<int64_t> **getTensorShape**(**const** *TensorId* *id*)
Return an ONNX graph tensor shape, from either the input, output, or value_info lists in the GraphProto.

Return A vector of tensor dimensions.

Parameters

- `id`: Tensor id.

bool **isInitializer**(**const** *TensorId* *id*) **const**
Returns true if the ONNX tensor is in the initializer list of the GraphProto.

Return A boolean.

Parameters

- `id`: Tensor id.

std::string **getTensorDtypeString**(**const** *TensorId* *id*)
Return an ONNX graph tensor type as a lower case string, from either the input, output, or value_info lists in the GraphProto.

Return A lower case string of tensor type.

Parameters

- `id`: Tensor id.

DataType **getTensorDataType**(**const** *TensorId* *id*)
Return a tensor type from either the input, output, or value_info lists in the GraphProto.

Return A tensor type.

Parameters

- `id`: Tensor id.

void **pushNameScope**(**const** std::string &*name*)
Push a name onto the name scope stack.

The names of tensors and nodes added to the ONNX graph will be prefixed with a concatenation of the names in the name stack.

void **popNameScope**()
Remove the last entry in the name scope stack.

std::string **getNameScope**(**const** std::string &*name* = "") **const**
Get the current namescope stack using the default delimiter.

Return A string of the concatenated namescope stack.

Parameters

- `name`: Optional string to concatenate to the end of the stack

void **setGraphName**(**const** std::string &*name*)
Specifies a graph name.

Parameters

- `name`: String to name the graph.

void **setParent**(*Builder* **parent*)
Sets the parent graph of this builder.

Parameters

- `parent`: the builder to become a parent.

Builder ***getParent**() **const**
Returns the parent graph of this graph or null if there is no parent.

bool **hasParent**() **const**
Returns true if this builder represents a subgraph.

Public Static Functions

std::unique_ptr<*Builder*> **create**()
Create a builder for an ONNX model.

std::unique_ptr<*Builder*> **createFromOnnxModel**(**const** std::string &*modelProtoOrFilename*)
Create a builder which loads a serialized ONNX ModelProto into the builder and validates it.

Parameters

- `modelProtoOrFilename`: Either an ONNX model protobuf, or the name of a file containing an ONNX model protobuf.

class popart::**AiGraphcoreOpset1** : **public** *popart*::DomainOpSet

Public Functions

AiGraphcoreOpset1(std::unique_ptr<BuilderImpl> &*impl_*)

TensorId **copyvarupdate**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})
Copies a tensor to an initalised tensor (variable)

This is used to update an initalised tensor (a variable created using addInitializedInputTensor) which retains its value between iterations, by setting the value to the value of another tensor (the updater). The purpose is to manually update the tensor in use cases for variables other than trained parameters (weights) or tensors used by other ops.

Return An alias to the updated variable: to ensure correct ordering of the updated variable, you should use this variable for any op which should operate on the updated variable.

Parameters

- `args`: A vector of the input tensors [tensor to update, updater]
- `debugContext`: Optional debug context.

std::vector<*TensorId*> **groupnormalization**(**const** std::vector<*TensorId*> &*args*, int64_t *num_groups*, float *epsilon* = 1e-05f, **const** *DebugContext* &*debugContext* = {})

Add a group normalization operation to the model.

This is a Poplar extension.

The group will be created from a strided input.

Return A vector of tensors: [y, mean, var].

Parameters

- `args`: A vector of input tensors: [x, scale, bias].
- `num_groups`: The number of groups to separate the channels into.
- `epsilon`: The epsilon value to use to avoid division by zero.
- `debugContext`: Optional debug context.

std::vector<*TensorId*> **multiconv**(**const** *MultiConvInputs* &*tensors*, **const** *MultiConvDilations* &*dilations* = {}, **const** *MultiConvDilations* &*inDilations* = {}, **const** *MultiConvPads* &*pads* = {}, **const** *MultiConvPads* &*outPads* = {}, **const** *MultiConvStrides* &*strides* = {}, **const** std::vector<float> &*availableMemoryProportions* = {}, **const** std::vector<std::string> &*partialsTypes* = {}, **const** nonstd::optional<std::string> *planType* = nonstd::nullopt, **const** nonstd::optional<int> *perConvReservedTiles* = nonstd::nullopt, **const** nonstd::optional<float> *cycleBackOff* = nonstd::nullopt, **const** std::vector<int64_t> *enableConvDithering* = {}, **const** *DebugContext* &*debugContext* = {})

Add a multi-convolution to the model.

Using this multi-convolution API ensures that the convolutions are executed in parallel on the device.

Functionally, a multi-convolution is equivalent to a series of single convolutions. Using this multi-convolution API is always equivalent to calling the single-convolution API (conv) once for each argument.

For example, calling:

```
A0 = conv({X0, W0, B0})
A1 = conv({X1, W1})
```

Is functionally equivalent to calling:

```
{A0, A1} = multiconv({{X0, W0, B0}, {X1, Q1}).
```

It is possible that any two convolutions cannot be executed in parallel due to topological constraints. For example, the following:

```
B = conv({A, W0});
C = B + A
D = conv({C, W1});
```

Cannot be converted to:

```
{B, D} = multiconv({{A, W0}, {C, W1}}).
```

Note that it is not possible to create such a cycle by adding a multi-convolution with this API.

Calls to multiconv() are mapped to poplar::poplin::multiconv::convolution().

All input vectors must be either empty, or equal in length to the number of convolutions. Note that groups for each convolution are automatically inferred from the shapes of the data and weight inputs.

Parameters

- `tensors`: List of [DataId, WeightId, BiasId (optional)] for each convolution.
- `dilations`: The dilations attributes for each convolution.
- `inDilations`: The input dilations attributes for each convolution.
- `pads`: The pads for each convolution.
- `outPads`: The output padding for each convolution.
- `strides`: The strides for each convolution.
- `availableMemoryProportions`: The available memory proportions per conv, each [0, 1).
- `partialsTypes`: The partials type per convolution.
- `planType`: Run convolutions in parallel or series.
- `perConvReservedTiles`: Tiles to reserve per convolution when planning.
- `cycleBackOff`: Cycle back-off proportion, [0, 1).
- `enableConvDithering`: Enable convolution dithering per convolution. If true, then convolutions with different parameters will be laid out from different tiles in an effort to improve tile balance in models.
- `debugContext`: Optional debug context.

Return The TensorId of the output tensor from each convolution.

TensorId **subsample**(**const** std::vector<*TensorId*> &*args*, **const** std::vector<int64_t> &*strides*, **const** *DebugContext* &*debugContext* = {})

Add a sub-sample operation to the model.

This is a Poplar extension.

If multiple tensors are provided that strides will applied to them all.

Return The name of the result tensor.

Parameters

- `args`: Vector of tensor ids to sub-sample.
- `strides`: The strides to use.
- `debugContext`: Optional debug context.

TensorId **printtensor**(**const** std::vector<*TensorId*> &*args*, int64_t *print_gradient* = 1, **const** *DebugContext* &*debugContext* = {}, **const** std::string &*title* = {})

Add a print tensor operation to the model.

This is a Poplar extension.

Return The name of the result tensor.

Parameters

- `args`: Vector of tensor ids to print.
- `print_gradient`:
- `debugContext`: Optional debug context.
- `title`:

TensorId **nop**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add a no-op operation to the model.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids.
- `debugContext`: Optional debug context.

TensorId **scale**(**const** std::vector<*TensorId*> &*args*, float *scale*, **const** *DebugContext* &*debugContext* = {})
Add a scale operation to the model.

This is a Poplar extension.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids.
- `scale`: The scale to apply.
- `debugContext`: Optional debug context.

TensorId **scaledadd**(**const** std::vector<*TensorId*> &*args*, float *scale0*, float *scale1*, **const** *DebugContext* &*debugContext* = {})
Add a scaled add operation to the model.

X = scale0 * T0 + scale1 * T1

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids: [T0, T1, scale0, scale1].
- `scale0`: The scale to apply (if no `scale0` tensor is supplied).
- `scale1`: The scale to apply (if no `scale1` tensor is supplied).
- `debugContext`: Optional debug context.

std::vector<*TensorId*> **lstm**(**const** std::vector<*TensorId*> &*args*, int64_t *outputFullSequence*, **const** *DebugContext* &*debugContext* = {})

TensorId **gelu**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})
Add a GELU operation to the model.

This is a Poplar extension.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids.
- `debugContext`: Optional debug context.

TensorId **detach**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})
Add a detach operation to the model.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids.
- `debugContext`: Optional debug context.

TensorId **depthtospace**(**const** std::vector<*TensorId*> &*args*, int64_t *blocksize*, **const** std::string &*mode* = "DCR", **const** *DebugContext* &*debugContext* = {})

Add the `DepthToSpace` to the model.

(This allows DepthToSpace_11 to be targeted from earlier opsets.)

The purpose of Depth to Space, also known as pixel shuffling, is to rearrange data from the depth (channels) dimension into the spacial (width and height) dimensions. It is an efficient means of learning upsampling alongside mixing convolution with bilinear interpolation and using transpose convolution.

https://github.com/onnx/onnx/blob/master/docs/Operators.md#DepthToSpace

Return A tensor which is a rearrangement of the input tensor.

Parameters

- `args`: Vector containing single tensor input id.
- `blocksize`: Indicates the scale factor: if the input is [N, C, H, W] and the blocksize is B, the output will be [N, C/(B*B), H*B, W*B].
- `mode`: Specifies how the data is rearranged:
  - "DCR": depth-column-row order
  - "CRD": column-row-depth order
- `debugContext`: Optional debug context.

TensorId **round**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add a `Round` operation to the model.

(This allows `Round_11` to be targeted from earlier opsets.)

https://github.com/onnx/onnx/blob/master/docs/Operators.md#Round

Return The normalized output tensor ids.

Parameters

- `args`: Vector of input tensor ids.
- `debugContext`: Optional debug context.

TensorId **init**(*Attributes*::*Ints* *shape*, *Attributes*::*Int* *data_type*, *Attributes*::*Int* *init_type*, *Attributes*::*Int* *batch_axis*, **const** *DebugContext* &*debugContext* = {})

Add an init operation to the model.

Return The name of the result tensor.

Parameters

- `shape`: Shape of the tensor to initialise.
- `data_type`: Data type to initialise tensor with.
- `init_type`: Mode of tensor initialisations.
- `batch_axis`: Axis relative to batch size.
- `debugContext`: Optional debug context.

TensorId **init**(*Attributes*::*Ints* *shape*, *Attributes*::*Int* *data_type*, *Attributes*::*Int* *init_type*, **const** *DebugContext* &*debugContext* = {})

Add an init operation to the model.

Return The name of the result tensor.

Parameters

- `shape`: Shape of the tensor to initialise.
- `data_type`: Data type to initialise tensor with.
- `init_type`: Mode of tensor initialisations.
- `debugContext`: Optional debug context.

TensorId **dynamicslice**(**const** std::vector<*TensorId*> &*args*, *Attributes*::*Ints* *axes*, *Attributes*::*Ints* *sizes*, *Attributes*::*Int* *noOverlap*, **const** *DebugContext* &*debugContext* = {})

Add a dynamic slice operation to the model.

Creates a new slice tensor. For example:

```
slice = tensor[offset]
```

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids: [tensor, offset].
- `axes`: Axes along which to slice.
- `sizes`: Size of the slice in each axis.
- `debugContext`: Optional debug context.

TensorId **dynamicupdate**(**const** std::vector<*TensorId*> &*args*, *Attributes*::*Ints* *axes*, *Attributes*::*Ints* *sizes*, *Attributes*::*Int* *noOverlap*, **const** *DebugContext* &*debugContext* = {})

Add a dynamic update operation to the model.

Creates a copy of a `tensor` with a `slice` inserted at `offset`. For example:

```
out = tensor, out[offset] = slice
```

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids: [tensor, offset, slice].
- `axes`: Axes along which to update.
- `sizes`: Size of the slice in each axis.
- `debugContext`: Optional debug context.

TensorId **dynamiczero**(**const** std::vector<*TensorId*> &*args*, *Attributes*::*Ints* *axes*, *Attributes*::*Ints* *sizes*, **const** *DebugContext* &*debugContext* = {})

Add a dynamic zero operation to the model.

Creates a copy of `tensor` with a slice at `offset` set to zero. For example:

```
out = tensor, out[offset] = 0.0
```

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids [tensor, offset].
- `axes`: Axes along which to erase.
- `sizes`: Size of the slice in each axis.
- `debugContext`: Optional debug context.

TensorId **dynamicadd**(**const** std::vector<*TensorId*> &*args*, *Attributes*::*Ints* *axes*, *Attributes*::*Ints* *sizes*, **const** *DebugContext* &*debugContext* = {})

Add a dynamic add operation to the model.

Creates a copy of `tensor` with `slice` added at `offset`. For example:

```
out = tensor, out[offset] += slice
```

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids: [tensor, offset, slice].
- `axes`: Axes along which to add.
- `sizes`: Size of the slice in each axis.
- `debugContext`: Optional debug context.

TensorId **sequenceslice**(**const** std::vector<*TensorId*> &*args*, *Attributes*::*Int* *zeroUnused*, **const** *DebugContext* &*debugContext* = {})

Slice a 2D tensor based on offsets specified by a tensor.

The outermost dimension is sliced; tOut[tOutOffset:tOutOffset+tN][...] = tIn[tInOffset:tInOffset+tN][...] for each entry in tN/tInOffset/tOutOffset; entries after the first tN==0 may be ignored. Unreferenced elements of tOut are zeroed if zeroUnused is set. The same output element should not be written by multiple inputs.

tIn and tOut must have rank greater than or equal to 2. The outer dimension is sliced; the product of the inner dimensions must match. tInOffset, tOutOffset and tN must be 1d and the same size.

Parameters

- `[sourcedestinationNsourceOffsetdestinationOffset]`:
- `zeroUnused`: Whether to zero unreferenced tOut elements.
- `debugContext`: Optional debug context.

std::vector<*TensorId*> **call**(**const** std::vector<*TensorId*> &*args*, unsigned *num_outputs*, **const** *Builder* &*callee*, **const** *DebugContext* &*debugContext* = {})

Add a call operation to the model.

This is a Poplar extension, to expose manual code re-use to the builder.

Return A vector of tensors; the subgraph outputs.

Parameters

- `args`: Vector of input tensor ids.
- `callee`: The subgraph to call into.
- `debugContext`: Optional debug context.

TensorId **replicatedallreduce**(**const** std::vector<*TensorId*> &*args*, **const** nonstd::optional<std::vector<int64_t>> &*commGroup* = nonstd::nullopt, **const** *DebugContext* &*debugContext* = {})

Add a replicated all-reduce operation to the model.

This is a Poplar extension, to expose manual code re-use to the builder.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids to reduce across.

- `commGroup`: GCL CommGroup parameter.
- `debugContext`: Optional debug context.

TensorId **l1loss**(**const** std::vector<*TensorId*> &*args*, **const** float *lambda*, **const** ReductionType *reduction* = ReductionType::Mean, **const** *DebugContext* &*debugContext* = {})

Add an `l1` loss operation to the model.

Calculates the mean absolute error between each element in the input with a zero target.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids.
- `lambda`: Scale factor of L1 loss.
- `reduction`: Type of reduction to perform on the individual losses.
- `debugContext`: Optional debug context.

TensorId **nllloss**(**const** std::vector<*TensorId*> &*args*, **const** ReductionType *reduction* = ReductionType::Mean, **const** nonstd::optional<int> *ignoreIndex* = nonstd::nullopt, bool *inputIsLogProbability* = false, **const** *DebugContext* &*debugContext* = {})

Add a negative log-likelihood loss operation to the model.

Calculates the nll loss given a probability tensor over classes, and a target tensor containing class labels.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids: probability and tensor.
- `reduction`: Type of reduction to perform on the individual losses.
- `ignoreIndex`: Optional class index to ignore in loss calculation.
- `inputIsLogProbability`: Specifies if the input tensor contains log-probabilities or raw probabilities (false, default).
- `debugContext`: Optional debug context.

TensorId **identityloss**(**const** std::vector<*TensorId*> &*args*, **const** ReductionType *reduction* = ReductionType::Mean, **const** *DebugContext* &*debugContext* = {})

Add an identity loss operation to the model.

Calculates the loss using the identity operator.

Return The name of the result tensor

Parameters

- `args`: Vector of input tensor ids.
- `reduction`: Type of reduction to perform on the individual losses.
- `debugContext`: Optional debug context.

TensorId **ctcloss**(**const** std::vector<*TensorId*> &*args*, **const** ReductionType *reduction* = ReductionType::Mean, **const** unsigned *blank* = 0, **const** std::string &*outDataType* = "UNDEFINED", **const** *DebugContext* &*debugContext* = {})

Add a connectionist temporal classification (CTC) loss operation to the model.

With T being maximum input length, N being batch size, C being number of classes, S being a maximum target length, this op calculates the CTC loss for a logarithmised probabilities tensor with shape [T, N, C], a class target tensor with shape [N, S], an input lengths tensor [N] and a target lengths tensor [N].

Note that C includes a blank class (default=0). The probabilities tensor is padded as required. Target sequences are also padded and are populated with values less than equal to C, not including the blank class, up to their respective target lengths. Note that target lengths cannot exceed input lengths.

Return The name of the result tensor

Parameters

- `args`: [log_probs,targets,input_lengths,target_lengths]
- `reduction`: Type of reduction to perform on the individual losses
- `blank`: The integer representing the blank class.
- `debugContext`: Optional debug context

std::vector<*TensorId*> **_ctcloss**(**const** std::vector<*TensorId*> &*args*, **const** ReductionType *reduction* = ReductionType::Mean, **const** unsigned *blank* = 0, **const** std::string &*outDataType* = "UNDEFINED", **const** *DebugContext* &*debugContext* = {})

std::vector<*TensorId*> **ctcbeamsearchdecoder**(**const** std::vector<*TensorId*> &*args*, unsigned *blank* = 0, unsigned *beamWidth* = 100, unsigned *topPaths* = 1, **const** *DebugContext* &*debugContext* = {})

Add a connectionist temporal classification (CTC) beam search decoder operation to the model.

Calculate the most likely `topPaths` labels and their probabilities given the input `logProbs` with lengths `dataLengths`.

Return The names of the result tensors. These are [labelProbs, labelLengths, decodedLabels], where labelProbs is of shape [batchSize, topPaths], labelLengths is of shape [batchSize, topPaths], and decodedLabels is of shape [batchSize, topPaths, maxTime].

Parameters

- `args`: Vector of input tensor ids. These are [logProbs, dataLengths], where logProbs is of shape [maxTime, batchSize, numClasses], and dataLengths is of shape [batchSize].
- `blank`: The integer representing the blank class.
- `beamWidth`: The number of beams to use when decoding.
- `topPaths`: The number of most likely decoded paths to return, must be less than or equal to `beamWidth`.
- `debugContext`: Optional debug context.

TensorId **shapeddropout**(**const** std::vector<*TensorId*> &*args*, **const** std::vector<int64_t> &*shape*, float *ratio* = 0.5f, **const** *DebugContext* &*debugContext* = {})

Add a shaped dropout operation to the model.

Applies a shaped dropout to the input tensor. This operator requires a shape parameter that is used to define the shape of the dropout mask so that strongly correlated features in the input tensor can be preserved. The provided shape must be broadcastable to the input tensor. Note that this operation targets the poprand library function of the same name.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids.
- `shape`: Shape of dropout mask. Must be broadcastable to the input.
- `ratio`: Probability of dropping an input feature (default = 0.5).
- `name`: Optional identifier for operation.

TensorId **atan2**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})
Add an `atan2` operation to the model.

Returns the element-wise angle theta as a tensor, -pi < theta <= pi, such that for two input tensors x and y and given r != 0, x = r cos theta, and y = r sin theta, element-wise.

In the case of x > 0, theta = arctan(y/x).

Return The name of the result tensor containing element wise theta values.

Parameters

- `args`: Vector of input tensor ids: [y, x].
- `name`: Optional identifier for operation.

TensorId **expm1**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})
Add `expm1` operation to the model.

It computes exp(x) - 1. Calculates the element-wise exponential of the input tensor and subtracts one.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids.
- `name`: Optional identifier for operation.

TensorId **log1p**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})
Add `log1p` operation to the model.

It computes log(x + 1). This calculates the element-wise logarithm of the input tensor plus one.

Return The name of the result tensor.

Parameters

- `args`: Vector of input tensor ids.
- `name`: Optional identifier for operation.

TensorId **reshape**(**const** *TensorId* &*arg*, **const** *Attributes*::*Ints* &*shape*, **const** *DebugContext* &*debugContext* = {})
Add reshape operation to the model.

Reshape the input tensor. This reshape takes the shape to reshape into as an attribute instead of a tensor input as the ONNX reshape op.

Return The name of the result tensor.

Parameters

- `arg`: Vector with single input tensor id.
- `shape`: The shape of the output Tensor. The output Tensor must contain the same number of elements as the input Tensor.
- `name`: Optional identifier for operation.

TensorId **fmod**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})
Add fmod operation to the model.

This is equivalent to C's fmod function. The result has the same sign as the dividend.

Return Computes the element-wise remainder of division. The remainder has the same sign as the dividend.

Parameters

- `args`: Input tensors.

TensorId **remainder**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add remainder operation to the model.

This is equivalent to Python's modulo operator %. The result has the same sign as the divisor.

Return Computes the element-wise remainder of division. The remainder has the same sign as the divisor.

Parameters

- `args`: Input tensors.

TensorId **reverse**(**const** std::vector<*TensorId*> &*args*, **const** std::vector<int64_t> &*dimensions*, **const** *DebugContext* &*debugContext* = {})

Add a reverse operator to the model.

Reverse, or 'flip', the tensor along the specified dimensions

Return The name of the result tensor.

Parameters

- `args`: Input tensors.
- `dimensions`: Dimensions along which to reverse the tensor. If this is empty then this is equivalent to the identity operator

TensorId **packedDataBlock**(**const** std::vector<*TensorId*> &*args*, **const** std::vector<int64_t> &*maxSequenceLengths*, int64_t *resultSize*, int64_t *callbackBatchSize*, **const** *Builder* &*callback*, **const** *DebugContext* &*debugContext* = {})

Add a packedDataBlock operator to the model.

Unpack packed sequences of data according to lengths and offsets tensors, and call the callback on the unpacked sequences.

Return The name of the result tensor.

Parameters

- `args`: Input tensors.
- `maxSequenceLengths`: The maximum length of a sequence in each of the data inputs.
- `resultSize`: The size of the first dimension of the result tensor.
- `callbackBatchSize`: The number of batches to pass to the callback.
- `callback`: The callback function.
- `debugContext`: Optional debug information.

void **abort**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add abort operation to the model.

The operation can be conditional or unconditional.

Parameters

- `args`: Optional input tensor to test condition

TensorId **bitwisenot**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add a bitwise NOT operation to the model.

The operation computes the bitwise NOT of a given integer tensor.

Return The name of the result tensor.

Parameters

- `args`: Input tensor of type integer.

TensorId **bitwiseand**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add a bitwise AND operation to the model.

The operation computes the bitwise AND of given two integer tensors.

Return The name of the result tensor.

Parameters

- `args`: Two broadcastable input tensors of type integer.

TensorId **bitwiseor**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add a bitwise OR operation to the model.

The operation computes the bitwise OR of given two integer tensors.

Return The name of the result tensor.

Parameters

- `args`: Two broadcastable input tensors of type integer.

TensorId **bitwisexor**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add a bitwise XOR operation to the model.

The operation computes the bitwise XOR of given two integer tensors.

Return The name of the result tensor.

Parameters

- `args`: Two broadcastable input tensors of type integer.

TensorId **bitwisexnor**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})

Add a bitwise XNOR operation to the model.

The operation computes the bitwise XNOR of given two integer tensors.

Return The name of the result tensor.

Parameters

- `args`: Two broadcastable input tensors of type integer.

std::vector<*TensorId*> **reducemedian**(**const** std::vector<*TensorId*> &*args*, **const** nonstd::optional<std::vector<int64_t>> &*axes* = nonstd::nullopt, int64_t *keepdims* = 1, **const** *DebugContext* &*debugContext* = {})

TensorId **scatterreduce**(**const** std::vector<*TensorId*> &*args*, *Attributes*::*Int* *axis_size*, *Attributes*::*Int* *axis* = -1, ScatterReduction *reduction* = ScatterReduction::Sum, **const** *DebugContext* &*debugContext* = {})

Add a scatterreduce operation to the model.

Reduces all the values from the src tensor at the indices specified along the given axis.

for i in range(axis_size): output[i] = reduce(src[index == i])

Return The name of the result tensor.

Parameters

- `args`: list of [src, index] tensors
- `axis_size`: Size in the reduced axis
- `axis`: Axis to reduce along (default = -1)
- `reduction`: The type of reduction to apply (default = “sum”)

TensorId **swish**(**const** std::vector<*TensorId*> &*args*, **const** *DebugContext* &*debugContext* = {})
Add a swish operation to the model.

The operation computes the swish activation function, also known as the SiLU activation.

Return The name of the result tensor.

Parameters

- `args`: Vector with single input tensor id.

2.4 Data flow

```
#include <popart/dataflow.hpp>
```

enum popart::**AnchorReturnTypeId**
An anchor tensor is a tensor that the user wants returned after a call to *Session::run()*.

Each call to *Session::run()* results in `batchesPerStep x accumulationFactor x replicationFactor` of such tensors being computed. We refer to the samples associated with each such computation as a micro batch. The dimensions are user-specified by the following parameters:

- `batchesPerStep` is the value in *DataFlow*.
- `accumulationFactor` is the value defined by *SessionOptions::accumulationFactor*.
- `replicationFactor` is the value defined by *SessionOptions::replicatedGraphCount*.

This enum type describes the strategy with which the micro batch values for anchor tensors (or summaries thereof) are written or to the *IStepIO* instance passed to *Session::run*.

See also: *AnchorReturnType*.

NOTE: Anchors are essentially what TensorFlow calls “fetches”.

Values:

enumerator Final = 0
Only return the tensor value for the last micro batch of the *Session::run* call for each replica.

The buffer shape required for this anchor in *IStepIO* is [`replicationFactor`, <anchorTensorShape>] (with dimensions of size 1 removed).

enumerator EveryN
Return the tensor value for every *N*\ th global batch for each replica and for all accumulation steps in that global batch.

Note that the value of *N* is captured by *AnchorReturnType*.

The buffer shape required for this anchor in *IStepIO* is [`batchesPerStep // N`, `accumulationFactor`, `replicationFactor`, <anchorTensorShape>] (with dimensions of size 1 removed).

enumerator All
Return the tensor value for *all* micro batches for each replica.

The buffer shape required for this anchor in *IStepIO* is [`batchesPerStep`, `accumulationFactor`, `replicationFactor`, <anchorTensorShape>] (with dimensions of size 1 removed).

enumerator Sum
Return one tensor value for each replica, doing a sum reduction over the `batchesPerStep` and `accumulationFactor` dimensions.

The buffer shape required for this anchor in *IStepIO* is [`replicationFactor`, <anchorTensorShape>] (with dimensions of size 1 removed).

class popart::**AnchorReturnType**

A class that captures an #AnchorReturnTypeId value and, when this value is `AnchorReturnTypeId::EVERYN`, the associated *N* value.

The constructor takes `std::string` values and converts them as appropriate.

Public Functions

AnchorReturnType(std::string *artString*, *TileSet* *tileSet* = *TileSet*::*Compute*, ExchangeStrategy *exchangeStrategy* = ExchangeStrategy::JustInTime)

Constructor.

NOTE: Attempting to construct an *AnchorReturnType* for `AnchorReturnTypeId::EVERYN` using this constructor will result in an error. Use the constructor that also specifies the return period.

Parameters

- `artString`: - the string to convert to an #AnchorReturnTypeId value. The following values are acceptable (case insensitive):
  - "final" = `AnchorReturnTypeId::FINAL`
  - "all" = `AnchorReturnTypeId::ALL`
  - "sum" = `AnchorReturnTypeId::SUM`

AnchorReturnType(std::string *artString*, int *returnPeriod*, *TileSet* *tileSet* = *TileSet*::*Compute*, ExchangeStrategy *exchangeStrategy* = ExchangeStrategy::JustInTime)

Constructor.

Parameters

- `artString`: The string to convert to an #AnchorReturnTypeId value. The following values are acceptable (case insensitive):
  - "final" = `AnchorReturnTypeId::FINAL`
  - "everyn" = `AnchorReturnTypeId::EVERYN`
  - "all" = `AnchorReturnTypeId::ALL`
  - "sum" = `AnchorReturnTypeId::SUM`
- `returnPeriod`: The value of *N* in the case of `AnchorReturnTypeId::EVERYN`.

AnchorReturnTypeId **id**() **const**

Return the associated #AnchorReturnTypeId, not currently part of public API.

int **rp**() **const**

Return the associated return period (*N*) if the #AnchorReturnTypeId is `AnchorReturnTypeId::EVERYN`, not currently part of public API.

std::size_t **hash**() **const**

const std::string &**str**() **const**

const *TileSet* &**tileSet**() **const**

const ExchangeStrategy &**exchangeStrategy**() **const**

class popart::**DataFlow**

This class specifies parameters for host-device data streams.

The parameters are used to control the amount input data processed each step (that is: each *Session::run* call) determines how data is returned to the user.

See also: *AnchorReturnType*, #AnchorReturnTypeId.

Public Functions

DataFlow()
Default constructor, sets `batchesPerStep` to 0 and does not have any anchors.

DataFlow(int *batchesPerStep*)
Construct *DataFlow* instance without anchor tensors.

Parameters

- `batchesPerStep`: - the number of global batches to run the inference or training session for per call to *Session::run* before returning control to the caller.

DataFlow(int *batchesPerStep*, **const** AnchorReturnTypeMap &*anchorMap*)
Constructor *DataFlow* instance with anchor tensors.

Parameters

- `batchesPerStep`: The number of global batches to run the inference or training session for per call to *Session::run* before returning control to the caller.
- `anchorMap`: A mapping from output tensor TensorId to *AnchorReturnType* indicating the strategy with which to write the anchor tensor values to the *IStepIO* object provided to *Session::run*.

DataFlow(int *batchesPerStep*, **const** std::vector<*TensorId*> *anchorTensorIds*, **const** *AnchorReturnType* &*anchorReturnType* = *AnchorReturnType*("All"))
Constructor *DataFlow* instance with anchor tensors.

Parameters

- `batchesPerStep`: The number of global batches to run the inference or training session for per call to *Session::run* before returning control to the caller.
- `anchorTensorIds`: The tensor ID of anchor tensors.
- `anchorReturnType`: The strategy with which to write anchor tensor values to the *IStepIO* object provided to *Session::run*.

DataFlow(**const** *DataFlow* &*rhs*) = default

DataFlow &**operator=**(**const** *DataFlow* &*rhs*) = default

bool **isAnchored**(*TensorId*) **const**

bool **isBatchCountingRequired**() **const**

const std::vector<*TensorId*> &**anchors**() **const**

const std::vector<int> &**rps**() **const**

int **nAnchors**() **const**

int **batchesPerStep**() **const**

AnchorReturnType **art**(*TensorId* *anchorId*) **const**

unsigned **numOutFetchesPerRepl**(**const** **struct** *SessionOptions* &*opts*, **const** *TensorId* &*anchorId*) **const**

std::size_t **hash**() **const**

const AnchorReturnTypeMap &**getAnchorReturnTypeMap**() **const**

2.5 Device manager

```
#include <popart/devicemanager.hpp>
```

enum popart::**DeviceType**
Values:

enumerator IpuModel = 0

enumerator Cpu

enumerator Ipu

enumerator OfflineIpu

enumerator Sim

enum popart::**DeviceConnectionType**
Values:

enumerator Always = 0

enumerator OnDemand

enumerator Never

enum popart::**SyncPattern**
Values:

enumerator Full = 0

enumerator SinglePipeline

enumerator ReplicaAndLadder

class popart::**DeviceInfo**
Represents a device.

Subclassed by popart::popx::DevicexInfo, popart::popx::DevicexOfflineIpuInfo

Public Functions

DeviceInfo(*DeviceProvider* &*_provider*, *DeviceType* *_type*, *DeviceConnectionType* *_connectionType*, **const** poplar::OptionFlags &*_flags*)

~DeviceInfo()

bool **attach**() = 0
Attach to the device.

Return True if successfully attached to the device.

void **detach**() = 0
Detach from the device.

bool **isAttached**() **const** = 0
True if attached.

DeviceType **getType**() **const**
Get the type of the device.

DeviceConnectionType **getConnectionType**() **const**
Get the connection type of the device.

std::string **toString**() **const**
Return a description of the device.

int **getId**() **const** = 0
Get the device id.

std::string **getVersion**() **const** = 0
Get the version of the software on the IPU.

int **getNumIpus**() **const** = 0
Get the number of IPUs in the device.

int **getTilesPerIPU**() **const** = 0
Get the number of tiles per IPU.

int **getNumWorkerContexts**() **const** = 0
Get the number of worker contexts per tile.

std::vector<unsigned> **getDriverIds**() **const** = 0

const poplar::Target &**getTarget**() **const** = 0

bool **canCompileOffline**() **const**

const poplar::OptionFlags &**getOptionFlags**() **const**

void **setOnDemandAttachTimeout**(**const** unsigned *seconds*)

const unsigned &**getOnDemandAttachTimeout**() **const**

bool **tryAttachUntilTimeout**()

class popart::**DeviceManager**
A class to manage devices.

Public Functions

void **registerDeviceProvider**(*DeviceProvider* **provider*)
Used to register a device provider.

Parameters

- `provider`: A provider.

std::shared_ptr<*DeviceInfo*> **getDevice**(*SyncPattern syncPattern* = *SyncPattern*::*Full*, uint32_t *deviceManagerId* = 0, *DeviceConnectionType connectionType* = *DeviceConnectionType*::*Always*)
Get the Device object of a device by ID.

Return List of requested IPUs.

Parameters

- `syncPattern`: Sync pattern.
- `deviceManagerId`: Number of IPUs to request.

std::vector<std::shared_ptr<*DeviceInfo*>> **enumerateDevices**(*SyncPattern pattern* = *SyncPattern*::*Full*, int *numIpus* = 1, *DeviceType deviceType* = *DeviceType*::*Ipu*, *DeviceConnectionType connectionType* = *DeviceConnectionType*::*Always*, int *tilesPerIPU* = 0)
Get the list of all devices fulfilling the specified criteria.

Return List of requested IPUs.

Parameters

- `pattern`: Sync pattern.

- `numIpus`: Number of IPUs to request.
- `deviceType`: Type of device required.
- `tilesPerIPU`: The number of tiles per IPU required.

std::shared_ptr<*DeviceInfo*> **acquireAvailableDevice**(int *numIpus* = 1, int *tilesPerIPU* = 0, *SyncPattern* *pattern* = *SyncPattern*::*Full*, *DeviceConnectionType* *connectionType* = *DeviceConnectionType*::*Always*, DeviceSelectionCriterion *selectionCriterion* = DeviceSelectionCriterion::First)

Finds the first available hardware device, with a certain number of IPUs.

This method will attach to the device.

Return A device, which can be used with a session. Will return nullptr if no device is available.

Parameters

- `numIpus`: The number of IPUs on the device [=1].
- `tilesPerIPU`: The number of tiles per IPU (0 will match any number) [=0]

std::shared_ptr<*DeviceInfo*> **acquireDeviceById**(int *id*, *SyncPattern* *pattern* = *SyncPattern*::*Full*, *DeviceConnectionType* *connectionType* = *DeviceConnectionType*::*Always*)

Allocates the hardware device by id.

This id can be found running `gc-info -l`. This method will attach to the device.

Return A device. Will return nullptr if the device is not available.

Parameters

- `id`: The index of the IPU to be used.

std::shared_ptr<*DeviceInfo*> **createCpuDevice**()

Create a 'simulated' CPU device.

Return A device.

std::shared_ptr<*DeviceInfo*> **createIpuModelDevice**(std::map<std::string, std::string> &*options*)

Create a 'simulated' IPU Model device.

The following options are supported:

- `numIPUs`: The number of IPUs to simulate [=1]
- `ge`: The number of tiles per IPU [=defaultFewTiles]
- `compileIPUCode`: Whether or not to compile real IPU code for modelling

Return A device.

Parameters

- `options`: Configuration settings for the IPU Model.

std::shared_ptr<*DeviceInfo*> **createSimDevice**(std::map<std::string, std::string> &*options*)

std::shared_ptr<*DeviceInfo*> **createOfflineIPUDevice**(std::map<std::string, std::string> &*options*)

Create a device resembling an IPU for offline compilation, The following options are supported:

- `numIPUs`: The number of IPUs to compile for
- `ge`: The number of tiles per IPU [=defaultManyTiles]
- `ipuVersion`: The ipu architecture [="ipu1"]

- `syncPattern`: The sync pattern to use: full/singlePipline/replicaAndLadder, defaults to full

Return A device.

Parameters

- `options`: Configuration settings for the IPU Model.

void **setOnDemandAttachTimeout**(**const** unsigned *seconds*)
If unable to attach to a device on first try, the attach timeout set here is the length of time (in seconds) that the *DeviceManager* will wait to try and attach.

Note: this only takes effect when trying to attach with a DeviceConnectionType::OnDemand DeviceConnectionType.

Parameters

- `seconds`: The attach timeout in seconds.

Public Static Functions

DeviceManager &**createDeviceManager**()
Accessor for the device manager.

Return A reference to the *DeviceManager*.

class popart::**DeviceProvider**
The interface for device providers which are registered with the device manager.

Subclassed by popart::popx::DevicexManager

Public Functions

~DeviceProvider()

std::shared_ptr<*DeviceInfo*> **getDevice**(*SyncPattern syncPattern*, unsigned *deviceManagerId*, *DeviceConnectionType connectionType*) = 0

void **enumerate**(std::vector<std::shared_ptr<*DeviceInfo*>> &*devices*, uint32_t *requiredNumIPUs*, *SyncPattern syncPattern*, *DeviceType type*, *DeviceConnectionType connectionType*, uint32_t *requiredTilesPerIPU*) = 0
Get the list of all devices fulfilling the specified criteria.

Parameters

- `devices`: Devices to get.
- `requiredNumIPUs`: Number of IPUs to request.
- `syncPattern`: Sync pattern.
- `requiredTilesPerIPU`: Number of tiles per IPU to request.

std::shared_ptr<*DeviceInfo*> **createHostDevice**(*DeviceType type*, **const** std::map<std::string, std::string> &*options*, *SyncPattern syncPattern* = *SyncPattern*::*Full*) = 0
Create a host device for testing.

class popart::popx::**Devicex**

Public Functions

const Ir &**ir**() **const**

const IrLowering &**lowering**() **const**

IrLowering &**lowering**()

Devicex(Executablex &*exe*, std::shared_ptr<*DeviceInfo*> *deviceInfo*)

~Devicex()

void **prepare**()

void **weightsFromHost**()

void **remoteBufferWeightsFromHost**()

void **optimizerFromHost**()

void **setRandomSeedFromHost**()

void **setRngStateFromHost**()

std::vector<uint32_t> **getRngStateToHost**()

void **setRngStateValue**(**const** std::vector<uint32_t>)

std::map<std::string, std::vector<uint64_t>> **cycleCountTensorToHost**()

void **run**(*IStepIO*&, std::string *debugName* = "")

void **weightsToHost**()

void **remoteBufferWeightsToHost**()

void **weightsToHost**(**const** std::map<*TensorId*, *MutableVoidData*>&)

void **readWeights**(**const** IWeightsIO &*dst*)

void **writeWeights**(**const** IWeightsIO &*src*)

std::string **getSummaryReport**(bool *resetProfile* = true) **const**

std::string **getSerializedGraph**() **const**

pva::Report **getReport**() **const**

bool **isEngineLoaded**() **const**

void **setEngineIsLoaded**(bool *isLoaded*)

void **connectRandomSeedStream**()

void **connectRngStateStream**()

void **connectStreamToCallback**(**const** std::string &*streamHandle*, std::function<void)void* > *callback*, unsigned *index*

void **connectStream**(**const** std::string &*streamHandle*, void **host_buffer*)

void **copyFromRemoteBuffer**(**const** PopStreamId *buffer*, void **w*, int *repeat_index*, unsigned *replication_index* = 0)

void **copyToRemoteBuffer**(void **w*, **const** PopStreamId *buffer*, int *repeat_index*, unsigned *replication_index* = 0)

unsigned **getReplicationFactor**() **const**

unsigned **getAccumulationFactor**() **const**

unsigned **getGlobalReplicaOffset**() **const**

unsigned **getGlobalReplicationFactor**() **const**

bool **isReplicatedGraph**() **const**

const *DeviceInfo* ***getDeviceInfo**() **const**

DeviceInfo ***getDeviceInfo**()

std::set<*TensorId*> **getLinearlyCreatedInputTensors**() **const**

std::set<*TensorId*> **getEfficientlyCreatedInputTensors**() **const**

bool **prepareHasBeenCalled**() **const**

void **loadEngineAndConnectStreams**()

Public Members

poplin::PlanningCache **convCache**

poplin::matmul::PlanningCache **matmulCache**

bool **prePlanConvolutions** = true

bool **prePlanMatMuls** = true

2.6 Op creation

2.6.1 Op definition for PopART IR

```
#include <popart/op.hpp>
```

class popart::**Op** : **public** *popart*::Vertex

Subclassed by popart::AbortOp, popart::AbsGradOp, popart::AdaDeltaUpdaterOp, popart::AdamUpdaterOp, popart::AddBiasOp, popart::ArgExtremaOp, popart::AveragePoolGradOp, popart::BasePadOp, popart::BaseSliceOp, popart::BaseSortOp, popart::BatchNormGradOp, popart::BatchNormOp, popart::BinaryComparisonOp, popart::BoundaryOp, popart::CastOp, popart::ConcatGradOp, popart::ConcatOp, popart::ConvFlipWeightsOp, popart::ConvTransposeOp, popart::CoshOp, popart::CtcBeamSearchDecoderOp, popart::CtcGradOp, popart::CumSumGradOp, popart::CumSumOp, popart::ElementWiseBinaryBaseOp, popart::ElementWiseBinaryGradOp, popart::ElementWiseNonLinearUnaryGradOp, popart::ElementWiseUnaryBooleanOp, popart::ElementWiseUnaryOp, popart::ExpandGradOp, popart::ExpandOp, popart::ExpGradOp, popart::Expm1GradOp, popart::GatherGradOp, popart::GatherOp, popart::GetRandomSeedOp, popart::GlobalAveragePoolGradOp, popart::GlobalAveragePoolOp, popart::GlobalMaxPoolGradOp, popart::GlobalMaxPoolOp, popart::GroupNormGradOp, popart::GroupNormOp, popart::GRUGradOp, popart::GRUOp, popart::HasReceptiveFieldOp, popart::HistogramOp, popart::IdentityLossGradOp, popart::IfOp, popart::InitOp, popart::InstanceNormGradOp, popart::InstanceNormOp, popart::IoTileCopyOp, popart::IpuCopyOp, popart::L1GradOp, popart::LambSquareOp, popart::LogSoftmaxGradOp, popart::LossOp, popart::LossScaleUpdateOp, popart::LRNGradOp, popart::LRNOp, popart::LSTMGradOp, popart::LSTMOp, popart::MatMulBaseOp, popart::MaxPoolGradOp, popart::ModifyRandomSeedOp, popart::MultiConvBaseOp, popart::MultiConvDataGradBaseOp, popart::MultiConvWeightsGradBaseOp, popart::NllGradOp, popart::NllWithSoftmaxGradDirectOp, popart::OnehotGradOp, popart::OnehotOp, popart::PackedDataBlockOp, popart::PlaceholderOp, popart::PopartLSTMGradOp, popart::PopartLSTMOp, popart::ReduceGradOp, popart::ReduceOp, popart::ReluGradOp, popart::RemoteExchangeOp, popart::RemoteLoadOp, popart::RemoteStoreOp, popart::ReshapeBaseOp, popart::ResizeOp, popart::RestoreOp, popart::ReverseBaseOp, popart::RMSPropUpdaterOp, popart::ScaledAddOp, popart::ScatterDataGradOp, popart::ScatterOp, popart::ScatterReduceGradOp, popart::ScatterReduceOp, popart::ScatterUpdateGradOp, popart::SequenceSliceOp, popart::ShapeOrLikeOp, popart::SigmoidGradOp, popart::SoftmaxGradDirectOp, popart::SoftmaxGradOp, popart::SplitGradOp, popart::SplitOp, popart::SqrtGradOp, popart::StashOp, popart::SubgraphOp, popart::SubsampleBaseOp, popart::SubsampleGradOp, popart::SyncOp, popart::TanhGradOp, popart::TileOp, popart::TopKGradOp, popart::TransposeBaseOp,

popart::UpsampleOp, popart::VariadicGradOp, popart::VariadicOp, *popart::VarUpdateOp*, popart::WhereOp, popart::WhereXGradOp, popart::WhereYGradOp

Public Types

using SubgraphInSig = std::tuple<*Op**, fwtools::subgraph::OutIndex, std::string>

Public Functions

Settings &**getSettings**()

const *Settings* &**getSettings**() **const**

Settings **getInSettings**(*InIndex*) **const**

Settings **getOutSettings**(*OutIndex*) **const**

Settings **adjustInSettings**(*InIndex*, *Op*::*Settings*) **const**

Settings **adjustOutSettings**(*InIndex*, *Op*::*Settings*) **const**

const OptionalVGraphId **getOptionalVGraphId**() **const**

VGraphId **getVirtualGraphId**() **const**

VGraphIdAndTileSet **getIntrospectionInVirtualGraphId**(*InIndex*) **const**

VGraphIdAndTileSet **getIntrospectionOutVirtualGraphId**(*OutIndex*) **const**

VGraphIdAndTileSet **getIntrospectionInVirtualGraphId**(*InIndex*, std::set<*OpId*> &*visited*) **const**

VGraphIdAndTileSet **getIntrospectionOutVirtualGraphId**(*OutIndex*, std::set<*OpId*> &*visited*) **const**

void **setVirtualGraphId**(**const** OptionalVGraphId)

bool **hasVirtualGraphId**() **const**

const OptionalExecutionPhase **getOptionalExecutionPhase**() **const**

ExecutionPhase **getExecutionPhase**() **const**

void **setExecutionPhase**(**const** OptionalExecutionPhase)

bool **hasExecutionPhase**() **const**

const OptionalBatchSerializedPhase **getOptionalBatchSerializedPhase**() **const**

BatchSerializedPhase **getBatchSerializedPhase**() **const**

void **setBatchSerializedPhase**(**const** OptionalBatchSerializedPhase)

bool **hasBatchSerializedPhase**() **const**

bool **isExcludedFromPattern**(**const** Pattern*) **const**

void **setPipelineStage**(OptionalPipelineStage)

bool **hasPipelineStage**() **const**

PipelineStage **getPipelineStage**() **const**

OptionalPipelineStage **getOptionalPipelineStage**() **const**

int **getInBatchAxis**(*InIndex*) **const**

int **getOutBatchAxis**(*OutIndex*) **const**

void **inheritPlacementAttributes**(bool *inheritSerializations*, AliasModel &*aliasModel*)

Helper function to set Op's placement attributes by inheriting them from other ops in the graph.

Attributes that are set include:

- Pipeline stage
- Execution phase
- Virtual graph ID
- Batch serial phase (optional)

Parameters

- `inheritSerializations`: Set batch serialial phase or not.
- `aliasModel`: An AliasModel containing alias info for this op's graph.

Ir &**getIr**()

const Ir &**getIr**() **const**

Graph &**getGraph**()

const Graph &**getGraph**() **const**

const Scope &**getScope**() **const**

void **setScope**(**const** Scope &*scope*)

const std::string &**getName**() **const**

void **setName**(**const** std::string &*name*)

const OpDebugInfo &**getDebugInfo**() **const**

bool **isNorm**() **const**

bool **isElementWiseUnary**() **const**

bool **canBeReplacedByIdentity**() **const**

Op(**const** OperatorIdentifier &*_opid*, **const** *Op*::Settings &*settings*)

Op(**const** *Op*&)

Op &**operator=**(**const** *Op*&) = delete

~Op()

std::string **str**() **const final**

std::string **debugName**() **const**

void **createAndConnectOutTensor**(*OutIndex*, *TensorId*)

void **append**(std::stringstream &*ss*) **const**

void **toJSON**(std::stringstream &*ss*) **const**

int64_t **memOfOutputs**() **const**

std::set<*InIndex*> **optionalInputs**() **const**

void **defaultConnectInTensor**(*InIndex*, *TensorId*)

void **connectInTensor**(*InIndex*, *TensorId*)

void **connectOutTensor**(*OutIndex*, *TensorId*)

void **disconnectInTensor**(Tensor **tensor*)

void **disconnectInTensor**(*InIndex*, Tensor **tensor*)

void **disconnectInTensor**(*InIndex*)

void **disconnectOutTensor**(Tensor **tensor*)

void **disconnectAllInputs**()

void **disconnectAllOutputs**()

const std::string &**name**() **const**

void **setup**()

void **finalizeDebugInfo**()

void **setCalledSubgraphGradInfo**(**const** FwdGraphToBwdGraphInfo &*calledGraphsGradInfo*)

std::vector<std::unique_ptr<*Op*>> **getGradOps**()

std::vector<std::tuple<OperatorIdentifier, float>> **inplacePriorityDefault**() **const**

std::unique_ptr<*Op*> **getInplaceVariant**(**const** OperatorIdentifier&) **const**

void **growAliasModel**(AliasModel &*aliasModel*) **const**

For certain tasks which involve analysing how Tensors alias each other, such as inplacing, a poprithms::memory::inplace::Graph, which corresponds to this Op's Graph, is constructed.

The poprithms Graph can then be queried for aliasing information, and can have algorithms run on it.

To construct the poprithms Graph, each PopART Op defines what its poprithms equivalents Op are. This method inserts this Op's poprithms::memory::inplace::Op equivalents into the poprithms Graph, which is the container *popAliaser*.

Pre All input tensors of this `Op` have mappings in `aliasModel` before the call to `aliasModel`.

Post All output tensors of this `Op` have mappings in `aliasModel` after to the call to `aliasModel`.

See AliasModel

poprithms::memory::inplace::Proposal **mapInplaceProposal**(**const** AliasModel &*aliasModel*, OperatorIdentifier) **const**

Translate a PopART inplacing proposal, which replaces this non-inplace Op with an inplace Op of type #inplaceId, into an AliasModel equivalent.

This method is defined as a void method which sets a value passed by reference, as opposed to a getter method, so that no poprithms headers need to be included in this file.

Return A tuple where the first element corresponds to an alias gate in the AliasModel and the second element is a input index.

Parameters

- `aliasModel`: Contains the mapping between this Op's (PopART) Graph and the poprithms Graph.
- `inplaceId`: The OperatorIdentifier to translate to the AliasModel equivalent.

view::*Regions* **modifies**(*InIndex*) **const**

view::*Regions* **uses**(*InIndex*) **const**

view::*Regions* **aliases**(*InIndex*, *OutIndex*) **const**

view::*RegMap* **fwdRegMap**(*InIndex*, *OutIndex*) **const**

view::*RegMap* **bwdRegMap**(*InIndex*, *OutIndex*) **const**

bool **doesAlias**() **const**

Return True if there is an input which aliases an output.

bool **isOutplace**() **const**

bool **doesAlias**(*InIndex inIndex*, *OutIndex outIndex*) **const**

Return True if the input at `inIndex` aliases the output at `outIndex`.

bool **modifies**() **const**

Is modifies(i) non-empty for any input index i?

Return True if modifies(i) is non-empty for any i, false otherwise.

bool **modifiesIndex**(*InIndex in*) **const**
Check if an op modifies a tensor at a specific index in.

Return True if it modifies the tensor, false otherwise.

Parameters

- `in`: Index to check.

bool **overwritesTensor**(Tensor *t) **const**
Check if an op overwrites a tensor at a specific index in.

Return True if it overwrites the tensor, false otherwise.

Parameters

- `t`: Tensor to check.

bool **isInplaceViewChange**() **const**
Is this op a view changing op? E.g.

does it not modify it's input, and is it an inplace op? Set at each op level. Examples: ReshapeInplaceOp, IdentityInplace, TransposeInplaceOp

Return true If this is a view changing inplace op

Return false Otherwise

bool **isOutplaceViewChange**() **const**
Same as above for outplace (non-inplace) ops.

Examples: ReshapeOp, IdentityOp, TransposeOp

Return true If this is a view changing outplace op

Return false Otherwise

int **getNonGradInIndex**(int *gradOpOutIndex*) **const**

const std::vector<GradInOutMapper> &**gradInputInfo**() **const**

const std::map<int, int> &**gradOutToNonGradIn**() **const**

std::unique_ptr<*Op*> **clone**() **const** = 0

template<typename **T**>
bool **isConvertibleTo**() **const**

bool **isLossOp**() **const**

bool **isIpuCopyOp**() **const**

bool **copiesOptimizerTensors**() **const**

bool **isOptimizerOp**() **const**

bool **requiresRandomSeed**() **const**

InIndex **getSeedInIndex**() **const**

bool **hasInput**(*InIndex index*) **const**

bool **hasOutput**(*OutIndex index*) **const**

Tensor ***inTensor**(*InIndex index*)

const Tensor ***inTensor**(*InIndex index*) **const**

Tensor *`outTensor`(*OutIndex index*)

`const` Tensor *`outTensor`(*OutIndex index*) `const`

TensorId `inId`(*InIndex index*)

`const` *TensorId* `inId`(*InIndex index*) `const`

TensorId `outId`(*OutIndex index*)

`const` *TensorId* `outId`(*OutIndex index*) `const`

TensorInfo &`inInfo`(*InIndex index*)

`const` *TensorInfo* &`inInfo`(*InIndex index*) `const`

TensorInfo &`outInfo`(*OutIndex index*)

`const` *TensorInfo* &`outInfo`(*OutIndex index*) `const`

`const` *Shape* &`inShape`(*InIndex index*) `const`

`const` *Shape* &`outShape`(*OutIndex index*) `const`

size_t `inTensorCount`() `const`

size_t `outTensorCount`() `const`

Rank `inRank`(*InIndex index*) `const`

Rank `outRank`(*OutIndex index*) `const`

OutIndex `outIndex`(Tensor*) `const`

void `appendAttributes`(OpSerialiserBase&) `const`

void `appendOutlineAttributes`(OpSerialiserBase&) `const`

void `appendMore`(OpSerialiserBase&) `const`

Shape `prettyNpOut`(`const` *Shape* &*s0*, `const` *Shape* &*s1*) `const`

TensorInfo `prettyNpOut`(`const` *TensorInfo* &*i0*, `const` *TensorInfo* &*i1*) `const`

std::vector<`const` Graph*> `getCalledGraphs`() `const`

std::vector<GraphId> `getCalledGraphIds`() `const`

SubgraphIndex `getCalledGraphIndex`(`const` GraphId &*id*) `const`

InIndex `opInToSubgraphInIndex`(*SubgraphIndex subgraphIndex*, *InIndex inIndex*)

InIndex `subgraphInToOpInIndex`(*SubgraphIndex subgraphIndex*, *InIndex inIndex*)

OutIndex `opOutToSubgraphOutIndex`(*SubgraphIndex subgraphIndex*, *OutIndex outIndex*)

OutIndex `subgraphOutToOpOutIndex`(*SubgraphIndex subgraphIndex*, *OutIndex outIndex*)

std::string `getSubgraphEquivId`() `const`

std::map<fwtools::subgraph::InIndex, *SubgraphInSig*> `getSubgraphInputs`() `const`

std::map<fwtools::subgraph::OutIndex, std::set<*Op**>> `getSubgraphOutputs`() `const`

float `getSubgraphValue`() `const` = 0

`constexpr` float `getHighSubgraphValue`() `const`

`constexpr` float `getLowSubgraphValue`() `const`

float `calcAutoVirtualGraphCost`(std::set<int> &*inputs_seen*)

bool `isOutlineable`() `const`

bool `hasSideEffect`() `const`

bool `inputsUnmodifiable`() `const`

bool **consumesGraphOutput**() **const**

bool **producesGraphOutput**() **const**

bool **inputUnmodifiable**(*InIndex in*) **const**
Check if input is unmodifiable or aliases an unmodifiable tensor.

Return True if any connected variable tensor has a non-empty alias chain and is unmodifiable, false otherwise.

Parameters

- `in`: InIndex to check.

bool **hasAliasedModifiers**(*OutIndex out*) **const**
Check if output is modified by any consumer.

Return True if any consumer of any aliased tensor downstream modifies a non-empty region, false otherwise.

Parameters

- `out`: OutIndex to check.

bool **isParentOf**(**const** *Op**) **const**

bool **isChildOf**(**const** *Op**) **const**

bool **canShard**() **const**

ReductionType **getShardReductionType**(*OutIndex index*) **const**

float **getShardRescaleFactor**(*Op* ***const** *shardedOp*, *OutIndex index*) **const**

std::map<*TensorId*, std::vector<*TensorId*>> **shard**(**const** std::map<*TensorId*, std::vector<*TensorId*>> &*inputs*)

ShardingPlan **shard**(**const** ShardingPlan *plan*)

void **configureShardedOp**(*Op* ***const** *shardedOp*, **const** *Settings* ***const** *settings_*) **const**

ReplicatedTensorShardingIndices **getReplicatedTensorShardingIndices**() **const**

void **configureForReplicatedTensorSharding**(*ReplicatedTensorShardingIndices indices*, CommGroup *shardingDomain*)

void **transferBaseProperties**(*Op *to*)

Op ***getPrecedingOp**(*InIndex inIndex*)

Op ***getFollowingOp**(*OutIndex outIndex* = 0)

std::vector<*Op**> **getFollowingOps**(*OutIndex outIndex* = 0)

template<typename **T**>
T ***getPrecedingOp**(*InIndex inIndex*)

template<typename **T**>
T ***getFollowingOp**(*OutIndex outIndex* = 0)

template<typename **T**>
std::vector<*T**> **getFollowingOps**(*OutIndex outIndex* = 0)

Public Members

std::unique_ptr<TensorIndexMap> **input**

std::unique_ptr<TensorIndexMap> **output**

OpId **id** = {-1}

OperatorIdentifier **opid**

bool **pruneable** = true

Settings **settings**

OpDebugInfo **debugInfo**

struct Settings

Public Functions

Settings(Graph &*graph_*, **const** std::string &*name_*)

Settings(Graph &*graph_*, **const** std::string &*name_*, **const** Scope &*scope_*)

~Settings() = default

Settings(**const** *Settings*&) = default

Settings **copy**(**const** std::string &*new_name*)

void **setFromAttributes**(**const** *Attributes* &*attributes*)

Ir &**getIr**() **const**

Public Members

std::reference_wrapper<Graph> **graph**

std::string **name** = ""

Scope **scope**

RecomputeType **recomputeType** = RecomputeType::Undefined

OptionalTensorLocation **tensorLocation**

std::vector<std::tuple<std::string, float>> **inplacePriorityVeto**

std::unordered_set<std::string> **excludePatterns**

OptionalVGraphId **vgraphId**

OptionalPipelineStage **pipelineStage**

OptionalExecutionPhase **executionPhase**

OptionalBatchSerializedPhase **batchSerializedPhase**

TileSet **tileSet** = {*TileSet*::*Compute*}

ExecutionContext **executionContext** = {ExecutionContext::Normal}

std::map<*InIndex*, *InIndex*> **inferTensorMappingToFrom**

double **schedulePriority** = {0.0}

std::map<std::string, std::string> **extraOutlineAttributes**

uint64_t **debugInfoId** = {0}

bool **optimizerOp** = {false}

struct popart::**POpCmp**

To prevent non-determinism, *POpCmp* is used on any sets and maps that use pointers to operators as a set/map key.

Public Functions

bool **operator()**(*Op* ***const** &*a*, *Op* ***const** &*b*) **const**

```
#include <popart/opmanager.hpp>
```

class popart::**OpDefinition**

Public Types

using DataTypes = std::vector<*DataType*>

using Inputs = std::vector<*Input*>

using Outputs = std::vector<*Output*>

using Attributes = std::map<std::string, *Attribute*>

Public Functions

OpDefinition()

OpDefinition(*Inputs* *i*, *Outputs* *o*, *Attributes* *a*)

Public Members

Inputs **inputs**

Outputs **outputs**

Attributes **attributes**

struct Attribute

Public Functions

Attribute(std::string *regex*)

Public Members

std::string **supportedValuesRegex**

struct Input

Public Functions

Input(std::string *n*, std::vector<*DataType*> *t*, bool *_constant* = false)

Public Members

std::string **name**

std::vector<*DataType*> **supportedTensors**

bool **constant**

struct Output

Public Functions

Output(std::string *n*, std::vector<*DataType*> *t*)

Public Members

std::string **name**

std::vector<*DataType*> **supportedTensors**

class popart::**OpCreatorInfo**

Public Functions

OpCreatorInfo(**const** OperatorIdentifier &*_opid*, **const** *Op*::*Settings* &*_settings*, **const** *Attributes* &*_attributes*, **const** std::vector<*TensorId*> &*_inputIds*, **const** std::vector<*TensorId*> &*_outputIds*)

bool **hasInputIds**() **const**

bool **hasOutputIds**() **const**

const std::vector<*TensorId*> &**getInputIds**() **const**

const std::vector<*TensorId*> &**getOutputIds**() **const**

Tensor *__getInputTensor__(int *index*) **const**

TensorData *__getInputTensorData__(int *index*) **const**

TensorInfo &**getInputTensorInfo**(int *index*) **const**

bool **hasInputTensor**(int *index*) **const**

template<typename **T**>
std::vector<*T*> **getInputData**(int *index*, **const** std::set<*DataType*> &*acceptedTypes*) **const**

template<typename **T**>
std::vector<*T*> **getInputData**(int *index*) **const**

template<typename **T**>
T **getInputScalarValue**(int *index*) **const**

template<typename **T**>
T **getInputScalarValue**(int *index*, *T defaultValue*) **const**

Public Members

const OperatorIdentifier &**opid**

const *Op*::*Settings* &**settings**

const *Attributes* &**attributes**

class popart::**OpManager**

Public Types

using **OpFactoryFunc** = std::function<std::unique_ptr<*Op*>(**const** *OpCreatorInfo*&)>

using **ComplexOpFactoryFunc** = std::function<*Op**(**const** *OpCreatorInfo*&, Graph &graph)>

Public Functions

OpManager() = default

Public Static Functions

void **registerOp**(**const** *OpInfo* &*opInfo*)

Attributes **getAttributesFromAnyMap**(std::map<std::string, *popart*::any> *attributes*)

std::unique_ptr<*Op*> **createOp**(**const** *OpDomain* &*domain*, **const** *OpType* &*type*, **const** int *opsetVersion*, Graph &*graph*, **const** std::string &*name* = "", **const** Scope &*scope* = {}, **const** *Attributes* &*_attr* = {}, **const** std::vector<*TensorId*> &*inputIds* = {}, **const** std::vector<*TensorId*> &*outputIds* = {})

std::unique_ptr<*Op*> **createOp**(**const** OperatorIdentifier &*opid*, Graph &*graph*, **const** std::string &*name* = "", **const** *Attributes* &*_attr* = {})

std::unique_ptr<*Op*> **createOpWithInputs**(**const** OperatorIdentifier &*opid*, Graph &*graph*, **const** std::string &*name*, **const** *Attributes* &*_attr*, **const** std::vector<*TensorId*> &*inIds*)

Op ***createOpInGraph**(**const** *Node* &*node*, Graph &*graph*)

const std::vector<OperatorIdentifier> **getSupportedOperations**(bool *includePrivate*)

const std::vector<OperatorIdentifier> **getUnsupportedOperations**(int *opsetVersion*)

const OpDefinitions **getSupportedOperationsDefinition**(bool *includePrivate*)

OpVersion **getOpVersionFromOpSet**(**const** *OpDomain* &*opDomain*, **const** *OpType* &*type*, **const** int *opsetVersion*)

class **OpInfo**

Public Functions

OpInfo(**const** OperatorIdentifier &*_id*, bool *_isPublic*, **const** *OpDefinition* &*_details*, *OpFactoryFunc* *_f1*)

OpInfo(**const** OperatorIdentifier &*_id*, bool *_isPublic*, **const** *OpDefinition* &*_details*, *ComplexOpFactoryFunc* *_f2*)

OpFactoryFunc &**getSimpleFactory**()

ComplexOpFactoryFunc &**getComplexFactory**()

bool **hasComplexFactory**()

Public Members

bool **isPublic**

const OperatorIdentifier **id**

OpDefinition **details**

```
#include <popart/op/varupdate.hpp>
```

class popart::**VarUpdateOp** : **public** *popart*::*Op*

Base class used to define PopART ops that update variable tensors.

Subclassed by popart::AccumulatorScaleOp, popart::VarUpdateWithUpdaterOp

Public Functions

VarUpdateOp(**const** OperatorIdentifier&, **const** *Op*::*Settings*&)

void **setup**() **final**

view::*Regions* **aliases**(*InIndex* *in*, *OutIndex*) **const override**

view::*Regions* **modifies**(*InIndex*) **const override**

std::map<*InIndex*, *TensorId*> **optimizerInputs**() **const** = 0

bool **isOptimizerOp**() **const override**

ReplicatedTensorShardingIndices **getReplicatedTensorShardingIndices**() **const override**

void **growAliasModel**(AliasModel&) **const override**

Public Static Functions

InIndex **getVarToUpdateInIndex**()

OutIndex **getUpdatedVarOutIndex**()

2.6.2 Op definition for Poplar implementation

```
#include <popart/popx/opx.hpp>
```

class popart::popx::**Opx** : **public** *popart*::popx::PopOpx

Public Functions

Opx(*Op**, *Devicex**)

~Opx()

poplar::Tensor **createInput**(*InIndex* *index*, **const** poplar::DebugNameAndId &*dnai*) **const**

snap::Tensor **createInputTensor**(*popart*::*InIndex* *index*, **const** poplar::DebugNameAndId &*dnai*) **const final**

poplar::Tensor **unwindTensorLayout**(poplar::Tensor *tensor*, *InIndex*, *OutIndex*) **const**

snap::Tensor **unwindTensorLayout**(snap::Tensor *tensor*, *InIndex*, *OutIndex*) **const final**

bool **createsEquiv**(int *index0*, **const** *Opx* **opx1*, int *index1*) **const**

poplar::Tensor **cloneNcopy**(poplar::program::Sequence&, *TensorId*) **const**

poplar::Tensor **cloneNcopy**(poplar::program::Sequence&, **const** poplar::Tensor&, **const** std::string *name* = "") **const**

poplar::Tensor **broadcast**(**const** std::vector<int64_t>&, *TensorId*) **const**

poplar::Tensor **broadcast**(**const** std::vector<int64_t>&, poplar::Tensor) **const**

poplar::Graph &**graph**() **const**

const poplar::Tensor &**get**(*TensorId*) **const**

const poplar::Tensor &**getView**(*TensorId*) **const**

void **insert**(*TensorId*, **const** poplar::Tensor&) **const**

const poplar::Tensor &**getInTensor**(*InIndex index*) **const**

const poplar::Tensor &**getOutTensor**(*OutIndex index*) **const**

const poplar::Tensor &**getInView**(*InIndex index*) **const**

const poplar::Tensor &**getOutView**(*OutIndex index*) **const**

void **setOutTensor**(*OutIndex index*, **const** poplar::Tensor &*tensor*) **const**

poplar::Tensor **getConst**(**const** poplar::Type &*type*, **const** std::vector<size_t> &*shape*, double *val*, **const** std::string &*name*) **const**

poplar::Tensor **getScalarVariable**(**const** poplar::Type &*type*, **const** std::string &*name*) **const**

void **grow**(snap::program::Sequence&) **const final**

void **grow**(poplar::program::Sequence&) **const**

2.7 Utility classes

2.7.1 Tensor information

```
#include <popart/tensorinfo.hpp>
```

enum popart::**DataType**

There is a one-to-one correspondence between `popart::DataTypes` and `ONNX_NAMESPACE::TensorProto_DataTypes`, or `decltype`(ONNX_NAMESPACE::TensorProto().data_type()).

Values:

enumerator UINT8 = 0

enumerator INT8

enumerator UINT16

enumerator INT16

enumerator INT32

enumerator INT64

enumerator UINT32

enumerator UINT64

enumerator BOOL

enumerator FLOAT

enumerator FLOAT16

enumerator BFLOAT16

enumerator DOUBLE

enumerator COMPLEX64

enumerator COMPLEX128

enumerator STRING

enumerator UNDEFINED

class popart::**DataTypeInfo**

Public Functions

DataTypeInfo(*DataType* *type__*, int *nbytes__*, bool *isFixedPoint__*, std::string *name__*, std::string *lcasename__*)

DataType **type**() **const**

const int &**nbytes**() **const**

const std::string &**name**() **const**

const std::string &**lcasename**() **const**

bool **isFixedPoint**() **const**

class popart::**TensorInfo**

Public Functions

TensorInfo(*DataType*, **const** *Shape*&)

Create TensorInformation based on data type and shape.

Parameters

- `data_type`: - The data type.
- `shape`: - The actual shape of the tensor.

TensorInfo(*DataType* *data_type*, **const** *Shape* &*shape*, **const** *Shape* &*meta_shape*)

Create TensorInformation based on data type, shape and meta shape.

Parameters

- `data_type`: - The data type.
- `shape`: - The actual shape of the tensor.
- `meta_shape`: - The meta shape of the tensor, which can for example be used to store the original tensor shape before replicated tensor sharding was applied.

TensorInfo(std::string *data_type*, std::string *shape*)

TensorInfo(std::string *data_type*, **const** *Shape*&)

TensorInfo(**const** ONNX_NAMESPACE::TensorProto&)

TensorInfo(**const** ONNX_NAMESPACE::TypeProto&)

void **set**(**const** ONNX_NAMESPACE::TensorProto&)

void **set**(**const** ONNX_NAMESPACE::TypeProto&)

TensorInfo() = default

void **set**(*DataType*)

void **set**(*DataType*, **const** *Shape*&)

void **set**(*DataType*, **const** *Shape*&, **const** *Shape*&)

const *Shape* &**shape**() **const**

const *Shape* &**metaShape**() **const**

std::vector<size_t> **shape_szt**() **const**

Rank **rank**() **const**

int64_t **nelms**() **const**

int64_t **nbytes**() **const**

int64_t **dim**(int *i*) **const**

DataType **dataType**() **const**

const std::string &**data_type**() **const**

const std::string &**data_type_lcase**() **const**

void **append**(std::ostream&) **const**

bool **isSet**() **const**

bool **operator==**(**const** *TensorInfo*&) **const**

bool **operator!=**(**const** *TensorInfo*&) **const**

Shape **shapeFromString**(**const** std::string &*s*) **const**

ONNX_NAMESPACE::TypeProto **getOnnxTypeProto**() **const**

const *DataTypeInfo* ***getDataTypeInfo**() **const**

2.7.2 Tensor location

```
#include <popart/tensorlocation.hpp>
```

enum popart::**ReplicatedTensorSharding**
Enum type to specify whether to shard tensors over replicas.

Values:

enumerator Off = 0
Don't shard tensors over replicas.

enumerator On = 1
Do shard tensors over replicas.

enumerator N = 2
Number of values.

class popart::**TensorLocation**
Class that describes the memory characteristics of one or multiple tensors.

See also: *SessionOptions*.

Public Functions

TensorLocation()
Equivalent to calling *TensorLocation*(TensorStorage::Undefined, TileSet::Compute, TileSet::Compute, ReplicatedTensorSharding::Off)

TensorLocation(*TensorStorage* *storage*)
Equivalent to calling *TensorLocation*(storage, TileSet::Compute, TileSet::Compute, ReplicatedTensorSharding::Off)

TensorLocation(*TensorStorage* *storage*, *ReplicatedTensorSharding* *replicatedTensorSharding*)
Equivalent to calling *TensorLocation*(storage, TileSet::Compute, TileSet::Compute, replicatedTensorSharding)

TensorLocation(*TensorStorage* *storage*, *ReplicatedTensorSharding* *replicatedTensorSharding*, CommGroup *shardingDomain*)
Equivalent to calling *TensorLocation*(storage, TileSet::Compute, TileSet::Compute, replicatedTensorSharding, shardingDomain)

TensorLocation(*TensorStorage* *storage*, *TileSet* *loadTileSet*, *TileSet* *storageTileSet*, *ReplicatedTensorSharding* *replicatedTensorSharding*)
Construct a *TensorLocation* from parameters.

Parameters

- `storage`: The memory location of the tensor(s).
- `loadTileSet`: The tiles through which the tensor(s) are loaded onto the chip.
- `storageTileSet`: The tiles on which the tensor(s) are stored.
- `replicatedTensorSharding`: Whether to apply replicated tensor. sharding.

TensorLocation(*TensorStorage* *storage*, *TileSet* *loadTileSet*, *TileSet* *storageTileSet*, *ReplicatedTensorSharding* *replicatedTensorSharding*, CommGroup *shardingDomain*)
Construct a *TensorLocation* from parameters.

Parameters

- `storage`: The memory location of the tensor(s).
- `loadTileSet`: The tiles through which the tensor(s) are loaded onto the chip.
- `storageTileSet`: The tiles on which the tensor(s) are stored.
- `replicatedTensorSharding`: Whether to apply replicated tensor. sharding.
- `shardingDomain`: GCL communication group across which to shard the tensor. Perpendicular replicas will not shard, and reduce gradients normally (via AllReduce). Defaults to sharding across all replicas.

TensorLocation(std::vector<int64_t> *serialized*)

TensorLocation &**operator=**(**const** *TensorLocation* &*rhs*) = default

bool **operator==**(**const** *TensorLocation* &*rhs*) **const**

bool **operator!=**(**const** *TensorLocation* &*rhs*) **const**

std::vector<int64_t> **serialize**() **const**

bool **isRemote**() **const**

Public Members

TensorStorage **storage**
The memory location of the tensor(s).

TileSet **loadTileSet**
The tiles through which the tensor(s) are loaded onto the chip.

TileSet **storageTileSet**
The tiles on which the tensor(s) are stored.

ReplicatedTensorSharding **replicatedTensorSharding**
Whether to apply replicated tensor sharding (RTS) or not.

CommGroup **shardingDomain**
The GCL comm groups across which to shard the tensor.

enum popart::**TensorStorage**
Enum type that determines where a tensor is stored.

Values:

enumerator OnChip = 0
Store the tensor in on-chip memory.

enumerator OffChip = 1
Store the tensor in streaming memory.

enumerator N = 2
Number of values.

enum popart::**TileSet**
Enum type to specify a set of tiles.

Values:

enumerator Compute = 0
The set of tiles designated for compute operations.

enumerator IO = 1
The set of tiles designated for IO operations.

enumerator Undefined = 2
Undefined (no) tile set.

enumerator N = 3
Number of values.

2.7.3 Region

```
#include <popart/region.hpp>
```

class popart::view::**Region**
A rectangular sub-region of a Shape.

Public Functions

Region(**const** std::vector<int64_t> &*lower_*, **const** std::vector<int64_t> &*upper_*)

Region(**const** std::vector<int64_t> &*lower_*, **const** std::vector<int64_t> &*upper_*, **const** AccessType *accessType*)

int64_t **rank**() **const**

int64_t **nelms**() **const**

bool **isEmpty**() **const**

Region **intersect**(**const** *Region* &*rhs*) **const**

Region **transpose**(**const** *Shape shape*) **const**

Region **reverse**(**const** *Shape shape*, **const** *Shape dimensions*) **const**

Regions **sub**(**const** *Regions* &*rhs*, bool *include_empty* = false) **const**

Regions **sub**(**const** *Region* &*rhs*, bool *include_empty* = false) **const**

Regions **add**(**const** *Region* &*rhs*) **const**

Regions **cut**(**const** std::vector<std::set<int64_t>> &*cuts*, bool *include_empty* = false) **const**

Regions **reshape**(*Region fullInRegion*, *Region fullOutRegion*) **const**

std::pair<int64_t, *Region*> **merge**(**const** *Region* &*rhs*) **const**

bool **contains**(**const** std::vector<int64_t> &*index*) **const**

bool **contains**(**const** *Region* &*rhs*) **const**

int64_t **flatIndex**(**const** std::vector<int64_t> &*index*) **const**

std::vector<int64_t> **dimIndex**(int64_t *index*) **const**

void **checks**() **const**

bool **operator==**(**const** *Region*&) **const**

bool **operator!=**(**const** *Region*&) **const**

const std::vector<int64_t> &**getLower**() **const**

const std::vector<int64_t> &**getUpper**() **const**

void **append**(std::ostream &*ss*) **const**

AccessType **getAccessType**() **const**

void **setAccessType**(AccessType *at*)

Public Static Functions

Region **getEmpty**(int64_t *r*)

Region **getFull**(**const** *Shape* &*s*, AccessType *accessType* = AccessType::ReadWrite)

2.7.4 Error handling

```
#include <popart/error.hpp>
```

enum popart::**ErrorSource**
Values:

enumerator popart = 0

enumerator popart_internal

enumerator poplar

enumerator poplibs

enumerator unknown

class popart::**error** : **public** runtime_error
Exception class for popart.

Subclassed by popart::internal_error, *popart::memory_allocation_err*, popart::runtime_error

Public Functions

template<typename ...**Args**>
error(**const** char *s, **const** *Args*&... *args*)
Variadic constructor for error which allows the user to use a fmt string for the message.

throw error("This is an error reason {}", 42);

template<typename ...**Args**>
error(**const** std::string &s, **const** *Args*&... *args*)

template<typename ...**Args**>
error(ErrorUid *uid*, **const** char *s, **const** *Args*&... *args*)

template<typename ...**Args**>
error(ErrorUid *uid*, **const** std::string &s, **const** *Args*&... *args*)

const std::string &**stackreport**() **const**

ErrorUid **uid**() **const**

class popart::**memory_allocation_err** : **public** *popart*::*error*
Subclassed by popart::popx::devicex_memory_allocation_err

Public Functions

memory_allocation_err(**const** std::string &*info*)

std::unique_ptr<*memory_allocation_err*> **clone**() **const** = 0

std::string **getSummaryReport**() **const** = 0

std::string **getProfilePath**() **const** = 0

2.7.5 Debug context

```
#include <popart/debugcontext.hpp>
```

class popart::**DebugContext**

Public Functions

DebugContext(SourceLocation *loc* = SourceLocation::Current())

DebugContext(**const** char **name*, SourceLocation *loc* = SourceLocation::Current())

DebugContext(std::string *name*, SourceLocation *loc* = SourceLocation::Current())

DebugContext(**const** DebugInfo &*debugInfo*, std::string *name* = "", SourceLocation *loc* = SourceLocation::Current())

DebugContext(**const** DebugNameAndId &*debugNameAndId*, std::string *name* = "", SourceLocation *loc* = SourceLocation::Current())

DebugContext(*DebugContext*&&)

~DebugContext()

std::string **getPathName**() **const**

2.7.6 Attributes

```
#include <popart/attributes.hpp>
```

class popart::**Attributes**

Wrapper around the container of `ONNX_NAMESPACE::AtrributeProtos` of a `Node`.

Provides faster and cleaner reads of values from keys (strings) than `ONNX_NAMESPACE::AttributesProto`.

Public Types

using Ints = std::vector<int64_t>

The types of attributes as defined in the ONNX spec.

using Int = int64_t

using Floats = std::vector<float>

using Float = float

using Strings = std::vector<std::string>

using String = std::string

using Graphs = std::vector<ONNX_NAMESPACE::GraphProto>

using Graph = ONNX_NAMESPACE::GraphProto

Public Functions

Attributes(**const** *NodeAttributes*&)

Attributes() = default

const std::vector<std::string> &**getNames**() **const**

onnxAttPtr **at**(**const** std::string &*name*) **const**

void **append**(std::stringstream &*ss*, std::string *prefix* = "") **const**

template<typename **T**>
void **setIfPresent**(*T*&, **const** std::string &*key*) **const**

template<typename **T**>
void **set**(*T*&, **const** std::string &*key*) **const**

bool **hasAttribute**(**const** std::string &*key*) **const**

void **takeAttribute**(**const** std::string &*key*, **const** *Attributes* &*attributes*)
Take an attribute identified by `key` from the given `Attributes` object.

template<typename **UnaryPredicate**>
Attributes **filter**(*UnaryPredicate p*) **const**
Take the set of attributes that match the given predicate.

template<typename **T**>
T **getAttribute**(**const** std::string &*key*, **const** *T* &*defaultValue*) **const**

Attributes::*Graphs* **getAllGraphAttributes**() **const**

template<typename **T**>
T **getAttribute**(**const** std::string &*key*) **const**

template<typename **T**>
void **setAttribute**(**const** std::string &*key*, *T*&)

template<>
Attributes **filter**(**const** char **key*) **const**

template<>
Attributes **filter**(std::string *key*) **const**

template<>
void **setIfPresent**(std::vector<int64_t>&, **const** std::string &*key*) **const**

template<>
void **setIfPresent**(int64_t&, **const** std::string &*key*) **const**

template<>
void **setIfPresent**(bool &*v*, **const** std::string &*key*) **const**

template<>
void **setIfPresent**(std::string&, **const** std::string &*key*) **const**

template<>
void **setIfPresent**(float&, **const** std::string &*key*) **const**

template<>
void **set**(std::vector<int64_t> &*vs*, **const** std::string &*key*) **const**

template<>
void **set**(std::vector<float> &*vs*, **const** std::string &*key*) **const**

template<>
void **set**(std::vector<std::string> &*vs*, **const** std::string &*key*) **const**

template<>
void **set**(float &*v*, **const** std::string &*key*) **const**

template<>
void **set**(int64_t &*v*, **const** std::string &*key*) **const**

template<>
Attributes::*Ints* **getAttribute**(**const** std::string &*key*, **const** *Attributes*::*Ints* &*defaultValue*) **const**

template<>
Attributes::*Int* **getAttribute**(**const** std::string &*key*, **const** *Attributes*::*Int* &*defaultValue*) **const**

template<>
Attributes::*String* **getAttribute**(**const** std::string &*key*, **const** *Attributes*::*String* &*defaultValue*) **const**

template<>
Attributes::*Float* **getAttribute**(**const** std::string &*key*, **const** *Attributes*::*Float* &*defaultValue*) **const**

template<>
Attributes::*Ints* **getAttribute**(**const** std::string &*key*) **const**

template<>
Attributes::*Int* **getAttribute**(**const** std::string &*key*) **const**

template<>
Attributes::*String* **getAttribute**(**const** std::string &*key*) **const**

template<>
Attributes::*Strings* **getAttribute**(**const** std::string &*key*) **const**

template<>
Attributes::*Float* **getAttribute**(**const** std::string &*key*) **const**

template<>
Attributes::*Floats* **getAttribute**(**const** std::string &*key*) **const**

template<>
Attributes::*Graph* **getAttribute**(**const** std::string &*key*) **const**

template<>
void **setAttribute**(**const** std::string &*key*, *Attributes*::*Ints*&)

template<>
void **setAttribute**(**const** std::string &*key*, *Attributes*::*Int*&)

template<>
void **setAttribute**(**const** std::string &*key*, *Attributes*::*String*&)

2.7.7 Void data

```
#include <popart/voiddata.hpp>
```

class popart::**ConstVoidData**
A class to point to constant data.

Public Functions

ConstVoidData() = default

ConstVoidData(**const** void **data_*, **const** *TensorInfo* &*info_*)

bool **storesData**() **const**

void **store**(std::vector<char> &&*d*, **const** *TensorInfo* &*i*)

Public Members

const void ***data** = nullptr

TensorInfo **info**

class popart::**MutableVoidData**

A class to point to non-constant data.

Public Members

void ***data** = nullptr

TensorInfo **info**

2.7.8 Input shape information

```
#include <popart/inputshapeinfo.hpp>
```

class popart::**InputShapeInfo**

Public Functions

InputShapeInfo() = default

void **add**(*TensorId*, **const** *TensorInfo*&)

const *TensorInfo* &**get**(*TensorId*) **const**

bool **has**(*TensorId*) **const**

std::vector<*TensorId*> **getAllTensorIds**() **const**

const std::map<*TensorId*, *TensorInfo*> &**getInfos**() **const**

2.7.9 Patterns

```
#include <popart/patterns.hpp>
```

class popart::**Patterns**

Public Functions

Patterns(PatternsLevel *level*)

Patterns()

Patterns(std::vector<std::string> *patterns*)

bool **isPatternEnabled**(**const** std::type_index &*t*)

bool **isPatternEnabled**(**const** std::string &*t*)

Patterns &**enablePattern**(**const** std::type_index &*t*, bool *v*)

Patterns &**enablePattern**(**const** std::string &*t*, bool *v*)

bool **isInitAccumulateEnabled**()

bool **isPreUniReplEnabled**()

bool **isPostNReplEnabled**()

bool **isSoftMaxGradDirectEnabled**()

bool **isNlllWithSoftMaxGradDirectEnabled**()

bool **isSplitGatherEnabled**()

bool **isOpToIdentityEnabled**()

bool **isUpsampleToResizeEnabled**()

bool **isSubtractArg1GradOpEnabled**()

bool **isMulArgGradOpEnabled**()

bool **isReciprocalGradOpEnabled**()

bool **isAtan2Arg0GradOpEnabled**()

bool **isAtan2Arg1GradOpEnabled**()

bool **isDivArg0GradOpEnabled**()

bool **isDivArg1GradOpEnabled**()

bool **isPowArg0GradOpEnabled**()

bool **isPowArg1GradOpEnabled**()

bool **isSinGradOpEnabled**()

bool **isCosGradOpEnabled**()

bool **isInPlaceEnabled**()

bool **isUpdateInplacePrioritiesForIpuEnabled**()

bool **isSqrtGradOpEnabled**()

bool **isExpGradOpEnabled**()

bool **isExpm1GradOpEnabled**()

bool **isLog1pGradOpEnabled**()

bool **isLogGradOpEnabled**()

bool **isNegativeOneScaleEnabled**()

bool **isMatMulOpEnabled**()

bool **isMatMulLhsGradOpEnabled**()

bool **isMatMulRhsGradOpEnabled**()

bool **isRandomNormalLikeOpPatternEnabled**()

bool **isRandomUniformLikeOpPatternEnabled**()

bool **isZerosLikeOpPatternEnabled**()

bool **isDecomposeBinaryConstScalarEnabled**()

bool **isFmodArg0GradOpEnabled**()

bool **isLambSerialisedWeightEnabled**()

bool **isTiedGatherEnabled**()

bool **isTiedGatherAccumulateEnabled**()

Patterns &**enableInitAccumulate**(bool *v*)

Patterns &**enablePreUniRepl**(bool *v*)

Patterns &**enablePostNRepl**(bool *v*)

Patterns &**enableSoftMaxGradDirect**(bool *v*)

Patterns &**enableNlllWithSoftMaxGradDirect**(bool *v*)

Patterns &**enableSplitGather**(bool *v*)

Patterns &**enableOpToIdentity**(bool *v*)

Patterns &**enableUpsampleToResize**(bool *v*)

Patterns &**enableSubtractArg1GradOp**(bool *v*)

Patterns &**enableMulArgGradOp**(bool *v*)

Patterns &**enableReciprocalGradOp**(bool *v*)

Patterns &**enableAtan2Arg0GradOp**(bool *v*)

Patterns &**enableAtan2Arg1GradOp**(bool *v*)

Patterns &**enableDivArg0GradOp**(bool *v*)

Patterns &**enableDivArg1GradOp**(bool *v*)

Patterns &**enablePowArg0GradOp**(bool *v*)

Patterns &**enablePowArg1GradOp**(bool *v*)

Patterns &**enableSinGradOp**(bool *v*)

Patterns &**enableCosGradOp**(bool *v*)

Patterns &**enableInPlace**(bool *v*)

Patterns &**enableUpdateInplacePrioritiesForIpu**(bool *v*)

Patterns &**enableSqrtGradOp**(bool *v*)

Patterns &**enableExpGradOp**(bool *v*)

Patterns &**enableExpm1GradOp**(bool *v*)

Patterns &**enableLog1pGradOp**(bool *v*)

Patterns &**enableLogGradOp**(bool *v*)

Patterns &**enableNegativeOneScale**(bool *v*)

Patterns &**enableMatMulOp**(bool *v*)

Patterns &**enableMatMulLhsGradOp**(bool *v*)

Patterns &**enableMatMulRhsGradOp**(bool *v*)

Patterns &**enableRandomNormalLikeOpPattern**(bool *v*)

Patterns &**enableRandomUniformLikeOpPattern**(bool *v*)

Patterns &**enableZerosLikeOpPattern**(bool *v*)

Patterns &**enableDecomposeBinaryConstScalar**(bool *v*)

Patterns &**enableLambSerialisedWeight**(bool *v*)

Patterns &**enableTiedGather**(bool *v*)

Patterns &**enableTiedGatherAccumulate**(bool *v*)

Patterns &**enableRuntimeAsserts**(bool *b*)

std::vector<std::unique_ptr<PreAliasPattern>> **getPreAliasList**()

bool **operator==**(**const** *Patterns* &*p*) **const**

const std::map<std::type_index, bool> &**getSettings**() **const**

bool **getInplaceEnabled**() **const**

bool **getUpdateInplacePrioritiesForIpuEnabled**() **const**

bool **getRuntimeAssertsOn**() **const**

Public Static Functions

Patterns **create**(std::vector<std::string> *patterns*)

Friends

friend friend std::ostream & operator<< (std::ostream &os, const Patterns &patterns)

2.7.10 Type definitions

namespace onnx

namespace google

namespace protobuf

namespace popart

namespace view

Typedefs

using Regions = std::vector<*Region*>

using RegMap = std::function<*Regions*(**const** *Region*&)>

using LowBounds = std::vector<int64_t>

using UppBounds = std::vector<int64_t>

Typedefs

using Shape = std::vector<int64_t>

using Rank = int

typedef std::string **TensorId**

using DnfTensorIds = std::vector<std::set<*TensorId*>>

using OpName = std::string

using OpDomain = std::string

using OpType = std::string

using OpVersion = unsigned

using OpId = int

using ReturnPeriod = int

using SubgraphIndex = int

using SubgraphPartIndex = int

using OpxGrowPartId = int

using InIndex = int

using OutIndex = int

using **ReplicatedTensorShardingIndices** = std::set<std::pair<std::set<*InIndex*>, std::set<*OutIndex*>>>

using **PipelineCycle** = int64_t

using **VGraphId** = int64_t

using **PipelineStage** = int64_t

using **ExecutionPhase** = int64_t

using **BatchSerializedPhase** = int64_t

using **StashIndex** = int64_t

using **RemoteBufferId** = int64_t

using **RemoteBufferIndex** = int64_t

using **RandomReferenceId** = int64_t

using **ConvInputs** = std::vector<*TensorId*>

using **ConvDilations** = std::vector<int64_t>

using **ConvGroup** = int64_t

using **ConvPads** = std::vector<int64_t>

using **ConvStrides** = std::vector<int64_t>

using **ConvTruncs** = std::vector<int64_t>

using **MultiConvInputs** = std::vector<*ConvInputs*>

using **MultiConvDilations** = std::vector<*ConvDilations*>

using **MultiConvGroups** = std::vector<*ConvGroup*>

using **MultiConvPads** = std::vector<*ConvPads*>

using **MultiConvStrides** = std::vector<*ConvStrides*>

using **TensorInterval** = std::pair<size_t, size_t>

using **TensorIntervalList** = std::vector<*TensorInterval*>

using **onnxAttPtr** = **const** ONNX_NAMESPACE::AttributeProto*

using **NodeAttributes** = *google*::*protobuf*::RepeatedPtrField<ONNX_NAMESPACE::AttributeProto>

using **OnnxTensors** = std::map<*TensorId*, ONNX_NAMESPACE::TensorProto>

using **Node** = ONNX_NAMESPACE::NodeProto

using **OnnxTensorPtrs** = std::map<*TensorId*, **const** ONNX_NAMESPACE::TensorProto*>

using **OpsBeforeKey** = std::map<*Op**, std::vector<*Op**>, *POpCmp*>

CHAPTER

THREE

INDEX

CHAPTER

FOUR

TRADEMARKS & COPYRIGHT

INDEX

A

AccumulateOuterFragmentSchedule (*C++ enum*), 18
AccumulateOuterFragmentSchedule::OverlapCycleOptimized (*C++ enumerator*), 18
AccumulateOuterFragmentSchedule::OverlapMemoryOptimized (*C++ enumerator*), 18
AccumulateOuterFragmentSchedule::Scheduler (*C++ enumerator*), 18
AccumulateOuterFragmentSchedule::Serial (*C++ enumerator*), 18
AccumulateOuterFragmentSettings (*C++ struct*), 18
AccumulateOuterFragmentSettings::AccumulateOuterFragmentSettings (*C++ function*), 18
AccumulateOuterFragmentSettings::excludedVirtualGraphs (*C++ member*), 18
AccumulateOuterFragmentSettings::operator= (*C++ function*), 18
AccumulateOuterFragmentSettings::schedule (*C++ member*), 18
Adam (*C++ class*), 34
Adam::~Adam (*C++ function*), 37
Adam::Adam (*C++ function*), 36, 37
Adam::beta1s (*C++ function*), 38
Adam::beta2s (*C++ function*), 38
Adam::clone (*C++ function*), 38
Adam::createOp (*C++ function*), 38
Adam::epss (*C++ function*), 39
Adam::fromDefaultMap (*C++ function*), 39
Adam::getInputIds (*C++ function*), 38
Adam::getInverseLossScalingTensorId (*C++ function*), 36
Adam::getOptimizerInputs (*C++ function*), 38
Adam::getStoredValue (*C++ function*), 38
Adam::getUnsetBeta1 (*C++ function*), 39
Adam::getUnsetBeta2 (*C++ function*), 39
Adam::getUnsetEps (*C++ function*), 39
Adam::getUnsetLearningRate (*C++ function*), 39
Adam::getUnsetLossScaling (*C++ function*), 39
Adam::getUnsetMaxWeightNorm (*C++ function*), 39
Adam::getUnsetWeightDecay (*C++ function*), 39
Adam::getWeightDecayMode (*C++ function*), 39
Adam::hash (*C++ function*), 39
Adam::hasSpecific (*C++ function*), 36
Adam::insertSpecific (*C++ function*), 38
Adam::learningRates (*C++ function*), 38
Adam::maxWeightNorms (*C++ function*), 39
Adam::resetTensorData (*C++ function*), 38
Adam::setFactorsFromOptions (*C++ function*), 39
Adam::setStep (*C++ function*), 38
Adam::setTensorData (*C++ function*), 38
Adam::type (*C++ function*), 37
Adam::type_s (*C++ function*), 37
Adam::useScaledOptimizerState (*C++ function*), 39
Adam::validReplacement (*C++ function*), 38
Adam::weightDecays (*C++ function*), 38
AdamMode (*C++ enum*), 34
AdamMode::Adam (*C++ enumerator*), 34
AdamMode::AdaMax (*C++ enumerator*), 34
AdamMode::AdamNoBias (*C++ enumerator*), 34
AdamMode::Lamb (*C++ enumerator*), 34

AdamMode::LambNoBias (*C++ enumerator*), 34
Adaptive (*C++ class*), 39
Adaptive::~Adaptive (*C++ function*), 42
Adaptive::Adaptive (*C++ function*), 41, 42
Adaptive::alphas (*C++ function*), 43
Adaptive::clone (*C++ function*), 42
Adaptive::createOp (*C++ function*), 42
Adaptive::epss (*C++ function*), 43
Adaptive::fromDefaultMap (*C++ function*), 44
Adaptive::getInputIds (*C++ function*), 42
Adaptive::getInverseLossScalingTensorId (*C++ function*), 41
Adaptive::getOptimizerInputs (*C++ function*), 42
Adaptive::getStoredValue (*C++ function*), 42
Adaptive::getUnsetAlpha (*C++ function*), 43
Adaptive::getUnsetEps (*C++ function*), 43
Adaptive::getUnsetLearningRate (*C++ function*), 43
Adaptive::getUnsetLossScaling (*C++ function*), 44
Adaptive::getUnsetMomentum (*C++ function*), 43
Adaptive::getUnsetWeightDecay (*C++ function*), 43
Adaptive::hash (*C++ function*), 43
Adaptive::hasSpecific (*C++ function*), 41
Adaptive::insertSpecific (*C++ function*), 43
Adaptive::learningRates (*C++ function*), 43
Adaptive::momentums (*C++ function*), 43
Adaptive::resetTensorData (*C++ function*), 42
Adaptive::setStep (*C++ function*), 43
Adaptive::setTensorData (*C++ function*), 42
Adaptive::type (*C++ function*), 42
Adaptive::type_s (*C++ function*), 42
Adaptive::validReplacement (*C++ function*), 42
Adaptive::weightDecays (*C++ function*), 43
AdaptiveMode (*C++ enum*), 39
AdaptiveMode::AdaDelta (*C++ enumerator*), 39
AdaptiveMode::AdaGrad (*C++ enumerator*), 39
AdaptiveMode::CenteredRMSProp (*C++ enumerator*), 39
AdaptiveMode::RMSProp (*C++ enumerator*), 39
AiGraphcoreOpset1 (*C++ class*), 55
AiGraphcoreOpset1::_ctcloss (*C++ function*), 63
AiGraphcoreOpset1::abort (*C++ function*), 65
AiGraphcoreOpset1::AiGraphcoreOpset1 (*C++ function*), 55
AiGraphcoreOpset1::atan2 (*C++ function*), 63
AiGraphcoreOpset1::bitwiseand (*C++ function*), 66
AiGraphcoreOpset1::bitwisenot (*C++ function*), 65
AiGraphcoreOpset1::bitwiseor (*C++ function*), 66
AiGraphcoreOpset1::bitwisexnor (*C++ function*), 66
AiGraphcoreOpset1::bitwisexor (*C++ function*), 66
AiGraphcoreOpset1::call (*C++ function*), 61
AiGraphcoreOpset1::copyvarupdate (*C++ function*), 55
AiGraphcoreOpset1::ctcbeamsearchdecoder (*C++ function*), 63
AiGraphcoreOpset1::ctcloss (*C++ function*), 62
AiGraphcoreOpset1::depthtospace (*C++ function*), 58
AiGraphcoreOpset1::detach (*C++ function*), 58
AiGraphcoreOpset1::dynamicadd (*C++ function*), 60
AiGraphcoreOpset1::dynamicslice (*C++ function*), 60
AiGraphcoreOpset1::dynamicupdate (*C++ function*), 60
AiGraphcoreOpset1::dynamiczero (*C++ function*), 60
AiGraphcoreOpset1::expm1 (*C++ function*), 64
AiGraphcoreOpset1::fmod (*C++ function*), 64
AiGraphcoreOpset1::gelu (*C++ function*), 58
AiGraphcoreOpset1::groupnormalization (*C++ function*), 55
AiGraphcoreOpset1::identityloss (*C++ function*), 62
AiGraphcoreOpset1::init (*C++ function*), 59
AiGraphcoreOpset1::l1loss (*C++ function*), 62
AiGraphcoreOpset1::log1p (*C++ function*), 64
AiGraphcoreOpset1::lstm (*C++ function*), 58
AiGraphcoreOpset1::multiconv (*C++ function*), 56
AiGraphcoreOpset1::nllloss (*C++ function*), 62

AiGraphcoreOpset1::nop (*C++ function*), 57
AiGraphcoreOpset1::packedDataBlock (*C++ function*), 65
AiGraphcoreOpset1::printtensor (*C++ function*), 57
AiGraphcoreOpset1::reducemedian (*C++ function*), 66
AiGraphcoreOpset1::remainder (*C++ function*), 65
AiGraphcoreOpset1::replicatedallreduce (*C++ function*), 61
AiGraphcoreOpset1::reshape (*C++ function*), 64
AiGraphcoreOpset1::reverse (*C++ function*), 65
AiGraphcoreOpset1::round (*C++ function*), 59
AiGraphcoreOpset1::scale (*C++ function*), 58
AiGraphcoreOpset1::scaledadd (*C++ function*), 58
AiGraphcoreOpset1::scatterreduce (*C++ function*), 66
AiGraphcoreOpset1::sequenceslice (*C++ function*), 61
AiGraphcoreOpset1::shapeddropout (*C++ function*), 63
AiGraphcoreOpset1::subsample (*C++ function*), 57
AiGraphcoreOpset1::swish (*C++ function*), 66
AnchorReturnType (*C++ class*), 67
AnchorReturnType::AnchorReturnType (*C++ function*), 68
AnchorReturnType::exchangeStrategy (*C++ function*), 68
AnchorReturnType::hash (*C++ function*), 68
AnchorReturnType::id (*C++ function*), 68
AnchorReturnType::rp (*C++ function*), 68
AnchorReturnType::str (*C++ function*), 68
AnchorReturnType::tileSet (*C++ function*), 68
AnchorReturnTypeId (*C++ enum*), 67
AnchorReturnTypeId::All (*C++ enumerator*), 67
AnchorReturnTypeId::EveryN (*C++ enumerator*), 67
AnchorReturnTypeId::Final (*C++ enumerator*), 67
AnchorReturnTypeId::Sum (*C++ enumerator*), 67
Attributes (*C++ class*), 94
Attributes::append (*C++ function*), 95
Attributes::at (*C++ function*), 95
Attributes::Attributes (*C++ function*), 95
Attributes::filter (*C++ function*), 95
Attributes::Float (*C++ type*), 94
Attributes::Floats (*C++ type*), 94
Attributes::getAllGraphAttributes (*C++ function*), 95
Attributes::getAttribute (*C++ function*), 95, 96
Attributes::getNames (*C++ function*), 95
Attributes::Graph (*C++ type*), 94
Attributes::Graphs (*C++ type*), 94
Attributes::hasAttribute (*C++ function*), 95
Attributes::Int (*C++ type*), 94
Attributes::Ints (*C++ type*), 94
Attributes::set (*C++ function*), 95
Attributes::setAttribute (*C++ function*), 95, 96
Attributes::setIfPresent (*C++ function*), 95
Attributes::String (*C++ type*), 94
Attributes::Strings (*C++ type*), 94
Attributes::takeAttribute (*C++ function*), 95
AutomaticLossScalingSettings (*C++ struct*), 18
AutomaticLossScalingSettings::AutomaticLossScalingSettings (*C++ function*), 19
AutomaticLossScalingSettings::binEdgeLocation (*C++ member*), 19
AutomaticLossScalingSettings::enabled (*C++ member*), 19
AutomaticLossScalingSettings::hash (*C++ function*), 19
AutomaticLossScalingSettings::operator= (*C++ function*), 19
AutomaticLossScalingSettings::thresholdUpperCountProportion (*C++ member*), 19
AutomaticLossScalingSettings::toTrackTensors (*C++ member*), 19

B

BatchSerializationBatchSchedule (*C++ enum*), 19
BatchSerializationBatchSchedule::Isomorphic (*C++ enumerator*), 19
BatchSerializationBatchSchedule::N (*C++ enumerator*), 19
BatchSerializationBatchSchedule::OverlapOnCompute (*C++ enumerator*), 19
BatchSerializationBatchSchedule::OverlapOnIo (*C++ enumerator*), 19
BatchSerializationBatchSchedule::Scheduler (*C++ enumerator*), 19
BatchSerializationMethod (*C++ enum*), 19

BatchSerializationMethod::Loop (*C++ enumerator*), 20
BatchSerializationMethod::N (*C++ enumerator*), 20
BatchSerializationMethod::UnrollDynamic (*C++ enumerator*), 19
BatchSerializationMethod::UnrollStatic (*C++ enumerator*), 20
BatchSerializationSettings (*C++ struct*), 20
BatchSerializationSettings::batchSchedule (*C++ member*), 20
BatchSerializationSettings::BatchSerializationSettings (*C++ function*), 20
BatchSerializationSettings::concatOnExecutionPhaseChange (*C++ member*), 20
BatchSerializationSettings::concatOnPipelineStageChange (*C++ member*), 20
BatchSerializationSettings::concatOnVirtualGraphChange (*C++ member*), 20
BatchSerializationSettings::factor (*C++ member*), 20
BatchSerializationSettings::method (*C++ member*), 20
BatchSerializationSettings::operator= (*C++ function*), 20
BatchSerializationSettings::transformContext (*C++ member*), 20
BatchSerializationTransformContext (*C++ enum*), 20
BatchSerializationTransformContext::Bwd (*C++ enumerator*), 20
BatchSerializationTransformContext::Fwd (*C++ enumerator*), 20
BatchSerializationTransformContext::N (*C++ enumerator*), 21
BatchSerializedPhase (*C++ type*), 101
Builder (*C++ class*), 44
Builder::~Builder (*C++ function*), 44
Builder::addInitializedInputTensor (*C++ function*), 45
Builder::addInputTensor (*C++ function*), 44, 45
Builder::addInputTensorFromParentGraph (*C++ function*), 45
Builder::addNodeAttribute (*C++ function*), 49, 50
Builder::addOutputTensor (*C++ function*), 45
Builder::addUntypedInputTensor (*C++ function*), 45
Builder::aiGraphcoreOpset1 (*C++ function*), 46
Builder::aiOnnxMlOpset1 (*C++ function*), 46
Builder::aiOnnxOpset10 (*C++ function*), 46
Builder::aiOnnxOpset11 (*C++ function*), 46
Builder::aiOnnxOpset6 (*C++ function*), 45
Builder::aiOnnxOpset7 (*C++ function*), 45
Builder::aiOnnxOpset8 (*C++ function*), 45
Builder::aiOnnxOpset9 (*C++ function*), 45
Builder::checkpointOutput (*C++ function*), 47
Builder::clearAttribute (*C++ function*), 48
Builder::create (*C++ function*), 55
Builder::createFromOnnxModel (*C++ function*), 55
Builder::createSubgraphBuilder (*C++ function*), 44
Builder::customOp (*C++ function*), 46
Builder::excludePatterns (*C++ function*), 47
Builder::executionPhase (*C++ function*), 47, 49
Builder::getAllNodeAttributeNames (*C++ function*), 52
Builder::getAttribute (*C++ function*), 48
Builder::getBoolNodeAttribute (*C++ function*), 52
Builder::getExecutionPhase (*C++ function*), 48, 53
Builder::getFloatNodeAttribute (*C++ function*), 51
Builder::getFloatVectorNodeAttribute (*C++ function*), 51
Builder::getInputTensorIds (*C++ function*), 53
Builder::getInt64NodeAttribute (*C++ function*), 51
Builder::getInt64VectorNodeAttribute (*C++ function*), 51
Builder::getModelProto (*C++ function*), 53
Builder::getNameScope (*C++ function*), 54
Builder::getOutputTensorIds (*C++ function*), 53
Builder::getParent (*C++ function*), 55
Builder::getPartialsType (*C++ function*), 48
Builder::getPipelineStage (*C++ function*), 48
Builder::getRecomputeOutputInBackwardPass (*C++ function*), 46
Builder::getStringNodeAttribute (*C++ function*), 52
Builder::getStringVectorNodeAttribute (*C++ function*), 52
Builder::getTensorDataType (*C++ function*), 54
Builder::getTensorDtypeString (*C++ function*), 54
Builder::getTensorShape (*C++ function*), 54
Builder::getTrainableTensorIds (*C++ function*), 53
Builder::getValueTensorIds (*C++ function*), 53
Builder::getVirtualGraph (*C++ function*), 48, 52, 53

Builder::hasAttribute (*C++ function*), 48
Builder::hasParent (*C++ function*), 55
Builder::hasValueInfo (*C++ function*), 54
Builder::isInitializer (*C++ function*), 54
Builder::nodeHasAttribute (*C++ function*), 50
Builder::outputTensorLocation (*C++ function*), 46
Builder::pipelineStage (*C++ function*), 47
Builder::popNameScope (*C++ function*), 54
Builder::pushNameScope (*C++ function*), 54
Builder::recomputeOutput (*C++ function*), 46
Builder::recomputeOutputInBackwardPass (*C++ function*), 46
Builder::removeNodeAttribute (*C++ function*), 52
Builder::reshape_const (*C++ function*), 46
Builder::saveInitializersExternally (*C++ function*), 53
Builder::saveModelProto (*C++ function*), 53
Builder::setAttribute (*C++ function*), 48
Builder::setAvailableMemoryProportion (*C++ function*), 48
Builder::setEnableConvDithering (*C++ function*), 48
Builder::setGraphName (*C++ function*), 55
Builder::setInplacePreferences (*C++ function*), 48
Builder::setParent (*C++ function*), 55
Builder::setPartialsType (*C++ function*), 47
Builder::setSerializeMatMul (*C++ function*), 47
Builder::virtualGraph (*C++ function*), 47, 48

C

ClipNormSettings (*C++ class*), 28
ClipNormSettings::clipAllWeights (*C++ function*), 29
ClipNormSettings::ClipNormSettings (*C++ function*), 28
ClipNormSettings::clipWeights (*C++ function*), 29
ClipNormSettings::getMaxNorm (*C++ function*), 28
ClipNormSettings::getMode (*C++ function*), 28
ClipNormSettings::getWeightIds (*C++ function*), 28
ClipNormSettings::maxNorm (*C++ member*), 29
ClipNormSettings::Mode (*C++ enum*), 28
ClipNormSettings::Mode::ClipAllWeights (*C++ enumerator*), 28
ClipNormSettings::Mode::ClipSpecifiedWeights (*C++ enumerator*), 28
ClipNormSettings::operator!= (*C++ function*), 29
ClipNormSettings::operator== (*C++ function*), 28
ClipNormSettings::weightIds (*C++ member*), 29
ConstSGD (*C++ class*), 33
ConstSGD::ConstSGD (*C++ function*), 34
ConstVoidData (*C++ class*), 96
ConstVoidData::ConstVoidData (*C++ function*), 96
ConstVoidData::data (*C++ member*), 97
ConstVoidData::info (*C++ member*), 97
ConstVoidData::store (*C++ function*), 96
ConstVoidData::storesData (*C++ function*), 96
ConvDilations (*C++ type*), 101
ConvGroup (*C++ type*), 101
ConvInputs (*C++ type*), 101
ConvPads (*C++ type*), 101
ConvStrides (*C++ type*), 101
ConvTruncs (*C++ type*), 101

D

DataFlow (*C++ class*), 68
DataFlow::anchors (*C++ function*), 69
DataFlow::art (*C++ function*), 69
DataFlow::batchesPerStep (*C++ function*), 69
DataFlow::DataFlow (*C++ function*), 69
DataFlow::getAnchorReturnTypeMap (*C++ function*), 69
DataFlow::hash (*C++ function*), 69
DataFlow::isAnchored (*C++ function*), 69
DataFlow::isBatchCountingRequired (*C++ function*), 69
DataFlow::nAnchors (*C++ function*), 69

DataFlow::numOutFetchesPerRepl (*C++ function*), 69
DataFlow::operator= (*C++ function*), 69
DataFlow::rps (*C++ function*), 69
DataType (*C++ enum*), 87
DataType::BFLOAT16 (*C++ enumerator*), 87
DataType::BOOL (*C++ enumerator*), 87
DataType::COMPLEX128 (*C++ enumerator*), 88
DataType::COMPLEX64 (*C++ enumerator*), 87
DataType::DOUBLE (*C++ enumerator*), 87
DataType::FLOAT (*C++ enumerator*), 87
DataType::FLOAT16 (*C++ enumerator*), 87
DataType::INT16 (*C++ enumerator*), 87
DataType::INT32 (*C++ enumerator*), 87
DataType::INT64 (*C++ enumerator*), 87
DataType::INT8 (*C++ enumerator*), 87
DataType::STRING (*C++ enumerator*), 88
DataType::UINT16 (*C++ enumerator*), 87
DataType::UINT32 (*C++ enumerator*), 87
DataType::UINT64 (*C++ enumerator*), 87
DataType::UINT8 (*C++ enumerator*), 87
DataType::UNDEFINED (*C++ enumerator*), 88
DataTypeInfo (*C++ class*), 88
DataTypeInfo::DataTypeInfo (*C++ function*), 88
DataTypeInfo::isFixedPoint (*C++ function*), 88
DataTypeInfo::lcasename (*C++ function*), 88
DataTypeInfo::name (*C++ function*), 88
DataTypeInfo::nbytes (*C++ function*), 88
DataTypeInfo::type (*C++ function*), 88
DebugContext (*C++ class*), 94
DebugContext::~DebugContext (*C++ function*), 94
DebugContext::DebugContext (*C++ function*), 94
DebugContext::getPathName (*C++ function*), 94
DeviceConnectionType (*C++ enum*), 70
DeviceConnectionType::Always (*C++ enumerator*), 70
DeviceConnectionType::Never (*C++ enumerator*), 70
DeviceConnectionType::OnDemand (*C++ enumerator*), 70
DeviceInfo (*C++ class*), 70
DeviceInfo::~DeviceInfo (*C++ function*), 70
DeviceInfo::attach (*C++ function*), 70
DeviceInfo::canCompileOffline (*C++ function*), 71
DeviceInfo::detach (*C++ function*), 70
DeviceInfo::DeviceInfo (*C++ function*), 70
DeviceInfo::getConnectionType (*C++ function*), 70
DeviceInfo::getDriverIds (*C++ function*), 71
DeviceInfo::getId (*C++ function*), 70
DeviceInfo::getNumIpus (*C++ function*), 71
DeviceInfo::getNumWorkerContexts (*C++ function*), 71
DeviceInfo::getOnDemandAttachTimeout (*C++ function*), 71
DeviceInfo::getOptionFlags (*C++ function*), 71
DeviceInfo::getTarget (*C++ function*), 71
DeviceInfo::getTilesPerIPU (*C++ function*), 71
DeviceInfo::getType (*C++ function*), 70
DeviceInfo::getVersion (*C++ function*), 71
DeviceInfo::isAttached (*C++ function*), 70
DeviceInfo::setOnDemandAttachTimeout (*C++ function*), 71
DeviceInfo::toString (*C++ function*), 70
DeviceInfo::tryAttachUntilTimeout (*C++ function*), 71
DeviceManager (*C++ class*), 71
DeviceManager::acquireAvailableDevice (*C++ function*), 72
DeviceManager::acquireDeviceById (*C++ function*), 72
DeviceManager::createCpuDevice (*C++ function*), 72
DeviceManager::createDeviceManager (*C++ function*), 73
DeviceManager::createIpuModelDevice (*C++ function*), 72
DeviceManager::createOfflineIPUDevice (*C++ function*), 72
DeviceManager::createSimDevice (*C++ function*), 72
DeviceManager::enumerateDevices (*C++ function*), 71
DeviceManager::getDevice (*C++ function*), 71

DeviceManager::registerDeviceProvider (*C++ function*), 71
DeviceManager::setOnDemandAttachTimeout (*C++ function*), 73
DeviceProvider (*C++ class*), 73
DeviceProvider::~DeviceProvider (*C++ function*), 73
DeviceProvider::createHostDevice (*C++ function*), 73
DeviceProvider::enumerate (*C++ function*), 73
DeviceProvider::getDevice (*C++ function*), 73
DeviceType (*C++ enum*), 70
DeviceType::Cpu (*C++ enumerator*), 70
DeviceType::Ipu (*C++ enumerator*), 70
DeviceType::IpuModel (*C++ enumerator*), 70
DeviceType::OfflineIpu (*C++ enumerator*), 70
DeviceType::Sim (*C++ enumerator*), 70
DnfTensorIds (*C++ type*), 100
DotCheck (*C++ enum*), 21
DotCheck::All (*C++ enumerator*), 21
DotCheck::Bwd0 (*C++ enumerator*), 21
DotCheck::Final (*C++ enumerator*), 21
DotCheck::Fwd0 (*C++ enumerator*), 21
DotCheck::Fwd1 (*C++ enumerator*), 21
DotCheck::N (*C++ enumerator*), 21
DotCheck::PreAlias (*C++ enumerator*), 21

E

error (*C++ class*), 93
error::error (*C++ function*), 93
error::stackreport (*C++ function*), 93
error::uid (*C++ function*), 93
ErrorSource (*C++ enum*), 93
ErrorSource::popart (*C++ enumerator*), 93
ErrorSource::popart_internal (*C++ enumerator*), 93
ErrorSource::poplar (*C++ enumerator*), 93
ErrorSource::poplibs (*C++ enumerator*), 93
ErrorSource::unknown (*C++ enumerator*), 93
ExecutionPhase (*C++ type*), 101
ExecutionPhaseIOSchedule (*C++ enum*), 21
ExecutionPhaseIOSchedule::N (*C++ enumerator*), 21
ExecutionPhaseIOSchedule::OnDemand (*C++ enumerator*), 21
ExecutionPhaseIOSchedule::Preload (*C++ enumerator*), 21
ExecutionPhaseSchedule (*C++ enum*), 22
ExecutionPhaseSchedule::Batch (*C++ enumerator*), 22
ExecutionPhaseSchedule::BatchClusteredIO (*C++ enumerator*), 22
ExecutionPhaseSchedule::Interleaving (*C++ enumerator*), 22
ExecutionPhaseSchedule::N (*C++ enumerator*), 22
ExecutionPhaseSettings (*C++ struct*), 21
ExecutionPhaseSettings::accumulatorIOSchedule (*C++ member*), 22
ExecutionPhaseSettings::activationIOSchedule (*C++ member*), 22
ExecutionPhaseSettings::ExecutionPhaseSettings (*C++ function*), 21
ExecutionPhaseSettings::operator= (*C++ function*), 21
ExecutionPhaseSettings::optimizerStateIOSchedule (*C++ member*), 22
ExecutionPhaseSettings::phases (*C++ member*), 22
ExecutionPhaseSettings::schedule (*C++ member*), 22
ExecutionPhaseSettings::stages (*C++ member*), 22
ExecutionPhaseSettings::weightIOSchedule (*C++ member*), 22

G

google (*C++ type*), 100
google::protobuf (*C++ type*), 100

I

InferenceSession (*C++ class*), 6
InferenceSession::~InferenceSession (*C++ function*), 7
InferenceSession::createFromIr (*C++ function*), 7
InferenceSession::createFromOnnxModel (*C++ function*), 7
InIndex (*C++ type*), 100
InputShapeInfo (*C++ class*), 97

InputShapeInfo::add (*C++ function*), 97
InputShapeInfo::get (*C++ function*), 97
InputShapeInfo::getAllTensorIds (*C++ function*), 97
InputShapeInfo::getInfos (*C++ function*), 97
InputShapeInfo::has (*C++ function*), 97
InputShapeInfo::InputShapeInfo (*C++ function*), 97
Instrumentation (*C++ enum*), 22
Instrumentation::Inner (*C++ enumerator*), 22
Instrumentation::N (*C++ enumerator*), 22
Instrumentation::Outer (*C++ enumerator*), 22
IrSerializationFormat (*C++ enum*), 23
IrSerializationFormat::JSON (*C++ enumerator*), 23
IStepIO (*C++ class*), 7
IStepIO::~IStepIO (*C++ function*), 8
IStepIO::assertNumElements (*C++ function*), 9
IStepIO::enableRuntimeAsserts (*C++ function*), 9
IStepIO::in (*C++ function*), 8
IStepIO::inComplete (*C++ function*), 8
IStepIO::out (*C++ function*), 8
IStepIO::outComplete (*C++ function*), 9
IStepIO::runtimeAssertsEnabled (*C++ function*), 9

M

memory_allocation_err (*C++ class*), 93
memory_allocation_err::clone (*C++ function*), 93
memory_allocation_err::getProfilePath (*C++ function*), 93
memory_allocation_err::getSummaryReport (*C++ function*), 93
memory_allocation_err::memory_allocation_err (*C++ function*), 93
MergeVarUpdateType (*C++ enum*), 23
MergeVarUpdateType::All (*C++ enumerator*), 23
MergeVarUpdateType::AutoLoose (*C++ enumerator*), 23
MergeVarUpdateType::AutoTight (*C++ enumerator*), 23
MergeVarUpdateType::N (*C++ enumerator*), 23
MergeVarUpdateType::None (*C++ enumerator*), 23
MultiConvDilations (*C++ type*), 101
MultiConvGroups (*C++ type*), 101
MultiConvInputs (*C++ type*), 101
MultiConvPads (*C++ type*), 101
MultiConvStrides (*C++ type*), 101
MutableVoidData (*C++ class*), 97
MutableVoidData::data (*C++ member*), 97
MutableVoidData::info (*C++ member*), 97

N

Node (*C++ type*), 101
NodeAttributes (*C++ type*), 101

O

onnx (*C++ type*), 100
onnxAttPtr (*C++ type*), 101
OnnxTensorPtrs (*C++ type*), 101
OnnxTensors (*C++ type*), 101
Op (*C++ class*), 75
Op::~Op (*C++ function*), 77
Op::adjustInSettings (*C++ function*), 76
Op::adjustOutSettings (*C++ function*), 76
Op::aliases (*C++ function*), 78
Op::append (*C++ function*), 77
Op::appendAttributes (*C++ function*), 80
Op::appendMore (*C++ function*), 80
Op::appendOutlineAttributes (*C++ function*), 80
Op::bwdRegMap (*C++ function*), 78
Op::calcAutoVirtualGraphCost (*C++ function*), 80
Op::canBeReplacedByIdentity (*C++ function*), 77
Op::canShard (*C++ function*), 81
Op::clone (*C++ function*), 79

Op::configureForReplicatedTensorSharding (*C++ function*), 81
Op::configureShardedOp (*C++ function*), 81
Op::connectInTensor (*C++ function*), 77
Op::connectOutTensor (*C++ function*), 77
Op::consumesGraphOutput (*C++ function*), 80
Op::copiesOptimizerTensors (*C++ function*), 79
Op::createAndConnectOutTensor (*C++ function*), 77
Op::debugInfo (*C++ member*), 82
Op::debugName (*C++ function*), 77
Op::defaultConnectInTensor (*C++ function*), 77
Op::disconnectAllInputs (*C++ function*), 77
Op::disconnectAllOutputs (*C++ function*), 77
Op::disconnectInTensor (*C++ function*), 77
Op::disconnectOutTensor (*C++ function*), 77
Op::doesAlias (*C++ function*), 78
Op::finalizeDebugInfo (*C++ function*), 78
Op::fwdRegMap (*C++ function*), 78
Op::getBatchSerializedPhase (*C++ function*), 76
Op::getCalledGraphIds (*C++ function*), 80
Op::getCalledGraphIndex (*C++ function*), 80
Op::getCalledGraphs (*C++ function*), 80
Op::getDebugInfo (*C++ function*), 77
Op::getExecutionPhase (*C++ function*), 76
Op::getFollowingOp (*C++ function*), 81
Op::getFollowingOps (*C++ function*), 81
Op::getGradOps (*C++ function*), 78
Op::getGraph (*C++ function*), 77
Op::getHighSubgraphValue (*C++ function*), 80
Op::getInBatchAxis (*C++ function*), 76
Op::getInplaceVariant (*C++ function*), 78
Op::getInSettings (*C++ function*), 76
Op::getIntrospectionInVirtualGraphId (*C++ function*), 76
Op::getIntrospectionOutVirtualGraphId (*C++ function*), 76
Op::getIr (*C++ function*), 77
Op::getLowSubgraphValue (*C++ function*), 80
Op::getName (*C++ function*), 77
Op::getNonGradInIndex (*C++ function*), 79
Op::getOptionalBatchSerializedPhase (*C++ function*), 76
Op::getOptionalExecutionPhase (*C++ function*), 76
Op::getOptionalPipelineStage (*C++ function*), 76
Op::getOptionalVGraphId (*C++ function*), 76
Op::getOutBatchAxis (*C++ function*), 76
Op::getOutSettings (*C++ function*), 76
Op::getPipelineStage (*C++ function*), 76
Op::getPrecedingOp (*C++ function*), 81
Op::getReplicatedTensorShardingIndices (*C++ function*), 81
Op::getScope (*C++ function*), 77
Op::getSeedInIndex (*C++ function*), 79
Op::getSettings (*C++ function*), 76
Op::getShardReductionType (*C++ function*), 81
Op::getShardRescaleFactor (*C++ function*), 81
Op::getSubgraphEquivId (*C++ function*), 80
Op::getSubgraphInputs (*C++ function*), 80
Op::getSubgraphOutputs (*C++ function*), 80
Op::getSubgraphValue (*C++ function*), 80
Op::getVirtualGraphId (*C++ function*), 76
Op::gradInputInfo (*C++ function*), 79
Op::gradOutToNonGradIn (*C++ function*), 79
Op::growAliasModel (*C++ function*), 78
Op::hasAliasedModifiers (*C++ function*), 81
Op::hasBatchSerializedPhase (*C++ function*), 76
Op::hasExecutionPhase (*C++ function*), 76
Op::hasInput (*C++ function*), 79
Op::hasOutput (*C++ function*), 79
Op::hasPipelineStage (*C++ function*), 76
Op::hasSideEffect (*C++ function*), 80
Op::hasVirtualGraphId (*C++ function*), 76

Op::id (*C++ member*), 82
Op::inheritPlacementAttributes (*C++ function*), 76
Op::inId (*C++ function*), 80
Op::inInfo (*C++ function*), 80
Op::inplacePriorityDefault (*C++ function*), 78
Op::input (*C++ member*), 82
Op::inputsUnmodifiable (*C++ function*), 80
Op::inputUnmodifiable (*C++ function*), 81
Op::inRank (*C++ function*), 80
Op::inShape (*C++ function*), 80
Op::inTensor (*C++ function*), 79
Op::inTensorCount (*C++ function*), 80
Op::isChildOf (*C++ function*), 81
Op::isConvertibleTo (*C++ function*), 79
Op::isElementWiseUnary (*C++ function*), 77
Op::isExcludedFromPattern (*C++ function*), 76
Op::isInplaceViewChange (*C++ function*), 79
Op::isIpuCopyOp (*C++ function*), 79
Op::isLossOp (*C++ function*), 79
Op::isNorm (*C++ function*), 77
Op::isOptimizerOp (*C++ function*), 79
Op::isOutlineable (*C++ function*), 80
Op::isOutplace (*C++ function*), 78
Op::isOutplaceViewChange (*C++ function*), 79
Op::isParentOf (*C++ function*), 81
Op::mapInplaceProposal (*C++ function*), 78
Op::memOfOutputs (*C++ function*), 77
Op::modifies (*C++ function*), 78
Op::modifiesIndex (*C++ function*), 79
Op::name (*C++ function*), 78
Op::Op (*C++ function*), 77
Op::operator= (*C++ function*), 77
Op::opid (*C++ member*), 82
Op::opInToSubgraphInIndex (*C++ function*), 80
Op::opOutToSubgraphOutIndex (*C++ function*), 80
Op::optionalInputs (*C++ function*), 77
Op::outId (*C++ function*), 80
Op::outIndex (*C++ function*), 80
Op::outInfo (*C++ function*), 80
Op::output (*C++ member*), 82
Op::outRank (*C++ function*), 80
Op::outShape (*C++ function*), 80
Op::outTensor (*C++ function*), 79, 80
Op::outTensorCount (*C++ function*), 80
Op::overwritesTensor (*C++ function*), 79
Op::prettyNpOut (*C++ function*), 80
Op::producesGraphOutput (*C++ function*), 81
Op::pruneable (*C++ member*), 82
Op::requiresRandomSeed (*C++ function*), 79
Op::setBatchSerializedPhase (*C++ function*), 76
Op::setCalledSubgraphGradInfo (*C++ function*), 78
Op::setExecutionPhase (*C++ function*), 76
Op::setName (*C++ function*), 77
Op::setPipelineStage (*C++ function*), 76
Op::setScope (*C++ function*), 77
Op::settings (*C++ member*), 82
Op::Settings (*C++ struct*), 82
Op::Settings::~Settings (*C++ function*), 82
Op::Settings::batchSerializedPhase (*C++ member*), 82
Op::Settings::copy (*C++ function*), 82
Op::Settings::debugInfoId (*C++ member*), 82
Op::Settings::excludePatterns (*C++ member*), 82
Op::Settings::executionContext (*C++ member*), 82
Op::Settings::executionPhase (*C++ member*), 82
Op::Settings::extraOutlineAttributes (*C++ member*), 82
Op::Settings::getIr (*C++ function*), 82
Op::Settings::graph (*C++ member*), 82

Op::Settings::inferTensorMappingToFrom (*C++ member*), 82
Op::Settings::inplacePriorityVeto (*C++ member*), 82
Op::Settings::name (*C++ member*), 82
Op::Settings::optimizerOp (*C++ member*), 82
Op::Settings::pipelineStage (*C++ member*), 82
Op::Settings::recomputeType (*C++ member*), 82
Op::Settings::schedulePriority (*C++ member*), 82
Op::Settings::scope (*C++ member*), 82
Op::Settings::setFromAttributes (*C++ function*), 82
Op::Settings::Settings (*C++ function*), 82
Op::Settings::tensorLocation (*C++ member*), 82
Op::Settings::tileSet (*C++ member*), 82
Op::Settings::vgraphId (*C++ member*), 82
Op::setup (*C++ function*), 78
Op::setVirtualGraphId (*C++ function*), 76
Op::shard (*C++ function*), 81
Op::str (*C++ function*), 77
Op::SubgraphInSig (*C++ type*), 76
Op::subgraphInToOpInIndex (*C++ function*), 80
Op::subgraphOutToOpOutIndex (*C++ function*), 80
Op::toJSON (*C++ function*), 77
Op::transferBaseProperties (*C++ function*), 81
Op::uses (*C++ function*), 78
OpCreatorInfo (*C++ class*), 84
OpCreatorInfo::attributes (*C++ member*), 85
OpCreatorInfo::getInputData (*C++ function*), 84
OpCreatorInfo::getInputIds (*C++ function*), 84
OpCreatorInfo::getInputScalarValue (*C++ function*), 84
OpCreatorInfo::getInputTensor (*C++ function*), 84
OpCreatorInfo::getInputTensorData (*C++ function*), 84
OpCreatorInfo::getInputTensorInfo (*C++ function*), 84
OpCreatorInfo::getOutputIds (*C++ function*), 84
OpCreatorInfo::hasInputIds (*C++ function*), 84
OpCreatorInfo::hasInputTensor (*C++ function*), 84
OpCreatorInfo::hasOutputIds (*C++ function*), 84
OpCreatorInfo::OpCreatorInfo (*C++ function*), 84
OpCreatorInfo::opid (*C++ member*), 85
OpCreatorInfo::settings (*C++ member*), 85
OpDefinition (*C++ class*), 83
OpDefinition::Attribute (*C++ struct*), 83
OpDefinition::Attribute::Attribute (*C++ function*), 83
OpDefinition::Attribute::supportedValuesRegex (*C++ member*), 83
OpDefinition::attributes (*C++ member*), 83
OpDefinition::Attributes (*C++ type*), 83
OpDefinition::DataTypes (*C++ type*), 83
OpDefinition::Input (*C++ struct*), 83
OpDefinition::Input::constant (*C++ member*), 84
OpDefinition::Input::Input (*C++ function*), 84
OpDefinition::Input::name (*C++ member*), 84
OpDefinition::Input::supportedTensors (*C++ member*), 84
OpDefinition::inputs (*C++ member*), 83
OpDefinition::Inputs (*C++ type*), 83
OpDefinition::OpDefinition (*C++ function*), 83
OpDefinition::Output (*C++ struct*), 84
OpDefinition::Output::name (*C++ member*), 84
OpDefinition::Output::Output (*C++ function*), 84
OpDefinition::Output::supportedTensors (*C++ member*), 84
OpDefinition::outputs (*C++ member*), 83
OpDefinition::Outputs (*C++ type*), 83
OpDomain (*C++ type*), 100
OpId (*C++ type*), 100
OpManager (*C++ class*), 85
OpManager::ComplexOpFactoryFunc (*C++ type*), 85
OpManager::createOp (*C++ function*), 85
OpManager::createOpInGraph (*C++ function*), 85
OpManager::createOpWithInputs (*C++ function*), 85
OpManager::getAttributesFromAnyMap (*C++ function*), 85

OpManager::getOpVersionFromOpSet (*C++ function*), 85
OpManager::getSupportedOperations (*C++ function*), 85
OpManager::getSupportedOperationsDefinition (*C++ function*), 85
OpManager::getUnsupportedOperations (*C++ function*), 85
OpManager::OpFactoryFunc (*C++ type*), 85
OpManager::OpInfo (*C++ class*), 85
OpManager::OpInfo::details (*C++ member*), 86
OpManager::OpInfo::getComplexFactory (*C++ function*), 85
OpManager::OpInfo::getSimpleFactory (*C++ function*), 85
OpManager::OpInfo::hasComplexFactory (*C++ function*), 85
OpManager::OpInfo::id (*C++ member*), 86
OpManager::OpInfo::isPublic (*C++ member*), 86
OpManager::OpInfo::OpInfo (*C++ function*), 85
OpManager::OpManager (*C++ function*), 85
OpManager::registerOp (*C++ function*), 85
OpName (*C++ type*), 100
OpsBeforeKey (*C++ type*), 101
Optimizer (*C++ class*), 25
Optimizer::~Optimizer (*C++ function*), 25
Optimizer::clone (*C++ function*), 26
Optimizer::createOp (*C++ function*), 26
Optimizer::getAccumulationFactor (*C++ function*), 26
Optimizer::getClipNormSettings (*C++ function*), 27
Optimizer::getFinalLossScalingVal (*C++ function*), 26
Optimizer::getInputIds (*C++ function*), 26
Optimizer::getInverseLossScalingTensorId (*C++ function*), 26
Optimizer::getLossScalingTensorId (*C++ function*), 27
Optimizer::getLossScalingVal (*C++ function*), 26
Optimizer::getOptimizerInputs (*C++ function*), 26
Optimizer::getReplicatedGraphCount (*C++ function*), 26
Optimizer::gradientAccumulationEnabled (*C++ function*), 26
Optimizer::hash (*C++ function*), 27
Optimizer::hasSpecific (*C++ function*), 27
Optimizer::lossMeanReplicationEnabled (*C++ function*), 26
Optimizer::lossScaling (*C++ function*), 26
Optimizer::meanGradientAccumulationEnabled (*C++ function*), 26
Optimizer::meanReductionEnabled (*C++ function*), 26
Optimizer::Optimizer (*C++ function*), 26
Optimizer::postMeanAccumulationEnabled (*C++ function*), 26
Optimizer::postMeanReplicationEnabled (*C++ function*), 26
Optimizer::resetTensorData (*C++ function*), 26
Optimizer::setFactorsFromOptions (*C++ function*), 26
Optimizer::setTensorData (*C++ function*), 26
Optimizer::type (*C++ function*), 26
Optimizer::type_s (*C++ function*), 26
Optimizer::validReplacement (*C++ function*), 26
OptimizerReductionType (*C++ enum*), 27
OptimizerReductionType::AcclReduce (*C++ enumerator*), 27
OptimizerReductionType::AccumReduce (*C++ enumerator*), 27
OptimizerReductionType::GradReduce (*C++ enumerator*), 27
OptimizerReductionType::None (*C++ enumerator*), 27
OptimizerType (*C++ enum*), 27
OptimizerType::Adam (*C++ enumerator*), 27
OptimizerType::Adaptive (*C++ enumerator*), 27
OptimizerType::NTYPES (*C++ enumerator*), 27
OptimizerType::SGD (*C++ enumerator*), 27
OptimizerValue (*C++ class*), 27
OptimizerValue::isConst (*C++ function*), 28
OptimizerValue::operator== (*C++ function*), 28
OptimizerValue::OptimizerValue (*C++ function*), 28
OptimizerValue::val (*C++ function*), 28
OptimizerValue::validReplacement (*C++ function*), 28
OpType (*C++ type*), 100
OpVersion (*C++ type*), 100
OpxGrowPartId (*C++ type*), 100
OutIndex (*C++ type*), 100

P

Patterns (*C++ class*), 97
Patterns::create (*C++ function*), 100
Patterns::enableAtan2Arg0GradOp (*C++ function*), 99
Patterns::enableAtan2Arg1GradOp (*C++ function*), 99
Patterns::enableCosGradOp (*C++ function*), 99
Patterns::enableDecomposeBinaryConstScalar (*C++ function*), 99
Patterns::enableDivArg0GradOp (*C++ function*), 99
Patterns::enableDivArg1GradOp (*C++ function*), 99
Patterns::enableExpGradOp (*C++ function*), 99
Patterns::enableExpm1GradOp (*C++ function*), 99
Patterns::enableInitAccumulate (*C++ function*), 98
Patterns::enableInPlace (*C++ function*), 99
Patterns::enableLambSerialisedWeight (*C++ function*), 99
Patterns::enableLog1pGradOp (*C++ function*), 99
Patterns::enableLogGradOp (*C++ function*), 99
Patterns::enableMatMulLhsGradOp (*C++ function*), 99
Patterns::enableMatMulOp (*C++ function*), 99
Patterns::enableMatMulRhsGradOp (*C++ function*), 99
Patterns::enableMulArgGradOp (*C++ function*), 99
Patterns::enableNegativeOneScale (*C++ function*), 99
Patterns::enableNlllWithSoftMaxGradDirect (*C++ function*), 98
Patterns::enableOpToIdentity (*C++ function*), 99
Patterns::enablePattern (*C++ function*), 97
Patterns::enablePostNRepl (*C++ function*), 98
Patterns::enablePowArg0GradOp (*C++ function*), 99
Patterns::enablePowArg1GradOp (*C++ function*), 99
Patterns::enablePreUniRepl (*C++ function*), 98
Patterns::enableRandomNormalLikeOpPattern (*C++ function*), 99
Patterns::enableRandomUniformLikeOpPattern (*C++ function*), 99
Patterns::enableReciprocalGradOp (*C++ function*), 99
Patterns::enableRuntimeAsserts (*C++ function*), 99
Patterns::enableSinGradOp (*C++ function*), 99
Patterns::enableSoftMaxGradDirect (*C++ function*), 98
Patterns::enableSplitGather (*C++ function*), 99
Patterns::enableSqrtGradOp (*C++ function*), 99
Patterns::enableSubtractArg1GradOp (*C++ function*), 99
Patterns::enableTiedGather (*C++ function*), 99
Patterns::enableTiedGatherAccumulate (*C++ function*), 99
Patterns::enableUpdateInplacePrioritiesForIpu (*C++ function*), 99
Patterns::enableUpsampleToResize (*C++ function*), 99
Patterns::enableZerosLikeOpPattern (*C++ function*), 99
Patterns::getInplaceEnabled (*C++ function*), 99
Patterns::getPreAliasList (*C++ function*), 99
Patterns::getRuntimeAssertsOn (*C++ function*), 99
Patterns::getSettings (*C++ function*), 99
Patterns::getUpdateInplacePrioritiesForIpuEnabled (*C++ function*), 99
Patterns::isAtan2Arg0GradOpEnabled (*C++ function*), 98
Patterns::isAtan2Arg1GradOpEnabled (*C++ function*), 98
Patterns::isCosGradOpEnabled (*C++ function*), 98
Patterns::isDecomposeBinaryConstScalarEnabled (*C++ function*), 98
Patterns::isDivArg0GradOpEnabled (*C++ function*), 98
Patterns::isDivArg1GradOpEnabled (*C++ function*), 98
Patterns::isExpGradOpEnabled (*C++ function*), 98
Patterns::isExpm1GradOpEnabled (*C++ function*), 98
Patterns::isFmodArg0GradOpEnabled (*C++ function*), 98
Patterns::isInitAccumulateEnabled (*C++ function*), 97
Patterns::isInPlaceEnabled (*C++ function*), 98
Patterns::isLambSerialisedWeightEnabled (*C++ function*), 98
Patterns::isLog1pGradOpEnabled (*C++ function*), 98
Patterns::isLogGradOpEnabled (*C++ function*), 98
Patterns::isMatMulLhsGradOpEnabled (*C++ function*), 98
Patterns::isMatMulOpEnabled (*C++ function*), 98
Patterns::isMatMulRhsGradOpEnabled (*C++ function*), 98
Patterns::isMulArgGradOpEnabled (*C++ function*), 98
Patterns::isNegativeOneScaleEnabled (*C++ function*), 98
Patterns::isNlllWithSoftMaxGradDirectEnabled (*C++ function*), 98

Patterns::isOpToIdentityEnabled (*C++ function*), 98
Patterns::isPatternEnabled (*C++ function*), 97
Patterns::isPostNReplEnabled (*C++ function*), 97
Patterns::isPowArg0GradOpEnabled (*C++ function*), 98
Patterns::isPowArg1GradOpEnabled (*C++ function*), 98
Patterns::isPreUniReplEnabled (*C++ function*), 97
Patterns::isRandomNormalLikeOpPatternEnabled (*C++ function*), 98
Patterns::isRandomUniformLikeOpPatternEnabled (*C++ function*), 98
Patterns::isReciprocalGradOpEnabled (*C++ function*), 98
Patterns::isSinGradOpEnabled (*C++ function*), 98
Patterns::isSoftMaxGradDirectEnabled (*C++ function*), 97
Patterns::isSplitGatherEnabled (*C++ function*), 98
Patterns::isSqrtGradOpEnabled (*C++ function*), 98
Patterns::isSubtractArg1GradOpEnabled (*C++ function*), 98
Patterns::isTiedGatherAccumulateEnabled (*C++ function*), 98
Patterns::isTiedGatherEnabled (*C++ function*), 98
Patterns::isUpdateInplacePrioritiesForIpuEnabled (*C++ function*), 98
Patterns::isUpsampleToResizeEnabled (*C++ function*), 98
Patterns::isZerosLikeOpPatternEnabled (*C++ function*), 98
Patterns::operator== (*C++ function*), 99
Patterns::Patterns (*C++ function*), 97
PipelineCycle (*C++ type*), 101
PipelineStage (*C++ type*), 101
popart (*C++ type*), 100
POpCmp (*C++ struct*), 82
POpCmp::operator() (*C++ function*), 83
popx::Devicex (*C++ class*), 73
popx::Devicex::~Devicex (*C++ function*), 74
popx::Devicex::connectRandomSeedStream (*C++ function*), 74
popx::Devicex::connectRngStateStream (*C++ function*), 74
popx::Devicex::connectStream (*C++ function*), 74
popx::Devicex::connectStreamToCallback (*C++ function*), 74
popx::Devicex::convCache (*C++ member*), 75
popx::Devicex::copyFromRemoteBuffer (*C++ function*), 74
popx::Devicex::copyToRemoteBuffer (*C++ function*), 74
popx::Devicex::cycleCountTensorToHost (*C++ function*), 74
popx::Devicex::Devicex (*C++ function*), 74
popx::Devicex::getAccumulationFactor (*C++ function*), 74
popx::Devicex::getDeviceInfo (*C++ function*), 75
popx::Devicex::getEfficientlyCreatedInputTensors (*C++ function*), 75
popx::Devicex::getGlobalReplicaOffset (*C++ function*), 74
popx::Devicex::getGlobalReplicationFactor (*C++ function*), 74
popx::Devicex::getLinearlyCreatedInputTensors (*C++ function*), 75
popx::Devicex::getReplicationFactor (*C++ function*), 74
popx::Devicex::getReport (*C++ function*), 74
popx::Devicex::getRngStateToHost (*C++ function*), 74
popx::Devicex::getSerializedGraph (*C++ function*), 74
popx::Devicex::getSummaryReport (*C++ function*), 74
popx::Devicex::ir (*C++ function*), 74
popx::Devicex::isEngineLoaded (*C++ function*), 74
popx::Devicex::isReplicatedGraph (*C++ function*), 74
popx::Devicex::loadEngineAndConnectStreams (*C++ function*), 75
popx::Devicex::lowering (*C++ function*), 74
popx::Devicex::matmulCache (*C++ member*), 75
popx::Devicex::optimizerFromHost (*C++ function*), 74
popx::Devicex::prepare (*C++ function*), 74
popx::Devicex::prepareHasBeenCalled (*C++ function*), 75
popx::Devicex::prePlanConvolutions (*C++ member*), 75
popx::Devicex::prePlanMatMuls (*C++ member*), 75
popx::Devicex::readWeights (*C++ function*), 74
popx::Devicex::remoteBufferWeightsFromHost (*C++ function*), 74
popx::Devicex::remoteBufferWeightsToHost (*C++ function*), 74
popx::Devicex::run (*C++ function*), 74
popx::Devicex::setEngineIsLoaded (*C++ function*), 74
popx::Devicex::setRandomSeedFromHost (*C++ function*), 74
popx::Devicex::setRngStateFromHost (*C++ function*), 74
popx::Devicex::setRngStateValue (*C++ function*), 74

popx::Devicex::weightsFromHost (*C++ function*), 74
popx::Devicex::weightsToHost (*C++ function*), 74
popx::Devicex::writeWeights (*C++ function*), 74
popx::Opx (*C++ class*), 86
popx::Opx::~Opx (*C++ function*), 86
popx::Opx::broadcast (*C++ function*), 87
popx::Opx::cloneNcopy (*C++ function*), 86
popx::Opx::createInput (*C++ function*), 86
popx::Opx::createInputTensor (*C++ function*), 86
popx::Opx::createsEquiv (*C++ function*), 86
popx::Opx::get (*C++ function*), 87
popx::Opx::getConst (*C++ function*), 87
popx::Opx::getInTensor (*C++ function*), 87
popx::Opx::getInView (*C++ function*), 87
popx::Opx::getOutTensor (*C++ function*), 87
popx::Opx::getOutView (*C++ function*), 87
popx::Opx::getScalarVariable (*C++ function*), 87
popx::Opx::getView (*C++ function*), 87
popx::Opx::graph (*C++ function*), 87
popx::Opx::grow (*C++ function*), 87
popx::Opx::insert (*C++ function*), 87
popx::Opx::Opx (*C++ function*), 86
popx::Opx::setOutTensor (*C++ function*), 87
popx::Opx::unwindTensorLayout (*C++ function*), 86

R

RandomReferenceId (*C++ type*), 101
Rank (*C++ type*), 100
RecomputationType (*C++ enum*), 23
RecomputationType::N (*C++ enumerator*), 23
RecomputationType::None (*C++ enumerator*), 23
RecomputationType::NormOnly (*C++ enumerator*), 23
RecomputationType::Pipeline (*C++ enumerator*), 23
RecomputationType::RecomputeAll (*C++ enumerator*), 23
RecomputationType::Standard (*C++ enumerator*), 23
RemoteBufferId (*C++ type*), 101
RemoteBufferIndex (*C++ type*), 101
ReplicatedTensorSharding (*C++ enum*), 89
ReplicatedTensorSharding::N (*C++ enumerator*), 89
ReplicatedTensorSharding::Off (*C++ enumerator*), 89
ReplicatedTensorSharding::On (*C++ enumerator*), 89
ReplicatedTensorShardingIndices (*C++ type*), 100
ReturnPeriod (*C++ type*), 100

S

Session (*C++ class*), 2
Session::~Session (*C++ function*), 2
Session::compileAndExport (*C++ function*), 2
Session::connectStream (*C++ function*), 3
Session::connectStreamToCallback (*C++ function*), 3
Session::getCycleCount (*C++ function*), 3
Session::getDevice (*C++ function*), 5
Session::getExecutable (*C++ function*), 5
Session::getInfo (*C++ function*), 4
Session::getIr (*C++ function*), 5
Session::getIrLowering (*C++ function*), 5
Session::getReport (*C++ function*), 4
Session::getRNGState (*C++ function*), 2
Session::getSerializedGraph (*C++ function*), 4
Session::getSummaryReport (*C++ function*), 4
Session::hasInfo (*C++ function*), 4
Session::loadEngineAndConnectStreams (*C++ function*), 3
Session::loadExecutableFromFile (*C++ function*), 2
Session::loadExecutableFromStream (*C++ function*), 3
Session::modelToHost (*C++ function*), 4
Session::prepareDevice (*C++ function*), 3

Session::readWeights (*C++ function*), 5
Session::resetHostWeights (*C++ function*), 4
Session::run (*C++ function*), 3
Session::serializeIr (*C++ function*), 5
Session::setRandomSeed (*C++ function*), 2
Session::setRNGState (*C++ function*), 2
Session::updateExternallySavedTensorLocations (*C++ function*), 3
Session::weightsFromHost (*C++ function*), 3
Session::weightsToHost (*C++ function*), 3
Session::writeWeights (*C++ function*), 5
SessionOptions (*C++ struct*), 11
SessionOptions::accumulateOuterFragmentSettings (*C++ member*), 14
SessionOptions::accumulationAndReplicationReductionType (*C++ member*), 13
SessionOptions::accumulationFactor (*C++ member*), 13
SessionOptions::accumulatorTensorLocationSettings (*C++ member*), 17
SessionOptions::activationTensorLocationSettings (*C++ member*), 16
SessionOptions::aliasZeroCopy (*C++ member*), 14
SessionOptions::autodiffSettings (*C++ member*), 14
SessionOptions::automaticLossScalingSettings (*C++ member*), 17
SessionOptions::autoRecomputation (*C++ member*), 12
SessionOptions::autoRecomputationEnabled (*C++ function*), 11
SessionOptions::batchSerializationSettings (*C++ member*), 14
SessionOptions::cachePath (*C++ member*), 14
SessionOptions::compilationProgressLogger (*C++ member*), 17
SessionOptions::compilationProgressTotal (*C++ member*), 17
SessionOptions::compileEngine (*C++ member*), 14
SessionOptions::constantWeights (*C++ member*), 14
SessionOptions::convolutionOptions (*C++ member*), 15
SessionOptions::customCodeletCompileFlags (*C++ member*), 15
SessionOptions::customCodelets (*C++ member*), 15
SessionOptions::decomposeGradSum (*C++ member*), 16
SessionOptions::defaultPrefetchBufferingDepth (*C++ member*), 13
SessionOptions::delayVarUpdates (*C++ member*), 14
SessionOptions::developerSettings (*C++ member*), 17
SessionOptions::disableGradAccumulationTensorStreams (*C++ member*), 14
SessionOptions::dotChecks (*C++ member*), 11
SessionOptions::dotOpNames (*C++ member*), 11
SessionOptions::enableDistributedReplicatedGraphs (*C++ member*), 16
SessionOptions::enableEngineCaching (*C++ member*), 14
SessionOptions::enableExplicitMainLoops (*C++ member*), 17
SessionOptions::enableFloatingPointChecks (*C++ member*), 14
SessionOptions::enableFullyConnectedPass (*C++ member*), 15
SessionOptions::enableGradientAccumulation (*C++ member*), 13
SessionOptions::enableLoadAndOffloadRNGState (*C++ member*), 16
SessionOptions::enableMergeExchange (*C++ member*), 17
SessionOptions::enableNonStableSoftmax (*C++ member*), 13
SessionOptions::enableOutlining (*C++ member*), 12
SessionOptions::enableOutliningCopyCostPruning (*C++ member*), 12
SessionOptions::enablePipelining (*C++ member*), 13
SessionOptions::enablePrefetchDatastreams (*C++ member*), 13
SessionOptions::enableReplicatedGraphs (*C++ member*), 13
SessionOptions::enableSerializedMatmuls (*C++ member*), 15
SessionOptions::enableStableNorm (*C++ member*), 15
SessionOptions::enableStochasticRounding (*C++ member*), 14
SessionOptions::enableSupportedDataTypeCasting (*C++ member*), 17
SessionOptions::engineOptions (*C++ member*), 15
SessionOptions::executionPhaseSettings (*C++ member*), 14
SessionOptions::explicitPipeliningEnabled (*C++ function*), 11
SessionOptions::explicitRecomputation (*C++ member*), 14
SessionOptions::exportPoplarComputationGraph (*C++ member*), 11
SessionOptions::exportPoplarVertexGraph (*C++ member*), 11
SessionOptions::finalDotOp (*C++ member*), 11
SessionOptions::firstDotOp (*C++ member*), 11
SessionOptions::gclOptions (*C++ member*), 15
SessionOptions::getAccumulationFactor (*C++ function*), 11
SessionOptions::getGlobalReplicationFactor (*C++ function*), 11
SessionOptions::getPrefetchBufferingDepth (*C++ function*), 11

SessionOptions::globalReplicaOffset (*C++ member*), 16
SessionOptions::globalReplicationFactor (*C++ member*), 16
SessionOptions::groupHostSync (*C++ member*), 16
SessionOptions::groupNormStridedChannelGrouping (*C++ member*), 17
SessionOptions::hardwareInstrumentations (*C++ member*), 14
SessionOptions::implicitPipeliningEnabled (*C++ function*), 11
SessionOptions::instrumentWithHardwareCycleCounter (*C++ member*), 14
SessionOptions::kahnTieBreaker (*C++ member*), 16
SessionOptions::logDir (*C++ member*), 11
SessionOptions::looseThresholdAtPeak (*C++ member*), 12
SessionOptions::lstmOptions (*C++ member*), 15
SessionOptions::matmulOptions (*C++ member*), 15
SessionOptions::meanAccumulationAndReplicationReductionStrategy (*C++ member*), 13
SessionOptions::mergeVarUpdate (*C++ member*), 12
SessionOptions::mergeVarUpdateMemThreshold (*C++ member*), 12
SessionOptions::NumIOTiles (*C++ class*), 17
SessionOptions::numIOTiles (*C++ member*), 14
SessionOptions::NumIOTiles::NumIOTiles (*C++ function*), 17
SessionOptions::NumIOTiles::operator int (*C++ function*), 17
SessionOptions::NumIOTiles::operator= (*C++ function*), 17
SessionOptions::NumIOTiles::operator== (*C++ function*), 17
SessionOptions::operator= (*C++ function*), 11
SessionOptions::optimizerStateTensorLocationSettings (*C++ member*), 16
SessionOptions::opxAliasChecking (*C++ member*), 16
SessionOptions::opxModifyChecking (*C++ member*), 16
SessionOptions::outlineSequenceBreakCost (*C++ member*), 12
SessionOptions::outlineThreshold (*C++ member*), 12
SessionOptions::partialsTypeMatMuls (*C++ member*), 15
SessionOptions::prefetchBufferingDepthMap (*C++ member*), 13
SessionOptions::rearrangeAnchorsOnHost (*C++ member*), 12
SessionOptions::rearrangeStreamsOnHost (*C++ member*), 12
SessionOptions::replicatedGraphCount (*C++ member*), 13
SessionOptions::reportOptions (*C++ member*), 15
SessionOptions::scheduleNonWeightUpdateGradientConsumersEarly (*C++ member*), 15
SessionOptions::separateCallOpPdfs (*C++ member*), 11
SessionOptions::serializedPoprithmsShiftGraphsDir (*C++ member*), 15
SessionOptions::shouldDelayVarUpdates (*C++ function*), 11
SessionOptions::strictOpVersions (*C++ member*), 16
SessionOptions::subgraphCopyingStrategy (*C++ member*), 12
SessionOptions::swapLimitScheduler (*C++ member*), 15
SessionOptions::syntheticDataMode (*C++ member*), 13
SessionOptions::tensorLocationSettingsOverride (*C++ member*), 17
SessionOptions::timeLimitScheduler (*C++ member*), 15
SessionOptions::transitiveClosureOptimizationThreshold (*C++ member*), 16
SessionOptions::useHostCopyOps (*C++ member*), 16
SessionOptions::virtualGraphMode (*C++ member*), 13
SessionOptions::weightTensorLocationSettings (*C++ member*), 16
SGD (*C++ class*), 29
SGD::~SGD (*C++ function*), 31
SGD::clone (*C++ function*), 31
SGD::createOp (*C++ function*), 32
SGD::dampenings (*C++ function*), 33
SGD::fromDefaultMap (*C++ function*), 33
SGD::getInputIds (*C++ function*), 32
SGD::getInverseLossScalingTensorId (*C++ function*), 33
SGD::getOptimizerInputs (*C++ function*), 32
SGD::getSGDAccumulatorAndMomentum (*C++ function*), 31
SGD::getStoredValue (*C++ function*), 32
SGD::getUnsetDampening (*C++ function*), 33
SGD::getUnsetLearningRate (*C++ function*), 33
SGD::getUnsetLossScaling (*C++ function*), 33
SGD::getUnsetMomentum (*C++ function*), 33
SGD::getUnsetVelocityScaling (*C++ function*), 33
SGD::getUnsetWeightDecay (*C++ function*), 33
SGD::hash (*C++ function*), 33
SGD::hasSpecific (*C++ function*), 33
SGD::insertSpecific (*C++ function*), 32

SGD::learningRates (*C++ function*), 33
SGD::momentums (*C++ function*), 33
SGD::resetTensorData (*C++ function*), 32
SGD::setTensorData (*C++ function*), 32
SGD::SGD (*C++ function*), 30, 31
SGD::type (*C++ function*), 31
SGD::type_s (*C++ function*), 31
SGD::validReplacement (*C++ function*), 32
SGD::velocityScalings (*C++ function*), 33
SGD::weightDecays (*C++ function*), 33
Shape (*C++ type*), 100
StashIndex (*C++ type*), 101
StepIOCallback (*C++ class*), 9
StepIOCallback::assertNumElements (*C++ function*), 10
StepIOCallback::in (*C++ function*), 10
StepIOCallback::inComplete (*C++ function*), 10
StepIOCallback::InputCallback (*C++ type*), 10
StepIOCallback::InputCompleteCallback (*C++ type*), 10
StepIOCallback::out (*C++ function*), 10
StepIOCallback::outComplete (*C++ function*), 10
StepIOCallback::OutputCallback (*C++ type*), 10
StepIOCallback::OutputCompleteCallback (*C++ type*), 10
StepIOCallback::StepIOCallback (*C++ function*), 10
StepIOGeneric (*C++ class*), 9
StepIOGeneric::advance (*C++ function*), 9
StepIOGeneric::assertNumElements (*C++ function*), 9
StepIOGeneric::get (*C++ function*), 9
StepIOGeneric::getTensorInfo (*C++ function*), 9
StepIOGeneric::in (*C++ function*), 9
StepIOGeneric::inComplete (*C++ function*), 9
StepIOGeneric::out (*C++ function*), 9
SubgraphCopyingStrategy (*C++ enum*), 23
SubgraphCopyingStrategy::JustInTime (*C++ enumerator*), 23
SubgraphCopyingStrategy::N (*C++ enumerator*), 24
SubgraphCopyingStrategy::OnEnterAndExit (*C++ enumerator*), 23
SubgraphIndex (*C++ type*), 100
SubgraphPartIndex (*C++ type*), 100
SyncPattern (*C++ enum*), 70
SyncPattern::Full (*C++ enumerator*), 70
SyncPattern::ReplicaAndLadder (*C++ enumerator*), 70
SyncPattern::SinglePipeline (*C++ enumerator*), 70
SyntheticDataMode (*C++ enum*), 24
SyntheticDataMode::N (*C++ enumerator*), 24
SyntheticDataMode::Off (*C++ enumerator*), 24
SyntheticDataMode::RandomNormal (*C++ enumerator*), 24
SyntheticDataMode::Zeros (*C++ enumerator*), 24

T

TensorId (*C++ type*), 100
TensorInfo (*C++ class*), 88
TensorInfo::append (*C++ function*), 89
TensorInfo::data_type (*C++ function*), 89
TensorInfo::data_type_lcase (*C++ function*), 89
TensorInfo::dataType (*C++ function*), 89
TensorInfo::dim (*C++ function*), 89
TensorInfo::getDataTypeInfo (*C++ function*), 89
TensorInfo::getOnnxTypeProto (*C++ function*), 89
TensorInfo::isSet (*C++ function*), 89
TensorInfo::metaShape (*C++ function*), 89
TensorInfo::nbytes (*C++ function*), 89
TensorInfo::nelms (*C++ function*), 89
TensorInfo::operator!= (*C++ function*), 89
TensorInfo::operator== (*C++ function*), 89
TensorInfo::rank (*C++ function*), 89
TensorInfo::set (*C++ function*), 88
TensorInfo::shape (*C++ function*), 88
TensorInfo::shape_szt (*C++ function*), 89

TensorInfo::shapeFromString (*C++ function*), 89
TensorInfo::TensorInfo (*C++ function*), 88
TensorInterval (*C++ type*), 101
TensorIntervalList (*C++ type*), 101
TensorLocation (*C++ class*), 89
TensorLocation::isRemote (*C++ function*), 90
TensorLocation::loadTileSet (*C++ member*), 91
TensorLocation::operator!= (*C++ function*), 90
TensorLocation::operator= (*C++ function*), 90
TensorLocation::operator== (*C++ function*), 90
TensorLocation::replicatedTensorSharding (*C++ member*), 91
TensorLocation::serialize (*C++ function*), 90
TensorLocation::shardingDomain (*C++ member*), 91
TensorLocation::storage (*C++ member*), 91
TensorLocation::storageTileSet (*C++ member*), 91
TensorLocation::TensorLocation (*C++ function*), 90
TensorLocationSettings (*C++ struct*), 24
TensorLocationSettings::location (*C++ member*), 24
TensorLocationSettings::minElementsForOffChip (*C++ member*), 24
TensorLocationSettings::minElementsForReplicatedTensorSharding (*C++ member*), 24
TensorLocationSettings::operator= (*C++ function*), 24
TensorLocationSettings::TensorLocationSettings (*C++ function*), 24
TensorStorage (*C++ enum*), 91
TensorStorage::N (*C++ enumerator*), 91
TensorStorage::OffChip (*C++ enumerator*), 91
TensorStorage::OnChip (*C++ enumerator*), 91
TileSet (*C++ enum*), 91
TileSet::Compute (*C++ enumerator*), 91
TileSet::IO (*C++ enumerator*), 91
TileSet::N (*C++ enumerator*), 91
TileSet::Undefined (*C++ enumerator*), 91
TrainingSession (*C++ class*), 5
TrainingSession::~TrainingSession (*C++ function*), 5
TrainingSession::copyFromRemoteBuffer (*C++ function*), 6
TrainingSession::copyToRemoteBuffer (*C++ function*), 6
TrainingSession::createFromIr (*C++ function*), 6
TrainingSession::createFromOnnxModel (*C++ function*), 6
TrainingSession::updateOptimizerFromHost (*C++ function*), 5

V

VarUpdateOp (*C++ class*), 86
VarUpdateOp::aliases (*C++ function*), 86
VarUpdateOp::getReplicatedTensorShardingIndices (*C++ function*), 86
VarUpdateOp::getUpdatedVarOutIndex (*C++ function*), 86
VarUpdateOp::getVarToUpdateInIndex (*C++ function*), 86
VarUpdateOp::growAliasModel (*C++ function*), 86
VarUpdateOp::isOptimizerOp (*C++ function*), 86
VarUpdateOp::modifies (*C++ function*), 86
VarUpdateOp::optimizerInputs (*C++ function*), 86
VarUpdateOp::setup (*C++ function*), 86
VarUpdateOp::VarUpdateOp (*C++ function*), 86
VGraphId (*C++ type*), 101
view (*C++ type*), 100
view::LowBounds (*C++ type*), 100
view::Region (*C++ class*), 91
view::Region::add (*C++ function*), 92
view::Region::append (*C++ function*), 92
view::Region::checks (*C++ function*), 92
view::Region::contains (*C++ function*), 92
view::Region::cut (*C++ function*), 92
view::Region::dimIndex (*C++ function*), 92
view::Region::flatIndex (*C++ function*), 92
view::Region::getAccessType (*C++ function*), 92
view::Region::getEmpty (*C++ function*), 92
view::Region::getFull (*C++ function*), 92
view::Region::getLower (*C++ function*), 92
view::Region::getUpper (*C++ function*), 92

view::Region::intersect (*C++ function*), 92
view::Region::isEmpty (*C++ function*), 92
view::Region::merge (*C++ function*), 92
view::Region::nelms (*C++ function*), 92
view::Region::operator!= (*C++ function*), 92
view::Region::operator== (*C++ function*), 92
view::Region::rank (*C++ function*), 92
view::Region::Region (*C++ function*), 92
view::Region::reshape (*C++ function*), 92
view::Region::reverse (*C++ function*), 92
view::Region::setAccessType (*C++ function*), 92
view::Region::sub (*C++ function*), 92
view::Region::transpose (*C++ function*), 92
view::Regions (*C++ type*), 100
view::RegMap (*C++ type*), 100
view::UppBounds (*C++ type*), 100
VirtualGraphMode (*C++ enum*), 24
VirtualGraphMode::Auto (*C++ enumerator*), 24
VirtualGraphMode::ExecutionPhases (*C++ enumerator*), 24
VirtualGraphMode::Manual (*C++ enumerator*), 24
VirtualGraphMode::N (*C++ enumerator*), 24
VirtualGraphMode::Off (*C++ enumerator*), 24

W

WeightDecayMode (*C++ enum*), 27
WeightDecayMode::Decay (*C++ enumerator*), 27
WeightDecayMode::L2Regularization (*C++ enumerator*), 27